UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Algoritmos Paralelos para la Reducción de Sistemas Lineales de Control Estables

Tesis doctoral de

David Guerrero López

Director: Dr. José E. Román Moltó

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Septiembre 2015

Dedicado a mi madre.

¡Mamá!
Gracias por todo. ☺

Agradecimientos

Gracias, Vicente.
Muchas gracias, Jose.

Hace ya varios lustros que comencé mi labor investigadora incorporándome en el grupo COPA (grupo de COmputación PAralela, que actualmente es el GRyCAP), liderado por Vicente Hernández y Antonio M. Vidal. A ellos se debe en parte la atracción que siento hacia la computación paralela, que espero siga conmigo por muchos años. Gracias a ambos.

Vicente además se prestó a dirigirme la tesis, aunque le ha llegado la jubilación antes de que yo la terminara. Siempre he recibido su apoyo. Su sabiduría me ha resuelto multitud de dudas, muchas veces dirigiéndome hacia donde las pudiera resolver por mí mismo. Gracias, Vicente.

En la última etapa de esta tesis ha sido José E. Román su sustituto en la difícil tarea de dirigirme. Quiero agradecerle, primero, que aceptara la dirección, y luego, que en todo momento me ayudara con todo lo relacionado con la tesis. Su conocimiento en ciertos campos numéricos es admirable, lo que me enriquece cada vez que los abordamos, cosa que espero siga ocurriendo por mucho tiempo. Es un verdadero lujo recibir clases suyas en seminarios dedicados a temas numéricos concretos. Muchas gracias, Jose. Y gracias también por todo lo que comento ahora de otros compañeros.

Quiero también dar las gracias por su comprensión y motivación a algunos investigadores importantes en el campo de la teoría de control con los que he tenido el privilegio de coincidir en reuniones de proyectos y algunos congresos: Enrique Quintana, Peter Benner, Vasile Sima, Andras Varga y especialmente Sven Hammarling, autor del método para resolver ecuaciones de Lyapunov que he paralelizado en esta tesis, que tuvo la gentileza de obsequiarme una versión impresa del artículo en que lo presenta. Todos ellos han sido muy cordiales tanto en lo social como en lo profesional.

Aunque no llegamos a conocernos, quiero agradecer a Thilo Penzl su dedicación a la resolución de la ecuación de Lyapunov. Uno de sus artículos fue de mis primeros contactos con este problema. Conservo este artículo en forma de un cuaderno muy gastado. Fue objeto de consulta durante largo tiempo.

Ya en el tema personal, quiero al menos nombrar a una pequeña parte (y son muchos) de los compañeros que me alegran el día a día. Puede que no me vea en otra situación en la que pueda hacerlo.

Doy gracias a todos esos compañeros de investigación a los que presté mi ayuda sin darme cuenta de lo mucho que aprendía mientras les ayudaba: Pedro, William, Georgina, Juan Manuel y, sobre todo, Carlos Campos. La mayoría de ellos ya son orgullosos doctores.

También quiero aprovechar para acordarme de antiguos miembros del grupo de investigación, cuya camaradería fue muy beneficiosa para todos: Kike, Héctor, Rafa, Gabi, ... Dejaron el grupo para dirigirse hacia otros destinos, que espero que les deparen muchos éxitos.

Y por supuesto doy las gracias también a todos los miembros actuales del grupo de investigación GRyCAP (grupo de GRid y Computación de Altas Prestaciones): José Miguel, Germán, Damián, Carlos, Miguel, Enrique, Erik, ... y de forma especial a Ignacio Blanquer, que siempre me ha ayudado cuanto ha podido en todo. Gracias, Nacho.

En el tema docente he tenido la suerte de dar siempre (hasta ahora) asignaturas que me gustan y con gente extraordinaria. He aprendido mucho tanto de las asignaturas impartidas como de los compañeros con los que las he compartido. Gracias, Víctor, Vicent, Javier, ... y especialmente Vicente y Álvaro, no tanto por su ayuda en la docencia, que también, sino por su conocimiento de temas tan diversos y entretenidos que siempre es un placer conversar con ellos.

Me he dejado para el final los compañeros más cercanos en el departamento con los que tropiezo con más frecuencia y siempre es un placer compartir breves momentos: Pedro, Fernando, Salva, Carmen, Alex, ... Gracias a todos.

Por último, quiero pedir disculpas a todas las personas no mencionadas explícitamente. Doy gracias, porque afortunadamente son muchas las personas que tengo a mi alrededor haciéndome más agradable el trabajo ... y la vida.

Resumen

En el campo de la teoría de control en ocasiones aparecen modelos de sistemas con un tamaño elevado (muchas variables de estado). Cuando se pretende simular, estudiar o controlar uno de estos sistemas de orden elevado, conviene realizar un trabajo previo de reducción del modelo del sistema con el propósito de reducir los costes (económicos/temporales) necesarios en un tratamiento posterior. El proceso de obtención de un sistema de orden reducido que represente adecuadamente el sistema original suele ser costoso, ya que necesariamente se tiene que hacer con el sistema original sin reducir. Por esto, resulta conveniente disponer de implementaciones de altas prestaciones para el problema de reducción de sistemas lineales de control.

En esta tesis se han desarrollado implementaciones de altas prestaciones para algunos métodos de reducción de modelos. Se han analizado los algoritmos existentes para la reducción de modelos de sistemas lineales de control estables y sus implementaciones en la librería de control SLICOT. Se han propuesto nuevos algoritmos paralelos para los métodos cuyo núcleo principal es la resolución de ecuaciones de Lyapunov. Las nuevas rutinas desarrolladas se incorporan a la librería de computación de altas prestaciones para control PSLICOT.

Aparte de las funciones principales a cargo de la reducción de modelos, se han tenido que paralelizar también todas aquellas operaciones numéricas que aparecen en este problema y de las que no se disponía de versiones de altas prestaciones. De estas operaciones, cabe destacar rutinas paralelas para la resolución de la ecuación de Lyapunov en su forma estándar obteniendo directamente el factor de Cholesky de la solución, que es lo que se necesita para la reducción de modelos. El método utilizado es una versión paralela del método de Hammarling. Las rutinas implementadas resuelven en paralelo y para matrices densas las cuatro variantes posibles de la ecuación de Lyapunov: en su forma discreta y continua, traspuestas y sin trasponer.

Todas las rutinas paralelizadas ofrecen una interfaz como la de las rutinas de la librería ScaLAPACK, para que puedan ser usadas con facilidad por el usuario habituado a trabajar con este tipo de librerías. Se permiten las mismas distribuciones de datos que en esta librería: una distribución cíclica 2D por bloques, que engloba muchas otras distribuciones.

Gracias al trabajo desarrollado, ahora se dispone de versiones paralelas de altas prestaciones para reducir sistemas lineales de control mediante diferentes variantes del método de balanceado y truncamiento de la raíz cuadrada (*the Square-Root Balance & Truncate model reduction method*): con o sin balanceado y con o sin usar

las fórmulas de perturbación singular. Se trata de versiones paralelas de los mismos algoritmos y métodos que se utilizan en las versiones secuenciales de la librería SLICOT. Esto permitirá reducir de forma eficiente modelos de sistemas lineales de control de gran tamaño.

También se ha mejorado la aplicabilidad del software existente en ScaLAPACK para el problema de valores propios cubriendo casos que no se contemplaban. Se ha trabajado en la solución del problema generalizado (mediante su transformación a forma estándar, lo que no es aplicable en todos los casos) y también en el cálculo de los vectores propios. Ambas operaciones se han utilizado en un problema real de simulación de flujos oceánicos. En esta aplicación se requiere el cálculo de todos los valores y vectores propios de un problema generalizado de gran dimensión. Como consecuencia, ha sido posible resolver problemas de valores propios generalizados enormes (con matrices de más de 400000 filas y columnas) que no habían podido resolverse con anterioridad, permitiendo así un estudio más preciso del comportamiento de las corrientes oceánicas.

Resum

Algoritmes paral·lels per a la reducció de sistemes lineals de control estables

En el camp de la teoria de control de vegades apareixen models de sistemes amb un tamany elevat (moltes variables d'estat). Quan es pretén simular, estudiar o controlar un d'aquests sistemes d'ordre elevat, convé realitzar un treball previ de reducció del model del sistema amb el propòsit de reduir els costos (econòmics/temporals) necesaris en un tractament posterior. El procés d'obtenció d'un sistema d'ordre reduit que represente adequadament el sistema original sol ser costós, perque necessàriament ha de fer-se amb el sistema original sense reduir. Per aquest motiu, resulta convenient disposar d'implementacions d'altes prestacions per al problema de reducció de sistemes lineals de control.

En aquesta tesis s'han desenvolupat implementacions d'altes prestacions per a alguns mètodes de reducció de models. S'han anal·litzat els algoritmes existents per a la reducció de models de sistemes lineals de control estables i les seues implementacions en la llibreria de control SLICOT. S'han proposat nous algoritmes paral·lels per als mètodes basats en la resolució d'equacions de Lyapunov. Les noves rutines desenvolupades s'incorporen a la llibreria de computació d'altes prestacions per a control PSLICOT.

Apart de les funcions principals a càrrec de la reducció de models, s'han hagut de paral·lelitzar també totes aquelles operacions numèriques que apareixen en aquest problema i per a les que no es disposava de versions d'altes prestacions. D'aquestes operacions, destaquen rutines paral·leles per a la resolució de l'equació de Lyapunov en la seua forma estàndard obtenint directament el factor de Cholesky de la solució, que és el que es necessita per a la reducció de models. El mètode emprat és una versió paral·lela del mètode de Hammarling. Les rutines implementades resolen en paral·lel i per a matrius denses les quatre variants possibles de l'equació de Lyapunov: en la seua forma discreta i contínua, traspostes i sense trasposar.

Totes les rutines paral·lelitzades ofereixen una interfaç com la de les rutines de la llibreria ScaLAPACK, per a que puguen ser usades fàcilment per l'usuari acostumat a treballar amb aquest tipus de llibreries. Es permeten les mateixes distribucions de dades que en aquesta llibreria: distribució cíclica 2D per blocs, que engloba moltes altres distribucions.

Gràcies al treball desenvolupat, ara es disposa de versions paral·leles d'altes prestacions per a reduir sistemes lineals de control mitjançant diferents variants del mètode de balancejat i truncament de l'arrel quadrada (*the Square-Root Balance & Truncate model reduction method*): amb o sense balancejat i amb o sense usar les fórmules de perturbació singular. Son versions paral·leles dels mateixos algoritmes i mètodes que s'utilitzen en les versions sequencials de la llibreria SLICOT. Això permetrà reduir de forma eficient models de sistemes lineals de control de gran tamany.

També s'ha mitjorat l'aplicabilitat del software existent en ScaLAPACK per al problema de valors propis cobrint casos que no es contemplaven. S'ha treballat en la solució del problema generalitzat (mitjançant la seua transformació a forma estàndard, cosa que no es pot fer sempre) i també en el càlcul dels vectors propis. Ambdues operacions s'han utilitzat en un problema real de simulació de fluxos oceànics. En aquesta aplicació es requereix el càlcul de tots els valors i vectors propis d'un problema generalitzat de gran dimensió. Com a conseqüència, ha sigut possible resoldre problemes de valors propis generalitzats molt grans (amb matrius de més de 400000 files i columnes) que no s'havien pogut resoldre anteriorment, permetent així un estudi més precís del comportament de les corrents oceàniques.

Summary

Parallel algorithms for the reduction of stable linear control systems

In the field of control theory, sometimes system models of big size (with many state variables) appear. When one of these high order systems needs to be simulated, studied or controlled, it is convenient to perform a previous work of model reduction in order to reduce the necessary (economic and temporal) costs. This process of obtaining a low order adequate representation of the original system usually has a high cost, because it has to be done with the original unreduced system. Thus, it is important to have high performance implementations for the problem of reducing linear control systems.

In this thesis high performance implementations for some methods of model reduction have been developed. Current algorithms for model reduction of stable linear control systems and their implementation in the control library SLICOT have been analysed. New parallel algorithms for the methods strongly based on solving Lyapunov equations have been proposed. The new developed routines are incorporated in the high performance library for control PSLICOT.

Apart from the main functions in charge of model reduction, all operations that appear in the problem and do not have a high performance version yet have also been parallelised. One of these operations is the solution of Lyapunov equations in standard form. Parallel routines for solving these equations have been developed. These routines solve the equation obtaining directly the Cholesky factor of the solution, which fits better their application in model reduction. For this, Hammarling's method has been parallelised. The new routines solve in parallel and for dense matrices the four possible variants of standard Lyapunov equations: discrete and continuous versions, both transposed and not transposed.

Interfaces offered by all the parallelised routines are similar to that of the existing routines in ScaLAPACK library, so they are easy to use from a user of this kind of libraries. The new routines work with the same data distribution used in this library: 2D block cyclic distribution, which allows many other distributions.

Thanks to the developed work, now there are available high performance parallel routines to reduce linear control systems by using different variants of *the Square-Root Balance & Truncate model reduction method*: with or without balancing and

with or without using the singular perturbation approximation formulas. They are parallel implementations of the same algorithms and methods used in the sequential routines of the SLICOT library. This allows to efficiently reduce models of linear control systems of big size.

Moreover, existing software in ScaLAPACK for the eigenvalue problem has been improved by covering cases not treated there: the solution of the generalised problem (by transforming it into standard form, which is not always possible) and the computation of the eigenvectors. This part of the work has been applied to a real problem of simulation of oceanic flows. Here, it is necessary to compute all the eigenvalues and eigenvectors of a generalised eigenvalue problem with a very big dimension. As a consequence, enormous eigenvalue problems have been solved (with matrices of order greater than 400000), that could not be solved previously. Solving them allows to improve the precision in the studies of the behaviour of oceanic flows.

Índice

Capítulo 1

Introducción

En este capítulo se describe brevemente el trabajo que se ha pretendido cubrir con la realización de esta tesis. Inicialmente se explica resumidamente el problema a resolver y el interés en hacerlo mediante una implementación de altas prestaciones. A continuación se enumeran los objetivos perseguidos en la tesis. Luego se describe la estructura que sigue el resto de la memoria. Por último, se comentan algunos aspectos a tener en cuenta en las descripciones de las nuevas rutinas implementadas que se realizan en próximos capítulos.

1.1. Escenario

En la actualidad abundan los sistemas de control automáticos, en los que un determinado sistema es controlado por una máquina (ya sea mecánica o electrónica). En el trabajo con estos sistemas aparece la necesidad de representarlos matemáticamente para poder manejarlos y desarrollar el control pertinente. Existen múltiples formas de representar un sistema de control, sin embargo, todas ellas tienen algo en común y es que a medida que el sistema representado trata de aproximarse más al modelo físico real, las dimensiones del modelo crecen.

El hecho de que un modelo de un sistema sea *más grande* que otro no es algo que carezca de importancia. Al contrario. Todos los procesos que vayan a realizarse sobre ese modelo van a tener que trabajar con su representación. Y una representación *más grande* va a implicar un mayor coste en su procesado. Este *coste* puede entenderse bien como un mayor tiempo a la hora de realizar un trabajo con el modelo, o bien como unos mayores requerimientos del sistema en que se va a realizar ese procesado.

En particular, si se trabaja por ejemplo con un proceso que requiera un control en tiempo real, va a ser muy importante el coste de este control. Existirán unos requerimientos temporales para su procesado. Básicamente hay dos formas de *reducir* los requerimientos temporales de un proceso de control:

- utilizar máquinas más potentes para realizar el control,
- *reducir* el proceso de control que se está realizando.

En el caso de optar por utilizar unas máquinas más potentes, esto va a encarecer el sistema de control. Además, siempre va a llegar un momento en que no se podrá mejorar el rendimiento de las máquinas que realizan el control, bien porque no existan máquinas más potentes, bien porque el precio se vuelva prohibitivo, o bien porque se desee también controlar el consumo energético necesario.

Es más habitual empezar por simplificar el modelo de control. Con esto suele ser suficiente para lograr el objetivo del tiempo real y si no, es un paso en la dirección adecuada que hará que no haya que incrementar excesivamente la potencia de cálculo necesaria.

Normalmente, para lograr simplificar el control de un proceso, lo primero es empezar simplificando el modelo matemático que representa el proceso a controlar. Al simplificar el modelo del proceso, el control necesario resultará más sencillo.

Además, otra ventaja de la simplificación de la representación del sistema a controlar es que cualquier proceso a realizar sobre el modelo requerirá de un menor coste (ya sea tiempo o potencia computacional o consumo energético). Cuanto más pequeño sea el modelo que representa el sistema con el que se trabaja, más fácil será trabajar con él. En algunos casos incluso es requisito imprescindible realizar la reducción del modelo del sistema para poder trabajar con él, debido a que el modelo sin reducir requiere más memoria/potencia de la que se dispone.

Por todo esto, los algoritmos de reducción de modelos son importantes y necesarios en la teoría de control.

Sin embargo, el proceso de reducir un modelo suele ser bastante costoso, en parte debido a que debe enfrentarse con el modelo del sistema sin reducir, que puede ser de gran tamaño. Es por esta razón, por la que el empleo de cualquier técnica que permita reducir los costes necesarios para reducir un modelo resulta muy interesante.

Esta tesis se centra en la implementación de *algoritmos de altas prestaciones para la reducción de modelos*. Con *algoritmos de altas prestaciones* se está queriendo indicar algoritmos que permitan disminuir el tiempo necesario para reducir un sistema o bien que permitan enfrentarse a problemas que por su tamaño no pueden ser abordados con algoritmos tradicionales. Básicamente, se trata de versiones paralelas que reparten el trabajo entre varios procesadores para lograr estas ventajas.

Actualmente existen bastantes implementaciones secuenciales de algoritmos conocidos para la reducción de modelos en sistemas de control. Sin embargo, hay pocas implementaciones de altas prestaciones de estos algoritmos.

Se ha seleccionado una de las múltiples librerías que tienen implementaciones eficientes y robustas de varios algoritmos de reducción de modelos, que es la librería SLICOT. Esta librería está basada en otras y lleva un tiempo razonable de existencia y desarrollo, lo que es un punto a favor de su estabilidad y corrección.

Los algoritmos de reducción de modelos presentes en esta librería trabajan con la representación en el espacio de estados de los sistemas lineales de control. Esta representación está basada en el uso de una serie de matrices para cada sistema, lo que resulta cómodo desde el punto de vista computacional.

En ella hay varios algoritmos de reducción de modelos. Por tratar de centrar la tesis, se ha trabajado sobre el subconjunto de algoritmos orientados a sistemas lineales de control estables. Estos algoritmos son un paso previo necesario para abordar

los algoritmos dirigidos a sistemas inestables.

Los algoritmos seleccionados para este trabajo operan con matrices en forma densa y utilizan como núcleo computacional básico la resolución de la ecuación de Lyapunov, obteniendo el factor de Cholesky de la solución. Implementaciones de altas prestaciones para llevar a cabo esta operación forman el núcleo central del presente trabajo.

Además de esa operación numérica, otras operaciones con las que se ha tenido que trabajar incluyen el cálculo de la forma real de Schur de una matriz y la descomposición en valores singulares.

1.2. Objetivos

El objetivo principal perseguido en la elaboración de esta tesis es el siguiente:

- Desarrollar algoritmos de altas prestaciones para la reducción de modelos de sistemas lineales de control.

Sin embargo, este objetivo es muy amplio y abarca mucho más de lo pretendido aquí. Este objetivo muy general puede dirigirse mediante la especialización a un objetivo más concreto que es:

- Desarrollar algoritmos de altas prestaciones para la reducción de modelos de sistemas lineales de control estables, mediante métodos basados en la resolución de la ecuación de Lyapunov sobre matrices densas.

Al pretender desarrollar algoritmos de altas prestaciones para la reducción de modelos, resulta conveniente comenzar por los algoritmos dirigidos a sistemas estables y, de ellos, a aquellos que trabajan en el campo denso.

Este objetivo puede desglosarse en dos importantes tareas:

- Implementar rutinas paralelas para la resolución de ecuaciones de Lyapunov obteniendo el factor de Cholesky de la solución.
- Paralelizar las rutinas de reducción de modelos estables de la librería SLICOT que están basadas en la resolución de la ecuación de Lyapunov, obteniendo así versiones de altas prestaciones de los siguientes métodos de reducción de modelos[1]:
  - SR B&T (*the Square-Root Balance & Truncate model reduction method*): el método de balanceado y truncamiento de la raíz cuadrada.
  - BFSR B&T (*the Balancing-Free Square-Root Balance & Truncate model reduction method*): el método de balanceado y truncamiento de la raíz cuadrada sin balanceado.

[1] Los nombres en castellano de los métodos de reducción de modelos listados pueden confundir al lector acostumbrado a material en inglés. Por esta razón se mantiene entre paréntesis la nomenclatura inglesa.

- SR SPA (*the Square-Root Singular Perturbation Approximation model reduction method*): el método de aproximación de perturbación singular de la raíz cuadrada.
- BFSR SPA (*the Balancing-Free Square-Root Singular Perturbation Approximation model reduction method*): el método de aproximación de perturbación singular de la raíz cuadrada sin balanceado.
- SPA (*the Singular Perturbation Approximation method*): el método de fórmulas de aproximación de perturbación singular.

El objetivo último es permitir a los usuarios realizar en un tiempo razonable la reducción de modelos de gran dimensión, formados por decenas de miles de variables de estado, o incluso más.

1.3. Estructura de la memoria

La memoria de la tesis se ha organizado en seis capítulos.

El capítulo 2 presenta algunos conceptos básicos necesarios para entender las explicaciones de computación de altas prestaciones realizadas en los otros capítulos. Comienza describiendo de forma breve algunos conceptos básicos sobre la computación de altas prestaciones. Se introducen los índices de prestaciones más comúnmente utilizados para comprobar la calidad de algoritmos de altas prestaciones, que posteriormente se utilizarán para medir los algoritmos desarrollados.

Para maximizar la portabilidad y eficiencia del código, es importante utilizar en lo posible las librerías software estándar disponibles, además de que el nuevo software desarrollado siga un esquema similar. Algunas de las librerías más comunes en el trabajo de computación de altas prestaciones con matrices densas se describen también en el capítulo 2.

Los capítulos 3, 4 y 5 representan el núcleo central del trabajo desarrollado en esta tesis. En ellos se explican los tres problemas a los que ha habido que enfrentarse en el desarrollo de los nuevos algoritmos de altas prestaciones:

- Problemas de valores propios, en el capítulo 3.
- La ecuación de Lyapunov, en el capítulo 4.
- La reducción de modelos, en el capítulo 5.

En todos ellos se empieza por hacer un recordatorio de los conceptos necesarios para afrontar cada uno de esos problemas, citando además las formas en que habitualmente han sido resueltos. Posteriormente se enumeran las nuevas rutinas de altas prestaciones desarrolladas para resolverlos, incluyendo alguna explicación sobre la forma en que han sido implementadas. Y finalmente se muestran algunos resultados experimentales obtenidos con las nuevas rutinas, seguidos de unas breves conclusiones.

Antes de mostrar los resultados experimentales por primera vez, en el capítulo 3, se describen las plataformas en las que se han ejecutado las pruebas realizadas con

las nuevas rutinas desarrolladas y de las que se muestran índices de prestaciones a lo largo de todo este documento.

En el último capítulo se puede leer un breve resumen del contenido de la tesis junto a las conclusiones obtenidas con el trabajo. Para finalizar se comentan algunas posibles líneas de trabajo futuro en las que continuar el trabajo iniciado en esta tesis.

1.4. Aspectos a considerar en la descripción de rutinas desarrolladas

En los capítulos 3, 4 y 5 se describen una a una las rutinas de altas prestaciones desarrolladas en el marco de este trabajo.

Generalmente se trata de versiones paralelas de rutinas secuenciales existentes en las librerías LAPACK o SLICOT. (En el capítulo 2 se presentan estas librerías). Se trata de versiones de altas prestaciones para aquellos algoritmos necesarios y para los que todavía no se disponía de una versión de altas prestaciones. Aparte obviamente de las propias rutinas para la reducción de modelos de sistemas lineales de control estables, también se han paralelizado rutinas para resolver problemas de valores propios, calcular la descomposición en valores singulares y resolver ecuaciones de Lyapunov.

También hay algunas rutinas que no tienen su versión secuencial, por ejemplo porque su implementación secuencial es trivial y no merece la pena tener una rutina para eso, aunque en paralelo ya no es trivial y ha resultado conveniente implementarlas aparte.

Aquellas rutinas correspondientes a paralelizaciones de rutinas secuenciales existentes en alguna de las librerías estándar mencionadas (LAPACK, SLICOT) conservan su nombre precedido de la letra `P` (de Paralelo). Las demás son nuevas rutinas auxiliares para operaciones que en el caso secuencial no revierten una especial dificultad, pero que en el caso paralelo resulta necesario o conveniente independizar en rutinas sueltas. Para ellas se han utilizado nombres relacionados con la labor que realizan.

Debe notarse que, si bien en la mayoría de las nuevas rutinas paralelas se realizan las mismas operaciones que en sus versiones secuenciales, en algunas de ellas no se ofrece su funcionalidad completa, sino sólo aquello que resulta necesario para su uso en los algoritmos de reducción de modelos que son el objetivo del trabajo.

Además, se ha procurado que todas las nuevas rutinas desarrolladas sigan el formato y se usen de forma similar a las rutinas paralelas ya existentes tanto en ScaLAPACK como en PSLICOT [BGH$^+$98]. Con esto, su uso resultará sencillo para cualquier usuario acostumbrado a trabajar con estas librerías.

Al explicar los algoritmos de las nuevas rutinas, se ha pretendido ofrecer la suficiente información como para entender la forma en que se realizan las operaciones principales y sus ventajas por hacerlas en paralelo, sin llegar al nivel de detalle presente en las implementaciones reales. Si en algún caso se desea conocer alguna operación de forma más profunda, siempre puede acudirse al código.

Se ha marcado en *negrita* aquellas rutinas más importantes dentro de cada grupo. Aquí se identifican como 'más importantes' las que se encuentran más altas en los diferentes árboles de llamadas mostrados en diferentes figuras a lo largo de la tesis (ejemplo: figura 5.2). Se trata de las rutinas que ofrecen operaciones de más alto nivel y que más probablemente puedan ser utilizadas directamente por un futuro usuario. Esto no quita que muchas de las otras rutinas también tengan suficiente funcionalidad como para poder ser utilizadas de forma independiente del resto. Nótese que las operaciones básicas implementadas representan algoritmos bastante utilizados y que pueden resultar útiles para otros problemas numéricos.

La dificultad a la hora de paralelizar y la relevancia del código de cada rutina no es relación directa de esta 'importancia'. Algunas rutinas para tareas más específicas no tienen nada que envidiar, en lo que a esfuerzo de implementación se refiere, a rutinas más generales que están más próximas al usuario. De hecho, en muchas ocasiones la dificultad de implementación de las rutinas es inversa a su cercanía a estas rutinas más próximas al usuario.

En el texto correspondiente se hace mención con cierta frecuencia a los elementos computacionales que procesarán las operaciones necesarias para cada algoritmo paralelo. Se ha procurado denominarlos *procesos*, por ser un término bastante genérico. En general es más correcto hablar de *procesos* en lugar del término más habitual *procesadores*. Esto es así porque cuando se habla de *procesos* no se especifica dónde se están ejecutando, permitiendo así englobar tanto una plataforma paralela virtual que esté siendo simulada en un único computador, como el caso más favorable en que realmente se esté trabajando sobre una plataforma paralela real y los *procesos* vayan a ejecutarse en *procesadores* diferentes. En este ámbito es más adecuado utilizar el término *procesos*, aunque si se piensa en trabajar únicamente en plataformas paralelas puede sustituirse por *procesadores*.

En múltiples ocasiones se presenta lo que aquí se denomina "esquema gráfico de las operaciones de un algoritmo". Esto no es más que una representación visual sencilla que trata de dar una idea de la sucesión de operaciones y comunicaciones que se producen en la malla de procesos. La plataforma paralela se representa como un conjunto de cuadrados donde cada uno simboliza un proceso dentro de la malla de procesos. Generalmente se trabajará en estos esquemas con una malla bidimensional de 4×4 procesos. En la figura 1.1 se ilustra esta malla de procesos junto a la distribución no cíclica de una supuesta matriz A en ella. Los diferentes a_{ij} no representan elementos de la matriz sino bloques de ella del tamaño elegido en la distribución. p_{ij} representa el proceso de la fila i y columna j de la malla de procesos.

En los esquemas que luego se presentan, la evolución temporal de los procesos se ilustra mediante la repetición de la malla de procesos múltiples veces. El orden para un tiempo creciente es el orden habitual de lectura: de izquierda a derecha y luego de arriba a abajo, de forma que se puede seguir como en un cómic los diferentes estados por los que pasa la malla de procesos.

Sin embargo, la situación que se muestra en los esquemas no tiene por qué coincidir exactamente con lo que va a suceder en la realidad. Se muestran diferentes etapas por las que se puede pasar en la ejecución de un algoritmo dado. El orden en que estas situaciones se producen en un proceso concreto se corresponde exactamente

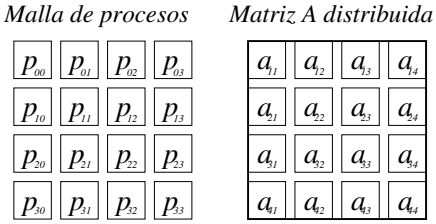

Figura 1.1: Ejemplo de malla de procesos y distribución no cíclica de una matriz en ella

con lo ilustrado. Pero no siempre estarán todos los procesos en una misma viñeta del esquema. En las implementaciones no se ha forzado ningún tipo de sincronismo que haga que los procesos vayan haciendo las mismas cosas a la vez. De haberlo hecho, se estaría "frenando" a algunos procesos. Los distintos procesos se sincronizan de forma natural cuando tienen que recibir información de otros. Pero mientras, cada uno puede adelantarse o retrasarse en sus operaciones. Teniendo en cuenta que los fragmentos de matrices que tiene cada proceso no tienen por qué ser del mismo tamaño en todos ellos, el coste de realizar las operaciones no será el mismo. En general, el tamaño de la porción de datos con la que se trabaja va cambiando a medida que avanza el algoritmo (típicamente irá reduciéndose). Por una u otra razón, es frecuente que el trabajo a realizar en cada proceso no suponga el mismo coste. Si se trabaja con problemas grandes, el desnivel existente será menor, pero existirá. Pues bien, tal como se han implementado los algoritmos, no se fuerza sincronismo innecesario. Cuando un proceso termine con las operaciones que esté realizando pasará a la siguiente tarea. De esta manera se facilita que todos los procesos se encuentren trabajando, aunque sea en etapas diferentes del mismo algoritmo. En la práctica tampoco es que vayan a estar muy distanciados en el algoritmo, puesto que en cuanto alguno requiera datos de otro, se quedará esperando hasta recibirlos.

En estos esquemas normalmente se ilustra el algoritmo en su versión no cíclica. Esto es así, porque en estos casos es más sencillo entender el algoritmo y además el esquema queda más simple. Pero las implementaciones se han desarrollado siempre permitiendo el caso de una distribución cíclica. Si se utiliza esta distribución cíclica, las operaciones que en los esquemas suelen involucrar a unos pocos procesos involucrarán a muchos más, porque ahora la matriz se encuentra más distribuida entre ellos. En general, el nivel de paralelismo real (por usar distribuciones cíclicas) es mayor que el que se aprecia visualmente en estos esquemas (por usar en ellos distribuciones no cíclicas).

Capítulo 2

Computación de altas prestaciones

En este capítulo se hace una breve introducción a la computación en paralelo, entendida como punto de referencia en la computación de altas prestaciones (aunque ésta también incluye algoritmos secuenciales optimizados), y se presentan algunas librerías software.

En un primer apartado se muestra la importancia del procesamiento en paralelo para la computación actual. Se presenta una clasificación de los computadores para procesamiento paralelo, lo que da una idea sobre las diferentes formas de programación de computadores paralelos. Se citan algunos índices de prestaciones ampliamente utilizados en computación en paralelo, que se emplearán en futuros capítulos. Y se analizan algunos elementos propios de la computación en paralelo que juegan un importante papel en las prestaciones de los algoritmos para máquinas paralelas.

Después se presentan algunas de las librerías software más utilizadas en computación numérica. La importancia de este tipo de librerías reside en que si se realizan programas utilizando llamadas a sus rutinas, puede conseguirse un alto grado de portabilidad y buenas prestaciones.

La portabilidad es la facilidad de migración del código de una máquina a otra. Gracias a la estandarización de estas librerías podrá obtenerse portabilidad, a nivel de código fuente, entre todas las máquinas para las que se disponga de una versión de las librerías.

La programación utilizando librerías fomenta también la obtención de buenas prestaciones. Se requiere diseñar los algoritmos orientándolos al uso de núcleos básicos de computación que ofrezcan las librerías. Para optimizar un programa implementado según este diseño en diferentes máquinas, bastará con optimizar esos núcleos básicos, que son precisamente parte de la librería y por tanto el fabricante o desarrollador de la librería se habrá encargado de optimizar para cada máquina concreta.

Otras ventajas que aporta la programación utilizando rutinas de librerías es la

legibilidad que se logra al simplificar el código de los programas sustituyendo múltiples líneas de código por una llamada a función. También mejora la robustez de los programas, haciendo más difícil la confusión en los índices y manejo de las matrices, ya que gran parte de las operaciones las realiza la librería de forma transparente al programador.

Por tanto, es conveniente utilizar librerías de álgebra lineal en la implementación de algoritmos para los que se deseen características de portabilidad, buenas prestaciones, legibilidad, robustez, ...

En este capítulo se presentan algunas de las librerías empleadas en la implementación de los algoritmos de reducción de modelos. Se pretende, por una parte, dar una idea de cómo se usan estas librerías y del potencial que ofrecen al programador de aplicaciones científicas. Por otra parte, es interesante observar la interfaz ofrecida por las librerías, puesto que se procura ofrecer una interfaz similar en las nuevas rutinas desarrolladas. El objetivo perseguido con esto es que el usuario acostumbrado a trabajar con esta clase de librerías no tenga ninguna dificultad a la hora de utilizar las nuevas rutinas.

Se ilustran por separado librerías secuenciales y librerías paralelas.

En el caso de las librerías secuenciales, se presentan librerías de álgebra lineal (BLAS y LAPACK) y librerías más orientadas a la teoría de control (SLICOT).

En el caso de las librerías de computación paralela, en primer lugar se introducen las librerías de comunicación más utilizadas (MPI, BLACS) y que ofrecen operaciones necesarias en cualquier librería paralela. Posteriormente se presentan algunas librerías paralelas de álgebra lineal (PBLAS, ScaLAPACK) y librerías paralelas dirigidas al trabajo con sistemas lineales de control (PSLICOT, PLICMR), además de citar muy brevemente otras librerías (SCASY, Elemental y libFLAME).

2.1. Conceptos básicos de la computación paralela

En los últimos años, aparecen cada vez más problemas que requieren un alto grado de computación para su resolución. Esto provoca la necesidad de fabricar ordenadores potentes para solucionarlos.

Los computadores secuenciales son aquellos en los que se van procesando las instrucciones de una en una y en orden secuencial una detrás de otra. Cada vez con más frecuencia se utilizan técnicas como segmentación o computación vectorial, para incrementar su productividad. También se aprovechan mejor los avances tecnológicos, que les permiten acelerar su velocidad de cómputo.

Sin embargo, a medida que un computador se va especializando, convirtiéndose así en un computador avanzado, va alcanzando un precio desorbitado pese a sus buenas prestaciones. Con otros ordenadores menos avanzados de coste inferior pueden obtenerse buenas prestaciones al unirlos conformando un multiprocesador. Además, la velocidad de un único computador secuencial siempre vendrá limitada por la tecnología existente. En un multiprocesador, puede aumentarse su capacidad computacional aumentando el número de procesadores que lo forman, por encima de las prestaciones de un único computador secuencial.

Los multiprocesadores ofrecen una buena relación calidad/precio frente a los supercomputadores.

Por todo esto surge la necesidad de trabajar con computación en paralelo, para mejorar las prestaciones obtenidas en máquinas secuenciales. Mediante la computación en paralelo se puede trabajar con multiprocesadores, utilizando así su capacidad para ejecutar múltiples instrucciones/programas al mismo tiempo.

Generalmente, serán más difíciles de programar los algoritmos para computadores paralelos que sus versiones para computadores secuenciales, pero permiten ofrecer unas mejores prestaciones.

Una máxima a perseguir en la programación en paralelo es repartir la carga del problema lo más equilibradamente posible entre todos los procesadores de que se disponga, para que al estar mejor repartido el trabajo, éste se resuelva en menos tiempo.

2.1.1. Arquitecturas paralelas

A la hora de realizar una catalogación de máquinas paralelas, suelen mencionarse dos clasificaciones típicas [KGGK94]: una atendiendo al mecanismo de control en los diferentes procesadores, y otra atendiendo a cómo se ve el espacio de direcciones en cada procesador.

Clasificación según el mecanismo de control

Según el mecanismo de control los computadores paralelos pueden clasificarse en:

- **SIMD** (Simple Instrucción Múltiples Datos)

  En este tipo de computadores, existe tan sólo una única unidad de control que indica la operación a realizar con múltiples datos. La operación es realizada por los elementos de proceso existentes, donde cada elemento de proceso ejecutará la operación sobre sus datos.

  Ejemplos de este tipo de computadores son los procesadores vectoriales y algunos procesadores sistólicos.

- **MIMD** (Múltiple Instrucción Múltiples Datos)

  Estos computadores poseen en cada elemento de proceso una unidad de control que se encarga de indicar en cada momento la operación a realizar. Cada elemento de proceso puede realizar una operación concreta sobre sus datos, independientemente de las operaciones que estén realizando el resto de elementos de proceso. Entre los diferentes elementos de proceso habrá algún tipo de comunicación que los haga trabajar de forma coordinada.

La diferencia fundamental entre ambos tipos de computadores paralelos es la posibilidad de los **MIMD** de ejecutar instrucciones diferentes sobre los múltiples datos, mientras que los **SIMD** ejecutan siempre una misma instrucción sobre los datos.

Los computadores **SIMD** requieren menos circuitería que los **MIMD**, ya que sólo necesitan de una unidad de control, mientras que los segundos necesitan una unidad de control por cada elemento de proceso. Además también necesitan menos memoria ya que al disponer de sólo una unidad de control, el programa sólo necesita estar almacenado una vez, mientras que en los ordenadores **MIMD** cada elemento de proceso necesita almacenar su programa.

Clasificación según la organización del espacio de direcciones

Según la organización del espacio de direcciones, los computadores paralelos se clasifican en:

- **Multiprocesadores de memoria compartida**

  Estos multiprocesadores se caracterizan por tener una única memoria principal compartida por todos los procesadores. La comunicación entre diferentes procesadores se realiza a través de la memoria común, a la que pueden acceder todos ellos. Puede verse su esquema en la figura 2.1.

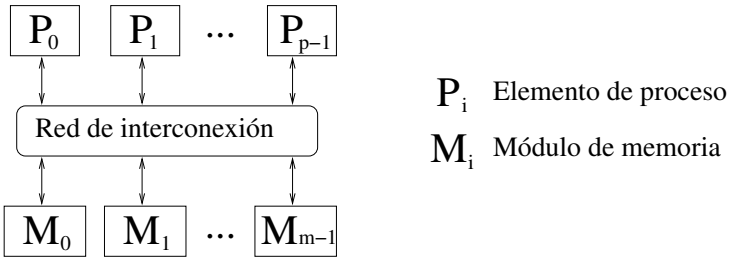

Figura 2.1: Esquema de multiprocesador con memoria compartida

  Es precisamente el acceso a esta memoria común y a la red que la interconecta el punto crítico en esta clase de computadores. Por ello en estas máquinas la memoria suele estructurarse en varios niveles de antememorias, memorias locales y memoria compartida, intentando que los programas trabajen al máximo con los datos más cercanos.

- **Multiprocesadores de memoria distribuida**

  En estos multiprocesadores, la sincronización entre procesadores diferentes debe realizarse mediante operaciones explícitas de comunicación, ya que cada uno de ellos posee su módulo de memoria individual al que no tienen acceso los demás. Puede verse su esquema en la figura 2.2.

  En los multiprocesadores, el cuello de botella del sistema es la red de interconexión entre los diferentes procesadores.

  La programación de estos multiprocesadores se realiza intercalando operaciones de comunicación entre operaciones de cálculo. Con estas operaciones de

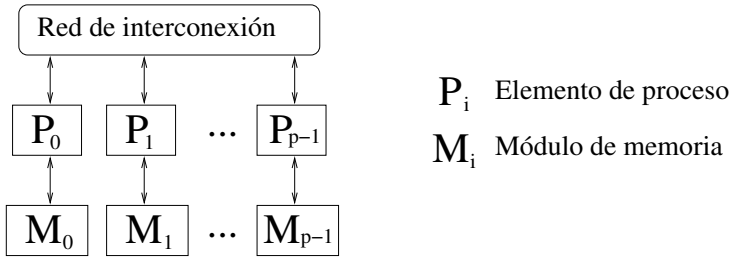

Figura 2.2: Esquema de multiprocesador con memoria distribuida

comunicación se consigue la sincronización necesaria entre los diferentes procesadores para la resolución de un problema en común.

Los multiprocesadores de memoria compartida pueden emular con facilidad a los de memoria distribuida, sin más que implementar el paso de mensajes a través de la memoria común. Es más difícil emular multiprocesadores de memoria compartida con multiprocesadores de memoria distribuida, aunque no por ello deja de hacerse. Por ejemplo, es común encontrar multiprocesadores con memoria *físicamente distribuida aunque lógicamente compartida*. Esto es que mediante circuitería *hardware* emulan un comportamiento de multiprocesadores de memoria compartida aunque tengan la memoria físicamente distribuida. Estos procesadores se conocen como multiprocesadores con memoria *virtualmente* compartida.

Dado que es fácil emular multiprocesadores de memoria distribuida en multiprocesadores de memoria compartida, se puede programar siguiendo el paradigma de memoria distribuida aunque luego se utilice el programa en un multiprocesador de memoria compartida.

Una ventaja que aportan los multiprocesadores de memoria distribuida es que permiten una mayor escalabilidad. Hay multiprocesadores de memoria distribuida con cientos de miles de procesadores. Es lo que se llama multiprocesadores *masivamente paralelos*. En el caso de los multiprocesadores de memoria compartida, algunos alcanzan los miles de procesadores. Pero ya supone una gran complejidad en la red de interconexión con memoria, porque todos deben tener acceso a ella. En los multiprocesadores de memoria distribuida, la red que conecta los procesadores puede ser tan sencilla como un anillo que conecte a cada procesador con el de al lado. Con esta red, la complejidad es mínima, aunque las prestaciones serán menores ya que para comunicar un mensaje entre procesadores muy alejados en el anillo se requerirá atravesar múltiples nodos. Habrá un compromiso entre el tipo de red utilizada para poder situar un gran número de procesadores y las prestaciones obtenidas.

2.1.2. Índices de prestaciones en computación paralela

A continuación se presentan los índices de prestaciones más habituales al trabajar en computación paralela: tiempo de ejecución, megaflops, speed-up y eficiencia.

De ellos, el tiempo de ejecución y los megaflops son índices de prestaciones que también se utilizan en computación secuencial.

Tiempo de ejecución

El *tiempo de ejecución* es el índice de prestaciones más intuitivo. En el caso de un programa secuencial, consiste en el tiempo transcurrido desde que se lanza su ejecución hasta que finaliza.

Es obvio que nos da una medida de las prestaciones del programa. Un programa será mejor que otro (si sólo se presta atención a este índice de prestaciones) en la medida en que tenga un menor tiempo de ejecución.

En el caso de computación en paralelo, este índice se toma como el tiempo invertido desde que el primero de los procesadores ha empezado a ejecutar el programa hasta que el último de los procesadores ha finalizado su ejecución.

Normalmente no suele incluirse en el tiempo de ejecución el tiempo empleado en la distribución de los datos y recogida de los resultados. De esta manera, los tiempos obtenidos son igualmente válidos para el caso más general en que el programa paralelo es una parte intermedia de un programa mayor, de tal forma que los datos ya están repartidos al principio, y al final están donde los necesitará el programa para un tratamiento posterior.

En el mejor de los casos el tiempo de ejecución de un programa en paralelo en p procesadores será p veces inferior al de su ejecución en un sólo procesador, teniendo todos los procesadores igual potencia de cálculo. Generalmente el tiempo nunca se verá reducido en un orden igual a p, el número de procesadores, ya que hay que contar con una sobrecarga extra que aparece al resolver el problema en varios procesadores, debida a sincronizaciones y dependencias entre las tareas que ejecutan.

En los multiprocesadores de memoria distribuida el tiempo dedicado a cálculos y el tiempo de comunicación pueden solaparse cuando las comunicaciones así lo permiten. Es bueno que se solapen comunicaciones y computaciones, porque así el procesador no pierde tiempo esperando enviar datos, tiempo que utiliza para realizar cálculos y acabar antes la computación.

Si ambos tiempos no se solapan, el tiempo de ejecución será la suma del tiempo de cálculo más el tiempo de comunicación. Si ambos tiempos se solapan completamente, el tiempo de ejecución será el mayor de ellos. Y si, lo más normal, los tiempos se encuentran en parte solapados y en parte no, el tiempo de ejecución será inferior a la suma de los tiempos de cálculo y comunicación.

El tiempo de ejecución de un algoritmo secuencial suele representarse mediante T_s, frente al tiempo de ejecución paralelo utilizando p procesadores T_p.

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:

$$S_p = \frac{T_s}{T_p}.$$

Dado que pueden haber distintas versiones secuenciales, se elige el T_s de la versión más rápida. En muchas ocasiones se toma como tiempo secuencial el tiempo del algoritmo paralelo ejecutado en 1 proceso.

El *speed-up* indica la ganancia de velocidad que se ha obtenido con la ejecución en paralelo.

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.

Cuanto mayor sea la ganancia de velocidad de un algoritmo paralelo, mejores serán sus prestaciones, puesto que está logrando reducir más el tiempo necesario para resolver el problema.

Eficiencia

La *eficiencia* es el cociente entre el speed-up y el número de procesadores:

$$E = \frac{S_p}{p}.$$

Mide el grado de aprovechamiento de los procesadores al ejecutar un determinado algoritmo paralelo para la resolución de un problema. Cuanto mayor sea la eficiencia, mejores prestaciones tiene el algoritmo paralelo.

Escalabilidad

La *escalabilidad* es la propiedad que tienen algunos algoritmos paralelos de mantener constante la eficiencia al aumentar el número de procesadores, aunque para ello haya que aumentar el tamaño del problema.

Dicho de una manera informal, un algoritmo paralelo para resolver un problema será escalable si al aumentar el número de procesadores empleados para resolverlo, el hecho de incrementar el tamaño del problema logra que todos 'sigan teniendo trabajo', de forma que se mantenga la eficiencia.

La escalabilidad responde a la pregunta de si se va a poder aprovechar la potencia aportada al aumentar el número de procesadores, aunque para ello haya que aumentar el tamaño del problema.

Si un algoritmo paralelo no es escalable, al aumentar el número de procesadores no se conseguirá incrementar ni mantener la eficiencia, con lo que cada vez se irá aprovechando menos la potencia de los procesadores.

En muchas ocasiones se trabaja con problemas en los que es relativamente fácil modificar su tamaño, por ejemplo por provenir de la discretización de sistemas reales. Aunque hay que tener cuidado, porque dependiendo del problema puede ocurrir que empeore su precisión o se aumente el número de pasos necesarios en su solución. Pero en algunos problemas se pueden obtener mejores resultados a costa de hacer

un mallado más fino o una discretización temporal con un paso de tiempo menor. En esta clase de problemas es especialmente importante la escalabilidad. Si el algoritmo paralelo con el que se resuelven es escalable, esto indica que a costa de aumentar los recursos computacionales se podrá trabajar de forma eficiente con problemas más grandes, que podrían ofrecer resultados de mejor calidad (mayor precisión).

A la hora de estudiar la escalabilidad de un algoritmo paralelo suelen distinguirse dos tipos de escalabilidad:

- La escalabilidad fuerte, *strong scaling*, en la que se estudia cómo afecta el número de procesadores en el tiempo de ejecución para resolver un problema de tamaño fijo.
- La escalabilidad débil, *weak scaling*, en la que se estudia cómo afecta el número de procesadores en el tiempo de ejecución para resolver un problema en el que se mantiene constante el tamaño del subproblema que corresponde a cada procesador.

2.1.3. Factores que afectan a la computación en paralelo

La forma de programar para máquinas paralelas es dependiente del tipo de plataforma a la que vaya destinado el programa. Sin embargo, en cualquiera de ellas es muy importante maximizar la reutilización de los datos más cercanos al procesador, esto es tener en cuenta la jerarquía de memorias, al igual que se hace en computación secuencial.

En multiprocesadores de memoria compartida, el programa recurrirá a la memoria compartida para la sincronización y comunicación entre procesadores. Algunos compiladores permiten en estos multiprocesadores programar de forma normal (secuencial), y ellos se encargan de paralelizar parte del código, normalmente los bucles más internos de los programas, para que se ejecute en diferentes procesadores. Sin embargo, aunque sencillo para el programador, estos métodos no suelen dar resultados óptimos, por la dificultad que supone para el compilador la paralelización del código.

En multiprocesadores de memoria distribuida, es el programador el que debe modificar los programas para que sigan, por ejemplo, el modelo de programación CSP (Procesos Secuenciales Comunicantes). En este modelo, los programas se ven como programas secuenciales que en algún momento requieren de comunicaciones para obtener datos de los que dependen próximas operaciones, o simplemente por motivos de sincronización. Es fácil aportar unas extensiones dedicadas al paso de mensajes, a los lenguajes de programación secuenciales, de forma que pasan a ser lenguajes de programación paralela. Una ventaja que aporta esta metodología es que los programadores no necesitan aprender un nuevo lenguaje para programar los multiprocesadores, sino que basta con aprender unas nuevas instrucciones dedicadas a la comunicación, para poder realizar programas paralelos.

En la computación en paralelo juegan un papel fundamental las redes de interconexión de los procesadores con la memoria común en el caso de multiprocesadores

de memoria compartida o de los procesadores entre sí en el caso de multiprocesadores de memoria distribuida. Características de la red como la latencia (tiempo de establecimiento de la comunicación) o la tasa de transmisión afectan directamente a las prestaciones de los programas paralelos.

Por ejemplo, si se tiene una red muy rápida con muy bajo tiempo de latencia y de comunicación, podrán obtenerse buenas prestaciones con algoritmos de grano fino (que realizan pocas operaciones de computación entre operaciones de comunicación).

También puede ocurrir la situación en la que el tiempo de latencia sea muy alto, aunque una vez establecida la comunicación la tasa de envío del mensaje sea rápida. En estos casos, un programa paralelo de grano fino tendrá unas malas prestaciones. Habrá que utilizar programas de grano grueso (muchas operaciones entre comunicaciones, aunque estas comunicaciones sean de más datos).

Otro de los factores importantes que afectan a las prestaciones de los programas paralelos es la distribución de los datos. La forma en que los datos estén distribuidos entre los diferentes procesadores marcará en gran medida las comunicaciones que harán falta durante el programa y entre qué procesadores.

Por lo general, convendrá repartir los datos uniformemente entre todos los procesadores, porque esto suele suponer una distribución equilibrada de la carga.

En el caso de grandes matrices, además suelen utilizarse distribuciones cíclicas. Una distribución no cíclica es aquélla en la que se parten las matrices en p bloques, y se da un bloque a cada uno de los p procesadores. En la distribución cíclica se parten las matrices en n bloques, siendo $n \gg p$, y se reparten estos bloques entre los procesadores de forma que a cada uno le corresponden bloques no consecutivos. Con esta distribución puede conseguirse mantener uniforme la carga hasta casi el final de la ejecución del programa. En la distribución no cíclica, suele suceder que a medida que avanza el algoritmo las matrices involucradas van disminuyendo su tamaño, lo que supone que el procesador en el que están los bloques de la zona de la matriz que ya no se utilizarán deja de trabajar.

2.2. Librerías software secuenciales

Existen muchas librerías software para realizar operaciones habituales del álgebra lineal. Sin embargo, centrándose en el álgebra lineal sobre matrices densas sobresalen con diferencia, tanto por su eficiencia como por su gran difusión, dos librerías: BLAS [Law78, DCHH88, DCDH89] y LAPACK [ABB+92, ADO92].

Son estas dos librerías las que se han utilizado en este trabajo para las operaciones básicas de álgebra lineal, tratando así de fomentar la portabilidad del código y sus buenas prestaciones, como ya se ha explicado anteriormente. Por esta razón se presentan en los próximos apartados.

Primeramente se muestra la librería BLAS. Se indica la nomenclatura típica usada en esta librería y su estructuración en niveles. Se presenta la forma de la interfaz que ofrece BLAS al programador.

En la sección dedicada a LAPACK, se muestra su alta dependencia con el BLAS,

la nomenclatura que utiliza, su estructura y diferentes tipos de rutinas, y la interfaz que ofrece. La librería LAPACK pretende ofrecer soluciones eficientes para problemas numéricos frecuentes. Su principal característica es la gran robustez de sus algoritmos fundamentada en un análisis numérico riguroso.

En cuanto a librerías software secuenciales orientadas al control de sistemas, se presenta la librería SLICOT.

2.2.1. BLAS

BLAS (*Basic Linear Algebra Subprograms*) [Law78] es una interfaz, que ha sido estandarizada, en la que se describen una serie de subrutinas de álgebra lineal básicas para realizar operaciones ampliamente utilizadas en algoritmos que resuelven problemas matriciales. Las operaciones que contiene son operaciones del estilo de productos entre vectores, productos de matrices, resolución de sistemas triangulares, ... Tiene una implementación de referencia (en FORTRAN) y múltiples implementaciones optimizadas como ATLAS [WP05], OpenBLAS [WZZY13], MKL [WZS+14], ..., así como otras implementaciones específicas de cada fabricante.

El propósito de la creación de este estándar es facilitar el diseño de programas de álgebra lineal portables y con altas prestaciones. Se debe programar maximizando el uso de llamadas a las subrutinas de BLAS, de forma que la portabilidad está garantizada al tener librerías que la implementan disponibles en múltiples plataformas. Además se obtendrán buenas prestaciones en diferentes máquinas sin tener que modificar el código, al estar habitualmente el BLAS optimizado para cada máquina.

Nomenclatura de BLAS

Las rutinas de BLAS tienen una nomenclatura específica que intenta dar la mayor información (en los nombres de las rutinas) con la limitación de 6 caracteres para los identificadores en FORTRAN-77. Las rutinas suelen tener un nombre que sigue el esquema `T Op` , donde `T` representa un carácter indicando el tipo de datos sobre el que se trabaja, y `Op` representa la operación a realizar.

BLAS permite trabajar con cuatro tipos de datos, correspondiéndose cada uno con un posible valor del carácter `T`:

- `S` : indica número real en Simple precisión,
- `D` : indica número real en Doble precisión,
- `C` : indica número Complejo en simple precisión,
- `Z` : indica número complejo en doble precisión.

Este tipo de datos indica el tipo de que se compone el vector o las matrices con los que operar.

Cuando se trabaja con matrices el formato es como el anterior, pero la parte `Op` que indica la operación se descompone en otras. El nombre de la subrutina queda

`TXXYYY` , donde `T` sigue siendo un carácter que indica el tipo de datos con los que operar.

`XX` indica el tipo de matrices con las que operar, pudiendo ser, entre otros, los siguientes caracteres:

- `GE` : indica matriz GEneral,
- `SY` : indica matriz Simétrica,
- `TR` : indica matriz TRiangular.

El BLAS permite además algunos tipos de matrices con almacenamiento empaquetado y banda.

`YYY` indica la operación a realizar.

Niveles de BLAS

Las diferentes subrutinas del BLAS se encuentran estructuradas en tres niveles. Las diferencias entre uno y otro nivel pueden asociarse con el tipo de objeto con el que se trabaja:

- **Nivel 1**: operaciones sobre vectores: vector-vector,
- **Nivel 2**: operaciones sobre matrices y vectores: matriz-vector,
- **Nivel 3**: operaciones sobre matrices: matriz-matriz.

Sin embargo, es más correcto justificar los diferentes niveles de BLAS en función del coste de las operaciones que realizan:

- **Nivel 1**: operaciones de $O(n)$ sobre $O(n)$ datos,
- **Nivel 2**: operaciones de $O(n^2)$ sobre $O(n^2)$ datos,
- **Nivel 3**: operaciones de $O(n^3)$ sobre $O(n^2)$ datos.

Utilizando esta descripción de los distintos niveles de BLAS se observa que es el nivel 3 de BLAS en el que se pueden lograr mejores prestaciones. En las máquinas actuales, la estructura de jerarquía de memorias supone la existencia de un cuello de botella en el bus entre la memoria y la CPU. Tanto en nivel 1 como en nivel 2 de BLAS, el número de operaciones es del mismo orden que el número de datos. Por lo tanto, se realizan igual orden de operaciones que de accesos a memoria, con lo que no se puede superar la velocidad de acceso a la memoria. En el caso de BLAS-3, se realizan $O(n^3)$ operaciones sobre $O(n^2)$ datos, con lo que se puede aprovechar mejor la localidad de los datos en las memorias más cercanas al procesador. La velocidad irá marcada por la velocidad del procesador y no de la memoria, ya que se realizan muchas más operaciones que accesos a memoria. Siempre que sea posible interesa por tanto utilizar rutinas de nivel 3 de BLAS antes que las de niveles inferiores.

A continuación se muestran algunas de las subrutinas de los diferentes niveles de BLAS.

Nivel 1 de BLAS

En este nivel de BLAS se encuentran la mayoría de operaciones aplicables a vectores, normas vectoriales y algunas operaciones de rotaciones planas.

Para especificar un vector en BLAS, se utilizan tres parámetros:

- `N` : Su tamaño, el número de componentes del vector.
- `X` : Dirección en memoria de la primera componente del vector.
- `INCX` : El *stride* del vector, esto es la distancia entre dos elementos adyacentes del vector. `INCX=1` implica que el vector está almacenado consecutivamente en memoria en orden creciente. Distintos valores de `INCX` permiten, por ejemplo, trabajar con un vector fila o columna de una matriz utilizando la misma función, con sólo cambiar el *stride*.

Algunas subrutinas de BLAS-1 son:

`SUBRUTINA xCOPY ( N, X,INCX, Y,INCY )`
$x \rightarrow y$
`SUBRUTINA xSWAP ( N, X,INCX, Y,INCY )`
$x \leftrightarrow y$
`SUBRUTINA xSCAL ( N, ALFA, X,INCX )`
$x \leftarrow \alpha x$
`SUBRUTINA xAXPY ( N, ALFA, X,INCX, Y,INCY )`
$y \leftarrow y + \alpha x$
`FUNCIÓN xDOT ( N, X,INCX, Y,INCY )`
dot$\leftarrow x^T y$
`FUNCIÓN xNRM2 ( N, X,INCX )`
nrm2$\leftarrow \|x\|_2$
`FUNCIÓN xASUM ( N, X,INCX )`
asum$\leftarrow \|x\|_1$
`FUNCIÓN IxAMAX ( N, X,INCX )`
amax$\leftarrow k$ / $|x_k| = \max_{i=1}^{n} |x_i|$
`SUBRUTINA xROTG ( A, B, C, S )`
Calcula una rotación plana.
`SUBRUTINA xROT ( N, X,INCX, Y,INCY, C,S )`
Aplica una rotación plana.

El carácter `x` indica cualquiera de los distintos caracteres de tipo que se han visto anteriormente.

Nivel 2 de BLAS

En este nivel de BLAS están las operaciones que involucran operaciones matriz-vector.

Para especificar una matriz en BLAS, se utilizan los parámetros:

- `M, N` : El tamaño de la matriz, el número de filas y de columnas.
- `A` : Dirección en memoria del primer elemento de la matriz.
- `LDA` : Dimensión principal de la matriz (del inglés *Leading Dimension*). Se utiliza el almacenamiento de FORTRAN para las matrices, es decir por columnas (elementos consecutivos de una misma columna son almacenados consecutivamente en memoria). La dimensión principal es la distancia en memoria entre dos elementos adyacentes de una misma fila de la matriz. Normalmente coincidirá con el número de filas de la matriz: `M`. En el caso de una submatriz, será el número de filas de la matriz a la que pertenezca. Siempre se cumple que `LDA`$\geq$`M`.

Además algunas subrutinas utilizan algunos de los siguientes parámetros para indicar casos especiales de matrices:

- `TRANS` : Indica si se quiere trabajar con la matriz tal cual, o con su transpuesta. Este parámetro puede valer "`No transpose`" o "`Transpose`".
- `UPLO` : Este parámetro es utilizado cuando se usan matrices triangulares o simétricas. Los valores posibles son "`Upper`" o "`Lower`". En el caso de matrices triangulares, especifica si se trata de matrices triangulares superiores o inferiores. En el caso de matrices simétricas, especifica qué parte de la matriz ha de accederse. Al ser simétrica no hace falta tener almacenada toda la matriz. Las subrutinas de BLAS sólo accederán a la parte indicada.
- `DIAG` : Se utiliza con las matrices triangulares e indica si la diagonal está formada por todo unos o no. Puede valer "`Nonunit`" o "`Unit`". En el caso de indicarse que la diagonal principal está formada por unos, BLAS no accederá a los elementos diagonales de la matriz, que no necesitan ser inicializados a la entrada de la subrutina.
- `SIDE` : En algunas operaciones con matrices triangulares o simétricas, se utiliza este parámetro para indicar si se actúa por la izquierda o por la derecha con estas matrices. Los valores posibles son "`Left`" o "`Right`".

En todos estos parámetros no es necesario usar toda la palabra indicada para las diferentes opciones. Sólo se mira el primer carácter de cada opción (El tipo del parámetro es `CHARACTER*1` en FORTRAN). Puede usarse la palabra completa para facilitar la legibilidad del código, pero sólo se comprueba el primer carácter, que además puede estar tanto en mayúsculas como en minúsculas.

Algunas de las subrutinas de BLAS de nivel 2 son:

`SUBRUTINA xGEMV (TRANS, M,N, ALFA,A,LDA, X,INCX, BETA,Y,INCY)`
$y \leftarrow \beta y + \alpha \ \mathrm{op}(A)x$
`SUBRUTINA xSYMV (UPLO, N, ALFA, A,LDA, X,INCX, BETA, Y,INCY)`
$y \leftarrow \beta y + \alpha A x$

`SUBRUTINA xTRMV (UPLO,TRANS,DIAG, N, A,LDA, X,INCX)`
$x \leftarrow \text{op}(A)x$
`SUBRUTINA xTRSV (UPLO,TRANS,DIAG, N, A,LDA, X,INCX)`
$x \leftarrow \alpha\ \text{op}(A)^{-1}x$
`SUBRUTINA xGER (M,N, ALFA, X,INCX, Y,INCY, A,LDA)`
$A \leftarrow A + \alpha xy^T$
`SUBRUTINA xSYR (UPLO, N, ALFA, X,INCX, A,LDA)`
$A \leftarrow A + \alpha xx^T$
`SUBRUTINA xSYR2 (UPLO, N, ALFA, X,INCX, Y,INCY, A,LDA)`
$A \leftarrow A + \alpha(xy^T + yx^T)$

Nivel 3 de BLAS

El nivel 3 de BLAS es donde se realizan las operaciones matriz-matriz. Son operaciones costosas que permiten aprovechar las características de cada máquina para mejorar las prestaciones. Realizan $O(n^3)$ operaciones sobre $O(n^2)$ datos, lo que permite superar la barrera impuesta por el ancho de banda de trasiego de datos entre procesador y memoria.

Algunas de las subrutinas de BLAS-3 son:

`SUBRUTINA xGEMM(TRANSA,TRANSB,M,N,K,ALFA,A,LDA,B,LDB,BETA,C,LDC)`
$C \leftarrow \beta C + \alpha\ \text{op}(A)\text{op}(B)$
`SUBRUTINA xSYMM(SIDE,UPLO, M,N,ALFA,A,LDA, B,LDB, BETA,C,LDC)`
$C \leftarrow \beta C + \alpha AB$ o $C \leftarrow \beta C + \alpha BA$
`SUBRUTINA xTRMM(SIDE,UPLO,TRANS,DIAG, M,N, ALFA,A,LDA, B,LDB)`
$B \leftarrow \alpha\ \text{op}(A)B$ o $B \leftarrow \alpha B\ \text{op}(A)$
`SUBRUTINA xTRSM(SIDE,UPLO,TRANS,DIAG, M,N, ALFA,A,LDA, B,LDB)`
$B \leftarrow \alpha\ \text{op}(A)^{-1}B$ o $B \leftarrow \alpha B\ \text{op}(A)^{-1}$
`SUBRUTINA xSYRK(UPLO,TRANS, N,K, ALFA, A,LDA, BETA, C,LDC)`
$C \leftarrow \beta C + \alpha AA^T$ o $C \leftarrow \beta C + \alpha A^T A$
`SUBRUTINA xSYR2K(UPLO,TRANS, N,K,ALFA,A,LDA,B,LDB,BETA,C,LDC)`
$C \leftarrow \beta C + \alpha(AB^T + BA^T)$ o $C \leftarrow \beta C + \alpha(A^T B + B^T A)$

2.2.2. LAPACK

LAPACK (*Linear Algebra PACKage*) [ABB+92] es una librería, inicialmente desarrollada en Fortran 77, que procura dar solución a los problemas más comunes de álgebra lineal.

LAPACK permite resolver sistemas de ecuaciones lineales, problemas de mínimos cuadrados, calcular los valores propios y valores singulares, ... También permite realizar algunas descomposiciones (como Cholesky, QR) y realizar estimaciones de números de condición.

Al igual que BLAS, permite operaciones con números reales en simple y doble precisión, y con números complejos también en simple y doble precisión (aunque fuera del estándar Fortran 77 para el que no hay números complejos de doble precisión).

LAPACK y BLAS

Las rutinas de LAPACK confían a BLAS el peso de la portabilidad y buenas prestaciones. Esto quiere decir que están hechas, en lo posible, mediante llamadas a las rutinas de BLAS, para así gozar de la portabilidad del BLAS. Precisamente por esta razón, es conveniente tener una versión de BLAS optimizada para la máquina en la que se vaya a utilizar LAPACK, porque un BLAS mal instalado puede provocar malas prestaciones en LAPACK.

Las subrutinas de LAPACK suelen estar basadas en realizar los algoritmos orientados a bloques. Esto es, que en lugar de trabajar elemento a elemento dentro de problemas de matrices, se utilizan algoritmos que trabajen bloque a bloque, para así poder utilizar BLAS-3 u otras rutinas de más bajo nivel del propio LAPACK.

Entre los parámetros modificables durante la instalación de LAPACK se encuentra el tamaño de bloque para cada rutina (especificado por la función `ILAENV`), lo que permite optimizar LAPACK para máquinas con muy diferentes tamaños de memoria y caché.

LAPACK utiliza todos los niveles de BLAS.

El primer nivel de BLAS es utilizado por LAPACK, pero más por motivos de portabilidad que por motivos de eficiencia, ya que son operaciones que suponen una parte insignificante de los cálculos.

El segundo nivel de BLAS también es empleado por LAPACK. En este nivel pueden obtenerse buenas prestaciones en muchos computadores vectoriales, pero en otros el movimiento de datos entre el procesador y memoria principal limita las prestaciones.

Es en el nivel 3 de BLAS donde pueden conseguirse mejores prestaciones, ya que se realizan $O(n^3)$ operaciones sobre $O(n^2)$ datos lo que permite aprovechar mejor la localidad de los datos: trabajar con ellos mientras están en memoria caché, y así superar el límite impuesto por la velocidad de comunicación con memoria.

Nomenclatura de LAPACK

Las rutinas LAPACK suelen tener esta estructura en su nombre: `XYYZZZ`.

El carácter `X` es como el primer carácter de las rutinas BLAS, con su mismo significado y sus mismos posibles valores. Indica el tipo de datos de las matrices involucradas: reales de simple precisión (`S`: tipo `REAL` de Fortran), reales de doble precisión (`D`: tipo `DOUBLE PRECISION`), complejos de simple precisión (`C`: tipo `COMPLEX`), y complejos de doble precisión (`Z`: tipo `COMPLEX*16` o `DOUBLE COMPLEX`).

Los siguientes dos caracteres, `YY`, representan el tipo de matriz con el que se va a trabajar. Algunos de sus valores posibles son:

`GE`	GEneral
`GG`	General, problema Generalizado: dos matrices generales
`GB`	Banda General
`TZ`	TrapeZoidal
`TR`	TRiangular
`TP`	Triangular emPaquetada
`SY`	Simétrica
`BD`	BiDiagonal
`HE`	HErmítica
`HS`	Hessenberg Superior
`OR`	ORtogonal

LAPACK permite muchos y variados tipos de matrices en sus subrutinas. En particular, LAPACK permite además de los tipos ya vistos en BLAS el uso de algunos formatos empaquetados de matrices, como son el almacenamiento de sólo la parte superior (inferior) de una matriz triangular superior (inferior), o el almacenamiento mediante vectores de matrices bidiagonales o tridiagonales, o el almacenamiento de sólo los elementos de la banda en una matriz banda.

En los nombres de las rutinas auxiliares, los caracteres `YY` suelen ser `LA`.

Las últimas tres letras (en algunas ocasiones sólo dos) indican la operación que realiza la subrutina.

En algunos casos en que las funciones de LAPACK realizan operaciones semejantes a las del BLAS, siguen una nomenclatura similar.

Estructura de LAPACK

La librería LAPACK está estructurada en tres grupos de rutinas: rutinas driver, rutinas computacionales, y rutinas auxiliares.

Rutinas driver

Son rutinas que resuelven un problema complejo por completo. Están realizadas mediante llamadas a las otras rutinas, especialmente a las rutinas computacionales.

Algunos de los problemas que resuelven son: sistemas de ecuaciones lineales, cálculo de valores propios de matrices generales, etc.

Ejemplos de rutinas driver pueden ser:

`xyySV`	Sistemas de ecuaciones lineales.
`xGELS`	Problemas de mínimos cuadrados.
`xGESVD`	Valores singulares.

Rutinas computacionales

Realizan diferentes tareas computacionales que utilizan las rutinas driver. Los programadores pueden necesitar estas rutinas para algunos problemas.

Por ejemplo, realizan la factorización LU, la reducción de una matriz simétrica real a forma tridiagonal, ...

Ejemplos de rutinas computacionales pueden ser:

`xyyTRF`	Factorizar sistemas de ecuaciones lineales
`xyyCON`	Estimar número de condición
`xGEQRF`	Descomposición QR
`xGEBRD`	Reducción a forma bidiagonal
`xBDSQR`	Valores singulares de una matriz bidiagonal

Rutinas auxiliares

Las rutinas auxiliares realizan tareas necesarias por las rutinas computacionales como son:

- Versiones no orientadas a bloques de los algoritmos.
- Cálculos de bajo nivel utilizados comúnmente, como el escalado de una matriz, normas matriciales, ... y en general operaciones del estilo de BLAS no presentes en esa librería.
- Extensiones de algunas operaciones BLAS sobre más tipos de datos, como rutinas para aplicar rotaciones planas complejas.

Algunos ejemplos de estas rutinas auxiliares son:

`xGEQR2`	Factorización QR no orientada a bloques
`xLADIV`	División compleja con aritmética real

Utilización de LAPACK

Las rutinas LAPACK se caracterizan por estar muy bien comentadas, de tal modo que son la mejor guía de LAPACK. Ante la duda de lo que hace o cuáles son los parámetros de una subrutina de LAPACK, siempre se puede ir al código fuente, donde cada rutina tiene muy bien especificados sus parámetros de entrada y de salida así como su funcionalidad.

En general, las rutinas de LAPACK tienen siempre un parámetro entero de salida, llamado `INFO`, que informa del tipo de terminación de la rutina. Si a la vuelta de la ejecución de la rutina el valor de `INFO` es cero, la rutina ha finalizado correctamente. Si el valor es negativo, suele indicar que el parámetro que en la llamada a la función ocupa la posición `-INFO` es erróneo. Si el valor es positivo, suele indicar un error durante la ejecución del algoritmo que se esté realizando, por ejemplo puede indicar la singularidad de una matriz cuando se está tratando de invertirla.

Normalmente, cuando `INFO` es negativo la rutina LAPACK efectúa una llamada a `XERBLA`, el manejador de errores de LAPACK. Este manejador por defecto informa del error y termina la ejecución del programa para evitar que se continúe con la ejecución normal, lo que podría conducir a resultados erróneos.

Otros parámetros especiales que llevan muchas de las rutinas LAPACK son `WORK` y `LWORK`. `WORK` es un vector cuyo tamaño mínimo es indicado por la rutina. Se trata de una zona que la rutina utiliza para cálculos auxiliares. Casi todas las rutinas devuelven en la primera componente de este vector lo que hubiera sido su tamaño óptimo. En el entero `LWORK` se le indica a la rutina el tamaño que tiene `WORK`, para

que no se salga de él. Si la rutina comprueba que `LWORK` es inferior al tamaño mínimo necesario para trabajar, devolverá un error.

Por lo demás, las rutinas de LAPACK se usan de forma idéntica a las rutinas de BLAS. Incluso la forma de pasarles vectores y matrices es como en BLAS (salvo cuando se utiliza alguno de los formatos empaquetados propios de LAPACK), esto es utilizando un valor de *stride* para los vectores y una dimensión principal para las matrices, que se almacenan por columnas.

2.2.3. SLICOT

SLICOT (*Subroutine LIbrary in systems and COntrol Theory*) [BMS+99] es una librería de libre distribución que proporciona implementaciones en Fortran 77 de algoritmos numéricos para las operaciones más comunes en teoría de control y sistemas. Basada en rutinas de álgebra lineal de BLAS y LAPACK, SLICOT ofrece métodos para el diseño y análisis de sistemas de control. Las ideas básicas que se persiguen en la librería son la utilidad de los algoritmos, robustez, estabilidad numérica y precisión, buenas prestaciones (refiriéndose tanto a velocidad como a requerimientos de memoria), portabilidad y reusabilidad, estandarización y facilidades para probar las rutinas.

La versión actual de SLICOT contiene unas 400 rutinas que abarcan diversos dominios en sistemas y control. La mayoría de estas rutinas tienen asociada documentación en línea. Unas 200 rutinas tienen además programas ejemplo, datos de entrada y sus resultados. Además se están implementando nuevas rutinas para la librería.

Gracias al uso de Fortran 77, el software obtenido es altamente reutilizable, de forma que SLICOT puede servir como núcleo para plataformas CACSD (*Computer Aided Control Systems Design*), tanto en la actualidad como en el futuro, así como para producción de software de calidad. Además, las rutinas de SLICOT pueden utilizarse desde MATLAB con la ayuda de un compilador de pasarelas (*gateway compiler*), como por ejemplo el *NAG Gateway Generator*.

Todavía se siguen desarrollando nuevas extensiones a la librería, con especial interés en aumentar la librería de *benchmarks* existentes, integrar la librería en un entorno más amigable (MATLAB) y el desarrollo de una versión paralela de SLICOT para multiprocesadores de memoria distribuida.

En la dirección web http://www.slicot.org/ puede consultarse más información sobre la librería SLICOT.

Organización de SLICOT

La librería SLICOT está organizada de igual modo que su manual. Cada capítulo se identifica por una única letra, agrupando las rutinas dedicadas a unas mismas tareas:

- `A`: Rutinas de análisis
- `B`: *Benchmarks* y problemas de prueba

- `C`: Control adaptativo
- `D`: Análisis de datos
- `F`: Filtrado
- `I`: Identificación
- `M`: Rutinas matemáticas
- `N`: Sistemas no lineales
- `S`: Rutinas de síntesis
- `T`: Rutinas de transformación
- `U`: Rutinas de utilidades

La documentación de SLICOT está disponible *on-line*. Contiene la descripción completa de todas las rutinas disponibles.

Rutinas de reducción de modelos en SLICOT

La librería SLICOT tiene múltiples rutinas dedicadas a la reducción de modelos [Var00]. Se encuentran en el capítulo `A`, dedicado a las rutinas de análisis, en el apartado `AB` donde se agrupan las rutinas de análisis en el espacio de estados.

SLICOT implementa la mayoría de los métodos de reducción de modelos más conocidos (algunos de los cuales se explican en capítulos posteriores de este documento): *Square-Root, Balancing-Free Square-Root, Singular Perturbation Approximation, Hankel-Norm Approximation, ...*

En SLICOT existen rutinas tanto para la reducción de modelos estables como para modelos inestables.

Las rutinas destinadas a la reducción de modelos de sistemas inestables utilizan técnicas de embebido, bien la técnica de descomposición aditiva o bien la técnica de factorización racional coprima (RCF), para obtener subsistemas estables del original y luego utilizar las rutinas de reducción de modelos de sistemas estables sobre estos subsistemas.

A continuación se presenta una breve descripción de algunas de las rutinas presentes en SLICOT para la reducción de modelos.

Reducción de modelos de sistemas estables

`AB09AD` calcula modelos balanceados de orden reducido (o mínimo) utilizando el método de truncamiento y balanceado SR o BFSR.

`AB09AX` calcula modelos balanceados de orden reducido (o mínimo) utilizando el método de truncamiento y balanceado SR o BFSR (trabaja sobre el sistema escalado con la matriz de estados en forma real de Schur).

`AB09BD` calcula modelos de orden reducido utilizando el método BFSR SPA.

`AB09BX` calcula modelos de orden reducido utilizando el método BFSR SPA (trabaja sobre el sistema escalado con la matriz de estados en forma real de Schur).

`AB09CD` calcula modelos de orden reducido utilizando el método óptimo HNA basado en balanceado SR.

`AB09CX` calcula modelos de orden reducido utilizando el método óptimo HNA basado en balanceado SR (trabaja sobre el sistema escalado con la matriz de estados en forma real de Schur).

`AB09DD` calcula un modelo de orden reducido utilizando las fórmulas de perturbación singular.

Reducción de modelos de sistemas inestables

`AB09ED` calcula modelos de orden reducido para sistemas inestables utilizando el método óptimo HNA en conjunción con descomposición espectral aditiva estable/inestable.

`AB09FD` calcula modelos de orden reducido para sistemas inestables utilizando el método BFSR B&T en conjunción con métodos de factorización coprima izquierda/derecha.

`AB09GD` calcula modelos de orden reducido para sistemas inestables utilizando el método BFSR SPA en conjunción con métodos de factorización coprima izquierda/derecha.

`AB09MD` calcula modelos de orden reducido para sistemas inestables utilizando el método B&T en conjunción con descomposición espectral aditiva estable/inestable.

`AB09ND` calcula modelos de orden reducido para sistemas inestables utilizando el método SPA en conjunción con descomposición espectral aditiva estable/inestable.

2.3. Librerías software para la computación en paralelo

Al igual que pasa con las librerías secuenciales, existen muchas librerías para la computación en paralelo. De todas ellas, se van a describir las que se han utilizado en este trabajo y están directamente relacionadas con operaciones sobre matrices densas en multiprocesadores de memoria distribuida.

Hay que mencionar que existen implementaciones de LAPACK capaces de ejecutarse en paralelo en multiprocesadores de memoria compartida. Suelen ser versiones en las que se ha compilado el código fuente de LAPACK utilizando extensiones de los compiladores para generar código multi-hilo y así aprovechar varios procesadores. Sin embargo, estas implementaciones paralelas no son siempre muy eficientes y además van dirigidas sólo a multiprocesadores de memoria compartida. Si bien es posible simular memoria compartida en multiprocesadores de memoria distribuida, no es lo habitual y además no se obtienen las mismas prestaciones que en una máquina real con memoria compartida.

Por todo esto, se trabaja con librerías dirigidas a multiprocesadores de memoria distribuida. Debe recordarse que aunque es difícil emular memoria compartida en

una plataforma de memoria distribuida, lo contrario sí es fácil y eficiente. Es decir, ejecutar algoritmos implementados para memoria distribuida en multiprocesadores de memoria compartida es sencillo y no suele implicar una pérdida de prestaciones. Por tanto, al enfocar el trabajo hacia multiprocesadores de memoria distribuida no se está perdiendo generalidad de aplicación.

A la hora de diseñar algoritmos paralelos para multiprocesadores de memoria distribuida, un factor clave es la necesidad de utilizar alguna capa de comunicaciones para realizar las sincronizaciones y trasiego de datos entre los diferentes procesadores. Se hace imprescindible trabajar con alguna librería de comunicaciones.

En este campo, y pensando siempre en el trabajo con matrices, destaca la librería BLACS. Esta librería de comunicaciones tiene como tipo de datos básico las matrices, lo que resulta muy apropiado en operaciones de álgebra lineal.

Es importante mencionar también la librería de comunicaciones MPI, sobre la que se apoya BLACS habitualmente.

Con cualquiera de estas librerías ayudándose del BLAS y el LAPACK pueden realizarse una gran cantidad de programas paralelos obteniendo buenas prestaciones. Sin embargo, será más cómodo utilizar rutinas que realicen cálculos directamente en paralelo, cuando las haya. Conviene utilizar en lo posible librerías de computación en paralelo.

De entre las muchas posibles, centrándose en el caso de operaciones sobre matrices densas, destaca la librería ScaLAPACK. Esta librería trata de ofrecer las mismas operaciones que ofrece LAPACK a nivel de una sola máquina (o un multiprocesador de memoria compartida), pero explotando arquitecturas de memoria distribuida. Para ello se apoya en la librería PBLAS (una versión paralela de BLAS para multiprocesadores de memoria distribuida) y ambas utilizan en lo posible BLAS y LAPACK para la computación y BLACS para las comunicaciones.

En cuanto a librerías de control paralelas, cabe mencionar la librería PSLICOT. Esta librería incluye unos pocos algoritmos paralelos para la reducción de modelos. En ella están las rutinas para este fin de la librería PLICMR y también las rutinas de reducción de modelos que se han desarrollado en esta tesis (aunque a día de hoy no están en su versión online las últimas versiones presentadas aquí). La librería PSLICOT utiliza en lo posible rutinas del resto de librerías comentadas en este documento (tanto secuenciales como paralelas).

Aunque estas son las librerías que se van a describir con algo más de detalle a continuación por ser las que guardan mayor relación con el trabajo realizado, conviene mencionar que también hay muchas otras librerías para realizar operaciones matriciales habituales en paralelo. Es conveniente citar brevemente al menos SCASY, Elemental y libFLAME, aunque algunas de estas librerias no estuvieran disponibles en el momento de empezar la tesis.

SCASY (*Parallel ScaLAPACK-style High Performance Library for Sylvester-Type Matrix Equations*) [GJK09] es una librería de computación de altas prestaciones que ofrece una interfaz similar a la de ScaLAPACK en rutinas para resolver 44 variantes (por cambios de signo y trasposiciones) de 8 ecuaciones matriciales de tipo Sylvester bastante comunes. Utiliza métodos basados en el método de Bartels-Stewart (que se describe en el capítulo 4) para resolver este tipo de ecuaciones.

Elemental [PMvdG+13] es una librería *open-source*, de desarrollo abierto a todo el mundo, para resolver problemas de álgebra lineal y optimización sobre matrices tanto densas como dispersas en ordenadores de memoria distribuida. Está construida, como ScaLAPACK, basándose en las librerías BLAS, LAPACK y MPI. Por ejemplo, ofrece rutinas para resolver ecuaciones de Lyapunov/Ricatti/Sylvester basadas en la función signo matricial.

libFLAME [ICQ+12] es una librería de altas prestaciones para realizar operaciones de álgebra lineal sobre matrices densas, utilizando paralelismo multi-hilo (es para multiprocesadores de memoria compartida). Su característica más novedosa es lo que llama la *metodología FLAME*, que pretende ofrecer una forma sistemática de desarrollo de librerías software para realizar operaciones sobre matrices densas.

2.3.1. BLACS

BLACS (*Basic Linear Algebra Communication Subprograms*) [DW95] es una interfaz de paso de mensajes orientada al álgebra lineal implementada eficientemente en una amplia gama de máquinas de memoria distribuida.

Entre sus objetivos figuran el hacer las aplicaciones en paralelo de álgebra lineal más fáciles de implementar y más portables. Pretende proporcionar la misma facilidad de uso y portabilidad que el BLAS ofrecía para la computación en álgebra lineal, pero ahora para las comunicaciones necesarias en máquinas de memoria distribuida para estas aplicaciones.

BLACS apareció como respuesta a la necesidad de la librería ScaLAPACK de una librería de comunicaciones eficiente y portable, especialmente centrada en las comunicaciones que aparecen en problemas de álgebra lineal.

Mediante la utilización de llamadas a BLAS para computación y llamadas a BLACS para comunicación pueden realizarse aplicaciones de álgebra lineal para ordenadores de memoria distribuida. Dado que en cada plataforma estarán optimizadas ambas librerías, este procedimiento es adecuado si se desean obtener buenas prestaciones y portabilidad en aplicaciones paralelas.

Existen varias versiones de BLACS que se basan en el uso de diferentes librerías de comunicaciones sobre las que se instala. Cabe destacar la versión de BLACS que trabaja utilizando MPI como capa de paso de mensajes.

MPI es un estándar para operaciones de paso de mensajes especialmente indicado para máquinas masivamente paralelas.

En los siguientes apartados, y tras describir brevemente algunas ideas básicas sobre la librería MPI, se comentan algunas características importantes de BLACS como son los datos y ámbitos de operación con que trabaja y las rutinas más esenciales que ofrece como interfaz al programador.

MPI

MPI (*Message Passing Interface*) [SOHL+96] [MPI94] es un estándar que define operaciones de paso de mensajes a utilizar en cualquier plataforma que pueda ser programada con este paradigma. Existen múltiples implementaciones cumpliendo

este estándar. Las de dominio público más utilizadas son MPICH, Open MPI y MVAPICH.

Ofrece muchas rutinas de comunicación punto a punto con diferentes tipos de operaciones. Por ejemplo, dispone de rutinas para envío y recepción bloqueantes y no bloqueantes. Las funciones más básicas (y más utilizadas) de comunicación tienen un comportamiento por defecto de envío asíncrono bloqueante y recepción bloqueante.

Una operación de comunicación (envío o recepción) se dice que es *bloqueante* cuando al retornar el control al programa llamante se puede trabajar con los datos enviados o recibidos sin problemas. Esto en una operación de envío implica que la llamada a la función de envío vuelve cuando o bien ya ha enviado los datos o bien se los ha copiado en un buffer interno, pero en cualquiera de los dos casos se pueden modificar los datos que se querían enviar. En el caso de una operación de recepción, el que sea bloqueante implica que la llamada a la función de recepción no retorna hasta no haber recibido y almacenado el mensaje en el espacio indicado. Si se hace un envío o recepción no bloqueantes, no se debe modificar/acceder a los datos hasta haber comprobado posteriormente (con alguna función auxiliar de la librería de comunicaciones) que se ha terminado de trabajar con el espacio proporcionado a la función.

En la recepción se indica el identificador del mensaje a recibir y el proceso que lo ha emitido, aunque existen unas constantes especiales que pueden usarse para especificar que se desea recibir mensajes con cualquier identificador de mensaje o provenientes de cualquier proceso o ambas cosas.

MPI permite el trabajo con unos tipos de datos básicos en cualquier lenguaje de programación, además de ofrecer la posibilidad de definir nuevos tipos de datos con algunas de sus rutinas.

Todas las operaciones de comunicación se realizan siempre dentro de un determinado ámbito de aplicación. Este ámbito se especifica en MPI mediante el uso de comunicadores. Un comunicador en MPI es un contexto que identifica a un conjunto de procesos. Existe un comunicador predefinido que representa el total de procesos que estaban ejecutándose al comienzo del programa.

Además de las operaciones de comunicación punto a punto, MPI ofrece también muchas y variadas rutinas de comunicación colectiva (involucrando a todos los procesos de un comunicador). Entre ellas están por supuesto operaciones de difusión y de reducción, pero también operaciones más complejas como son repartos y recolecciones.

En el caso de la operación de difusión, se especifica qué proceso realiza la difusión y se comunicará el mensaje a todos los procesos pertenecientes al comunicador especificado.

En el caso de la operación de reducción, se reducirán los datos de los procesos presentes en el comunicador especificado dejando el resultado en uno de ellos, que también se indicará, realizando alguna operación. Algunas de las operaciones que pueden utilizarse en la reducción son: máximo, mínimo, suma, producto, Y lógico, O lógico, ... Además, MPI permite definir nuevas funciones de reducción.

MPI ofrece también múltiples rutinas para la gestión de comunicadores. Casi todas ellas parten de un comunicador que engloba a una serie de procesos y permiten

separarlos en otros contextos o reordenarlos de alguna manera. Es la forma en que MPI trabaja con grupos de procesos: asociándolos a un nuevo comunicador.

Una característica interesante de MPI es la posibilidad que ofrece de definir topologías asociadas a comunicadores. Por ejemplo, se puede crear cualquier topología de malla n-dimensional de una forma sencilla. Y luego pueden obtenerse comunicadores que se correspondan con las filas o columnas de procesos. Esto es útil para orientar las comunicaciones al trabajo en algoritmos matriciales.

El modelo de programación que más se utiliza en MPI es el modelo SPMD (*Simple Program Multiple Data*), en el que todos los procesos ejecutan el mismo programa, que va tomando uno u otro camino según sea un proceso u otro. Sin embargo este modelo no es ninguna restricción, ya que se puede utilizar la información ofrecida por MPI para que cada proceso se diferencie de los demás y tome en consecuencia unas acciones u otras.

Características de BLACS

A continuación se describen algunas características de la librería BLACS. Se va a explicar el tipo de datos con que trabaja BLACS: las matrices. También se explican los distintos ámbitos donde se permiten realizar comunicaciones y lo que representa un contexto en BLACS.

Comunicación de matrices

En BLACS se usan las matrices como tipo de datos básico, ya que es una librería orientada a aplicaciones de álgebra lineal.

BLACS permite operar con los dos tipos de matrices densas más utilizados: matrices generales y matrices trapezoidales. Las matrices triangulares son un caso particular de las matrices trapezoidales.

Al igual que en las librerías BLAS y LAPACK, se utilizan una serie de parámetros para especificar una matriz. En el caso de matrices generales, bastará con su dirección de inicio en memoria, los datos sobre su tamaño `M`,`N` y su dimensión principal `LDA`. Para matrices trapezoidales se necesitará además `UPLO`, para diferenciar si es una matriz trapezoidal superior o inferior, y `DIAG`, para poder distinguir el caso de matriz trapezoidal unidad. En la tabla 2.1 puede verse cómo es el tipo de matriz trapezoidal en función de los distintos valores indicados.

Ámbitos de comunicación

En BLACS siempre se trabaja sobre una malla 2D en la que se encuentran todos los procesos. Esta malla no necesita ser una malla física real, sino que puede ser cualquier topología física, pero será vista como una malla lógica 2D.

Por tanto, en BLACS un proceso se identifica por dos índices que indican sus coordenadas dentro de la malla.

En las operaciones de BLACS que involucran a más de dos procesos, hay que especificar un *ámbito de aplicación* dentro de la malla. En la tabla 2.2 se especifican

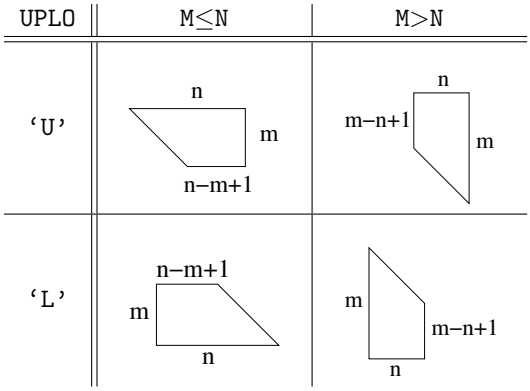

Tabla 2.1: Matrices trapezoidales en BLACS

los diferentes ámbitos de aplicación que BLACS utiliza. El carácter que se indica es el que se utilizará en las rutinas en las que haya que indicar un ámbito de aplicación.

Ámbito	Procesos que participan
Row (`'R'`)	Todos los de la fila
Column (`'C'`)	Todos los de la columna
All (`'A'`)	Todos los procesos de la malla

Tabla 2.2: Ámbitos de operación en BLACS

Contextos

Al igual que en MPI, BLACS trabaja con lo que llama *contextos de comunicación*. El concepto de contexto en BLACS es muy parecido a lo que en MPI era un comunicador: representa un conjunto de procesos asociado a una topología lógica.

Al crear mallas lógicas en BLACS se obtienen contextos asociados a ellas. En todas las operaciones de comunicación se especifica un contexto que identifica el conjunto de procesos sobre los que se debe realizar la comunicación.

Nomenclatura de BLACS

Como en muchas otras librerías, los nombres de las rutinas en BLACS siguen una nomenclatura especial que identifica fácilmente el tipo de datos involucrado y la operación a realizar.

BLACS utiliza una nomenclatura con una estructura fija en el caso de las operaciones de comunicación, siendo más flexible para otras operaciones. En cualquier caso, así como MPI inicia el nombre de sus rutinas con `MPI_`, todas las rutinas de

BLACS van precedidas en su nombre de `BLACS_` en FORTRAN o `Cblacs_` en lenguaje C.

Para las operaciones de comunicación punto a punto y las difusiones, las rutinas reciben el nombre indicado en la tabla 2.3.

`vXXYY2D`

v	Tipo de datos en la matriz
I	Números enteros
S	Números reales de simple precisión
D	Números reales de doble precisión
C	Números complejos de simple precisión
Z	Números complejos de doble precisión

XX	Forma de la matriz
GE	Matriz general
TR	Matriz trapezoidal

YY	Tipo de operación
SD	*Send*: Envío de un proceso a otro
RV	*Receive*: Recepción de un envío de otro proceso
BS	*Broadcast Send*: Envío a varios procesos (difusión)
BR	*Broadcast Receive*: Recepción de una difusión

Tabla 2.3: Nomenclatura de las rutinas de comunicación de BLACS

En la tabla 2.4 se muestra el formato de los nombres de las rutinas de reducción que ofrece BLACS. El carácter `v` conserva el significado explicado en la tabla 2.3.

`vGZZZ2D`

ZZZ	Operación a realizar en la reducción
AMX	Mayor valor absoluto
AMN	Menor valor absoluto
SUM	Suma

Tabla 2.4: Nomenclatura de las rutinas de reducción de BLACS

Interfaz de BLACS

Se enumeran a continuación algunas de las rutinas de BLACS, separadas en diferentes grupos según su utilidad.

Inicialización y finalización

`SUBRUTINA BLACS_PINFO(MYPNUM, NPROCS)`

Con esta rutina se obtiene el número de procesos activos (`NPROCS`) y el identificador del proceso que la llama `MYPNUM`.

`SUBRUTINA BLACS_GRIDINIT(ICONTXT, ORDER, NPROW,NPCOL)`
`SUBRUTINA BLACS_GRIDMAP(ICONTXT, USERMAP,LDUMAP, NPROW,NPCOL)`

Estas rutinas pueden utilizarse para crear una malla 2D de procesos que se utilizará en futuras comunicaciones. Es necesario crear esta malla en todos los programas que utilicen BLACS.

En `ICONTXT` se devolverá el contexto asociado a la nueva malla creada. `ORDER` indica cómo deben repartirse los procesadores en la nueva malla: en orden de filas (`'R'`), o en orden de columnas (`'C'`). `NPROW` y `NPCOL` son el tamaño de la nueva malla: número de filas y de columnas, respectivamente.

Puede verse como ejemplo la malla 2D de la figura 2.3 que se obtendría al llamar a la rutina `BLACS_GRIDINIT` indicando 2 filas, 3 columnas y orden por columnas.

	0	1	2
0	Proc.0	Proc.2	Proc.4
1	Proc.1	Proc.3	Proc.5

Figura 2.3: Malla 2D obtenida con `BLACS_GRIDINIT(ICONTXT,'C',2,3)`

La rutina `BLACS_GRIDMAP` se comporta como `BLACS_GRIDINIT`, pero en lugar de indicar con el parámetro `ORDER` la forma de repartir los procesos en la malla, se dice de forma explícita en la matriz descrita por `USERMAP,LDUMAP` de qué forma realizar esta asignación de procesos. En la posición `(i,j)` de esta matriz se indicará el número del proceso que ocupará esa posición de la malla.

`SUBRUTINA BLACS_GRIDEXIT(ICONTXT)`

Los contextos consumen recursos en BLACS. Por esto es conveniente liberarlos cuando no se van a utilizar más. Esta rutina libera los recursos reservados para el contexto `ICONTXT`.

`SUBRUTINA BLACS_ABORT(INFO, ERRORNUM)`
`SUBRUTINA BLACS_EXIT(CONTINUE)`

Estas rutinas son las utilizadas para finalizar la utilización de facilidades de la librería BLACS.

`BLACS_EXIT` es la forma normal de terminar una aplicación de BLACS. Si el parámetro `CONTINUE` es distinto de cero, se podrán seguir utilizando en el programa llamadas de paso de mensajes con la librería de comunicaciones por debajo de BLACS. Si su valor es cero, no se podrán realizar más comunicaciones (internamente se finalizará el uso de la otra librería de comunicaciones).

`BLACS_ABORT` es una rutina para finalizar todos los procesos BLACS. Se utilizará cuando se produzca alguna condición de error que imposibilite la continuación del programa. Ambos parámetros serán impresos en pantalla en un mensaje de error, por lo que pueden ser utilizados por el programador para diferenciar entre distintas situaciones de error.

Rutinas de comunicación

Es en estas rutinas donde BLACS ofrece una manera cómoda de realizar las comunicaciones, siempre orientada a matrices.

Se muestran a continuación las rutinas de comunicación de BLACS, donde el carácter `v` sigue siendo el indicador de tipo según la tabla 2.3.

```
SUBRUTINA BLACS_vGESD2D(ICONTXT, M,N, A,LDA, RDEST,CDEST)
SUBRUTINA BLACS_vTRSD2D(ICONTXT, UPLO,DIAG, M,N, A,LDA, RDEST,CDEST)
SUBRUTINA BLACS_vGERV2D(ICONTXT, M,N, A,LDA, RSRC,CSRC)
SUBRUTINA BLACS_vTRRV2D(ICONTXT, UPLO,DIAG, M,N, A,LDA, RSRC,CSRC)
```

Estas rutinas se usan para la comunicación punto a punto entre dos procesos. Para el envío de matrices generales y trapezoidales se utilizan `BLACS_vGESD2D` y `BLACS_vTRSD2D`, respectivamente. `BLACS_vGERV2D` y `BLACS_vTRRV2D` son las rutinas análogas para la recepción.

Como se ha descrito anteriormente, `M` y `N` indican el tamaño de la matriz; `A` y `LDA` son la dirección en memoria de la matriz y su dimensión principal; `UPLO` puede valer `'U'` o `'L'` para indicar respectivamente que la matriz es trapezoidal superior o inferior; `DIAG` indica si la diagonal es unitaria (`'U'`), en cuyo caso no necesita comunicarse, o no (`'N'`).

`ICONTXT` representa el contexto en el que se realiza la operación de comunicación. Los parámetros `(RDEST,CDEST)` o `(RSRC,CSRC)` son los índices de fila y columna en la malla de procesos del proceso al que enviar o del que recibir.

```
SUBRUTINA BLACS_vGEBS2D(ICONTXT, SCOPE,TOP, M,N, A,LDA)
SUBRUTINA BLACS_vTRBS2D(ICONTXT, SCOPE,TOP, UPLO,DIAG, M,N, A,LDA)
SUBRUTINA BLACS_vGEBR2D(ICONTXT, SCOPE,TOP, M,N, A,LDA, RSRC,CSRC)
SUBRUTINA BLACS_vTRBR2D(ICONTXT, SCOPE,TOP, UPLO,DIAG, M,N,A,LDA,
                                                        RSRC,CSRC)
```

Estas rutinas se usan para la comunicación entre un conjunto de procesos. Realizan difusiones: comunicación de un proceso a otros.

Con `SCOPE` se indica el ámbito de aplicación de la difusión con las opciones expuestas en la tabla 2.2. Todos los procesos del ámbito en que se realiza la operación deben llamar a la rutina de recepción de la difusión, excepto el que llama a la de emisión.

En las rutinas de recepción debe indicarse en `(RSRC,CSRC)` las coordenadas en la malla del proceso que realiza la difusión.

En estas rutinas aparece además un parámetro, `TOP`, que indica la topología a utilizar en la operacion. Algunos de los valores que permite son `' '` (un espacio) para

optimizar la realización de una única difusión o 'INCREASING RING', 'DECREASING RING', ... para optimizar varias difusiones aunque la primera sea algo más lenta. Si sólo se va a realizar una difusión convendrá utilizar la primera topología (basada en estructura de árbol). Si por el contrario se van a realizar múltiples difusiones en el mismo ámbito, convendrá utilizar una topología basada en anillo que, aunque pueda tardar más en realizar la primera difusión, deja los procesos preparados para realizar sucesivas difusiones.

```
SUBRUTINA BLACS_vGSUM2D(ICONTXT, SCOPE,TOP, M,N, A,LDA, RDEST,CDEST)
SUBRUTINA BLACS_vGAMX2D(ICONTXT,SCOPE,TOP,M,N,A,LDA, RA,CA,RCFLAG,
                                                      RDEST,CDEST)
SUBRUTINA BLACS_vGAMN2D(ICONTXT,SCOPE,TOP,M,N,A,LDA, RA,CA,RCFLAG,
                                                      RDEST,CDEST)
```

Estas rutinas realizan las operaciones de reducción que permite BLACS. En ellas, un grupo de procesos, dentro del ámbito especificado en la llamada, 'reduce' información de forma que se obtiene el resultado de aplicar algún tipo de operación a la información de todos los procesos. En BLACS se permite que sea uno o varios los que obtengan el resultado de la operación de reducción.

El resultado que se obtiene para cada rutina es: la suma (`BLACS_vGSUM2D`), el elemento con mayor valor absoluto (`BLACS_vGAMX2D`) y el elemento con menor valor absoluto (`BLACS_vGAMN2D`).

Las operaciones de reducción se realizan elemento a elemento en las matrices involucradas. Esto quiere decir que se obtendrá una matriz de dimensión igual a las matrices de entrada en la que cada elemento tendrá como valor el resultado de aplicar la operación indicada a los elementos que se encuentran en su misma posición en cada una de las matrices de entrada.

En `RDEST,CDEST` se especifican el índice de fila y columna del proceso que obtendrá el resultado de la reducción. Si alguno de estos valores es `-1` se entenderá que todos los procesos del ámbito de la operación van a recibir el resultado.

En el caso de las operaciones de mayor o menor valor absoluto, pueden obtenerse también los índices de fila (`RA`) y columna (`CA`) que indican para cada elemento de la matriz qué proceso tenía el elemento de mayor o menor valor absoluto. Si `RCFLAG` vale `-1` no se calcularán estos índices ni se accederá a las matrices `RA` y `CA`. Si `RCFLAG` tiene otro valor, éste será tomado como la dimensión principal de `RA` y `CA`, y estas matrices se rellenarán en los procesos que obtengan el resultado.

```
SUBRUTINA BLACS_BARRIER(ICONTXT, SCOPE)
```

La operación realizada por esta rutina no es propiamente de comunicación. Esta rutina sincroniza todos los procesos del ámbito `SCOPE`. Cada proceso que la llame quedará bloqueado en ella hasta que todos los procesos de ese ámbito la hayan llamado.

Obtención de información

Se presentan a continuación algunas rutinas para la obtención de información de BLACS.

`SUBRUTINA BLACS_GET(ICONTXT, WHAT, VAL)`

Esta rutina se utiliza para obtener el valor de algunos parámetros de BLACS. Mediante el entero `WHAT` se indica el valor que se quiere consultar, el cual es devuelto en `VAL`. Por ejemplo, si `WHAT` vale `0` se devolverá el contexto por defecto del sistema.

```
SUBRUTINA BLACS_GRIDINFO(ICONTXT, NPROW,NPCOL, MYPROW,MYPCOL)
FUNCIÓN BLACS_PNUM(ICONTXT, PROW,PCOL)
SUBRUTINA BLACS_PCOORD(ICONTXT, PNUM, PROW,PCOL)
```

Estas rutinas devuelven información sobre la malla asociada a un contexto.

`BLACS_GRIDINFO` devuelve en `NPROW,NPCOL` su tamaño (número de filas y de columnas) y en `MYPROW,MYPCOL` el identificador del proceso que la llama.

Las otras dos funciones se usan para pasar de identificador de proceso a coordenadas en la malla (`BLACS_PCOORD`) o de coordenadas a identificador de proceso (`BLACS_PNUM`).

Además de las rutinas que se han expuesto, BLACS dispone de unas rutinas específicas para tareas de inicialización y mantenimiento de la librería de paso de mensajes sobre la que esté funcionando. Es lo que se conoce como *rutinas no oficiales* de BLACS.

2.3.2. ScaLAPACK

ScaLAPACK (*Scalable Linear Algebra PACKage*) [BCC+97] es una librería estándar de álgebra lineal para multiprocesadores de memoria distribuida y en general para cualquier plataforma que permita la programación mediante paso de mensajes. Puede utilizarse en cualquier máquina para la que se disponga de las librerías de comunicación MPI o BLACS.

ScaLAPACK es una continuación del proyecto LAPACK [ABB+92], donde se pretendía ofrecer rutinas para resolver problemas comunes de álgebra lineal de una forma eficiente, portable y fiable. LAPACK se utiliza en estaciones de trabajo, supercomputadores vectoriales y multiprocesadores de memoria compartida. ScaLAPACK ofrece la funcionalidad de LAPACK (no toda) pero en multiprocesadores de memoria distribuida y en computadores heterogéneos en los que se disponga de MPI o BLACS. El tipo de problemas que se pueden resolver son sistemas de ecuaciones lineales, problemas de mínimos cuadrados y problemas de valores propios.

ScaLAPACK ofrece una interfaz para sus rutinas muy semejante a la ofrecida por LAPACK, para que el usuario de LAPACK se adapte con facilidad a su uso.

La portabilidad y buenas prestaciones se ven favorecidas por el uso de librerías como son BLAS, LAPACK y BLACS.

A continuación se muestran los componentes que forman ScaLAPACK y su estructura, que es semejante a la de LAPACK. También se describen otros aspectos como la distribución de matrices que utiliza (distribución por bloques cíclica 2D).

Componentes de ScaLAPACK

Tanto para lograr buenas prestaciones como por motivos de portabilidad, ScaLAPACK realiza todas sus operaciones mediante llamadas a rutinas de librerías: BLAS, LAPACK, BLACS y PBLAS.

PBLAS es una librería para la computación matricial en paralelo. Es el equivalente a BLAS pero para matrices distribuidas. En ScaLAPACK se han utilizado las rutinas de la librería PBLAS como núcleos básicos de igual modo que en LAPACK se usaba la librería BLAS. Así se consigue que el código de ScaLAPACK sea similar al de LAPACK.

En la figura 2.4 se muestra la relación existente entre los diversos componentes de ScaLAPACK.

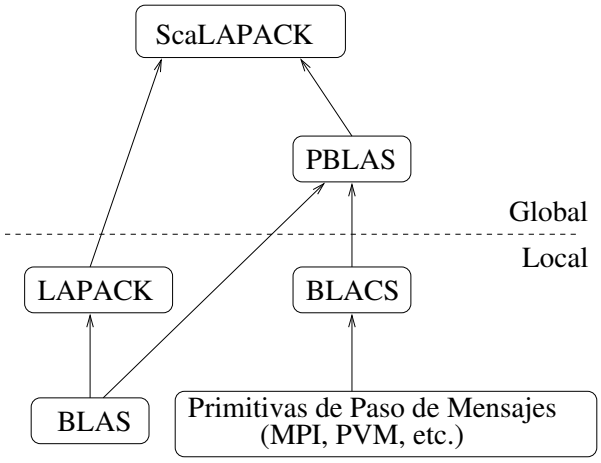

Figura 2.4: Jerarquía de la librería ScaLAPACK

Estos componentes pueden distinguirse en dos grupos, que en la figura se han denominado *global* y *local*, en función de si realizan operaciones que deben ser llamadas por todos o por algunos de los procesos. Por ejemplo, cuando se quiera realizar un producto de dos matrices en paralelo, todos los procesos llamarán a la rutina de PBLAS que realiza esta operación, aunque dentro de esta rutina sólo algunos procesos llamarán a rutinas de comunicación (BLACS) y/o operarán con las matrices (BLAS).

Todas estas librerías constituyen un paquete importante para la computación de altas prestaciones. Fundamentalmente se apoyan en las librerías BLAS para computación y BLACS para comunicación. Debido a la alta disponibilidad de ambas li-

brerías y a la optimización de BLAS en muchas plataformas, mediante su uso se pueden obtener buenas prestaciones y alta portabilidad.

PBLAS: Parallel Basic Linear Algebra Subprograms

PBLAS (*Parallel Basic Linear Algebra Subprograms*) [CDOP96] es una librería con rutinas para realizar operaciones básicas de cálculo matricial en paralelo. Es una versión del BLAS para el trabajo con matrices distribuidas.

PBLAS trata de ofrecer una interfaz parecida a la interfaz de BLAS, lo que permite que las rutinas de ScaLAPACK se asemejen, en lo posible, a sus versiones secuenciales de LAPACK. Cuando en LAPACK se llama a rutinas de BLAS, en ScaLAPACK se llamará a rutinas de PBLAS.

PBLAS trata de suministrar un estándar de operaciones matriciales para memoria distribuida al igual que BLAS lo ha hecho para memoria compartida.

La librería PBLAS realiza las operaciones mediante llamadas a las rutinas de las librerías BLAS y BLACS. La comunicación queda muchas veces oculta dentro de la implementación de esta librería. Este hecho también hace que algunas de las rutinas de ScaLAPACK se asemejen bastante a las de LAPACK, al no aparecer explícitamente llamadas a rutinas de comunicación en el código fuente.

PBLAS utiliza el mismo tipo de distribución que ScaLAPACK: distribución por bloques cíclica 2D (más adelante se muestra cómo es esta distribución).

La interfaz de las rutinas de PBLAS es como la de BLAS, pero con algunos parámetros adicionales para identificar un vector o una matriz distribuidos.

Para especificar una matriz en PBLAS, y también en ScaLAPACK, se utilizan los siguientes parámetros:

- `M, N` : El tamaño de la submatriz de la matriz que se pasa con la que se va a trabajar. El número de filas y de columnas (puede ser toda la matriz, pero el hecho de permitir especificar una submatriz es más general).
- `A` : Dirección de inicio en memoria de la parte local de la matriz distribuida a la que pertenece la submatriz.
- `IA, JA` : Los índices de fila y columna de la submatriz con la que trabajar dentro de la matriz distribuida.
- `DESCA` : Es un descriptor que indica cómo está distribuida la matriz a la que pertenece la submatriz con la que se va a trabajar. Algunos de los datos que contiene son la dimensión principal de la parte local de la matriz (el `LDA` de BLAS) y el tamaño de la matriz distribuida (más adelante al hablar de la distribución usada en ScaLAPACK, se explica el contenido de este descriptor).

En el caso de vectores, los parámetros para identificarlos son los mismos que los de las matrices, a los que se añade `INCX`, que indica la distancia en memoria entre dos elementos consecutivos del vector. También se incluyen los índices `IX, JX` que indican dónde comienza el vector dentro de la matriz. Tanto en PBLAS como en ScaLAPACK los vectores sólo pueden ser (sub)vectores fila o columna de una matriz

distribuida. La forma de diferenciar si un (sub)vector está en una fila o columna de la matriz es precisamente mediante el parámetro `INCX`. Si su valor coincide con la dimensión principal de la matriz distribuida, se tratará de un vector fila, y si su valor es 1, será un vector columna (elementos consecutivos en memoria).

El resto de parámetros de las rutinas de BLAS se mantienen con su mismo significado y valores posibles.

Las rutinas de PBLAS tienen el nombre de las rutinas de BLAS precedido de una `P` y sus mismos parámetros, tan sólo cambiando aquéllos que identifican a matrices o vectores de la forma que se ha explicado.

A continuación se muestran las cabeceras de algunas de las rutinas de PBLAS. La operación que realizan es la misma que sus análogas de BLAS, sólo que en este caso trabajan con (sub)matrices o (sub)vectores distribuidos.

Nivel 1 de PBLAS:

```
SUBRUTINA PxCOPY(N, X,IX,JX,DESCX,INCX, Y,IY,JY,DESCY,INCY)
SUBRUTINA PxSWAP(N, X,IX,JX,DESCX,INCX, Y,IY,JY,DESCY,INCY)
SUBRUTINA PxSCAL(N, ALFA, X,IX,JX,DESCX,INCX)
SUBRUTINA PxAXPY(N,ALFA,X,IX,JX,DESCX,INCX,Y,IY,JY,DESCY,INCY)
SUBRUTINA PxDOT(N,DOT, X,IX,JX,DESCX,INCX, Y,IY,JY,DESCY,INCY)
SUBRUTINA PxNRM2(N, NORM2, X,IX,JX,DESCX,INCX)
SUBRUTINA PxASUM(N, ASUM, X,IX,JX,DESCX,INCX)
SUBRUTINA IxAMAX(N, AMAX,INDX, X,IX,JX,DESCX,INCX)
```

Nivel 2 de PBLAS:

```
SUBRUTINA PxGEMV(TRANS, M,N, ALFA,A,IA,JA,DESCA, X,IX,JX,DESCX,INCX,
                           BETA,Y,IY,JY,DESCY,INCY)
SUBRUTINA PxSYMV(UPLO, N, ALFA,A,IA,JA,DESCA, X,IX,JX,DESCX,INCX,
                         BETA,Y,IY,JY,DESCY,INCY)
SUBRUTINA PxTRMV(UPLO,TRANS,DIAG, N, A,IA,JA,DESCA, X,IX,JX,DESCX,INCX)
SUBRUTINA PxTRSV(UPLO,TRANS,DIAG, N,A,IA,JA,DESCA, X,IX,JX,DESCX,INCX)
SUBRUTINA PxGER(M,N, ALFA, X,IX,JX,DESCX,INCX, Y,IY,JY,DESCY,INCY,
                          A,IA,JA,DESCA)
SUBRUTINA PxSYR(UPLO, N, ALFA, X,IX,JX,DESCX,INCX, A,IA,JA,DESCA)
SUBRUTINA PxSYR2(UPLO, N, ALFA, X,IX,JX,DESCX,INCX, Y,IY,JY,DESCY,INCY,
                               A,IA,JA,DESCA)
```

Nivel 3 de PBLAS:

```
SUBRUTINA PxGEMM(TRANSA,TRANSB, M,N,K,ALFA,A,IA,JA,DESCA, B,IB,JB,DESCB,
                                     BETA,C,IC,JC,DESCC)
SUBRUTINA PxSYMM(SIDE,UPLO, M,N,ALFA,A,IA,JA,DESCA, B,IB,JB,DESCB,
                               BETA,C,IC,JC,DESCC)
SUBRUTINA PxTRMM(SIDE,UPLO,TRANS,DIAG, M,N,ALFA,A,IA,JA,DESCA,
                                           B,IB,JB,DESCB)
SUBRUTINA PxTRSM(SIDE,UPLO,TRANS,DIAG, M,N,ALFA,A,IA,JA,DESCA,
                                           B,IB,JB,DESCB)
```

```
SUBRUTINA PxSYRK(UPLO,TRANS, N,K,ALFA,A,IA,JA,DESCA, BETA,C,IC,JC,DESCC)
SUBRUTINA PxSYR2K(UPLO,TRANS, N,K,ALFA,A,IA,JA,DESCA, B,IB,JB,DESCB,
                                  BETA,C,IC,JC,DESCC)
SUBRUTINA PxTRAN(M,N, ALFA,A,IA,JA,DESCA, BETA,C,IC,JC,DESCC)
```

Al igual que en BLAS, se permite trabajar con diferentes tipos de datos: números reales de simple y doble precisión, y números complejos de simple y doble precisión. En los nombres de las rutinas, el carácter `x` tiene distintos valores según el tipo de los datos: `S,D,C` y `Z`, respectivamente.

En PBLAS se ha añadido una nueva rutina para la transposición de matrices, `PxTRAN`. Esta operación, que es obvia en el caso secuencial, requiere comunicaciones en el caso de matrices distribuidas.

En la mayoría de implementaciones de PBLAS existen una serie de restricciones de alineamiento entre las (sub)matrices y/o (sub)vectores con los que se pueden utilizar las rutinas. Esto quiere decir que sus distribuciones deben cumplir unas normas para que se pueda realizar la operación. Si no, se producirá un error. Las restricciones de alineamiento se refieren siempre a la posición de inicio y los tamaños de bloque de las submatrices/subvectores con los que operar. Pueden exigirse, por ejemplo, que los dos (sub)vectores de entrada a una rutina tengan su primer elemento en el mismo procesador o misma fila de procesadores y que estén distribuidos con igual patrón.

Estas restricciones hacen que PBLAS y por extensión ScaLAPACK no sean tan flexibles como en un principio era deseable. Para casi todas las operaciones, las matrices involucradas deberán respetar algunas condiciones de alineamiento. Si bien, se espera que estas restricciones vayan desapareciendo en próximas versiones.

Estructura de ScaLAPACK

ScaLAPACK es una librería análoga a LAPACK pero para ejecución en ordenadores paralelos. Trata de ofrecer la misma funcionalidad y sigue su misma estructura (rutinas driver, computacionales y auxiliares).

No todas las facilidades de LAPACK están cubiertas en ScaLAPACK. Aquellas operaciones que sí están, reciben en ScaLAPACK el nombre que tenían en LAPACK precedido por el carácter `P`. Al igual que en PBLAS, el único cambio en los parámetros se produce en los de identificación de matrices o vectores, que ahora tienen la forma explicada en el punto anterior.

En el caso del espacio de trabajo, su funcionalidad es la misma que en LAPACK, pero ahora en cada rutina se indica la cantidad de memoria necesaria localmente en cada proceso. Además, suele venir expresada en función del tamaño de la parte local de las matrices con las que se trabaja.

Al igual que en LAPACK, sigue utilizándose un parámetro `INFO` donde se indicará si ha habido error. De ser así, se llama a la rutina `PXERBLA`, el equivalente a la rutina `XERBLA` de LAPACK. Pero en el caso de ScaLAPACK esta rutina no siempre aborta el programa. Si se detecta un error en rutinas de alto nivel de ScaLAPACK (rutinas driver o computacionales), no se detiene la ejecución, sino que se devuelve

el control al programa llamante, que generalmente podrá recuperarse del error. Sin embargo, si el error se produce en rutinas de bajo nivel (rutinas auxiliares o rutinas de PBLAS o BLACS), el error se considera irrecuperable y se aborta la ejecución.

Distribución de datos en ScaLAPACK

La distribución de datos que utiliza ScaLAPACK es la distribución por bloques cíclica en 2 dimensiones [Wal95].

En LAPACK se procuraba hacer un uso eficiente de la jerarquía de memoria. Esto suponía maximizar la reutilización de datos, lo que, por ejemplo en un procesador con memoria caché, supone minimizar la transferencia de datos entre memoria principal y memoria caché.

En ScaLAPACK se lleva a cabo una aproximación análoga para máquinas de memoria distribuida. Se utilizan algoritmos con una partición de las matrices en bloques, para reducir la frecuencia con que los datos deberán ser transferidos entre distintos procesadores. Estos algoritmos pueden implementarse de forma natural con una distribución por bloques cíclica en 2 dimensiones.

En ScaLAPACK se utiliza esta distribución, excepto en el caso de las rutinas para resolver sistemas lineales con estructura banda y sistemas tridiagonales. Estas rutinas trabajan con una distribución por bloques cíclica en 1 dimensión, que puede verse como un caso particular de la distribución en 2 dimensiones.

La distribución por bloques cíclica 2D de una matriz es la que se obtiene tras particionar la matriz en bloques de tamaño fijo y distribuir estos bloques a filas y columnas de procesadores de manera cíclica. Todos los bloques de una misma fila(columna) de bloques de la matriz estarán en una misma fila(columna) de la malla de procesadores tras la distribución.

Puede verse un ejemplo de este tipo de distribución en la figura 2.5. Se ha distribuido una matriz de tamaño 5×5 en una malla de 2×3 procesadores utilizando bloques de tamaño 2×2. Se ha empezado la distribución en el procesador $(0, 0)$.

Matriz a distribuir

$$\left(\begin{array}{cc|ccc} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ \hline a_{20} & a_{21} & a_{22} & a_{23} & a_{24} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} \\ \hline a_{40} & a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right)$$

Distribución en la malla

	0		1		2
0	a_{00}	a_{01}	a_{02}	a_{03}	a_{04}
	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
	a_{40}	a_{41}	a_{42}	a_{43}	a_{44}
1	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}
	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}

Figura 2.5: Ejemplo de distribución cíclica por bloques en una malla de procesadores

El elemento (i, j) de una matriz distribuida de forma cíclica en bloques de tamaño $mb \times nb$ en una malla de $P \times Q$ procesos estará en la posición (i_p, j_q) de la submatriz que corresponde al proceso con índices (p, q) en la malla, según las

siguientes fórmulas (utilizando índices que empiezan en cero tanto para la matriz como para los procesos):

$$\begin{aligned}(p,q) &= \left(\left\lfloor\frac{i}{mb}\right\rfloor \bmod P, \left\lfloor\frac{j}{nb}\right\rfloor \bmod Q\right),\\ (i_p, j_q) &= \left(mb\left\lfloor\frac{i}{P\cdot mb}\right\rfloor + i \bmod mb, nb\left\lfloor\frac{j}{Q\cdot nb}\right\rfloor + j \bmod nb\right).\end{aligned}$$

Para explicar la distribución mostrada en la figura 2.5, se han utilizado unos parámetros necesarios para identificar una distribución cíclica por bloques 2D. Estos parámetros característicos son:

- `M, N` : El tamaño de la matriz a distribuir. El número de filas y de columnas.
- `MB, NB` : El tamaño de los bloques en que se particiona la matriz para su distribución. El número de filas y de columnas de los bloques.
- `RSRC, CSRC` : Los índices en la malla del procesador en que se empieza la distribución. Índice de filas y de columnas.

En ScaLAPACK se utilizan estos datos para identificar la distribución de cada matriz con la que se va a trabajar. Forman lo que se conoce como el *descriptor* de la matriz. En las rutinas de ScaLAPACK y PBLAS en que se pasa como parámetro una matriz, siempre se pasa acompañada de su descriptor asociado. El descriptor es un vector en el que están almacenados los datos necesarios para identificar el tipo de distribución que se ha utilizado con una matriz determinada. Además de los parámetros que se acaban de explicar, también aparecen los siguientes:

- `LLD` : Es la dimensión principal de la submatriz local en cada procesador. Normalmente coincidirá con el número de filas de la matriz que le corresponden a ese procesador.
- `CTXT` : Es el contexto de la malla de procesadores sobre la que se ha distribuido la matriz. Es necesario porque ScaLAPACK permite trabajar con múltiples contextos a la vez.
- `DTYPE` : Un identificador del tipo de descriptor. El descriptor que se ha explicado aquí se corresponde con el tipo 1 de descriptores de ScaLAPACK. Es el descriptor utilizado en una distribución por bloques cíclica 2D. Como ya se ha mencionado, ScaLAPACK utiliza distribuciones especiales (cíclicas en 1 sola dimensión) para algunas rutinas que trabajan con matrices banda o tridiagonales. Para estas distribuciones se utilizan otros tipos de descriptor con otros campos característicos.

Para rellenar los datos de un descriptor puede utilizarse la rutina de ScaLAPACK:

```
SUBRUTINA DESCINIT(DESC,M,N,MB,NB,RSRC,CSRC,CTXT,LLD,INFO)
```

Cuando en LAPACK se pasa un parámetro erróneo a una rutina, se devuelve en `INFO` el índice de este parámetro. En ScaLAPACK también se utiliza `INFO` de esta

forma. Pero si el parámetro erróneo es un descriptor, en `INFO` se devuelve el valor $-(i*100+j)$, siendo i el orden que ocupa el descriptor en el resto de parámetros de la rutina, y j el índice de la componente errónea dentro del descriptor. De esta forma puede saberse con precisión en qué parámetro se ha cometido un error, incluso si se trata de una componente de un descriptor y hay varios descriptores en los parámetros de la rutina.

Utilización de ScaLAPACK

La librería ScaLAPACK funciona sobre BLACS. Esto exige que antes de llamar a ninguna de las rutinas de ScaLAPACK hay que inicializar la librería BLACS. Y por lo mismo habrá que finalizar BLACS al final del programa. Además ScaLAPACK necesita de una malla lógica 2D inicializada en BLACS.

En la sección de BLACS (la 2.3.1) ya se ha visto cómo se crea una malla utilizando esta librería (`BLACS_GRIDINIT`, `BLACS_GRIDMAP`) y cómo debe finalizarse (`BLACS_GRIDEXIT` y `BLACS_EXIT`).

Las rutinas de ScaLAPACK trabajan con las matrices ya distribuidas. Por tanto, previamente a la llamada a estas rutinas se tendrán que haber distribuido las matrices siguiendo la distribución especificada en los descriptores. ScaLAPACK parte de las matrices ya distribuidas, porque así pueden llamarse consecutivamente múltiples rutinas sin necesidad de realizar comunicaciones entre ellas. Normalmente las matrices se distribuirán al principio de la aplicación, se trabajará con ellas utilizando varias rutinas de ScaLAPACK y al final se recogerá el resultado en un procesador.

Para la distribución de las matrices, ScaLAPACK se acompaña de unas rutinas auxiliares que, bajo la denominación de `REDIST`, facilitan esta tarea. Por ejemplo, puede utilizarse la rutina `PDGEMR2D`, que copia una (sub)matriz `GE`neral de números reales de `D`oble precisión en otra, cualesquiera que sean sus distribuciones. En particular, es muy útil utilizar para una de las matrices un descriptor con una distribución con tamaño de bloque el tamaño de la matriz y empezando a distribuir en el procesador que la tiene. Este descriptor especifica que toda la matriz está en un único procesador. Si el descriptor de la matriz destino especifica la distribución deseada, llamando a esta rutina se distribuirá la matriz. De igual modo si se invierte el orden de las matrices en la llamada, se conseguirá copiar una matriz distribuida en una matriz local.

Para obtener información sobre una rutina concreta de ScaLAPACK, siempre puede consultarse el código fuente. Con el mismo estilo que en LAPACK, en el código fuente de cada rutina se describen sus parámetros, las restricciones de alineamiento, el tamaño necesario para el espacio de trabajo y la tarea que realiza.

ScaLAPACK es una librería muy útil en álgebra lineal ya que soluciona muchos problemas comunes. Ante un problema determinado, si existe alguna rutina o grupo de rutinas de ScaLAPACK que lo resuelvan directamente, su utilización es bastante sencilla, obteniéndose buenas prestaciones y portabilidad con un esfuerzo de desarrollo muy pequeño.

Sin embargo, si no existe rutina para resolver el problema en ScaLAPACK, habrá que diseñar una versión en paralelo partiendo, posiblemente, de una versión

secuencial. Gracias a ScaLAPACK y PBLAS, podrá obtenerse de forma relativamente fácil una versión paralela sustituyendo en el programa secuencial las llamadas a LAPACK por llamadas a ScaLAPACK y llamadas a BLAS por llamadas a PBLAS. Esta forma de trabajar generalmente dará buenos resultados, aunque en algunas ocasiones puede no ser suficiente, necesitándose en estos casos un estudio más profundo del problema para implementar una versión paralela más optimizada. Aún en este caso, sigue siendo recomendable la utilización de librerías de un nivel inferior: BLAS, LAPACK y BLACS. Con ello se aprovecharán su portabilidad y buenas prestaciones. Si además se ofrece una interfaz semejante a la de ScaLAPACK, podrá utilizarse la nueva rutina en futuros programas en que se necesite resolver este problema más otros que ya resuelve ScaLAPACK.

2.3.3. PSLICOT

PSLICOT (*Parallel SLICOT*) [BGH+98] es una librería en desarrollo que pretende ofrecer funcionalidades similares a las presentes en la librería secuencial SLICOT en plataformas paralelas. Para poder enfrentarse a esta clase de problemas resulta necesario realizar una aproximación desde el punto de vista de la computación de altas prestaciones. Este enfoque permitirá reducir los costes necesarios, en cuanto a tiempo de ejecución y requerimientos de memoria y potencia computacional, para afrontar problemas de gran dimensión.

De todos los problemas que trata de resolver la librería SLICOT, es en los problemas de reducción de modelos donde es más previsible la necesidad de trabajar con problemas de gran dimensión. Esto es así porque en un problema de reducción de modelos obligatoriamente se tiene que partir del modelo que se quiere reducir y que muy posiblemente sea de un gran tamaño. Es por esta razón por la que se ha comenzado a desarrollar PSLICOT empezando por paralelizar los algoritmos de reducción de modelos presentes en SLICOT.

Conservando la estructura básica de los algoritmos de reducción de modelos presentes en SLICOT, pueden realizarse dos enfoques a la hora de paralelizarlos, en función de los métodos básicos utilizados para resolver las ecuaciones de Lyapunov necesarias.

En una primera aproximación, se pueden conservar los algoritmos básicos ya utilizados en el caso secuencial para esta labor, paralelizándolos. Este es el enfoque cuyo trabajo se presenta en esta tesis y es descrito en más detalle en los próximos capítulos. Se han desarrollado e incluido en la librería PSLICOT todas las rutinas que se muestran en la figura 5.2 y que se van a ir explicando en los próximos capítulos.

Otra forma de proceder consiste en modificar los métodos básicos utilizados para resolver las ecuaciones de Lyapunov que aparecen. Siguiendo este camino se ha desarrollado una mini-librería llamada PLICMR [BQQ00b, BQQ99] en la que se han paralelizado algunas de las rutinas de reducción de modelos de SLICOT, utilizando la función signo matricial como núcleo computacional básico. He aquí una breve presentación de las rutinas que ofrece esta librería PLICMR, que también está incluida en PSLICOT.

Rutinas de reducción de modelos de PSLICOT basadas en métodos iterativos

`PAB09AX` calcula modelos balanceados de orden reducido (o mínimo) utilizando el método de truncamiento y balanceado SR o BFSR.

`PAB09BX` calcula modelos de orden reducido utilizando el método óptimo BFSR o el SR SPA.

`PAB09CX` calcula modelos de orden reducido utilizando el método óptimo HNA basado en balanceado SR.

`PAB09DD` aplica las fórmulas de aproximación de perturbación singular a un sistema general.

`PSB03ODC` resuelve ecuaciones de Lyapunov estables usando el método de la función signo matricial.

`PSB03ODD` resuelve ecuaciones de Stein estables usando la iteración de Smith o el método de la función signo matricial.

`PSB04MD` resuelve la ecuación de Sylvester estable usando el método de la función signo matricial.

`PMB05RD` calcula la función signo de una matriz utilizando la iteración de Newton.

`PMB03TD` calcula la descomposición en valores singulares del producto de dos matrices.

`PMB03OX` realiza una estimación del rango de una matriz triangular usando un estimador de condición incremental.

Capítulo 3

Cálculo de valores propios

En este capítulo se presenta el trabajo desarrollado alrededor del problema del cálculo de valores propios, tanto en su versión estándar como en su versión generalizada.

Primeramente se recuerda en qué consiste este problema y se citan algunos métodos habituales para su resolución. Después se presentan las rutinas de altas prestaciones desarrolladas para resolverlo. En el apéndice pueden consultarse algunos nuevos códigos necesarios para el problema real práctico que se ha utilizado como caso real donde aplicar el desarrollo.

Aparte de su uso en el problema central de esta tesis, que es la reducción de modelos, el cálculo de valores propios también aparece en muchos otros problemas. Esto ha permitido utilizar las nuevas rutinas de altas prestaciones para resolver un problema de simulación del flujo oceánico, problema que es descrito a continuación de las nuevas rutinas.

Por último, se muestran algunos resultados experimentales obtenidos tanto en los problemas de reducción de modelos que se presentarán en próximos capítulos como en la solución de este problema real de simulación de flujos oceánicos.

Antes de empezar, es interesante hacer notar que el problema de valores propios que resulta útil para este trabajo es el que opera con matrices **densas** y pretende calcular **todos** los valores propios y no sólo unos pocos como ocurre en otras aplicaciones. En este trabajo, se aborda el problema de valores propios desde dos puntos de vista complementarios:

- Para el caso de las nuevas rutinas de reducción de modelos desarrolladas, el problema de valores propios que necesita resolverse es el cálculo de **todos** los valores propios y también la matriz ortogonal de transformación que ha permitido obtenerlos. Esto es lo que se conoce como la transformación a forma real de Schur, incluyendo el cálculo de los vectores de Schur. Además, en este caso se trabaja únicamente con el problema estándar de valores propios (una sola matriz de entrada al problema).

- Para el caso práctico real de simulación del flujo oceánico, el problema de

valores propios que necesita resolverse sigue siendo con matrices densas, pero ahora se trabaja con el problema generalizado (se parte de dos matrices) y además se necesitan también los vectores propios y no sólo los valores propios.

Es sobre estas dos tipologías del problema de valores propios sobre lo que versa en su mayoría este capítulo.

3.1. El problema de valores propios

El *problema de valores propios* consiste en calcular los valores λ para los que el sistema lineal de ecuaciones

$$Ax = \lambda x$$

tiene soluciones distintas de la trivial ($x = 0$). A es una matriz real de tamaño $n \times n$, x es un vector de tamaño n (lo que posteriormente se definirá como un *vector propio*) y λ es un escalar (que luego será llamado *valor propio*). Aunque la matriz A sea de números reales, x y λ pueden ser números complejos. Si la matriz A es de números reales y además simétrica, entonces sus valores y vectores propios serán reales.

Nota: En lo que sigue se describe el problema de valores propios siempre pensando en una matriz A de números reales, que es lo que sucede en los problemas hacia los que va orientado este trabajo. En general, la matriz A podría ser de números complejos.

Este sistema de ecuaciones puede reescribirse en la forma

$$(\lambda I - A)x = 0.$$

Aquí se observa que para que el sistema tenga soluciones distintas a la trivial, la matriz $\lambda I - A$ debe ser singular, es decir

$$\det(\lambda I - A) = 0.$$

Expandiendo este determinante se obtiene lo que se conoce como la ecuación característica de la matriz A:

$$\lambda^n + a_{n-1}\lambda^{n-1} + ... + a_1\lambda + a_0 = 0.$$

Su parte izquierda es un polinomio de grado n en λ cuyos ceros darán los *valores propios* de la matriz. Aunque no se calculan de esta manera.

Es interesante ver que, como los valores propios son los ceros de un polinomio con coeficientes reales (si trabajamos con una matriz A real), los valores propios complejos, de haber alguno, vendrán a pares. Si un número complejo es un valor propio de A, su conjugado también lo será.

Al problema de encontrar los valores λ y los vectores x que cumplen estas ecuaciones se le conoce como *el problema de valores propios* [GV13] [Saa11].

Será *el problema de valores propios estándar* si se parte únicamente de una matriz A o *el problema de valores propios generalizado* si se parte de un par de matrices (A, B) (como luego se verá).

Existen muchos problemas reales en los que se necesita resolver problemas de valores propios, en campos tan diversos como: teoría de control, cálculo de estructuras, acústica, dinámica de fluidos, electromagnetismo, ...

A continuación se describe nuevamente el problema y se citan algunos métodos habituales usados para resolverlo.

Definición 3.1 *Los* **valores propios** *de una matriz $A \in \mathbb{R}^{n\times n}$ son las n raíces (reales y/o complejas) de su polinomio característico $p(z) = \det(zI - A)$. El conjunto formado por estas raíces es lo que se conoce como* **espectro** *de la matriz A y se denota por $\lambda(A)$.*

En los problemas que se resolverán en los próximos capítulos se utiliza el concepto de matriz (semi)definida positiva, que está fuertemente relacionado con el signo de los valores propios de la matriz.

Definición 3.2 *Se dice que una matriz $A \in \mathbb{R}^{n\times n}$ es* definida positiva *si cumple*

$$u^T Au > 0 \quad \forall\ u \in \mathbb{R}^n, u \neq 0.$$

Proposición 3.1 (Criterio de Sylvester)
Una matriz simétrica P es definida positiva *si y sólo si cumple*

$$p_{11} > 0, det \begin{bmatrix} p_{11} & p_{12} \\ p_{21} & p_{22} \end{bmatrix} > 0, \ldots, det(P) > 0$$

Proposición 3.2 (Criterio de los valores propios positivos)
Una matriz A es definida positiva *si y sólo si todos sus valores propios son positivos.*

Si en esta definición y criterios para ver si una matriz es definida positiva se relajan las condiciones permitiendo el cero, las matrices que los cumplen se dice que son *semidefinidas positivas*. Por ejemplo, una matriz será semidefinida positiva si todos sus valores propios son no negativos.

Definición 3.3 *Para los λ que sean valores propios de $A \in \mathbb{R}^{n\times n}$, los vectores no nulos $x \in \mathbb{C}^n$ que cumplen*

$$Ax = \lambda x$$

son lo que se conoce como **vectores propios**.

Estos vectores son también conocidos como vectores propios **derechos** *de A, en contraposición con los vectores propios* **izquierdos** *(que cumplen $x^H A = \lambda x^H$).*

Nótese que dado un vector propio x para un valor propio λ de la matriz A, el vector αx para cualquier valor α no nulo también será un vector propio de la matriz para ese valor propio. Los vectores propios son en realidad direcciones, que son escaladas (por su valor propio) pero no rotadas al ser transformadas por la matriz.

Definición 3.4 *Dos matrices $A \in \mathbb{R}^{n\times n}$ y $B \in \mathbb{R}^{n\times n}$ se dice que son* **semejantes** *si existe una matriz $S \in \mathbb{R}^{n\times n}$ no singular tal que $B = S^{-1}AS$. En este caso, los valores propios de ambas matrices coinciden.*

Además, si una matriz A puede particionarse de la forma

$$A = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix},$$

se cumple que sus valores propios pueden obtenerse a partir de los valores propios de sus dos submatrices diagonales:

$$\lambda(A) = \lambda(A_{11}) \cup \lambda(A_{22}).$$

Muchos algoritmos de cálculo de valores propios utilizan la propiedad de semejanza para transformar la matriz original en otra en la que sea (más) fácil calcular los valores propios. Por ejemplo, puede tratar de anularse un bloque a la izquierda y abajo desde la subdiagonal y luego aplicar esta última propiedad para reducir el problema a la solución de dos subproblemas más pequeños. Esto es una forma de lo que se conoce como *deflación*, que consiste en ir reduciendo el problema a problemas más pequeños.

O, si lo que se necesita es obtener tan sólo unos cuantos valores y vectores propios, se pueden utilizar técnicas más eficientes, que además suelen permitir aprovechar una posible dispersión en la matriz original.

Las técnicas comúnmente utilizadas para resolver problemas de valores propios suelen dividirse en dos grandes frentes en función de lo que se pretende obtener:

- Cuando el objetivo es obtener *todos* los valores propios y posiblemente también sus vectores propios asociados, suelen usarse técnicas que van transformando la matriz (o matrices en el caso del problema generalizado) a casos más sencillos en los que la extracción de los valores propios sea inmediata. Este tipo de técnicas no suelen poder explotar una posible dispersión de los datos de la matriz, así que suelen trabajar realizando transformaciones con matrices densas.

- Cuando el objetivo es obtener unos pocos valores propios y posiblemente también sus vectores propios asociados, suelen usarse técnicas que habitualmente van obteniendo de forma iterativa aproximaciones de los vectores propios. Normalmente estas técnicas no modifican la matriz (o matrices) sino que van realizando operaciones usando la matriz (típicamente productos matriz vector). En estos casos suele ser fácil aprovechar una posible estructura dispersa de la matriz y beneficiarse tanto de no necesitar tener toda la matriz en formato denso en memoria como de realizar las operaciones teniendo en cuenta sólo los elementos no nulos de la matriz. Son técnicas que resultan muy apropiadas cuando la matriz (o matrices) es dispersa (tiene un alto número de elementos nulos).

En el caso de trabajar con el problema *simétrico* de valores propios (esto es que la matriz inicial sea simétrica), en ambos tipos de técnicas se puede aprovechar tanto la simetría de la matriz como el hecho de saber que los valores propios resultantes serán reales [Par98].

Los problemas de valores propios que aparecen en esta tesis son problemas no simétricos y en los que se desea conocer todo el espectro. Se trata pues de problemas en los que se utilizarán técnicas de transformación a casos más sencillos, que suelen basarse en el *el método iterativo* QR (o el QZ en el caso del problema generalizado). En los próximos apartados se describen brevemente algunas de estas técnicas.

3.1.1. Obtención de la forma real de Schur

Definición 3.5 *Una matriz cuadrada* $U \in \mathbb{C}^{n\times n}$ *se dice que es una* **matriz unitaria** *cuando su inversa es ella misma traspuesta y conjugada y por tanto se cumple que* $U^H U = UU^H = I$. *Una* **matriz ortogonal** *es una matriz unitaria con todos sus elementos reales.*

Definición 3.6 *Dada una matriz* $A \in \mathbb{R}^{n\times n}$ *con valores propios* $\lambda_1, \lambda_2, ..., \lambda_n \in \mathbb{C}$, *existe una matriz unitaria* $U \in \mathbb{C}^{n\times n}$ *tal que el producto* $U^H AU \in \mathbb{C}^{n\times n}$ *es una matriz triangular superior cuya diagonal son los valores propios de la matriz* A *(teorema 7.1.3. de [GV13]). A esta matriz triangular superior con los valores propios en su diagonal se la conoce como* **forma de Schur** *de la matriz* A. *Y las columnas de la matriz* U *son los* **vectores de Schur** *de la matriz* A.

A pesar de que una matriz real puede tener (y frecuentemente los tendrá) valores propios complejos, es muy habitual no querer introducir aritmética compleja simplemente para extraer estos valores propios. En estos casos, lo que se hace es realizar todas las operaciones siempre con números reales y la matriz resultado ya no es triangular superior, estrictamente hablando, pero a cambio todo son números reales.

Definición 3.7 *Dada una matriz* $A \in \mathbb{R}^{n\times n}$, *existe una matriz ortogonal* $U \in \mathbb{R}^{n\times n}$ *tal que el producto* $U^T AU \in \mathbb{R}^{n\times n}$ *es una matriz casi-triangular superior cuya diagonal a bloques tiene como valores propios los de la matriz* A. *A esta matriz casi-triangular superior se la conoce como* **forma real de Schur** *de la matriz* A. *Se trata de una matriz triangular superior a bloques en la que la diagonal principal son bloques* 1×1 *o* 2×2.

En la *forma real de Schur* se tienen todos los valores propios de la matriz original en una matriz de números **reales**. Los valores propios complejos se encuentran agrupados en bloques 2×2 de la diagonal a bloques de la matriz.

En algunos sitios, por ejemplo en algunas rutinas de trabajo con valores propios, se distingue entre *forma real de Schur* y *forma real de Schur estandarizada*.

Para la primera, simplemente se habla de tener una matriz casi-triangular superior, cuyos valores propios de los bloques de la diagonal son los de la matriz original.

Pero, por ejemplo, se permiten bloques 2×2 que tengan dos valores propios reales en lugar de un par de valores propios complejos conjugados.

Lo más habitual es (aún sin llamarlo así) tener los bloques diagonales de la matriz estandarizados. Esto quiere decir que los bloques diagonales 2×2 siempre son de un par de valores propios complejos conjugados y que además en estos bloques los elementos diagonales son idénticos y los no diagonales tienen un producto negativo (para que los valores propios asociados sean complejos). Un bloque 2×2 en forma estándar presenta la forma

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix}$$

y sus valores propios son $a \pm \sqrt{b \cdot c}$.

Cuando se desean calcular todos los vectores y valores propios de una matriz, normalmente se obtiene primero su forma (real) de Schur y luego es fácil sacar de ella esa información. Los métodos más eficientes para obtener la forma real de Schur de una matriz tienen como base el conocido como *método de la iteración QR*. Su algoritmo en su forma más básica actúa de esta manera:

Algoritmo 3.1 *Método de la iteración QR para la obtención de la forma real de Schur de una matriz A*

Partiendo de una aproximación inicial $T_0 = A$, iterar para $k = 1, 2, ...$ hasta que la matriz T_k esté en forma real de Schur:

1. *Calcular la descomposición QR de T_{k-1}: $T_{k-1} = Q_k R_k$.*
2. *Calcular la nueva T_k como el producto $R_k Q_k$.*

En este método se van obteniendo transformaciones de semejanza de la matriz original ($T_{k-1} = QR \rightarrow T_k = RQ = Q^{-1} T_{k-1} Q$), que convergen a la forma real de Schur de la matriz.

Esta es la versión básica del algoritmo, que nunca suele utilizarse. Lo habitual es encontrarse con versiones que aportan múltiples mejoras a esta:

- Para empezar, suele partirse de una matriz que no es directamente la original sino una semejante a ella y mucho más cerca de ser casi-triangular (típicamente la forma de Hessenberg).
- Además, se puede acelerar el proceso de convergencia mediante técnicas que se basan en realizar desplazamientos de la matriz.
- También hay algoritmos que procuran hacer múltiples iteraciones simultáneas.

Definición 3.8 *Una matriz H se dice que está en* **forma de Hessenberg** *cuando cumple que $h_{ij} = 0,\ i > j + 1$. Es decir, cuando todos sus elementos por debajo de la primera subdiagonal son nulos.*

Esta es la forma preferida para usar como partida en el algoritmo iterativo QR para el cálculo de la forma real de Schur de una matriz. Utilizar una matriz en forma

de Hessenberg permite que el coste por iteración sea $O(n^2)$ en lugar de $O(n^3)$, lo que supone una reducción muy importante y necesaria en el coste del proceso total.

Se puede obtener una matriz en forma de Hessenberg semejante a la matriz A de partida mediante una sucesión de transformaciones de semejanza que van anulando los elementos por debajo de la subdiagonal principal. Los elementos de cada columna son eliminados mediante una transformación de Householder aplicada por la izquierda, que es también aplicada por la derecha para mantener la semejanza con la matriz original.

Algoritmo 3.2 *Transformación a forma de Hessenberg mediante transformaciones de Householder*

Partiendo de $H_0 = A \in \mathbb{R}^{n\times n}$*, para* $k = 1, 2, ..., n-2$ *hacer:*

1. *Calcular una transformación de Householder* P_k *que anule los elementos de la columna* k *por debajo de la subdiagonal principal.*
2. *Aplicarla a* A_{k-1} *tanto por la izquierda como por la derecha:* $A_k \leftarrow P_k A_{k-1} P_k^T$

Lógicamente, en la implementación del algoritmo debe tenerse en cuenta que el tamaño de las transformaciones de Householder va reduciéndose en cada iteración lo que permite aplicarlo cada vez a una porción menor de la matriz. Además, es bastante habitual que la matriz en forma de Hessenberg se vaya calculando sobre la propia matriz original A, con lo que también se ahorra espacio de memoria.

El coste de este algoritmo es de $10/3n^3$ más $4/3n^3$ si se necesita construir la acumulación de las matrices de Householder [MW68]. También hay versiones optimizadas para hacer un mayor uso de BLAS nivel 3 [DSH89]. Pueden consultarse [KKQQ08], [TND10] y [KK11] para ver desarrollos más recientes del proceso de transformación a forma de Hessenberg.

Habitualmente el cálculo de la forma de Hessenberg de la matriz original es el primer paso en la resolución de problemas de valores propios. Y después se utiliza el algoritmo de la iteración QR optimizado para acelerar su convergencia [Wat07, Wat08]. En esta versión del algoritmo se utilizan desplazamientos (*shift*) de la matriz en cada iteración. Aunque un único desplazamiento daría problemas con valores propios complejos, así que se usa un desplazamiento doble, pero tiene mayor coste, razón por la que se utiliza en su forma implícita, cuyo coste es menor. Esta forma implícita fue descrita primeramente por Francis en lo que se conoce como *la iteración de Francis* [Fra61]. El algoritmo iterativo QR que suele utilizarse emplea esta *iteración de Francis* con lo que incluye desplazamientos y "persecución de barrigas"(*bulge chasing*) basados en el teorema de la Q implícita (teorema 7.4.2 de [GV13]). El coste total de obtener la forma real de Schur con este algoritmo mejorado (incluyendo la transformación a forma de Hessenberg), haciendo un cálculo muy aproximado, es de $25n^3$ si se calculan también las matrices ortogonales necesarias o de $10n^3$ si sólo se quieren los valores propios. Téngase en cuenta que al tratarse de un algoritmo iterativo, no es posible conocer a priori el número de iteraciones que va a requerir. Estos costes están extraídos de [GV13], donde se explica que en su cálculo se ha supuesto (tras observación empírica) que de media sólo se necesitan 2 iteraciones de Francis para

poder hacer deflación (eliminar una porción final del problema, correspondiente a un bloque 1×1 o 2×2 en función de si se ha encontrado un valor propio real o complejo, respectivamente).

De las primeras implementaciones en ordenador del *método iterativo QR* cabe destacar la de Martin, Petersen y Wilkinson [MPW70]. Una traducción a FORTRAN fue incluida en la librería EISPACK [SBD+76]. La versión inicial del método en la librería LAPACK (rutina `DHSEQR`) se basa en trabajo desarrollado por Bai y Demmel [BD89], cuya principal novedad era la introducción del uso de técnicas con múltiples desplazamientos para mejorar la localidad de los datos. Esta rutina fue mejorada en sucesivas versiones [AT97], hasta que en la versión 3.1 de LAPACK fue sustituida por una implementación que hacía uso de múltiples *barrigas* simultáneas y técnicas agresivas de deflación tempranas procedente del trabajo de Braman, Byers y Mathias [BBM02a, BBM02b, Bye07].

Con respecto a las implementaciones paralelas para el *método iterativo QR*, ha habido muchas versiones [VdG88, vdGH89, vdG93, SOH94, Wat94, HVDG96], aunque la primera versión altamente difundida fue la presente en la versión 1.5 de ScaLAPACK [BCC+97], en la forma de su rutina `PDLAHQR`, basada en el trabajo de Henry, Watkins y Dongarra [HWD02]. Desde entonces, Granat y Kagstrom están realizando múltiples nuevas versiones [GKK10, GKKS15] con el ánimo de sustituir esta rutina por versiones donde se explote mejor la simultaneidad de *barrigas* con desplazamientos múltiples y las técnicas agresivas de deflación tempranas.

De este último trabajo se han cogido las rutinas para la transformación a forma real de Schur que se han utilizado en esta tesis, aunque ha habido que corregir pequeños errores de implementación.

3.1.2. Cálculo de vectores propios a partir de la forma real de Schur

Si el problema de valores propios se resuelve utilizando la transformación a forma de Schur y aparte de los valores propios se precisan también los vectores propios, estos pueden calcularse a partir de los vectores propios de la matriz en forma de Schur [Dem97]. Y los vectores propios del problema en forma de Schur son más sencillos de calcular.

Si $Q^T AQ = T$ es la transformación a forma de Schur de la matriz A y x es un vector propio de la forma de Schur T, se cumple $Tx = \lambda x$. De la expresión de la transformación se tiene que $AQ = QT$, que puede aplicarse a la expresión del vector propio: $AQx = QTx = \lambda Qx$, de donde se deduce que Qx será un vector propio de A.

Por tanto, para calcular los vectores propios de A bastará con calcular los vectores propios de T y premultiplicarlos por la matriz ortogonal Q que permitió la transformación a forma de Schur.

El cálculo de los vectores propios de una matriz T en forma de Schur es más sencillo que calcularlos sobre una matriz genérica.

Por ejemplo, para calcular el vector propio x asociado a un valor propio real

$\lambda = t_{ii}$ de multiplicidad 1, habría que resolver el sistema de ecuaciones $(T-\lambda I)x = 0$:

$$\begin{pmatrix} T_{11}-\lambda I & T_{12} & T_{13} \\ 0 & 0 & T_{23} \\ 0 & 0 & T_{33}-\lambda I \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} (T_{11}-\lambda I)x_1 + T_{12}x_2 + T_{13}x_3 \\ T_{23}x_3 \\ (T_{33}-\lambda I)x_3 \end{pmatrix} = 0.$$

Nótese que $T_{22} = \lambda$, razón por la que en la matriz de coeficientes aparece cero en el centro ($T_{22} - \lambda = 0$).

Como en este ejemplo se asume multiplicidad 1 para λ, $T_{33} - \lambda I$ es no singular y la solución de $(T_{33} - \lambda I)x_3$ es $x_3 = 0$ (del tamaño adecuado).

Por la misma razón, $T_{11}-\lambda I$ es también no singular y la solución de $(T_{11}-\lambda I)x_1 = -T_{12}x_2$ para cualquier valor no nulo de x_2 dará el vector propio buscado. Tradicionalmente se tomará un valor arbitrario, por ejemplo $x_2 = 1$, y se resolverá este sistema de ecuaciones obteniendo que el vector propio correspondiente de la matriz T sería

$$x = \begin{pmatrix} -(T_{11}-\lambda I)^{-1}T_{12} \\ 1 \\ 0 \end{pmatrix}.$$

Es importante notar que la matriz de coeficientes del sistema de ecuaciones a resolver no es genérica, sino en forma real de Schur, es decir una matriz casi-triangular superior. Por tanto, la resolución del sistema es más sencilla que la de uno genérico. Además, su tamaño no es igual para todos los vectores, sino que va de 1 a $n-1$ siendo n el orden de la matriz T, lo que también debe explotarse para tener una implementación eficiente.

En el caso de que el valor propio no fuera real sino un par de valores propios complejos, las operaciones se complican un poco, pero el proceso global sigue manteniendo la filosofía aquí descrita.

Este es el método que hay implementado en la rutina `DTREVC` de LAPACK y es el que se ha utilizado en la nueva rutina desarrollada para realizar esta operación en paralelo: la rutina `PDTREVC`, que se presenta posteriormente.

3.1.3. Problema de valores propios generalizado

En ocasiones, en lugar de partirse de una sola matriz A se parte de dos matrices $A, B \in \mathbb{R}^{n\times n}$ (que también podrían ser complejas). Cuando sucede esto, el problema es conocido como *problema de valores propios generalizado.*

Definición 3.9

Se dice que el par de números complejos $(\alpha, \beta) \neq (0,0)$ es un valor propio generalizado del par de matrices (A,B) si existe un vector $x \neq 0$ para el que se cumple $\alpha Ax = \beta Bx$.

Dos valores propios generalizados $(\alpha, \beta), (\gamma, \delta)$ se consideran iguales si y sólo si se cumple que $\alpha\delta = \gamma\beta$.

Dadas dos matrices A y B de tamaño $n \times n$, se conoce como *haz de matrices* al conjunto de todas las matrices $A - \lambda B$ con $\lambda \in \mathbb{C}$. Los valores propios (α_i, β_i) del haz de matrices $\alpha A - \beta B$ representan los valores propios λ_i del haz $A - \lambda B$, siendo $\lambda_i = \beta_i / \alpha_i$ si $\alpha_i \neq 0$, ó $\lambda_i = \infty$ si $\alpha_i = 0$.

$\lambda(A, B)$ denota el conjunto formado por todos los valores propios generalizados del haz de matrices $\alpha A - \beta B$.

Definición 3.10 *Dado un valor propio generalizado $\lambda = \beta/\alpha$ para un par de matrices A y B ($A, B \in \mathbb{R}^{n \times n}$), el vector $x \in \mathbb{C}^n$ es un* vector propio *de $A - \lambda B$ si es no nulo y cumple $\alpha Ax = \beta Bx$.*

Definición 3.11
Se dice que un haz de matrices $\alpha A - \beta B$ es un haz regular si y sólo si existe un par de numeros complejos (α', β') que hacen que la matriz $\alpha' A - \beta' B$ sea no singular (invertible).

Dadas A y B de tamaño $n \times n$, hay exactamente n valores propios si el rango de B es n [GV13]. Si B es de rango deficiente, no se puede asegurar nada sobre la cantidad de valores propios, que podrían ser un conjunto vacío, finito o incluso infinito de valores complejos.

Si la matriz B es no singular, entonces se cumple que los valores propios del problema generalizado (A, B) coinciden con los del problema estándar asociado a la matriz $B^{-1}A$: $\lambda(A, B) = \lambda(B^{-1}A)$. Esto sugiere un primer método para resolver el problema generalizado cuando la matriz B sea no singular [GV13]:

Algoritmo 3.3 *Cálculo de valores propios generalizados partiendo de una matriz B no singular*

1. *Obtener la matriz C tal que $BC = A$. Esto es resolver un sistema de ecuaciones lineales con múltiples partes derechas.*

2. *Usar el algoritmo iterativo QR para calcular los valores propios de C que coincidirán con los valores propios generalizados de (A, B).*

Este algoritmo no es adecuado si la matriz B está mal condicionada, ya que se introducirán errores en la resolución del sistema que afectarán negativamente a la precisión de los valores propios calculados.

Actualmente no hay rutinas en la librería ScaLAPACK para resolver el problema generalizado. Así que se han desarrollado nuevas rutinas para resolverlo siguiendo este algoritmo (teniendo en cuenta que no siempre es aplicable). La resolución del sistema de ecuaciones que aparece se ha implementado usando la descomposición QR para minimizar los posibles errores en esa parte del algoritmo.

Cuando la matriz B es singular o está mal condicionada, puede acudirse a la *forma de Schur generalizada*, similar a la forma ya vista para el caso estándar, pero ahora generalizada [MS73].

Definición 3.12 *Dadas dos matrices $A, B \in \mathbb{R}^{n\times n}$ existen dos matrices unitarias $Q, Z \in \mathbb{C}^{n\times n}$ tales que los productos $Q^H AZ = T$ y $Q^H BZ = S$ son matrices triangulares superiores. El par de matrices (T, S) es la* **forma de Schur generalizada**.

Además, si alguno de los elementos de la diagonal principal es a la vez nulo en T y S, los valores propios generalizados de A y B son todo el conjunto de los números complejos. Si no, los valores propios generalizados de A y B son el conjunto formado por los números t_{ii}/s_{ii} para los s_{ii} no nulos

$$\lambda(A, B) = \{t_{ii}/s_{ii} : s_{ii} \neq 0\}$$

y además el valor propio infinito, con multiplicidad igual al número de s_{ii} nulos.

Al igual que pasaba en el caso estándar, muchas veces se trabaja con matrices reales y aunque el resultado puede estar en el campo de los complejos, suele representarse de forma que se hagan todas las operaciones siempre con números reales.

Definición 3.13 *Dadas dos matrices $A, B \in \mathbb{R}^{n\times n}$ existen dos matrices ortogonales $Q, Z \in \mathbb{R}^{n\times n}$ tales que el producto $Q^T AZ = T$ es una matriz casi-triangular superior y el producto $Q^T BZ = S$ es triangular superior. El par de matrices (T, S) es la* **forma real de Schur generalizada**.

Al igual que pasaba con el caso estándar, el método más utilizado para calcular *todos* los valores propios generalizados pasa por obtener la forma real de Schur generalizada. En el caso estándar se empezaba transformando la matriz A a forma de Hessenberg superior. En el caso generalizado se hace una transformación similar de A y B que deja la matriz A en forma de Hessenberg superior y la matriz B en forma triangular superior. Con esto, la transformación posterior a forma real de Schur es más eficiente. Esta transformación en el caso estándar se hacía con el método iterativo QR y en el caso generalizado ahora se hace con el método similar conocido bajo el nombre de método iterativo QZ.

Los algoritmos básicos para todos estos pasos pueden consultarse, por ejemplo, en [GV13].

A pesar de que se están abordando estas transformaciones también desde implementaciones paralelas [ADK02, AKK07, AKK14], de momento no están muy difundidas. Por esto, en los nuevos desarrollos realizados en este trabajo se ha resuelto el problema usando la transformación a forma estándar antes comentada.

3.2. Rutinas de altas prestaciones desarrolladas

La transformación de una matriz de forma general a forma real de Schur es un paso importante en la resolución del problema de valores propios. También es necesaria esta transformación en muchos métodos de reducción de modelos, como se explicará en el capítulo 5.

Esta operación de transformación a forma real de Schur se realiza habitualmente en dos pasos: primero se pasa de forma general a Hessenberg y luego de Hessenberg a forma real de Schur.

La transformación de forma general a forma de Hessenberg en paralelo está presente en la librería ScaLAPACK y se ha utilizado (rutina `PDGEHRD`).

El paso de forma de Hessenberg a forma real de Schur en paralelo estaba en un punto muy básico en ScaLAPACK (rutina `PDLAHQR`) al comienzo de este trabajo.

Con estas rutinas para realizar los dos pasos básicos, se ha desarrollado una nueva que las usa para calcular la transformación a forma real de Schur de una matriz (nueva rutina `PDGEES`) ayudándose de alguna nueva rutina auxiliar para operaciones intermedias.

En una primera aproximación se utilizó la versión básica de ScaLAPACK para pasar de forma de Hessenberg a forma real de Schur (rutina `PDLAHQR`), añadiéndole alguna operación que faltaba.

Afortunadamente se ha avanzado mucho en nuevos algoritmos paralelos para la etapa de transformación de forma de Hessenberg a forma real de Schur [GKK10, GKKS15]. Algunos de estos algoritmos ya están presentes en la librería ScaLAPACK. Sin embargo, tal como están en la librería no son muy usables ya que contienen múltiples errores.

Se ha realizado un intensivo trabajo de reparación de estas rutinas, cuyos errores se han comunicado a los autores de las mismas (que nos mencionan en sus agradecimientos en su artículo [GKKS15]).

Con estas nuevas rutinas arregladas, se ha podido utilizar los nuevos métodos para el paso de transformación de forma de Hessenberg a forma real de Schur, mediante una nueva versión de la rutina `PDGEES`, lo que ha supuesto una notable reducción del tiempo invertido en esta parte del algoritmo.

En esta nueva versión de la rutina `PDGEES`, básicamente se ha sustituido el uso de la antigua rutina de transformación a forma de Schur `PDLAHQR` por la nueva `PDHSEQR`.

Además, la reparación de estas rutinas ha permitido utilizarlas en problemas de gran dimensión de cálculo de valores propios, como el presente en la simulación de flujos oceánicos, que se explica como caso práctico en un apartado posterior de este mismo capítulo. Para este problema además se ha tenido que desarrollar una rutina (`PDGGEV`) que resuelva el problema de valores propios generalizado, a base de transformarlo a un problema estándar y luego resolver este.

Y también ha sido necesario el desarrollo de una versión paralela para el cálculo de los vectores propios (rutina `PDTREVC`), ya que esta operación no dispone de implementación paralela en ScaLAPACK.

Todas estas nuevas rutinas pueden verse en el grafo que muestra las llamadas entre ellas, en la figura 3.1.

En este grafo puede verse en la parte superior las dos nuevas rutinas principales: `PDGEES` para la transformación a forma real de Schur del problema estándar y `PDGGEV` para obtener los valores y vectores propios del problema generalizado.

La rutina `PDHSEQR` y todas las que tiene por debajo se corresponden con la última versión del algoritmo optimizado para calcular la forma real de Schur de una matriz. Esta rutina y todas las que aparecen llamadas por ella en el grafo **no** son rutinas desarrolladas en esta tesis. Se muestran en el grafo porque todas ellas han requerido alguna reparación, pero no se ha realizado su completa implementación, como sí ocurre con el resto de rutinas del grafo.

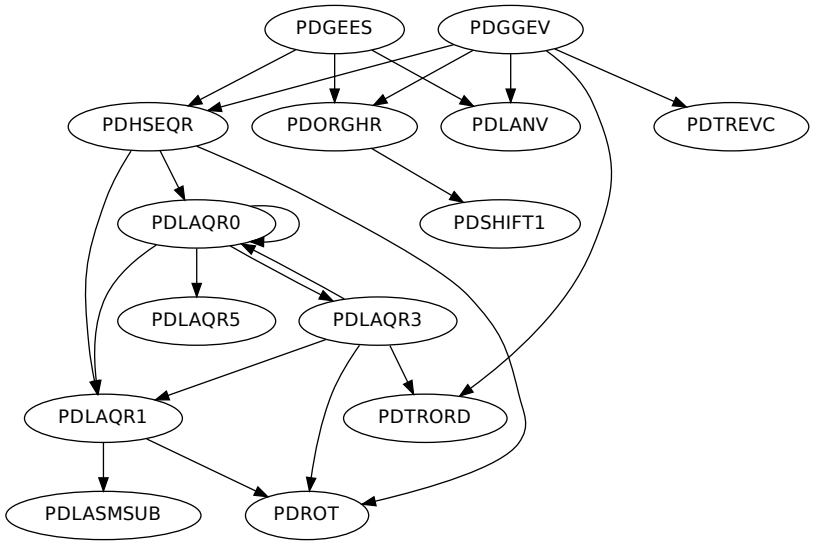

Figura 3.1: Rutinas de altas prestaciones desarrolladas (algunas sólo modificadas) para resolver problemas de valores propios

A continuación se describen las nuevas rutinas desarrolladas para poder utilizar los nuevos métodos de obtención de la forma real de Schur, así como nuevas herramientas que han sido necesarias para poder calcular los vectores y valores propios generalizados, que aparecen en la resolución del problema de simulación de flujos oceánicos.

3.2.1. PDGEES

```
PDGEES( JOBVS,SORT,SELECT,N,A,IA,JA,DESCA, SDIM, WR, WI,
                          Z,IZ,JZ,DESCZ, DWORK,LWORK,BWORK,INFO )
```

`PDGEES` calcula la factorización de Schur de una matriz cuadrada general A, calculando además sus valores propios y la matriz de vectores de Schur utilizada Z: $A = ZTZ^T$. `PDGEES` es una versión paralela, con menos funcionalidad, de la rutina `DGEES` de LAPACK. La rutina de LAPACK permite de forma opcional realizar un reordenamiento posterior de los valores propios dentro de la matriz, llevando a la parte superior izquierda aquellos que se indique mediante una función que se pasa como parámetro. Esto no ha sido implementado en la versión paralela, por no necesitarse esta funcionalidad en las rutinas de reducción de modelos implementadas.

Esta rutina permite calcular la forma real de Schur de una matriz (ver sección 3.1.1).

La rutina de LAPACK `DGEES` obtiene la forma de Schur de una matriz en dos pasos. Primero realiza una transformación de semejanza ortogonal de la matriz a forma de Hessenberg, mediante la rutina `DGEHRD`. Después realiza una transformación de forma de Hessenberg a forma de Schur, mediante la rutina `DHSEQR`.

Además, cuando se calculan los vectores de Schur, hay que obtener la matriz ortogonal de la transformación a forma de Hessenberg antes de realizar la transformación de Hessenberg a Schur. Esto es así, porque la rutina `DGEHRD`, como la mayoría de rutinas de LAPACK (y también de ScaLAPACK) que operan con transformaciones de Householder, no devuelve la matriz ortogonal como tal sino en forma de reflectores. Se devuelve una matriz que contiene todos los vectores correspondientes a las transformaciones de Householder realizadas (tradicionalmente llamados vectores v) y aparte un vector con los multiplicadores usados (tradicionalmente llamado valores *tau*). Se hace de esta manera porque se puede operar de forma eficiente con las matrices así representadas y muchas rutinas lo hacen. Por contra, cuando se necesita tener la matriz ortogonal que representan, hace falta realizar un trabajo extra para transformar los reflectores a matriz. Y este es el caso si se precisan calcular los vectores de Schur. La rutina `DHSEQR` espera recibir la matriz ortogonal para actualizarla con las operaciones necesarias para obtener la forma real de Schur. Así que entre las llamadas a ambas rutinas se utiliza la rutina `DORGHR` que obtiene la matriz ortogonal correspondiente a los reflectores elementales proporcionados.

En la versión paralela desarrollada se siguen estos mismos pasos básicos para realizar un proceso similar que permita obtener la forma real de Schur de una matriz. Sin embargo, en ScaLAPACK no se tenían versiones paralelas de las tres rutinas que acaban de comentarse. Tan sólo se ofrece implementada la rutina `PDGEHRD` correspondiente a la paralelización de la rutina `DGEHRD` a cargo de la transformación a forma de Hessenberg superior. Las otras dos han tenido que ser desarrolladas.

En el caso de la rutina `DORGHR`, se ha implementado la correspondiente versión paralela en `PDORGHR`.

Sin embargo, en el caso de la rutina `DHSEQR`, a pesar de no disponerse inicialmente de una rutina paralela equivalente en ScaLAPACK, sí que se tenía una primera versión paralela de la rutina de LAPACK `DLAHQR` en la rutina `PDLAHQR`. La rutina `PDLAHQR` puede utilizarse como alternativa paralela para la transformación de forma de Hessenberg a forma de Schur, siempre que se tengan en cuenta sus limitaciones. En particular, resulta necesario realizar un post-proceso que pase los pares de valores propios complejos a su forma estándar. Esto es lo que hace la nueva rutina desarrollada bajo el nombre `PDLANV`.

En su primera versión esta rutina hacía uso de la rutina `PDLAHQR`. Pero, como ya se ha comentado, actualmente sí que hay en ScaLAPACK una versión paralela de `DHSEQR` (`PDHSEQR`), que ha podido utilizarse tras reparar múltiples pequeños errores presentes en ella y las rutinas auxiliares que utiliza.

`PDGEES` utiliza las rutinas `PDORGHR` y `PDLANV`, que se explican a continuación.

3.2.2. PDORGHR

```
PDORGHR( N, ILO,IHI, A,IA,JA,DESCA, TAU, WORK,LWORK, INFO )
```

PDORGHR obtiene la matriz ortogonal Q de la transformación a forma de Hessenberg ($Q^T AQ = H$) realizada en la rutina de ScaLAPACK PDGEHRD a partir de los reflectores elementales devueltos por esta rutina. PDORGHR es una versión paralela de la rutina DORGHR de LAPACK.

La versión secuencial básicamente realiza un desplazamiento de una submatriz de la matriz de entrada una columna a la derecha, rellenando el resto como la matriz identidad y luego llama a la rutina DORGQR para transformar los reflectores de la submatriz en una matriz ortogonal.

Como DORGQR tiene en ScaLAPACK su versión paralela en PDORGQR, la versión paralela de DORGHR es muy similar a la secuencial.

Se podría hacer el desplazamiento, rellenar el resto con la matriz identidad y luego llamar a PDORGQR para la transformación a matriz ortogonal. Sin embargo, en la versión paralela hay que prestar atención al vector TAU que a la entrada a la rutina tiene los factores escalares de los reflectores tal como los devolvió PDGEHRD. Este vector en la versión paralela está distribuido y repartido entre los procesos siguiendo una distribución análoga a la de la submatriz con que se trabaja. Esto implica que cuando se realizara el desplazamiento de la submatriz, habría que desplazar también el vector TAU para que la rutina PDORGQR encuentre cada factor escalar coincidente con su reflector asociado. Pero desplazar el vector TAU, que está distribuido, supondría más comunicaciones. No es que vayan a ser muchas más comunicaciones, puesto que serían pocas. Pero sucede que evitar el tener que desplazar TAU es muy sencillo.

En la versión paralela desarrollada, primero se hace la llamada a PDORGQR para que transforme los reflectores a matriz y después se hace el desplazamiento. De esta manera se consigue que el vector TAU y la submatriz lleguen con una distribución correcta para la rutina PDORGQR, al mismo tiempo que se ahorra el desplazamiento del vector TAU. Simplemente con desplazar la matriz ortogonal después de su cálculo, ya se tiene donde debe ser dejada por esta rutina y se ahorran unas pocas comunicaciones.

Respecto a la inicialización del resto de la matriz como la matriz identidad, en la versión paralela se hace parte de esta inicialización delante y parte detrás de la transformación. Ahora tiene que hacerse así para que la submatriz se pase bien a la rutina de transformación y además se rellene el hueco final que queda después de desplazar.

PDORGHR utiliza la rutina PDSHIFT1 para realizar el desplazamiento de la submatriz.

3.2.3. PDSHIFT1

```
PDSHIFT1( DIR, M,N,A,IA,JA,DESCA, DWORK,LDWORK, INFO )
```

PDSHIFT1 realiza un *desplazamiento* del contenido de una matriz una fila hacia arriba (U) o hacia abajo (D), o una columna hacia la izquierda (L) o hacia la derecha

(`R`), según se le indique en el parámetro `DIR`. Es una rutina auxiliar que no se corresponde con la paralelización de ninguna rutina secuencial de BLAS, LAPACK o SLICOT.

Esta operación de desplazamiento de las filas o columnas de una matriz una posición resulta muy fácil en el caso secuencial. Sin embargo, en el caso paralelo esta operación no es trivial ya que requerirá de comunicaciones en los bordes de los fragmentos de matriz que posee cada proceso. Por esta razón ha sido conveniente independizar esta tarea en una rutina.

Utilizando una distribución cíclica, por cada bloque de filas/columnas de la matriz presente en un proceso, se tendrá que comunicar a los procesos adyacentes una fila/columna y también recibirla.

Esta operación de enviar a un proceso y recibir de otro esta implementada en algunas librerías de comunicación en una única rutina. Utilizar esta rutina sería lo más eficiente en caso de disponer de ella. Por ejemplo, en MPI se podría utilizar la rutina `MPI_Sendrecv` para hacer estas comunicaciones, o incluso usar la rutina `MPI_Sendrecv_replace` que es similar salvo por reutilizar el mismo *buffer* para el envío y la recepción. Sin embargo, hay que recordar que entre los objetivos del trabajo desarrollado se incluye fomentar la portabilidad, al menos permitiendo utilizar las nuevas rutinas allá donde se pueda utilizar ScaLAPACK. Esto requiere que para las comunicaciones deban utilizarse rutinas que con toda seguridad se dispongan en ScaLAPACK. Las comunicaciones en ScaLAPACK se efectúan mediante la librería de comunicaciones BLACS, aunque en muchas ocasiones resultará que ésta trabaje por encima de MPI. Y en BLACS no hay rutinas para realizar la operación de envío y recepción de una sola vez, por lo que hay que separar el envío y la recepción en dos operaciones distintas (esto mismo sucede también en la rutina `PDSHIFT` que se explica en un capítulo posterior).

Una forma eficiente de realizar el desplazamiento consiste en empezar copiando los bordes que deberán comunicarse en un *espacio de trabajo* (lo que en las rutinas estándar se conoce como `WORK`) adecuado. Posteriormente se realizaría el desplazamiento de los fragmentos de matriz que posee cada proceso, todo ello en paralelo y sin comunicaciones. Al final, sólo resta realizar una serie de envíos y sus correspondientes recepciones en el orden adecuado para lograr el desplazamiento deseado también en los bordes de las submatrices.

Este mismo proceso puede hacerse sin necesidad de un *espacio de trabajo*. En este caso, por cada bloque de la matriz que se tiene en un proceso, se empieza enviando la porción que quedará en otro proceso (puesto que va a sobreescribirse al desplazar), entonces se realiza el desplazamiento de este bloque en la dirección deseada y después se recibe el final de este bloque tras el desplazamiento que provendrá de otro proceso. Esta operación se debe repetir para cada bloque contiguo de la matriz almacenado en cada proceso. Al utilizar comúnmente una distribución cíclica y no usar *espacio de trabajo*, hay que ir comunicando antes y después de desplazar cada bloque de submatriz contiguo de la matriz global. En esta ocasión se requieren más comunicaciones que en el caso anterior, aunque cada una envía sólo una fila/columna de la matriz en lugar de un bloque.

Pero lo peor de este último método de realizar el desplazamiento no es el incre-

mento en el número de comunicaciones. El hecho de tener que hacer las operaciones en el orden indicado, para no sobreescribir una parte de la matriz antes de haberla utilizado, puede producir una secuencialización indeseada de todo el proceso. En aquellas plataformas en las que la implementación de las rutinas de comunicación sean síncronas, o aún cuando no siéndolo se use un *buffer* para las comunicaciones en el que no quepa una fila/columna completa, todo el proceso se realizaría de forma secuencial con un nivel de paralelismo inexistente.

Por tanto, resulta más eficiente el procedimiento indicado anteriormente en que sí se utiliza *espacio de trabajo*. Para este procedimiento se requiere un *espacio de trabajo* del tamaño de un bloque de filas/columnas con tantas filas/columnas como bloques de filas/columnas de la matriz se tengan en un mismo proceso. Esto puede ser un tamaño considerable al compararlo con la dificultad mínima de la operación que se realiza. Es por esto que en lugar de implementar de esta forma el desplazamiento, se ha optado por una versión intermedia.

En la rutina implementada se utiliza un *espacio de trabajo* correspondiente a una fila/columna de la matriz, lo que es mucho más razonable. Con esto se consigue eliminar la secuencialización que puede aparecer si no se usa *espacio de trabajo*, aunque se incrementa el número de comunicaciones necesarias con respecto a la versión que usa más espacio (pero ahora las comunicaciones son de menor tamaño).

Se trata de una solución intermedia que logra un buen paralelismo con un mínimo *espacio de trabajo*. Con un espacio mayor, se mejoraría el algoritmo pero se necesitaría más memoria. Cabe la posibilidad de implementar las dos versiones que usan más y menos *espacio de trabajo* y que la rutina elija en tiempo de ejecución una de ellas en función del espacio suministrado. Sin embargo, esto no se ha implementado. La versión implementada funciona suficientemente bien y no mejoraría mucho con más espacio. Sí que supondría una mejora significativa para esta rutina en el caso de encontrarse en una plataforma donde las comunicaciones tengan un tiempo de establecimiento de la comunicación importante, ya que se minimizaría el número de envíos.

A continuación se ilustra el algoritmo finalmente implementado (para el caso de un desplazamiento hacia arriba. Los algoritmos para las otras direcciones son análogos):

Algoritmo 3.4 `PDSHIFT1` *(con* `DIR=U`*)*

Cada proceso recorre de arriba abajo los bloques de filas que posee de la matriz global, haciendo para cada bloque lo siguiente:

1. *Copiar la primera fila en el* espacio de trabajo*.*
2. *Desplazar una fila hacia arriba todas las demás.*
3. *Si hay más de una fila de procesos, entonces:*
   a) *Si no se trata del primer bloque de filas de la matriz global, enviar la primera fila de este bloque (que está almacenada en el* `WORK`*) al proceso inmediatamente por encima en la malla lógica de procesos.*

b) *Si no se trata del último bloque de filas de la matriz global, recibir la última fila de este bloque del proceso inmediatamente por debajo en la malla de procesos.*

En la figura 3.2 se muestra un esquema de las operaciones de este algoritmo para desplazar hacia arriba una matriz distribuida de forma no cíclica. En el caso de una distribución cíclica (lo habitual), se repetirían las operaciones indicadas en la figura tantas veces como bloques de filas/columnas tenga cada proceso.

Figura 3.2: Esquema de operaciones en `PDSHIFT1` con `DIR=U` y una distribución no cíclica

Es conveniente, y así se ha hecho, realizar el recorrido de la matriz en cada caso en el sentido apropiado para que en las primeras comunicaciones siempre haya procesos que no envíen, que empiecen recibiendo. Dependiendo de la forma en la que se haya implementado en cada plataforma las rutinas de envío y recepción, podrían llegar a producirse interbloqueos si no se hubiera escogido un orden adecuado en las comunicaciones. Si a bajo nivel las comunicaciones son síncronas y al principio todos los procesos quieren enviar, se produciría un interbloqueo. Hoy en día será raro encontrar alguna plataforma en la que se hayan implementado las comunicaciones de esa manera. Lo normal es encontrar envíos asíncronos, con lo que presumiblemente no se presentaría este problema. Sin embargo, a la hora de desarrollar nuevo código hay que ponerse en el peor caso, esto es suponer que todas las comunicaciones son síncronas, e implementar un código que funcione para esta situación. Si, como es normal, luego se trabaja en una plataforma en la que las comunicaciones no son síncronas, pues más eficiente será el código. Pero se debe garantizar la portabilidad, vigilando que el código no pueda producir interbloqueos aún cuando las comunicaciones sean síncronas.

El orden adecuado para que este algoritmo no pueda provocar interbloqueos se corresponde con seguir el sentido inverso a aquél hacia dónde se desplaza la matriz. Este orden curiosamente coincide con el que la lógica exige en el caso de realizar esta operación en un único proceso.

Hay otros algoritmos diferentes al comentado aquí para realizar la operación de desplazamiento libre de interbloqueos. Algunos de ellos se basan por ejemplo en que cada vez que hay una comunicación siempre envían primero los procesadores impares y reciben los pares y luego al revés. Esta es otra forma de evitar el posible

interbloqueo.

3.2.4. PDLANV

```
PDLANV( N, A,IA,JA,DESCA, Z,IZ,JZ,DESCZ, WORK,LWORK )
```

La rutina PDLANV pasa a forma estandarizada los bloques 2×2 de la diagonal de la matriz A que ya está en forma real de Schur. Además actualiza convenientemente la matriz Z que acumula las transformaciones realizadas para llegar a la forma de Schur.

Como se comentó en la sección 3.1.1, la forma real de Schur se dice que está en forma estándar si los bloques 2×2 de la diagonal se corresponden efectivamente con pares de valores propios complejos (y no con 2 valores propios reales) y además presentan la forma:

$$\begin{pmatrix} a & b \\ c & a \end{pmatrix},$$

siendo $b \cdot c < 0$. Los valores propios de cada uno de estos bloques son $a \pm \sqrt{b \cdot c}$.

Esta rutina resulta necesaria porque las versiones preliminares de ScaLAPACK para el cálculo de la forma real de Schur no la devuelven (siempre) en su forma estandarizada.

En LAPACK está la rutina DLANV2 que realiza esta estandarización para un bloque 2×2. La forma de estandarizar toda la matriz es aplicar de forma sucesiva esta rutina sobre todos los bloques 2×2 y actualizar las transformaciones sobre la matriz Z. Esto que es muy sencillo en el caso secuencial, no lo es tanto en el caso paralelo, donde las matrices se encuentran distribuidas. Esto implica que una pequeña actualización deba probablemente ser realizada en múltiples procesos, al disponer cada uno de ellos de una porción de las matrices involucradas. Y además, en el caso paralelo puede suceder que varios de los bloques 2×2 de la diagonal de la matriz A se encuentran repartidos entre varios procesos, lo que complica su manejo.

Todo esto ha sido tenido en cuenta en la versión paralela de la rutina. Aunque esta rutina supone un tiempo de ejecución muy pequeño, no por ello ha sido menospreciada a la hora de paralelizarla. Se ha procurado realizar su paralelización de forma eficiente, lo que se refleja en que a pesar de tener un coste muy pequeño, se consigue reducir su tiempo de ejecución al ejecutarla en paralelo.

Los bloques se pasan a forma estándar en dos fases diferenciadas. Primero se procede con los bloques que están completamente almacenados en un proceso, lo que puede hacerse en paralelo con el resto de procesos. Después se opera, de forma secuencial, con los bloques que están partidos entre varios procesos. Antes de la primera fase se difunden por filas y por columnas los bloques de la diagonal de A, para que todos los procesos tengan los elementos de aquellos bloques que necesitan para actualizar sus filas y/o columnas (salvo los de bloques partidos entre varios procesos). Durante la segunda fase, los elementos de los bloques repartidos entre varios procesos son comunicados a los vecinos que los necesitarán.

El algoritmo empleado puede resumirse así:

Algoritmo 3.5 `PDLANV`

1. *Difundir los bloques* 2×2 *diagonales de* A *a los procesos que comparten filas o columnas con ellos (esto es difundir los elementos de las 3 diagonales centrales a los procesos de sus filas y columnas).*

2. *Estandarizar los bloques* 2×2 *que no están repartidos entre varios procesos, actualizando* A *y* Z *en paralelo.*

3. *Estandarizar los bloques partidos entre varios procesos. Esto supone ir viendo todos los cambios de bloque diagonal de la matriz* A *y en caso de detectar un corte de un bloque* 2×2*, comunicar a los 2 (o 4) procesos involucrados todo el bloque para poder realizar su estandarización y actualización de las matrices* A *y* Z*.*

3.2.5. PDGGEV

```
PDGGEV( JOBVL, JOBVR, N, A,IA,JA,DESCA, B,IB,JB,DESCB,
        ALPHAR, ALPHAI, BETA,
        VL,IL,JL,DESCL, VR,IR,JR,DESCR,
        WORK,LWORK, IWORK,LIWORK, INFO )
```

La rutina `PDGGEV` calcula para un par de matrices generales A y B sus valores propios generalizados y opcionalmente los vectores propios izquierdos y/o derechos. Es una versión paralela de la rutina secuencial de LAPACK equivalente: `DGGEV`.

Esta rutina ha sido desarrollada para su uso en el problema de simulación del flujo oceánico (ver sección 3.4.3). Aunque es completamente independiente de ese problema y por tanto puede utilizarse para su función en cualquier otra ocasión en que se necesite resolver el problema de valores propios generalizado. Eso sí, la matriz B del problema a resolver debe ser no singular y debe estar bien condicionada. Estas condiciones se cumplen en el problema de simulación del flujo oceánico donde se va a utilizar la rutina. La versión secuencial de LAPACK es más general y no exige estas restricciones.

En ScaLAPACK no hay todavía rutinas paralelas para la transformación a forma real de Schur generalizada. Por esta razón, en esta nueva rutina se resuelve el caso generalizado utilizando la transformación a Schur del caso estándar, lo que no es adecuado en todos los casos.

La transformación, en teoría, supone la inversión de la matriz B y su aplicación a la matriz A. En la implementación, esta operación se ha hecho con el mayor cuidado: se calcula la descomposición QR de B aplicando Q^T a A y resolviendo posteriormente un sistema triangular con múltiples partes derechas para obtener $B^{-1}A$ de forma precisa, siempre que el número de condición de R no sea demasiado elevado.

En cualquier caso, este proceso no es posible si B no es invertible. Se ha añadido una detección de singularidad de la matriz B, de forma que si algún elemento diagonal de la matriz R, obtenida de su descomposición QR, es menor en valor absoluto que un cierto valor, se finaliza la ejecución avisando del problema.

En ScaLAPACK tampoco está todavía la rutina encargada del cálculo de los vectores propios de una matriz quasi-triangular (`PDTREVC`). Así que esta rutina se ha desarrollado y se explica posteriormente. Para facilitar la implementación de esta nueva rutina, resulta conveniente garantizar que no se produce ningún corte de un bloque diagonal 2×2 de la matriz A correspondiente a un par de valores propios complejos. Es decir, que la distribución de la matriz haga que cada uno de todos esos bloques esté por completo en un solo proceso. El proceso genérico de reordenación de los bloques no es trivial, pero afortunadamente está implementado en ScaLAPACK, en la rutina `PDTRORD` [GKK09]. Utilizando esta rutina existe una forma sencilla de conseguir no cortar ningún bloque 2×2 mediante la realización de un reordenamiento previo al cálculo de los vectores propios. Para que este reordenamiento siempre sea posible, se ha optado por exigir que el tamaño de bloque utilizado en la distribución sea par. Con esto siempre existe un reordenamiento sencillo que consigue no cortar entre varios procesos ninguno de los bloques. Bastaría con situar en la parte superior de la matriz todos los valores propios complejos, que al ser bloques 2×2 encajarán en un tamaño de bloque par sin cortarse.

En realidad, no resulta necesario trasladar todos los bloques al principio de la matriz. Se ha realizado una pequeña optimización consistente en lo siguiente. Primero se busca desde el principio de la matriz el primer bloque 2×2 cortado entre varios procesos y se procede del mismo modo desde el final de la matriz. Con esto, es suficiente con reordenar la submatriz de A que va desde el primer corte hasta el último, dejando en ella los bloques de valores propios complejos en la parte superior.

A continuación se muestra una descripción del algoritmo paralelo implementado para la transformación a forma de Schur generalizada que se ha utilizado en esta rutina:

Algoritmo 3.6 `PDGGEV`

1. *Realizar un escalado previo de la matriz A, si su mayor elemento en valor absoluto es muy grande o muy pequeño. Ídem con la matriz B.*

2. *Obtener la descomposición QR de la matriz B y aplicar las transformaciones ortogonales a la matriz A (rutinas* `PDGEQRF` *y* `PDORMQR` *de ScaLAPACK).*

3. *Calcular el menor elemento en valor absoluto de la diagonal de R, para considerar B singular y abortar si este valor es muy pequeño.*

4. *Terminar la transformación de problema generalizado a problema estándar calculando $R^{-1}A$, usando para ello la resolución de un sistema triangular con múltiples partes derechas (rutina* `PDTRSM` *de PBLAS).*

5. *Si se desean los vectores propios izquierdos, inicializar estos a la matriz ortogonal de la descomposición QR previa, transformando los reflectores a matriz ortogonal (rutina* `PDORGQR` *de ScaLAPACK).*

6. *Transformar la nueva matriz A a forma Hessenberg superior (rutina* `PDGEHRD` *de ScaLAPACK).*

7. *Si se desean los vectores propios derechos, inicializar estos a la matriz ortogonal de la transformación a forma de Hessenberg superior, transformando los reflectores a matriz ortogonal (nueva rutina* `PDORGHR`*).*

8. *Obtener la forma real de Schur usando la rutina* `PDHSEQR` *de ScaLAPACK y luego la nueva rutina* `PDLANV` *para estandarizar los bloques* 2×2 *de valores propios complejos.*

9. *Si se desean vectores propios, reordenar la porción de matriz que presente cortes entre varios procesos de los bloques* 2×2 *de valores propios complejos (rutina* `PDTRORD` *de ScaLAPACK, indicándole qué hay que reordenar). Luego se calculan los vectores propios con la nueva rutina* `PDTREVC`*.*

10. *Deshacer los escalados, si se hizo alguno.*

3.2.6. PDTREVC

```
PDTREVC( SIDE, HOWMNY, SELECT, N, T,IT,JT,DESCT,
         VL,IL,JL,DESCL, VR,IR,JR,DESCR,
         MM, M, ALPHAR,ALPHAI,
         WORK,LWORK, INFO )
```

La rutina `PDTREVC` calcula los vectores propios izquierdos y/o derechos de una matriz en forma real de Schur (que se habrá obtenido externa y previamente a la llamada a esta rutina). Se permite tanto sólo calcularlos, como también acumularlos sobre matrices ya inicializadas. Esto permite aplicarlos sobre las matrices ortogonales usadas para la transformación previa a forma de Schur, obteniendo así los vectores propios de la matriz original. Se corresponde con una paralelización de la rutina equivalente de la librería LAPACK y que todavía no dispone de versión paralela en ScaLAPACK.

Además, para no complicar innecesariamente el código paralelo de la rutina, se asume que ningún bloque 2×2 correspondiente a un par de valores propios complejos se encuentra repartido entre varios procesos. Todos estos bloques deberán estar completamente almacenados en un único proceso. Esta asunción resulta adecuada, puesto que se puede garantizar fácilmente sin más que realizar una pequeña reordenación previamente a la llamada a esta rutina, cosa que se ha implementado en la rutina `PDGGEV`, que es la que llama a esta.

El procedimiento utilizado en esta rutina sigue el método descrito en la sección 3.1.2 y que también es el usado en la correspondiente rutina secuencial de LAPACK [Dem97]. Pero ahora se ha paralelizado este método tratando de hacer en paralelo entre varios procesos el mayor número de operaciones.

El algoritmo paralelo implementado en la rutina es:

Algoritmo 3.7 `PDTREVC`

1. *Recorrer los bloques diagonales de la matriz en forma real de Schur T (su tamaño vendrá dado por el tamaño de bloque usado en la distribución de datos) desde abajo hacia arriba. Para cada bloque diagonal:*

a) *Difundir todo el bloque de columnas asociado a todos los procesos de la fila.*

b) *Calcular en paralelo las porciones de todos los vectores propios contenidas en la fila de procesos donde está el bloque diagonal actual (resolviendo localmente sistemas casi-triangulares con múltiples partes derechas).*

c) *Si no es el último bloque diagonal:*

1) *En la fila donde se acaban de calcular porciones de todos los vectores propios, enviar estas porciones a todos los procesos de la misma columna de procesos.*

2) *En paralelo, en todos los procesos, actualizar la parte derecha de los sistemas que luego se resolverán para calcular las siguientes porciones de todos los vectores propios.*

2. *Si se ha indicado que se desea transformar los vectores propios usando una matriz proporcionada externamente (por ejemplo para obtener los vectores propios de una matriz general), calcular el producto de los vectores propios de la matriz en forma de Schur por la matriz de transformación facilitada (rutina* `PDTRMM` *de PBLAS).*

3. *Normalizar los vectores propios.*

En la figura 3.3 se muestra gráficamente la porción del algoritmo encargada tan sólo del cálculo de los vectores propios de la matriz en forma real de Schur. Se muestra para el caso de una distribución no cíclica, con el propósito de que sea un esquema más sencillo e ilustrativo de las operaciones que se realizan. No obstante, debe entenderse que en la realidad se trabaja (en general) con una distribución cíclica. Cuando esto ocurre, las difusiones mostradas en la figura involucrarán a todos los procesos de la fila o columna en la que se hagan. Y los cálculos, tanto de las porciones de los vectores propios como la actualización posterior, afectarán a más procesos. Precisamente la distribución cíclica favorece una mejor utilización de todos los procesos equilibrando mejor la carga y logrando que estén más tiempo todos activos.

3.3. Problema práctico real: simulación del flujo oceánico

El cálculo de valores propios aparece en multitud de aplicaciones, estando hoy en día asociado a un elevado porcentaje del tiempo de cálculo en los centros de supercomputación. Es el núcleo computacional principal en algunas aplicaciones relativas a los grandes desafíos científicos. Encontramos ejemplos en muchas áreas, entre ellas la geofísica o ciencia de la tierra, y en particular en el estudio de la dinámica de los océanos (mareas, corrientes, etc.), que tiene influencia en el clima, a corto, medio y largo plazo (cambio climático).

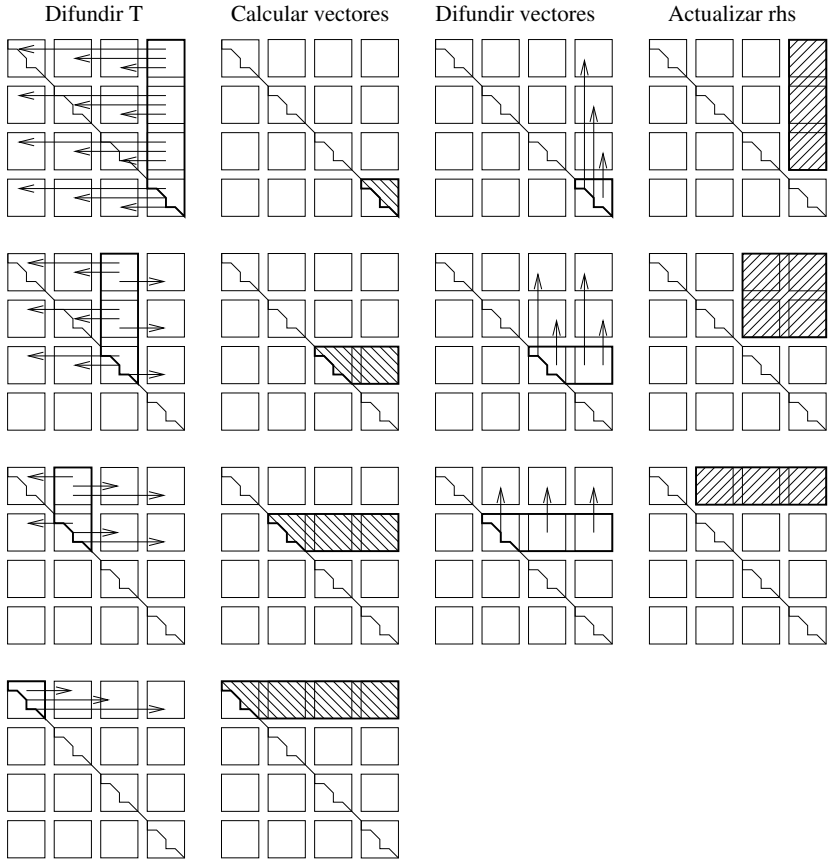

Figura 3.3: Esquema de operaciones del cálculo de vectores propios en `PDTREVC` con una distribución no cíclica

En el estudio de flujos oceánicos altamente turbulentos, el régimen no lineal del problema se puede mapear en el espectro de un problema linealizado alrededor de la zona de equilibrio. Resolver estos problemas permitirá obtener una aproximación que aporte luz al estudio de las dinámicas no lineales de los océanos. Para ello, es necesario encontrar el espectro completo del problema de valores propios asociado. Se trata de problemas no simétricos de valores propios generalizados de tamaños muy grandes. Es necesario por tanto una implementación de altas prestaciones que haga posible enfrentarse a estos problemas de gran tamaño.

Un grupo de investigación en el Imperial College de Londres se encuentra trabajando en estas simulaciones del flujo de los océanos y se ha colaborado con ellos para resolver el problema numérico necesario para estas simulaciones.

Este grupo investiga la variabilidad en baja frecuencia, *Low Frequency Variability* (LFV), de *la corriente del golfo*. Se trata de una corriente oceánica cálida que fluye

desde el golfo de Méjico en paralelo con la costa de Estados Unidos hacia Terranova (Canadá) y luego continúa a través del océano Atlántico hacia el noroeste de Europa, ya con otro nombre (*la deriva del Atlántico Norte*). Se trata de investigar los cambios en la corriente del golfo durante periodos de 10-20 años. En particular, no les interesa especialmente la variabilidad en baja frecuencia sino más bien los mecanismos físicos que la gobiernan. Hasta ahora, estos mecanismos suponen uno de los grandes enigmas en el campo de *dinámica de fluidos en Geofísica*.

La estrategia básica para analizar este problema se puede descomponer en tres etapas:

1. Se extraen los datos de variabilidad en baja frecuencia (LFV) con la ayuda del análisis de la función ortogonal empírica (EOF).

2. Se proyectan los modos de la EOF en la solución de un problema linealizado alrededor del flujo medio (aquí es donde aparece la necesidad de resolver un problema de valores propios).

3. Se extraen las soluciones que suponen mayor contribución a la LFV.

La variabilidad de baja frecuencia a gran escala de la circulación en los océanos en una escala temporal de años a decadas juega un papel muy importante en el cambio climático. En las últimas décadas se han invertido muchos esfuerzos en investigación para determinar cómo afectan los procesos físicos de la LFV en la circulación océanica guíada por el viento en latitudes medias. Estudios realizados sobre las corrientes más importantes, como la del golfo de Méjico o la de Kuroshio (una corriente cálida en el océano Pacífico que fluye al nordeste pasando por Japón en dirección a Alaska), revelan la LFV observada a gran escala [BM99, BHD07, KBD$^+$06, PDM14]. Sin embargo, los mecanismos físicos responsables de esto aún permanecen desconocidos o mal entendidos.

El problema de la LFV en latitudes medias del océano ha sido abordado desde diferentes frentes. El uso de la teoría de sistemas dinámicos ha permitido alcanzar algunos progresos. En particular, gracias al análisis de bifurcaciones sucesivas del modelo barotrópico casi-geotrófico (modelo QG) [SD02], se ha podido confirmar que los modos de giro oscilatorio muestran un comportamiento de baja frecuencia cuando se les compara con otros modos como los modos de cuenca de Rossby (*Rossby basin modes*), los modos de pared (*wall-trapped modes*) y los modos baroclínicos. Otra posibilidad es que la LFV se produzca por la aparición de órbitas homoclínicas, que pueden apreciarse en el modelo QG barotrópico [Mea00, CIGL01]. En [BM99] se ha aplicado el análisis de la función ortogonal empírica (EOF) al estudio de la LVF en modelos QG de 1.5 y 2 capas. En [KB15] se ha utilizado la aproximación estocástica para estudiar la LFV a gran escala cada década en el modelo QG de 3 capas.

El grupo de investigación de Londres está realizando una nueva aproximación para explorar la variabilidad de baja frecuencia del océano. La idea clave es extraer la LFV usando el método de la EOF y proyectar las principales EOFs sobre los modos de cuenca del modelo oceánico estudiado linealizado alrededor de un flujo de fondo no nulo. Entonces, se puede estimar la contribución de cada uno de estos

modos sobre la variabilidad de baja frecuencia mediante un análisis de la influencia de cada uno de ellos sobre la LFV. Esta aproximación es independiente del modelo, aunque la están usando para el modelo QG de 3 capas.

La parte numérica clave en todo este proceso es la resolución de un problema generalizado de valores propios. Se ha utilizado esta aplicación para analizar el comportamiento de los códigos desarrollados cuando se aplican a un problema de muy gran dimensión, tanto a nivel de precisión como de prestaciones paralelas.

En un principio se pretendía resolver el problema usando rutinas de la librería de altas prestaciones ScaLAPACK, pero ahora mismo no dispone de rutinas para el problema generalizado. Sí parece que estarán disponibles en el futuro, porque, aunque no se ha podido tener acceso a ellas, se están haciendo desarrollos para resolver este problema en paralelo mediante el algoritmo QZ [AKK14].

Dado que la matriz B del problema generalizado a resolver en este caso práctico es invertible, como ya se ha explicado en la sección 3.2.5 se ha optado por realizar una nueva rutina que resuelva el problema generalizado a base de transformarlo en un problema estándar. Con esta nueva rutina y las existentes en ScaLAPACK para resolver el problema de valores propios estándar, ya ha sido posible resolver la parte numérica de este problema.

Es importante hacer notar que, aunque en esta aplicación las matrices de datos iniciales son dispersas, como se requiere el cálculo de *todos* los valores y vectores propios, se ha resuelto el problema como denso.

Además, la aplicación necesita los vectores propios. No basta con los vectores de Schur, que es lo que proporcionan las rutinas de ScaLAPACK. Así que se ha desarrollado también una nueva rutina paralela para el cálculo de los vectores propios a partir de los vectores de Schur.

Gracias a todo este desarrollo se ha podido resolver el problema de simulación del flujo oceánico. En esta ocasión no se trataba de reducir el tiempo de ejecución necesario, sino de conseguir resolver el problema, que era demasiado grande para ser resuelto sin la ayuda de estas nuevas rutinas de computación de altas prestaciones. Se ha podido resolver un problema muy grande de valores propios generalizados donde las matrices involucradas han sido de tamaño 444675×444675.

3.4. Resultados experimentales

A continuación se muestran los resultados computacionales de algunas pruebas prácticas llevadas a cabo utilizando las nuevas rutinas desarrolladas. Primero se estudia el problema de reducción a forma real de Schur y posteriormente el proceso completo de resolución de un problema de valores propios generalizado, necesario en la simulación de flujos oceánicos.

Antes se describen las plataformas donde se han medido los índices de prestaciones mostrados.

3.4.1. Plataformas empleadas

Se han utilizado muchas plataformas diferentes que van desde simples ordenadores de sobremesa interconectados con una red ethernet hasta potentes supercomputadores dedicados exclusivamente al cálculo científico. Afortunadamente todo el trabajo realizado es portable gracias al uso de librerías portables, además de haber fomentado la portabilidad en el desarrollo de las nuevas rutinas.

Ya en las últimas pruebas se ha trabajado fundamentalmente en tres plataformas: *Kahan*, *Tirant* y *Archer*.

Cluster *Kahan*

El cluster *Kahan* es una plataforma utilizada en la docencia de asignaturas relacionadas con la computación de altas prestaciones en el Departamento de Sistemas Informáticos y Computación de la Universitat Politècnica de València.

Consta de 6 nodos biprocesador interconectados mediante una red infiniband. Cada nodo consta de:

- 2 procesadores AMD Opteron 16 Core 6272 a 2.1GHz.
- 32GB de memoria DDR3 1600.
- Disco duro de 500GB, SATA 6 GB/s.
- Controladora InfiniBand QDR 4X (40Gbps, tasa efectiva de 32Gbps).

En total son 12 procesadores con 192 núcleos de cómputo y un total de 192 GB de RAM (entre todos).

El disponer de un cluster tan cerca y con una apropiada facilidad de uso ha sido (y es) muy adecuado para facilitar la depuración y primeras ejecuciones de los nuevos códigos. Sin embargo, el número máximo de unidades de cómputo en *Kahan* (192) no resulta adecuado para realizar pruebas con muchos procesos. Este factor se ve agravado por el hecho de que el cluster no funciona especialmente bien al realizar comunicaciones de procesos dentro de un mismo nodo. Parece que es un problema del ancho de banda en accesos múltiples a memoria. Ahora mismo es más rápida la comunicación entre dos procesos ubicados en dos nodos diferentes de *Kahan* que si se tienen ubicados en un mismo nodo. Es probable que esto no ocurra en todos los casos, ya que posiblemente dependa del tamaño de los mensajes empleados. Pero sí ha ocurrido al usar los algoritmos necesarios en esta tesis. Por esto, trabajar con más de 6 procesos en *Kahan* (6 procesos aún pueden ubicarse cada uno en un nodo diferente) supone tener una merma importante en las prestaciones y que no es debida a la implementación realizada.

Por todo esto, este cluster se ha utilizado en pocas ocasiones para los resultados mostrados en esta tesis. Aunque ha sido una herramienta muy útil para el desarrollo y depuración de los códigos implementados.

Cluster *Tirant*

En marzo de 2007 se inauguró la *red española de supercomputación*. Esto propició la actualización del supercomputador *MareNostrum*. MareNostrum es el computador más potente de España, ubicado en Barcelona. En su actualización, se cambiaron los blades JS20 por blades JS21, todos ellos de IBM. Los blades sustituidos se utilizaron para crear una estructura distribuida de supercomputadores en diferentes emplazamientos del territorio español. Parte de estos blades se utilizaron para constituir el supercomputador de la Universitat de València conocido como *Tirant*, sobre el que después se han realizado actualizaciones.

Tirant es un supercomputador de memoria distribuida formado por 512 ordenadores blade JS21 de IBM. Cada uno consta de 2 procesadores PowerPC 970MP dual core de 64 bits a 2.2 GHz con 4 GB de memoria RAM (2 TByte en total). Los diferentes ordenadores están interconectados mediante una red Myrinet con un ancho de banda de 2+2 GBytes por segundo. El supercomputador tiene accceso a 44.9 TB de disco duro, exportado al cluster mediante un sistema de ficheros compartido (*General Parallel File System*).

Este es el cluster que se ha usado en la mayoría de los resultados mostrados en esta tesis.

Cuando se ha ejecutado con más de un proceso, el sistema de colas va asignando cada 4 procesos a un mismo nodo, de forma que se aprovechan las 4 unidades de cálculo que dispone cada nodo. Por ejemplo, cuando se ha lanzado en 7 procesos, el sistema ha utilizado 2 nodos: 1 nodo con 4 procesos y otro con 3 procesos. En esta ocasión no se han apreciado diferencias significativas entre las comunicaciones realizadas entre procesos asignados a un mismo nodo y procesos asignados a diferentes nodos.

Cluster *Archer*

En la cooperación con el Imperial College de Londres para resolver el problema de valores propios generalizado necesario para las simulaciones de los flujos oceánicos, se han utilizado varios de sus clusters dedicados a la computación de altas prestaciones. Si bien, el acceso a estos clusters ha sido casi siempre de forma indirecta. Normalmente el grupo de investigación con el que se ha colaborado y que es residente allí ha sido quien ha utilizado directamente estos clusters. Tan sólo cuando ha habido algún problema grave y que no era reproducible en otras plataformas hemos tenido acceso a sus clusters para solventar el problema.

Esta es la razón por la que los índices de prestaciones mostrados para esta plataforma son más escasos y menos concretos que para el resto. Se trata de resultados que nos han proporcionado desde Londres, pero que no hemos generado directamente nosotros (aunque se han obtenido con nuestro software).

Aunque se han utilizado múltiples clusters de esta universidad, los resultados para los problemas más grandes (que no hubieran podido resolverse sin ayuda de nuestro trabajo) se han obtenido en el cluster llamado *Archer* (http://www.archer.ac.uk/). *Archer* es un entorno de computación de altas prestaciones desarrollado alrededor de un supercomputador Cray XC30 con 4920 nodos. Cada nodo de cálculo en *Archer*

consta de dos procesadores a 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge). Se distingue entre nodos estándar, que tienen 64 GB de memoria compartida entre los dos procesadores y nodos con más memoria, que tienen 128 GB de memoria para los dos procesadores. *Archer* consta de 4544 nodos estándar (12 grupos, 109056 núcleos) y 376 nodos con más memoria (1 grupo, 9024 núcleos) lo que hace un total de 4920 nodos (13 grupos, 118080 núcleos).

Los diferentes nodos están interconectados usando el sistema *Cray Aries Interconnect*, que utiliza una topología de libélula (*Dragonfly*). Con esta topología, cada 4 nodos están conectados a un router Aries, cada 188 nodos se agrupan en un conjunto y cada 2 conjuntos forman lo que se denomina un grupo. La interconexión utilizada es una red de cobre de conexiones 2D de todos a todos entre los nodos de un mismo grupo, mientras que diferentes grupos se interconectan entre sí mediante una red óptica.

3.4.2. Reducción de modelos

En el capítulo 5 se hace más énfasis en el problema de la reducción de modelos de sistemas lineales de control y las operaciones necesarias para su resolución. Allí se muestran resultados experimentales con vistas a comprobar la correcta precisión de los resultados y también con vistas a ver que las nuevas rutinas van a permitir trabajar con problemas de gran tamaño. En el proceso completo de la reducción de modelos aparece la necesidad de transformar el sistema a uno en el que la matriz de estados se encuentre en forma real de Schur. Aquí es donde se utiliza la nueva rutina `PDGEES` para obtener la forma real de Schur de una matriz y luego aplicar las transformaciones empleadas al resto de matrices que representan el sistema. En este apartado se muestran los resultados obtenidos con esta rutina, ya que este es el capítulo dedicado al problema de valores propios. Pero es después donde se apreciará en su conjunto todo el proceso de reducción de modelos.

Se han generado sistemas lineales de control sintéticos de un tamaño tal que su procesamiento tenga un coste razonable, que permita observar la evolución con diferente número de procesos. Se utilizan matrices de tamaño 3000×3000 y con un tamaño de bloque para la distribución de datos de 32. En la sección 5.4 pueden consultarse tanto las características de las matrices generadas, como el breve estudio realizado para elegir el tamaño de bloque más adecuado a utilizar en este cluster para estas operaciones.

Como se ilustra en la sección 3.4.3, este tamaño de problema no es ni mucho menos grande para lo que se puede resolver con las nuevas rutinas de valores propios. Simplemente es el tamaño utilizado para estas rutinas dentro de su uso en el problema de reducción de modelos, donde el tiempo total del proceso ya es considerable al tener que sumarse muchas otras operaciones.

En la figura 3.4 se muestran los tiempos de ejecución (en *Tirant*) de la rutina `PDGEES` para estos problemas de reducción de modelos. Se muestra tanto el tiempo que necesita la primera versión de la rutina `PDGEES` (marcada como "antigua" en la gráfica) como el tiempo usado por la nueva versión. Recuérdese que la primera versión implementada de esta rutina utilizaba la rutina antigua de ScaLAPACK para

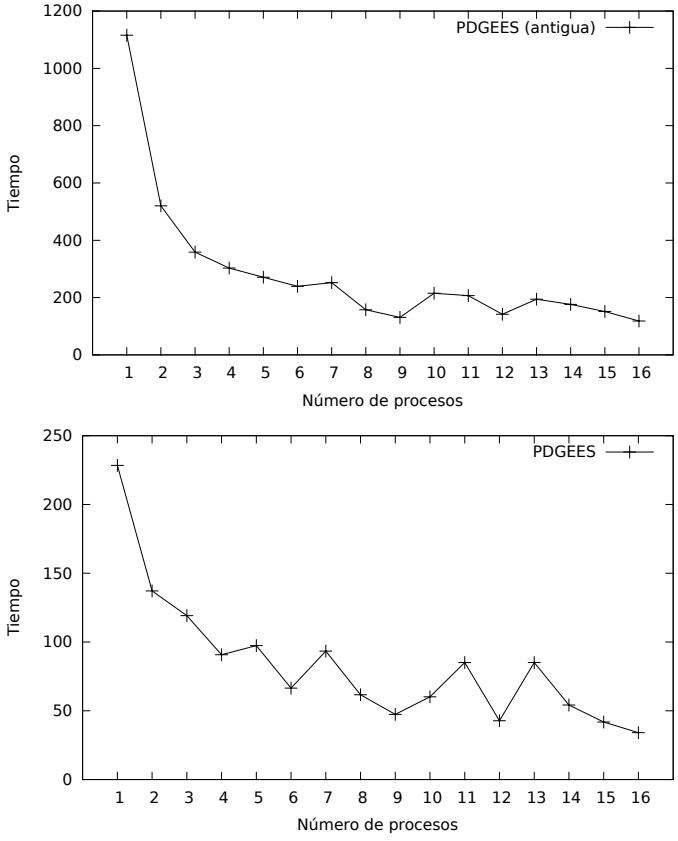

Figura 3.4: Tiempo de ejecución paralelo en segundos variando el número de procesos entre 1 y 16 para la rutina `PDGEES`. Se muestran resultados de las rutinas antiguas (parte superior) y las nuevas (parte inferior)

este problema: `PDLAHQR`, mientras que la nueva versión está basada en las nuevas implementaciones disponibles en la última versión de la librería y que están basadas en el uso de `PDHSEQR`. En esta rutina se implementan las nuevas optimizaciones de altas prestaciones para el problema de factorización de Schur [GKK10].

A simple vista las diferencias de tiempo de ejecución son muy notables. La nueva versión requiere un tiempo de ejecución mucho menor incluso cuando es ejecutada en un solo procesador.

Para cada número de procesos se han ejecutado todas las posibles combinaciones de mallas de procesos con esa cantidad y en la gráfica se muestra el mejor resultado. Por ejemplo, para 4 procesos se ha ejecutado en 1×4, 2×2 y 4×1.

Llama la atención que en la nueva versión el tiempo empeora especialmente al pasar a ejecutar en 5, 7, 11 o 13 procesos. Nótese que esto es un número primo

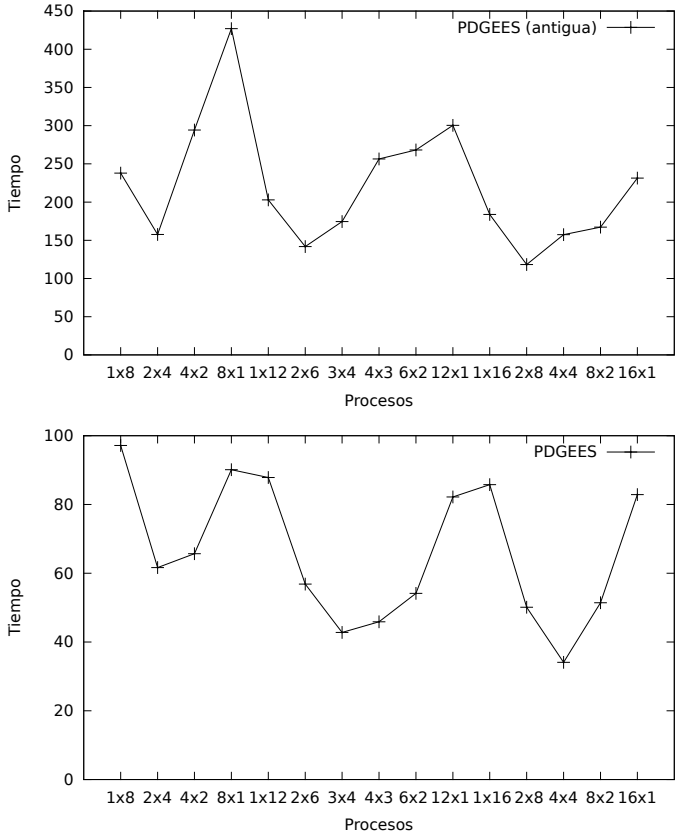

Figura 3.5: Tiempo de ejecución en segundos para 8, 12 y 16 procesos con malla de procesos variable para la rutina `PDGEES`. Se muestran resultados de las rutinas antiguas (parte superior) y las nuevas (parte inferior)

de procesos que no permite mallas más o menos cuadradas. En estas cantidades de procesos sólo hay dos opciones: una malla con todos los procesos en horizontal o una malla con todos los procesos en vertical. El hecho de que los tiempos empeoren en estos casos hace pensar que la rutina funciona mejor con mallas más cuadradas. Esto puede comprobarse en la figura 3.5. En esta figura se muestran los mismos tiempos de ejecución (sólo para un subconjunto del número de procesos mostrado anteriormente), pero para todas las configuraciones de malla de procesos posible para cada número de procesos. Como se intuía de la otra figura, la nueva versión funciona mejor con mallas de procesos cuadradas. La versión antigua funciona mejor con mallas un poco rectangulares, ni totalmente horizontal o vertical ni estrictamente cuadrada.

En la tabla 3.1 se muestran los speed-ups y eficiencias para ambas rutinas. Se ha

utilizado como tiempo secuencial el tiempo de ejecutar la rutina paralela en un solo proceso.

antigua	1	2	4	8	12	16
Speed-up	1.00	2.15	3.68	7.08	7.87	9.44
Eficiencia	100.00	107.26	92.05	88.52	65.61	59.01
nueva	1	2	4	8	12	16
Speed-up	1.00	1.67	2.52	3.70	5.34	6.70
Eficiencia	100.00	83.27	62.89	46.30	44.48	41.87

Tabla 3.1: Speed-ups y eficiencias para la rutina `PDGEES`

Atendiendo a esta tabla, se podría decir que la versión antigua tiene mejores prestaciones paralelas. Y es cierto que tiene mejores speed-ups y eficiencias, pero esto es así porque esa versión tiene un coste en un proceso muy elevado y se ha utilizado este para calcular los índices de prestaciones. La nueva versión necesita mucho menos tiempo para resolver el mismo problema en un solo proceso. Si se calcularan los speed-ups y eficiencias utilizando como tiempo secuencial el de la nueva versión, saldrían valores mucho peores de speed-ups y eficiencias.

Viendo las gráficas de tiempos (figura 3.4) es obvio que la nueva versión es mucho mejor. No gana tanto como la antigua al incrementar el número de procesos, pero atendiendo al tiempo total necesario para resolver el problema, este es mucho menor ahora.

Por ejemplo, aunque el speed-up de la nueva versión ejecutada en 16 procesos comparada con ella misma en un proceso es de 6,7, si se compara con el tiempo que se tenía en un proceso con la versión anterior, supone una ganancia de velocidad de 32,74. Esto significa que usando la nueva versión en 16 procesos se consigue resolver este problema 32 veces más rápido que con la antigua versión en un solo proceso, lo que supone una buena ganancia en tiempo de ejecución. Si bien, no todo es debido a su ejecución en paralelo. La nueva versión ejecutada en un solo proceso ya es 5 veces más rápida que la antigua versión.

3.4.3. Simulación del flujo oceánico

En el problema de simulación de los flujos oceánicos a resolver, el primer paso es la obtención de los valores y vectores propios generalizados de un par de matrices muy grandes. Aquí se muestran los resultados experimentales de esta primera fase del proceso.

Las matrices de entrada al problema son matrices dispersas obtenidas mediante discretización del problema real. Para tener una precisión suficientemente buena del cómputo global, los científicos requieren mallas de discretización cada vez más finas, lo que aumenta considerablemente el tamaño de estas matrices. Sin embargo, para el estudio de selección del tamaño de bloque más adecuado se han utilizado mallas gruesas en las que el tiempo de ejecución es más modesto.

Procesos	Bloque	n	Tiempo
32=8×4	256	12678	1606 s
64=8×8	256	12678	730 s
144=12×12	256	12678	352 s
144=12×12	512	12678	473 s
289=17×17	256	12678	456 s
144=12×12	64	12678	96 s
144=12×12	100	12678	86 s
144=12×12	128	12678	96 s
144=12×12	256	12678	108 s
144=12×12	512	12678	137 s
289=17×17	64	12678	78 s
289=17×17	100	12678	67 s
289=17×17	128	12678	89 s
64=8×8	256	49926	23 h
144=12×12	256	49926	11.6 h
289=17×17	256	49926	6.9 h
289=17×17	512	49926	11.9 h
144=12×12	64	49926	2242 s
144=12×12	100	49926	2115 s
144=12×12	128	49926	2163 s
289=17×17	100	49926	1272 s
576=24×24	64	49926	868 s
576=24×24	100	49926	757 s
576=24×24	128	49926	856 s
576=24×24	100	100000	1.3 h
576=24×24	100	200000	16.3 h
1156=34×34	100	200000	11.8 h
576=24×24	100	198147	12 h
960=30×32	100	309123	29.5 h
2704=52×52	100	444675	30 h

Tabla 3.2: Tiempos de ejecución paralelos del cálculo de valores y vectores propios generalizados

En la tabla 3.2 pueden verse los tiempos de ejecución de resolver en paralelo múltiples problemas de valores propios generalizados de diferentes tamaños y en diferentes configuraciones.

En la parte superior de la tabla se muestran resultados de múltiples casos de prueba para determinar valores adecuados de algunos parámetros, incluyendo el tamaño de bloque, a utilizar en los problemas más grandes.

Aunque no se dispone de muchos datos, puede apreciarse por ejemplo como para el tamaño 49926 se obtienen mejores tiempos usando como tamaño de bloque el valor 100. El tiempo de resolver este problema en 144 procesos es de 2115 segundos, que consiguen reducirse hasta 757 segundos al ejecutar en 576 procesos. Multiplicando por un factor de 4 el número de procesos, el speedup se ha multiplicado por un factor de 2115/757=2.79, que es mejor que el factor conseguido en la tabla 3.1 al pasar de 1 a 4 procesos.

En la parte inferior de la tabla (las tres últimas filas) se muestran los tiempos necesarios para resolver tres problemas especialmente grandes. Se trata de problemas que no habían podido ser resueltos hasta ahora. Su enorme tamaño (matrices de órdenes 198147, 309123 y 444675) hacía inviable su resolución mediante algoritmos secuenciales. Aún así, debe notarse que se ha necesitado un alto poder computacional. En el caso del problema más grande, han sido necesarios $2704 = 52 \times 52$ procesos durante 30 horas para ser capaces de calcular los valores y vectores propios generalizados de un par de matrices A y B de tamaños 444675×444675.

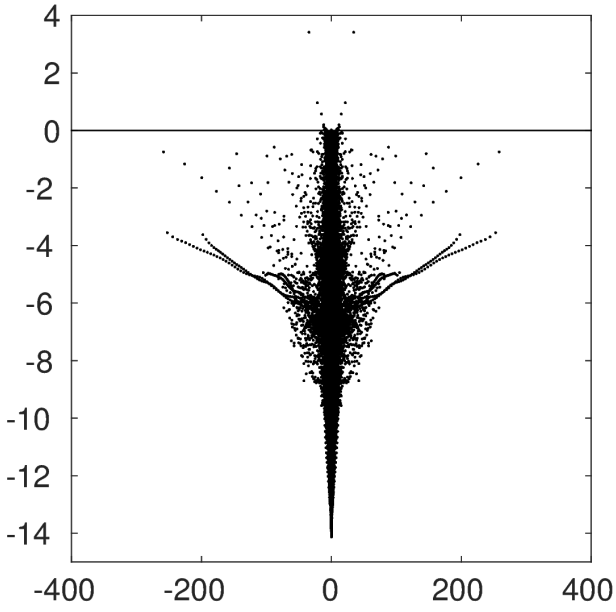

Figura 3.6: Espectro del problema de valores propios de tamaño 198147

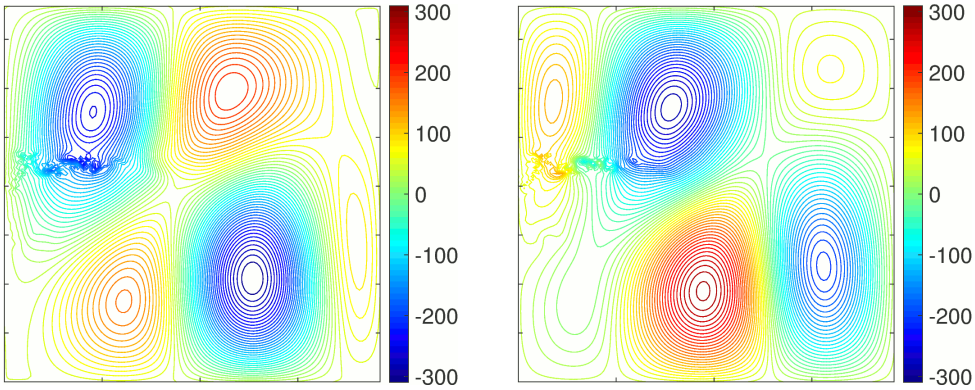

Figura 3.7: Uno de los vectores propios más importantes para la LFV en el problema de tamaño 49926 (parte real a la izquierda, parte imaginaria a la derecha)

A modo de ejemplo, se muestra en la figura 3.6 el espectro de los valores propios obtenidos para el problema de orden 198147. Nótese que en el eje x se representa la parte imaginaria de los valores propios, mientras que la parte real está sobre el eje y.

En la figura 3.7 se representa la parte real (a la izquierda) y la parte imaginaria (a la derecha) de uno de los vectores propios más importantes desde el punto de vista de su implicación sobre la variabilidad en baja frecuencia (LFV) del flujo oceánico, para el problema de tamaño 49926.

3.5. Conclusiones

Se ha trabajado en el desarrollo (y reparación) de rutinas que permitan calcular en paralelo valores y vectores propios tanto del problema estándar como del problema generalizado.

Para el problema estándar se ha implementado una rutina driver que permita obtener con facilidad la transformación a forma real de Schur de una matriz (esto es parte del problema estándar de valores propios). En esta rutina se utilizan las rutinas existentes en ScaLAPACK para resolver este problema, aunque han hecho falta algunas rutinas auxiliares, por ejemplo para pasar a forma estándar los bloques 2×2 de la matriz en forma real de Schur.

La primera versión de esta rutina driver usaba una transformación de forma de Hessenberg a forma real de Schur algo antigua que no ofrece muy buenas prestaciones. Por ello, se ha adaptado la rutina para utilizar nuevas versiones de las rutinas para realizar este paso, aunque ha habido que solventar algunos pequeños errores presentes en las nuevas rutinas.

Gracias a esto se dispone de una función para obtener la transformación a forma real de Schur que resulta mucho más eficiente que la versión anterior. Esto permite

trabajar con problemas más grandes, lo que permitirá reducir sistemas de control de mayor tamaño en las rutinas de reducción de modelos en que se utiliza esta operación.

Aparte de su uso en el campo de la reducción de sistemas lineales de control, estas rutinas permiten obtener la transformación a forma real de Schur en cualquier otro ámbito de aplicación. En particular, en cooperación con el Imperial College de Londres se ha trabajado en un problema de simulación de los flujos oceánicos. Es parte indispensable en estas simulaciones el resolver grandes problemas de valores propios generalizados.

Al tratarse de problemas de valores propios *generalizados*, no se puede usar directamente la nueva rutina, pero dadas las características del problema a resolver, sí que es factible su transformación a problema estándar y su resolución posterior usando el trabajo desarrollado. Se ha implementado una nueva rutina que resuelva el problema completo de valores propios generalizados mediante esta transformación.

Ha hecho falta completar la resolución del problema estándar de valores propios para obtener también los vectores propios. Se ha desarrollado una nueva rutina de altas prestaciones para obtener los vectores propios a partir de los vectores de Schur.

Además de las rutinas encargadas de resolver problemas numéricos, también se han desarrollado múltiples rutinas auxiliares para el trabajo con enormes matrices de datos, que son las que aparecen en este problema.

Gracias a todo este desarrollo se ha conseguido obtener los valores y vectores propios de problemas de gran tamaño, que de otro modo no hubieran podido obtenerse. Esto ha podido utilizarse, por ejemplo, para resolver problemas con matrices del orden de 444675 filas y columnas.

Con estas nuevas rutinas ya se tiene resuelto el paso de obtención de los valores y vectores propios y se ha podido continuar con el trabajo de estudio de los flujos oceánicos.

Capítulo 4

Resolución numérica de ecuaciones de Lyapunov

Este capítulo está dedicado a la resolución de la ecuación de Lyapunov en paralelo, con vistas a su utilización en los métodos de reducción de modelos que se explicarán en el siguiente capítulo.

En primer lugar se presenta la ecuación de Lyapunov, junto a los métodos clásicos para resolverla, haciendo mayor hincapié en los que se han paralelizado.

Después se describen las rutinas de altas prestaciones que se han implementado para resolver esta ecuación en paralelo. Se presentan tanto las rutinas principales a cargo de la solución de la ecuación de Lyapunov como las rutinas auxiliares utilizadas por estas.

Por último, se muestran algunos resultados de los experimentos llevados a cabo para comprobar tanto el buen funcionamiento de las nuevas rutinas como sus buenas prestaciones.

4.1. La ecuación de Lyapunov

La ecuación de Lyapunov es una ecuación matricial lineal de la cual existen múltiples versiones (continua y discreta y, para cada una de éstas, versiones estándar y generalizada). En los métodos de reducción de modelos, objetivo principal de esta tesis, tienen cabida todas ellas. Este trabajo va orientado a la resolución de la versión estándar de la ecuación, aunque se va a utilizar el caso generalizado de la ecuación para explicar los conceptos teóricos necesarios y los métodos habituales de resolución. Puede consultarse [GHRV97b] para ver una aproximación simplificada de resolución del problema generalizado en paralelo.

Además, para los algoritmos de reducción de modelos que se van a paralelizar, lo que se necesita es el factor de Cholesky de la solución de la ecuación de Lyapunov. Por esto se prestará especial atención al método de Hammarling para la resolución de la ecuación de Lyapunov, ya que en este método se obtiene el factor de Cholesky

de la solución directamente y sin necesidad de realizar una factorización posterior, como se explica más adelante.

A continuación se empieza describiendo la ecuación en su forma generalizada (más general) y estándar, para luego comentar los métodos habituales de resolución de este tipo de ecuaciones, poniendo un mayor énfasis en el método de Hammarling, que es el que resulta más adecuado para el objetivo de la tesis.

4.1.1. Ecuación de Lyapunov generalizada

Existen dos formas de la ecuación de Lyapunov generalizada, correspondientes a su versión para tiempo discreto y su versión para tiempo continuo. La Ecuación de Lyapunov Generalizada en tiempo Continuo (ELGC) tiene la forma

$$A^T X E + E^T X A = -Y, \tag{4.1}$$

y la Ecuación de Lyapunov Generalizada en tiempo Discreto (ELGD) (ecuación de Stein generalizada)

$$A^T X A - E^T X E = -Y. \tag{4.2}$$

En ambas ecuaciones, las matrices A, E e Y son matrices reales $n \times n$, siendo además Y simétrica, al igual que la solución, X, cuando es única.

Las ecuaciones de Lyapunov pueden verse como casos particulares de la ecuación generalizada de Sylvester

$$R^T X S + U^T X V = -Y. \tag{4.3}$$

En esta ecuación X e Y son en general matrices $n \times m$. La existencia y unicidad de solución de (4.3) (y por tanto de las ecuaciones de Lyapunov) viene determinada por la estructura de los valores propios generalizados de los pares de matrices (R,U) y (V,S).

El siguiente teorema (teorema 1 de [Chu87]) da condiciones necesarias y suficientes para la existencia y unicidad de la solución de la ecuación (4.3).

Teorema 4.1

La ecuación matricial (4.3) tiene solución y esta solución es única si y sólo si

1. $\alpha R - \beta U$ *y* $\alpha V - \beta S$ *son haces regulares y*
2. $\lambda(R, U) \cap \lambda(-V, S) = \emptyset$.

Si se aplica este teorema a las ecuaciones de Lyapunov (4.1) y (4.2), se obtiene el siguiente corolario.

Corolario 4.1

Sea $A - \lambda E$ *un haz regular. Entonces:*

1. *La ELGC (4.1) tiene solución y esta solución es única, si y sólo si todos los valores propios del haz* $A - \lambda E$ *son finitos y además* $\lambda_i + \lambda_j \neq 0$ *para cualquier par de valores propios* λ_i, λ_j *de* $A - \lambda E$.

2. *La ELGD (4.2) tiene solución y esta solución es única, si y sólo si* $\lambda_i\lambda_j \neq 1$ *para cualquier par de valores propios* λ_i, λ_j *de* $A - \lambda E$ *(bajo la convención* $0 \times \infty = 1$*).*

Como consecuencia de este corolario, se observa que en la ELGC la singularidad de una de las matrices A o E implica la singularidad de la ecuación (la no existencia de una solución única). Si A es singular, aparecerá el cero entre los valores propios generalizados del haz y no cumplirá el punto 1 del corolario anterior, ya que existen dos valores propios, el $\lambda_i = 0$ y $\lambda_j = 0$ $(= \lambda_i)$, que cumplen $\lambda_i + \lambda_j = 0$. Si E es singular, entre los valores propios generalizados de (A, E) se encontrará el infinito, por lo que nuevamente no se cumple el punto 1 del corolario ya que hay un valor propio generalizado que no es finito.

En el caso de la ELGD, puede concluirse de forma semejante que la ecuación será singular cuando ambas matrices A y E sean singulares, ya que en este caso aparecerán tanto el cero como el infinito entre los valores propios generalizados del haz y por tanto, no se cumple el punto 2 del corolario, existen dos valores propios, $\lambda_i = 0$ y $\lambda_j = \infty$, que cumplen $\lambda_i\lambda_j = 1$.

Así, centrándose en el estudio de las ecuaciones cuando tienen solución única, puede suponerse en la ELGC que ambas matrices son invertibles, y en la ELGD que al menos una de las matrices A o E es invertible. Además, dado el carácter simétrico de ambas ecuaciones (salvo un signo en la ELGD), puede suponerse que la matriz E es invertible, para obtener las siguientes ecuaciones, equivalentes a (4.1) y (4.2)

$$(AE^{-1})^T X + X(AE^{-1}) = -E^{-T} Y E^{-1} \tag{4.4}$$

$$(AE^{-1})^T X (AE^{-1}) - X = -E^{-T} Y E^{-1} \tag{4.5}$$

Estas ecuaciones son las ecuaciones estándar de Lyapunov, como luego se verá, aunque presenten una forma un tanto extraña. La ecuación (4.4) es la ecuación estándar de Lyapunov en tiempo continuo y la ecuación (4.5) es la ecuación estándar de Lyapunov en tiempo discreto.

Vistas las ecuaciones (4.1) y (4.2) en la forma (4.4) y (4.5), puede extenderse a las ecuaciones de Lyapunov generalizadas el resultado clásico sobre la solución definida positiva de la ecuación de Lyapunov [SZ70], en la forma del siguiente teorema (teorema 2 de [Pen98]):

Teorema 4.2

Sea la matriz E *no singular y la matriz* Y *(semi)definida positiva.*

1. *Si* $\mathrm{Re}(\lambda_i) < 0$ *para todos los valores propios generalizados* λ_i *del haz* $A - \lambda E$*, entonces la matriz solución* X *de la ELGC (4.1) es (semi)definida positiva.*

2. *Si* $|\lambda_i| < 1$ *para todos los valores propios generalizados* λ_i *del haz* $A - \lambda E$*, entonces la matriz solución* X *de la ELGD (4.2) es (semi)definida positiva.*

Puede haber una solución definida en el caso discreto, incluso si E es singular. En ese caso, si $|\lambda_i| < 1$ para todos los valores propios generalizados λ_i del haz $E - \lambda A$, entonces la matriz solución X es (semi)definida negativa.

Las ecuaciones de Lyapunov que cumplen las condiciones dadas por este teorema son conocidas como *ecuaciones de Lyapunov estables*.

Definición 4.1 *Las ecuaciones de Lyapunov 4.1 y 4.2 se dice que son* estables *cuando cumplen que la matriz E es no singular, la matriz Y es semidefinida positiva y*

- $\mathrm{Re}(\lambda_i) < 0$ *para todos los valores propios generalizados λ_i del haz $A - \lambda E$, en el caso de la ecuación en tiempo continuo.*
- $|\lambda_i| < 1$ *para todos los valores propios generalizados λ_i del haz $A - \lambda E$, en el caso de la ecuación en tiempo discreto.*

En las ecuaciones de Lyapunov en forma estándar, la matriz E es la identidad y estas propiedades de estabilidad se traducen directamente en propiedades a cumplir por los valores propios de la matriz A. En estos casos, se dice que *la matriz A es estable* cuando sus valores propios cumplen lo requerido para el caso de la ecuación continua (parte real negativa) o *convergente* cuando cumplen lo necesario para la ecuación discreta (módulo menor que uno).

4.1.2. Ecuación de Lyapunov estándar

La forma que tiene la ecuación estándar de Lyapunov en tiempo continuo es

$$A^T X + X A = -Y. \tag{4.6}$$

Y la ecuación estándar de Lyapunov en tiempo discreto es

$$A^T X A - X = -Y. \tag{4.7}$$

Obsérvese que estas ecuaciones se corresponden con las ecuaciones generalizadas (4.1) y (4.2), en el caso en que la matriz E sea la identidad. De hecho, todas las propiedades sobre existencia, unicidad y propiedades de la solución vistas para la ecuación generalizada son igualmente válidas para la ecuación estándar, asumiendo en ellas que la matriz E es la identidad.

Por ejemplo, la ecuación de Lyapunov estándar en tiempo continuo tiene solución y esta solución es única si y sólo si para cualesquiera valores propios (estándar) λ_i, λ_j de la matriz A, se cumple que $\lambda_i + \lambda_j \neq 0$. Además, la solución será (semi)definida positiva si también lo es la matriz Y y los valores propios de A son todos negativos. Nótese que si esto se cumple, se está cumpliendo la propiedad de existencia y unicidad de la solución. Con la generalizada también pasaba.

Por otro lado, la ecuación de Lyapunov estándar en tiempo discreto tiene solución y la solución es única si y sólo si para cualesquiera valores propios λ_i, λ_j de la matriz A, se cumple siempre que $\lambda_i \lambda_j \neq 1$ (con la convención de que $0 \times \infty = 1$). La

solución será (semi)definida positiva si lo es Y y los valores propios de A tienen todos módulo menor que uno. Al igual que pasa con el criterio para la ecuación continua, aquí también es cierto que si se cumple este criterio se cumple el de existencia y unicidad de la solución. Y también ocurría en la ecuación generalizada.

Es importante hacer notar que estas condiciones de existencia, unicidad y la propiedad de que la solución sea (semi)definida positiva se van a cumplir en las ecuaciones de Lyapunov que se resolverán en el próximo capítulo para la reducción de modelos:

- La matriz Y va a ser (semi)definida positiva puesto que se obtendrá como producto de una matriz por su transpuesta.
- Y la matriz A (que allí se conocerá como la *matriz de estado* del sistema) tendrá todos sus valores propios con parte real negativa en el caso continuo y con módulo menor que uno el caso discreto, puesto que se van a utilizar sistemas lineales de control estables.

4.1.3. Resolución de ecuaciones de Lyapunov

Una vez que se han presentado las ecuaciones de Lyapunov y algunas de sus propiedades, se mencionan a continuación algunos métodos clásicos para resolverlas.

Se van a describir con detalle los métodos dirigidos a matrices densas, en especial el método de Hammarling, que es el que se paralelizará posteriormente por resultar más adecuado para los métodos de reducción de modelos que se explicarán en el próximo capítulo.

La descripción se efectúa sobre la ecuación generalizada, por ser más general. Para el caso de la ecuación estándar, simplemente hay que recordar que es la forma generalizada con una matriz E igual a la identidad. Esto hará desaparecer los productos por E o por E_{ii}, que dejarían inalterable el resultado al ser multiplicaciones por la identidad, y anulará los factores en que aparezca E_{ij} con $i \neq j$, que serían matrices nulas.

En el caso de la transformación a forma real de Schur, habitual en estos métodos para simplificar la ecuación original, en lugar de aplicar el método iterativo QZ, para el caso estándar se aplicaría el método iterativo QR. Lo que después es la matriz Z resultaría igual a la matriz Q y la matriz E_s proveniente de triangularizar la matriz E sería la identidad, cumpliéndose lo que se acaba de explicar para E también para E_s.

En definitiva, los métodos descritos son directamente aplicables a la ecuación estándar, para la cual resultarán más sencillos, al simplificar las operaciones en que aparece la matriz E.

Métodos teóricos

Antes que nada, se presentan métodos de resolución que, aunque teóricamente sean correctos, no suelen ser aplicables en la práctica.

El primer método, que podría ser considerado como hacer uso de la 'fuerza bruta', es aplicable, en general, a cualquier ecuación matricial que sea lineal en las entradas de la matriz X solución. Consiste en desarrollar el sistema de ecuaciones lineales embebido en la ecuación matricial.

Por ejemplo, para el caso de la ecuación de Sylvester generalizada (4.3), se obtendría el siguiente sistema de ecuaciones lineales equivalente

$$(S^T \otimes R^T + V^T \otimes U^T)vec(X) = -vec(Y) \tag{4.8}$$

donde $\otimes$ representa el producto de Kronecker de dos matrices [LT85] y $vec(X)$ representa el vector columna asociado a una matriz obtenido al apilar las columnas de X, es decir

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix},$$

$$vec(X) = (x_{11}, \dots, x_{n1}, x_{12}, \dots, x_{n2}, \dots, x_{1m}, \dots, x_{nm})^T.$$

El problema por el que no es aplicable este método es que para una ecuación de Sylvester con matrices X e Y de $n \times m$ se obtiene un sistema de ecuaciones lineales con $n \times m$ ecuaciones e incógnitas, lo que resulta inabordable en la práctica excepto para valores muy pequeños de n y m.

Para el caso de las ecuaciones de Lyapunov generalizadas (4.1),(4.2), otro método que no suele utilizarse se basa en su paso previo a forma estándar (4.4),(4.5), mediante la inversión de la matriz E. Se realizaría la transformación indicada para así transformar las ecuaciones generalizadas a forma estándar (4.4),(4.5), y posteriormente poder resolver estas ecuaciones estándar de Lyapunov. Sin embargo, como se ha visto, esta transformación supondría calcular la inversa de la matriz E, que aunque invertible, en algunos casos podría estar mal condicionada evitando así la posibilidad de un cálculo fiable de su inversa. Las expresiones (4.4) y (4.5) son útiles para su aplicación teórica pero no suelen usarse en la práctica, salvo en casos muy concretos donde se conozcan las buenas propiedades de la matriz E.

Método de Bartels-Stewart

El método de Bartels-Stewart [BS72] se utiliza para la resolución de la ecuación estándar de Sylvester $AX + XB = C$. Sin embargo, puede generalizarse para la resolución de la ecuación de Lyapunov generalizada (4.1), (4.2). A continuación se explica el método aplicado a la ecuación continua (4.1). Para el caso discreto, se haría de forma similar.

La forma de trabajar es la descrita anteriormente. Primero se transforma la ecuación a una forma 'reducida'. Después, se resuelve ésta. Y por último, se realiza una transformación hacia atrás sobre la solución de la ecuación reducida, para obtener la solución de la ecuación original.

Transformación a forma reducida

El haz de matrices $A-\lambda E$ se reduce a la forma real de Schur generalizada $A_s-\lambda E_s$ utilizando el algoritmo iterativo QZ (ver sección 3.1.3). A partir de las matrices A y E, matrices generales, se obtendrá una matriz A_s, en forma casi-triangular superior, otra matriz E_s, triangular superior, y dos matrices ortogonales Q,Z que cumplen

$$\begin{aligned} A_s &= Q^T A Z, \\ E_s &= Q^T E Z. \end{aligned} \tag{4.9}$$

Como se comentó en la sección 3.1.1, el que la matriz A_s sea casi-triangular superior quiere decir que es una matriz triangular superior con la excepción de que su diagonal está formada por bloques 1×1 ó 2×2, que se corresponden con valores propios reales o pares de valores propios complejos conjugados, respectivamente, del haz $A - \lambda E$. Por tanto, en el caso de que aparezca en la diagonal algún bloque 2×2, habrá algún elemento no nulo por debajo de la diagonal. Véase un ejemplo de este tipo de matrices:

$$\begin{pmatrix} \boxed{a_{11}} & a_{12} & a_{13} & a_{14} & a_{15} \\ 0 & \boxed{a_{22}} & \boxed{a_{23}} & a_{24} & a_{25} \\ 0 & \boxed{a_{32}} & \boxed{a_{33}} & a_{34} & a_{35} \\ 0 & 0 & 0 & \boxed{a_{44}} & a_{45} \\ 0 & 0 & 0 & 0 & \boxed{a_{55}} \end{pmatrix}. \tag{4.10}$$

Con las matrices Q y Z provenientes de la reducción a la forma real de Schur (4.9), pueden definirse las siguientes matrices

$$\begin{aligned} Y_s &= Z^T Y Z, \\ X_s &= Q^T X Q, \end{aligned} \tag{4.11}$$

que transforman la ecuación (4.1) en

$$A_s^T X_s E_s + E_s^T X_s A_s = -Y_s. \tag{4.12}$$

Resolución de la ecuación en forma reducida

Para resolver la ecuación (4.12), se hace una partición por bloques de las matrices A_s, E_s, X_s e Y_s que sea conforme con los bloques cuadrados 1×1 ó 2×2 en la diagonal de A_s.

Para la matriz del ejemplo (4.10), la partición sería:

$$A_s = \left(\begin{array}{c|cc|c|c} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ \hline 0 & a_{22} & a_{23} & a_{24} & a_{25} \\ 0 & a_{32} & a_{33} & a_{34} & a_{35} \\ \hline 0 & 0 & 0 & a_{44} & a_{45} \\ \hline 0 & 0 & 0 & 0 & a_{55} \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ 0 & A_{22} & A_{23} & A_{24} \\ 0 & 0 & A_{33} & A_{34} \\ 0 & 0 & 0 & A_{44} \end{pmatrix},$$

$$E_s = \left(\begin{array}{c|c|cc|c} e_{11} & e_{12} & e_{13} & e_{14} & e_{15} \\ \hline 0 & e_{22} & e_{23} & e_{24} & e_{25} \\ 0 & 0 & e_{33} & e_{34} & e_{35} \\ \hline 0 & 0 & 0 & e_{44} & e_{45} \\ \hline 0 & 0 & 0 & 0 & e_{55} \end{array}\right) = \begin{pmatrix} E_{11} & E_{12} & E_{13} & E_{14} \\ 0 & E_{22} & E_{23} & E_{24} \\ 0 & 0 & E_{33} & E_{34} \\ 0 & 0 & 0 & E_{44} \end{pmatrix},$$

$$Y_s = \left(\begin{array}{c|c|cc|c} y_{11} & y_{12} & y_{13} & y_{14} & y_{15} \\ \hline y_{21} & y_{22} & y_{23} & y_{24} & y_{25} \\ y_{31} & y_{32} & y_{33} & y_{34} & y_{35} \\ \hline y_{41} & y_{42} & y_{43} & y_{44} & y_{45} \\ \hline y_{51} & y_{52} & y_{53} & y_{54} & y_{55} \end{array}\right) = \begin{pmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} \\ Y_{21} & Y_{22} & Y_{23} & Y_{24} \\ Y_{31} & Y_{32} & Y_{33} & Y_{34} \\ Y_{41} & Y_{42} & Y_{43} & Y_{44} \end{pmatrix},$$

$$X_s = \left(\begin{array}{c|c|cc|c} x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ \hline x_{21} & x_{22} & x_{23} & x_{24} & x_{25} \\ x_{31} & x_{32} & x_{33} & x_{34} & x_{35} \\ \hline x_{41} & x_{42} & x_{43} & x_{44} & x_{45} \\ \hline x_{51} & x_{52} & x_{53} & x_{54} & x_{55} \end{array}\right) = \begin{pmatrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ X_{41} & X_{42} & X_{43} & X_{44} \end{pmatrix}.$$

A partir de aquí, se realiza una resolución por bloques hacia adelante, para resolver las ecuaciones de Sylvester 1×1 ó 2×2 que aparecen

$$A_{kk}^T X_{kl} E_{ll} + E_{kk}^T X_{kl} A_{ll} = -\widehat{Y}_{kl},$$

cuya parte derecha es

$$\widehat{Y}_{kl} = Y_{kl} + \sum_{\substack{i=1,j=1 \\ (i,j) \neq (k,l)}}^{k,l} (A_{ik}^T X_{ij} E_{jl} + E_{ik}^T X_{ij} A_{jl}). \tag{4.13}$$

Sólo se resolverán para el caso de los bloques X_{kl} por encima y en la diagonal principal de X_s, dada la simetría de la solución.

La forma de resolverlas es mediante el sistema de ecuaciones lineales equivalente, como se veía en (4.8). Aunque entonces se vio que el método no era recomendable para ecuaciones de orden elevado, aquí sí es útil ya que el orden de las ecuaciones es sólo 1 ó 2

$$(E_{ll}^T \otimes A_{kk}^T + A_{ll}^T \otimes E_{kk}^T) vec(X_{kl}) = -vec(\hat{Y}_{kl}).$$

Sin embargo, esta forma de resolver la ecuación, calculando la expresión (4.13) para cada X_{kl} lleva a un coste total de orden n^4 lo que no es viable en la práctica. Hay una forma de reducir ese coste a cúbico [Pen98], consistente en ir calculando la expresión (4.13) al mismo tiempo para toda una columna de bloques de X_s, ya que tienen una gran parte de la expresión en común. Puede verse la expresión desglosada de la siguiente manera:

$$\hat{Y}_{kl} = Y_{kl} \quad + \quad \sum_{i=1}^{k} \left(A_{ik}^T \left(\sum_{j=1}^{l-1} X_{ij} E_{jl} \right) + E_{ik}^T \left(\sum_{j=1}^{l-1} X_{ij} A_{jl} \right) \right) +$$

$$+ \quad \sum_{i=1}^{k-1} (A_{ik}^T X_{il} E_{ll} + E_{ik}^T X_{il} A_{ll}).$$

Transformación hacia atrás

Finalmente, cuando se tiene la solución de la ecuación en su forma reducida, deshaciendo el cambio (4.11) se obtiene la solución de la ecuación original:

$$X = Q X_s Q^T.$$

Pueden consultarse algunas implementaciones paralelas básicas de la resolución de la ecuación de Lyapunov mediante el método de Bartels-Stewart en [BCHV97]. En [GK10b, GK10a] se describen unas implementaciones paralelas más elaboradas, que ofrecen rutinas con interfaz similar a la de la librería ScaLAPACK para resolver diferentes tipos de ecuaciones (de Lyapunov y de Sylvester) mediante variantes de este método. Se trata de las rutinas que forman la librería SCASY, que se mencionó brevemente en el capítulo 2.

Método de Hammarling

El método de Hammarling [Ham82] es una alternativa al método de Bartels-Stewart en el caso en que la ecuación de Lyapunov a resolver sea estable y su parte derecha semidefinida positiva. Además, el método asume que la parte derecha viene dada en la forma del producto de una matriz por su traspuesta. Todas estas restricciones se dan, por ejemplo, en los problemas de reducción de modelos [FGG92], que son objeto del capítulo 5.

El método permite resolver tanto la forma continua de la ecuación:

$$A^T X E + E^T X A = -B^T B, \tag{4.14}$$

como la forma discreta:

$$A^T X A - E^T X E = -B^T B.$$

En estas ecuaciones, B es una matriz real de orden $m \times n$ mientras que A, E y X son matrices cuadradas de orden n.

Bajo las condiciones de estabilidad y dado que la parte derecha es semidefinida positiva, la solución X es simétrica y semidefinida positiva (teorema 4.2) y utilizando el método de Hammarling puede obtenerse su descomposición de Cholesky sin necesidad de calcular el producto $B^T B$ ni de obtener previamente la solución completa X.

A continuación se explica una generalización del método de Hammarling para el caso de la ecuación generalizada continua [Pen98], la ecuación (4.14). El tratamiento sería similar para el caso de la ecuación discreta. La forma de proceder se parece a la del método de Bartels-Stewart. También consta de tres etapas: una transformación a forma reducida, la resolución de la forma reducida y una transformación hacia atrás.

Transformación a forma reducida

Al igual que en el método de Bartels-Stewart, se aplica el algoritmo iterativo QZ al haz de matrices $A - \lambda E$ para reducirlo a la forma real de Schur generalizada $A_s - \lambda E_s$. El algoritmo QZ devuelve las matrices A_s, en forma casi-triangular superior, E_s, triangular superior, y dos matrices ortogonales Q,Z que cumplen

$$\begin{aligned} A_s &= Q^T A Z, \\ E_s &= Q^T E Z. \end{aligned} \tag{4.15}$$

Como en el caso del método de Bartels-Stewart la ecuación (4.14) se reduce a la forma

$$A_s^T X_s E_s + E_s^T X_s A_s = -(BZ)^T (BZ)$$

siendo

$$X_s = Q^T X Q$$

pero ahora se pretende además simplificar la estructura del factor BZ.

La ecuación reducida a la que se pretende llegar es

$$A_s^T U_s^T U_s E_s + E_s^T U_s^T U_s A_s = -B_s^T B_s \tag{4.16}$$

siendo U_s el factor de Cholesky de la solución X_s y B_s una matriz en forma triangular superior.

Si $m \geq n$ (mayor número de filas que de columnas en la matriz B de la ecuación original), la matriz B_s se obtiene a partir de la siguiente descomposición QR

$$BZ = Q_B \begin{pmatrix} B_s \\ 0 \end{pmatrix}.$$

Si $m < n$, el procedimiento es algo más complicado. Primero se particiona el producto de BZ de la siguiente manera

$$BZ = \begin{pmatrix} \hat{B}_1 & \hat{B}_2 \end{pmatrix}.$$

Entonces se calcula la descomposición QR de la matriz $\hat{B}_1$ de tamaño $m \times m$

$$\hat{B}_1 = Q_B \hat{B}_3.$$

En este caso, la matriz B_s viene dada por

$$B_s = \begin{pmatrix} \hat{B}_3 & Q_B^T \hat{B}_2 \\ 0 & 0 \end{pmatrix}.$$

Resolución de la ecuación en forma reducida

Para resolver la ecuación (4.16), se particiona a las matrices involucradas A_s, E_s, B_s y la solución U_s de la forma

$$\begin{aligned} A_s &= \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}, \qquad E_s = \begin{pmatrix} E_{11} & E_{12} \\ 0 & E_{22} \end{pmatrix} \\ B_s &= \begin{pmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{pmatrix}, \qquad U_s = \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}. \end{aligned}$$

Los bloques superiores izquierdo de cada partición son bloques 1×1 ó 2×2 en función de si el primer bloque diagonal de la matriz casi-triangular A_s es 1×1 ó 2×2. Es decir, en función de si se corresponde con un valor propio real o un par de valores propios complejos conjugados del haz $A - \lambda E$.

Esta partición aplicada a la ecuación (4.16) conduce a las siguientes ecuaciones, donde se supone que U_{11} y E_{11} son no singulares:

$$\begin{aligned} A_{11}^T U_{11}^T U_{11} E_{11} + E_{11}^T U_{11}^T U_{11} A_{11} &= -B_{11}^T B_{11}, && (4.17) \\ A_{22}^T U_{12}^T + E_{22}^T U_{12}^T M_1 &= -B_{12}^T M_2 - A_{12}^T U_{11}^T - E_{12}^T U_{11}^T M_1, && (4.18) \\ A_{22}^T U_{22}^T U_{22} E_{22} + E_{22}^T U_{22}^T U_{22} A_{22} &= -B_{22}^T B_{22} - B_{12}^T B_{12} \\ &\quad -(A_{12}^T U_{11}^T + A_{22}^T U_{12}^T)(U_{11} E_{12} + U_{12} E_{22}) \\ &\quad -(E_{12}^T U_{11}^T + E_{22}^T U_{12}^T)(U_{11} A_{12} + U_{12} A_{22}) \\ &= -B_{22}^T B_{22} - Y Y^T. && (4.19) \end{aligned}$$

Las matrices M_1, M_2 e Y vienen dadas por:

$$\begin{aligned} M_1 &= U_{11} A_{11} E_{11}^{-1} U_{11}^{-1} && (4.20) \\ M_2 &= B_{11} E_{11}^{-1} U_{11}^{-1} && (4.21) \\ Y &= B_{12}^T - (E_{12}^T U_{11}^T + E_{22}^T U_{12}^T) M_2^T. \end{aligned}$$

Como se ha comentado anteriormente, se puede asumir que E es no singular (de lo contrario no está garantizada la existencia de solución única por el corolario 4.1), con lo que existe la inversa de E_{11}.

En el caso en que U_{11} fuera singular, siguen pudiendo utilizarse estas expresiones, pero hay que calcular las matrices M_1 y M_2 mediante otros cálculos que evitan la inversión de U_{11}. Puede consultarse el punto 6 de [Ham82] para conocer todos los detalles.

La ecuación (4.17) es una versión de la ecuación reducida (4.16), de orden 1 ó 2. Posteriormente se explica la forma de resolverla.

Cuando se ha resuelto (4.17), pueden calcularse las matrices M_1 y M_2. Entonces, la ecuación (4.18) se resuelve como una ecuación de Sylvester generalizada. Nótese que es una versión reducida de este tipo de ecuaciones, ya que las matrices coeficiente de la parte izquierda son triangulares o casi-triangulares.

Si la parte derecha de la ecuación (4.19) tuviera la forma de una matriz triangular superior, premultiplicada por su transpuesta, sería posible resolver esta nueva

ecuación de Lyapunov, en la que se ha conseguido reducir el tamaño del problema, aplicando el mismo método descrito para la ecuación (4.16).

Para ello, debe calcularse una matriz triangular superior $\tilde{B}_{22}$ que cumpla

$$B_{22}^T B_{22} + YY^T = \tilde{B}_{22}^T \tilde{B}_{22}.$$

Dicha matriz $\tilde{B}_{22}$ triangular superior puede obtenerse mediante la descomposición QR

$$\begin{pmatrix} B_{22} \\ Y^T \end{pmatrix} = Q_{\tilde{B}} \begin{pmatrix} \tilde{B}_{22} \\ 0 \end{pmatrix}. \tag{4.22}$$

Nota de implementación: En esta ecuación, la matriz B_{22} ya está en forma triangular, con lo que para calcular la descomposición QR indicada basta con aplicar transformaciones que anulen Y^T, que es menos costoso que una descomposición QR completa, necesaria si la matriz fuera densa.

Por tanto, la ecuación (4.19) se puede expresar como

$$A_{22}^T U_{22}^T U_{22} E_{22} + E_{22}^T U_{22}^T U_{22} A_{22} \quad = \quad -\tilde{B}_{22}^T \tilde{B}_{22}$$

y puede resolverse utilizando el mismo procedimiento empleado para la resolución de la ecuación reducida (4.16), pero ahora la matriz solución es de tamaño $n-1$ ó $n-2$.

La ecuación (4.16) puede resolverse de forma recursiva aplicando todos estos pasos.

Transformación hacia atrás

Una vez que se tiene la solución de la ecuación reducida, hay que realizar una transformación para obtener la solución de la ecuación original. En el caso del método de Hammarling, esta transformación consiste en obtener la descomposición QR del producto de la matriz U_s, resultado de la ecuación reducida, por la transpuesta de la matriz Q, proveniente de la transformación a la forma de Schur generalizada:

$$U_s Q^T = Q_U U.$$

Así se obtiene la matriz triangular superior U, descomposición de Cholesky de la solución X de la ecuación (4.14).

Utilizando este método no ha hecho falta calcular en ningún momento el producto $B^T B$, que podría añadir errores, ni tampoco se calcula la solución X para luego calcular su descomposición de Cholesky, lo que también añadiría errores.

Si lo que se precisa es la solución X de la ecuación de Lyapunov (4.1), donde la parte derecha está expresada en forma de una única matriz, se utilizará el método de Bartels-Stewart.

Si lo que se precisa es el factor de Cholesky de la solución de la ecuación (4.14), en la que además la parte derecha viene dada como producto de una matriz por su transpuesta, el método de Hammarling es más apropiado que el de Bartels-Stewart. Para este último, habría que multiplicar la parte derecha de la ecuación, y al final calcular la descomposición de Cholesky. Esto, además de aumentar la complejidad temporal del algoritmo, puede añadir error a la solución.

Resolución de la ecuación reducida de orden 2×2

Ha faltado ver la resolución de la ecuación (4.17), que se corresponde con una ecuación reducida de tamaño a lo sumo 2×2.

Si el orden de las matrices A_{11},E_{11},B_{11} y U_{11} es 1, la resolución es inmediata, ya que la ecuación (4.17) resulta una ecuación con escalares

$$2a_{11}e_{11}u_{11}^2 = -b_{11}^2,$$

de donde puede obtenerse fácilmente la solución

$$u_{11} = \frac{b_{11}}{\sqrt{-2a_{11}e_{11}}}. \tag{4.23}$$

En el caso en que las matrices A_{11},E_{11},B_{11} y U_{11} sean 2×2, el método utilizado para resolver la ecuación (4.17) es el mismo que el utilizado para la ecuación sin reducir (4.14), pero aprovechándose del reducido tamaño de todas las matrices involucradas. Se puede usar la forma de Schur compleja, que en un tamaño mayor supondría aumentar mucho el número de operaciones.

Utilizando aritmética compleja, primero el haz $A_{11} - \lambda E_{11}$ es reducido a la forma de Schur generalizada mediante las matrices unitarias $\hat{Q}$ y $\hat{Z}$ que cumplen

$$\hat{Q}^H A_{11} \hat{Z} = \hat{A}_{11} = \begin{pmatrix} a_{11} & a_{12} \\ 0 & a_{22} \end{pmatrix},$$
$$\hat{Q}^H E_{11} \hat{Z} = \hat{E}_{11} = \begin{pmatrix} e_{11} & e_{12} \\ 0 & e_{22} \end{pmatrix}.$$

Entonces, se calcula la descomposición QR del producto $B_{11}\hat{Z}$

$$B_{11}\hat{Z} = \hat{Q}_B \hat{B}_{11},$$

donde la matriz $\hat{Q}_B$ es unitaria, y las entradas de la diagonal principal de la matriz triangular superior $\hat{B}_{11}$ son reales no negativas.

La ecuación en forma reducida queda

$$\hat{A}_{11}^H \hat{U}_{11}^H \hat{U}_{11} \hat{E}_{11} + \hat{E}_{11}^H \hat{U}_{11}^H \hat{U}_{11} \hat{A}_{11} = -\hat{B}_{11}^H \hat{B}_{11}.$$

Si se particiona $\hat{B}_{11}$ y $\hat{U}_{11}$ como

$$\hat{B}_{11} = \begin{pmatrix} b_{11} & b_{12} \\ 0 & b_{22} \end{pmatrix},$$
$$\hat{U}_{11} = \begin{pmatrix} u_{11} & u_{12} \\ 0 & u_{22} \end{pmatrix},$$

la solución se obtiene de

$$\delta_1 \quad = \quad \sqrt{-\bar{a}_{11}e_{11} - \bar{e}_{11}a_{11}},$$

$$\begin{aligned} u_{11} &= \frac{b_{11}}{\delta_1}, \\ u_{12} &= -\frac{b_{12}\delta_1 + (\bar{a}_{11}e_{12} + a_{12}\bar{e}_{11})u_{11}}{a_{22}\bar{e}_{11} + e_{22}\bar{a}_{11}}, \\ \delta_2 &= \sqrt{-\bar{a}_{22}e_{22} - \bar{e}_{22}a_{22}}, \\ y &= \bar{b}_{12} - \frac{\delta_1}{\bar{e}_{11}}(\bar{e}_{12}u_{11} + \bar{e}_{22}\bar{u}_{12}), \\ u_{22} &= \frac{1}{\delta_2}\sqrt{b_{22}^2 + |y|^2}. \end{aligned}$$

Ahora sólo falta hacer la transformación hacia atrás, para obtener la solución de (4.17). Para ello, se realiza la descomposición QR de

$$\hat{U}_{11}\hat{Q} = Q_U U_{11}$$

donde Q_U es una matriz unitaria.

Pueden consultarse algunas implementaciones paralelas básicas de la resolución de la ecuación de Lyapunov generalizada mediante el método de Hammarling en [GHRV97a] y [GHRV97b]. En [Cla99, CH99] se analiza una implementación para resolver el caso discreto de la ecuación estándar mediante paso de mensajes usando una distribución de datos que maximice el grado de paralelismo (se va operando por bloques antidiagonales de la matriz, distribuida cíclicamente por columnas). En [SZ03] se proponen algunas optimizaciones del método básico. Y en [Kre08] puede verse una implementación secuencial orientada a bloques.

Métodos iterativos

Los métodos de Bartels-Stewart y de Hammarling recién presentados son los métodos numéricos de referencia para la resolución de ecuaciones de Lyapunov. Sin embargo, ambos métodos necesitan calcular la forma real de Schur de una matriz. Este proceso es especialmente costoso y además no permite explotar una posible dispersión de las matrices de entrada.

Se puede evitar el uso de la transformación a forma real de Schur mediante métodos iterativos para la obtención de la solución de la ecuación de Lyapunov.

El método Smith cuadrático [Smi68] y el método de la función signo matricial [Rob80] permiten calcular la solución de ecuaciones de Lyapunov de una forma iterativa y sin necesidad de calcular la forma de Schur. Sin embargo, no aprovechan el posible caso de que las matrices sean dispersas. A pesar de ello son buenas alternativas a los métodos de Bartels-Stewart y Hammarling en la resolución de problemas con matrices densas. Existen múltiples implementaciones paralelas de estos métodos [BQ99, GL91].

En problemas de dimensión muy grande, suele ocurrir que las matrices del problema sean dispersas. Resulta entonces conveniente utilizar métodos que permitan mantener y aprovechar esta dispersidad, como por ejemplo el método ADI (*Alternating Direction Implicit*) [PR55, Wac88].

El método ADI aprovecha y mantiene la dispersidad de los datos de entrada, pero al igual que los otros genera la matriz solución en forma densa, lo que también puede evitarse cuando se trata de problemas de bajo rango. Existen métodos llamados *de bajo rango* que aprovechan este hecho para obtener la solución X en la forma de un factor de Cholesky de bajo rango (*low rank Cholesky factor*). Esto es una matriz Z rectangular (no necesariamente triangular, habitualmente densa) que aproxima $ZZ^T \approx X$, siendo X de $n \times n$ y Z de $n \times r$ con $r \ll n$.

Con estos métodos de bajo rango se pueden llegar a resolver ecuaciones de Lyapunov de tamaños especialmente grandes.

La mayoría de métodos *de bajo rango* son métodos basados en subespacios de Krylov [JK94, HTP96, SG90, KS11, SS12, KPT13], aunque también los hay basados en otras técnicas, como por ejemplo en la iteración ADI [BLP08, BKS13]. Algunos de estos métodos se encuentran disponibles en el paquete para MATLAB LyaPack [Pen00], que permite resolver ecuaciones de Lyapunov y problemas numéricos relacionados con ella mediante métodos iterativos aplicados sobre matrices dispersas.

Las últimas tendencias en resolución de ecuaciones de Lyapunov son esfuerzos dirigidos a utilizar computadores gráficos (GPUs: *Graphics Processing Unit*). La tecnología actual tiende a integrar muchas unidades de cómputo en procesadores gráficos y a ofrecer formas de aprovecharlos no sólo para hacer gráficos sino también para computación general.

Por ejemplo, en [RS12] resuelven la ecuación de Lyapunov en GPUs usando los métodos de Bartels-Stewart, Smith, ADI y la función signo matricial.

Incluso hay algoritmos híbridos que aprovechan simultáneamente la CPU tradicional y la GPU para resolver ecuaciones de Lyapunov, como por ejemplo en [BEK+11, BRD+15].

4.2. Rutinas de altas prestaciones desarrolladas

En este apartado se describen las rutinas de altas prestaciones desarrolladas para la resolución de la ecuación de Lyapunov, dentro del marco de la reducción de modelos.

Como se verá en el próximo capítulo, la ecuación de Lyapunov a resolver en el marco de la reducción de modelos es estable y se desea obtener el factor de Cholesky de la solución. Bajo estas premisas, el método más adecuado es el presentado por Hammarling [Ham82] y que se corresponde con el implementado en las rutinas de SLICOT. Aquí se presentan versiones paralelas para las rutinas de SLICOT dirigidas a resolver este problema.

En la figura 4.1 se muestra el árbol de llamadas entre todas estas nuevas rutinas. Todas las rutinas que aparecen en esta figura son rutinas nuevas que no estaban presentes en las librerías existentes y resultan necesarias para la reducción de modelos.

Todas las nuevas rutinas utilizan algoritmos paralelos, salvo por la excepción de la rutina `SB03OT2`. Esta rutina es una versión especial de la rutina `SB03OT` presente en SLICOT. Sigue siendo secuencial, pero almacena algunos resultados intermedios que serán necesarios en las rutinas paralelas que la utilizan y que no se guardan en

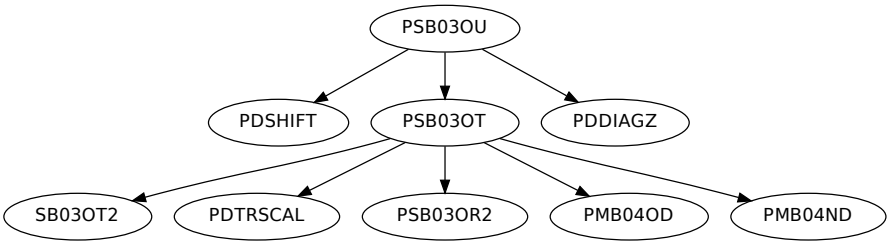

Figura 4.1: Rutinas de altas prestaciones desarrolladas para resolver la ecuación de Lyapunov

la rutina presente en SLICOT.

El principal objetivo perseguido con el desarrollo de estas rutinas es disponer de una implementación de algoritmos fiables y eficientes de resolución de la ecuación de Lyapunov por el método de Hammarling. Aunque aquí sólo se muestran las versiones finales, cabe mencionar que se ha trabajado hasta con siete versiones de los algoritmos centrales de este proceso numérico. Pese a ello, los cambios entre ellas son modificaciones en subtareas internas del algoritmo global, algunas de ellas mínimas. Tan sólo tres de estas versiones merecen ser mencionadas, una de ellas muy brevemente.

La primera versión paralela del algoritmo de resolución de la ecuación de Lyapunov se obtuvo realizando la utópica transformación que se sugiere en algunas ocasiones como fácil para paralelizar código basado en BLAS y LAPACK. Consiste en cambiar las llamadas a rutinas de BLAS y LAPACK por llamadas a las rutinas de PBLAS y ScaLAPACK. Se parte del código secuencial y, si todas las rutinas de BLAS y LAPACK utilizadas tienen su versión paralela implementada en PBLAS y ScaLAPACK, se añade una `P` por delante en los nombres de las rutinas invocadas, reescribiendo los argumentos, que cambian ligeramente en las versiones paralelas.

Sin embargo, debido a algunas restricciones de alineamiento y a la necesidad de tener algunos datos en procesos diferentes a aquellos donde se generan, la transformación no es tan sencilla. Se requiere fragmentar algunas operaciones en otras y realizar bastantes comunicaciones de forma explícita mediante llamadas a rutinas de BLACS.

De cualquier modo, una vez finalizada la 'paralelización' efectuada de esta manera, los resultados experimentales en cuanto a prestaciones han resultado muy deficientes. La causa de estas malas prestaciones reside en el hecho de que el algoritmo utilizado para resolver la ecuación de Lyapunov es de un grano muy fino. Está orientado a bloques de 1×1 ó 2×2 como se indicó al explicar el método de Hammarling en la sección 4.1.3. Si las operaciones que se realizan para cada bloque de estos tamaños (aunque también se incluye el trabajo con bloques de filas o columnas de la matriz,

que llegan a ser de $2 \times (n-2)$) se realizan en paralelo mediante llamadas a PBLAS o ScaLAPACK, resulta un grano demasiado fino.

Lo normal en estas librerías paralelas es que se repartan entre diferentes procesos las operaciones a realizar en cada rutina de alto nivel. Si se llama a estas rutinas con matrices de pequeño tamaño, sólo se logra cooperación paralela en operaciones con submatrices muy pequeñas que no consiguen justificar con suficiente carga computacional el peso de las comunicaciones necesarias. Hay que procurar que las llamadas a rutinas de alto nivel de las librerías paralelas se hagan con submatrices lo más grandes posibles, maximizando el trabajo a realizar y así la posibilidad de realizarlo en paralelo.

En general, si el algoritmo que se pretende paralelizar se compone de llamadas a rutinas de BLAS o LAPACK con submatrices de gran tamaño y que 'en la ejecución en paralelo estarán bastante bien distribuidas en la malla de procesos', sí que puede ser suficiente con realizar un cambio llamando ahora a rutinas de PBLAS y ScaLAPACK.

Pero si el algoritmo secuencial está orientado al trabajo con submatrices muy pequeñas o incluso elementos, hay que rediseñar el algoritmo para que pueda dar buenas prestaciones en paralelo. Esto es lo que sucede en el caso del algoritmo de resolución de la ecuación de Lyapunov por el método de Hammarling. En estos casos, hay que rediseñar el algoritmo agrupando operaciones para que quede *orientado a bloques*. Después se paralelizará este algoritmo orientado a bloques, consiguiendo así unas mejores prestaciones.

Al orientar a bloques de la forma tradicional (aumentando el tamaño de las submatrices con las que operar), se ha obtenido un algoritmo que tras ser paralelizado reporta buenas prestaciones. Sin embargo, al realizar las operaciones básicas del algoritmo con bloques mayores de 2×2, se ha perdido parte de la buena precisión que se tenía. En esta versión la precisión baja al aumentar el tamaño de bloque usado en la distribución, cuando se resuelven ecuaciones de Lyapunov en las que la parte derecha de la ecuación no sea definida positiva (sea semidefinida positiva) o esté mal condicionada, cosa que no ocurre en la versión secuencial.

Como consecuencia, se ha tenido que desarrollar una nueva versión orientada a bloques [GLR15], pero de una manera especial tal que conserve las buenas prestaciones de esta última versión [GHR02] y las buenas características en cuanto a precisión de la versión secuencial [Ham82]. En esta versión se han ido agrupando las operaciones que se realizan en el algoritmo secuencial en bloques, pero dentro de cada bloque se orientan las operaciones hacia submatrices 1×1 o 2×2. Para poder realizar esta tarea, se han añadido nuevas submatrices que almacenan resultados intermedios de las operaciones para utilizarlos posteriormente. Por esto se ha tenido que desarrollar una versión especial de la rutina `SB03OT`, con el nombre `SB03OT2`, que devuelva esos resultados intermedios que antes no eran necesarios.

La actualización a esta nueva versión a bloques afecta a las rutinas `PSB03OT` y `PSB03OR` (además de la nueva `SB03OT2`). `PSB03OT` ha cambiado internamente para hacer uso de los resultados intermedios de las operaciones realizadas sobre bloques. Lo mismo sucede con la nueva versión de `PSB03OR`, que ahora además necesita unos nuevos parámetros donde recoger esos resultados intermedios, que provienen de la

nueva SB03OT2, por lo que se le ha cambiado el nombre a PSB03OR2. No conviene ofrecer dos versiones de una misma rutina con diferente número de parámetros de entrada.

A continuación se describen las nuevas rutinas desarrolladas que guardan relación con la resolución de la ecuación de Lyapunov. Para las rutinas PSB03OT y PSB03OR2 se explica primero la penúltima versión, cuyos algoritmos paralelos se conservan en gran medida en la última versión, y luego se comentan los cambios realizados en la última versión para evitar los problemas de precisión que existían.

4.2.1. PSB03OU

```
PSB03OU( DISCR, LTRANS, N, M, A,IA,JA,DESCA,
         B,IB,JB,DESCB, TAU, U,IU,JU,DESCU,
                   SCALE, DWORK,LDWORK, INFO )
```

PSB03OU permite resolver las ecuaciones de Lyapunov estables tanto de tiempo continuo como de tiempo discreto en sus versiones traspuesta y no traspuesta, teniendo la matriz A en forma real de Schur. Es una versión paralela de la rutina SB03OU de SLICOT.

La ecuación de Lyapunov de tiempo continuo y su versión traspuesta presentan la forma

$$A^T X + XA = -\alpha^2 B^T B,$$
$$AX + XA^T = -\alpha^2 BB^T,$$

mientras que en tiempo discreto sin trasponer y traspuesta tienen la forma

$$A^T XA - X = -\alpha^2 B^T B,$$
$$AXA^T - X = -\alpha^2 BB^T.$$

El valor α de estas ecuaciones se corresponde con el parámetro SCALE de la rutina.

En lugar de obtener la solución X a estas ecuaciones, esta rutina devuelve su factor de Cholesky U, lo que resulta más adecuado para el uso posterior en algoritmos de reducción de modelos. En el caso de las ecuaciones sin trasponer, esta U cumple $X = U^T U$. En el caso de las ecuaciones traspuestas, $X = UU^T$.

El parámetro SCALE es de salida y es un factor de escala menor o igual a la unidad que se utiliza para evitar un posible desbordamiento (*overflow*) en la solución.

El proceso que sigue esta rutina es el de terminar la transformación de la ecuación de Lyapunov a forma reducida y llamar a la rutina especializada en resolver la ecuación reducida (PSB03OT). A la entrada ya se tiene la matriz A en forma real de Schur. Tan sólo resta calcular la descomposición QR de la matriz B (o la descomposición RQ si se trata de la ecuación traspuesta), para lo que se usa la correspondiente rutina de ScaLAPACK PDGEQRF (o PDGERQF). En el caso de la ecuación traspuesta resulta necesario además realizar un desplazamiento para dejar la matriz B como la espera la rutina PSB03OT. Para esta operación se ha desarrollado la rutina PDSHIFT.

Una vez que se tiene la ecuación en forma reducida, es resuelta en la rutina PSB03OT, tras lo cual se deben dejar todos los elementos de la diagonal de U no negativos. Esto se hace en el caso secuencial recorriendo la matriz y multiplicando por -1 las filas o columnas correspondientes a elementos diagonales negativos, dejando así estos elementos positivos. En el caso paralelo, dado que la matriz U se encontrará generalmente distribuida en diferentes procesos, la operación no es inmediata. Por esta razón se ha separado en una rutina independiente: PDDIAGZ.

PSB03OU utiliza las nuevas rutinas PDSHIFT, PSB03OT y PDDIAGZ, que se explican a continuación.

4.2.2. PDSHIFT

```
PDSHIFT( LZERO, K, M,N,A,IA,JA,DESCA )
```

PDSHIFT permite desplazar una submatriz distribuida un número determinado de columnas a la izquierda o a la derecha, rellenando con ceros o no la porción desplazada. Es una rutina auxiliar que no se corresponde con la paralelización de ninguna rutina secuencial de BLAS, LAPACK o SLICOT.

En el parámetro K se le indica el número de columnas hacia la derecha que hay que desplazar la submatriz. Si K tiene un valor negativo, se entenderá que hay que desplazar -K columnas a la izquierda.

La operación realizada en esta rutina es similar a la que se realiza en PDSHIFT1, que se presentó en el capítulo anterior, salvo por dos diferencias fundamentales.

La primera es que PDSHIFT1 permite realizar el desplazamiento también hacia arriba o hacia abajo además de a izquierda o a derecha.

La segunda y más importante es que en PDSHIFT1 siempre se realiza un desplazamiento de una fila o columna. Este hecho afecta significativamente, ya que en el desplazamiento de cada bloque en PDSHIFT1 siempre hay que enviar una fila/columna y el resto es desplazamiento local. Por esta razón el algoritmo utilizado en PDSHIFT1 es diferente del aquí expuesto. Entre otras cosas, allí se utiliza espacio de trabajo. Basta con una fila/columna de la submatriz a desplazar, lo que es aceptable. En la implementación de PDSHIFT no se usa espacio extra para realizar la operación. En caso de haberlo usado, su tamaño podría resultar excesivo dependiendo del número de columnas a desplazar.

A continuación se describe el algoritmo implementado para PDSHIFT. Se hace un recorrido en el orden adecuado de la submatriz por bloques, realizando el correspondiente desplazamiento de cada bloque. Este desplazamiento puede implicar comunicaciones o no en función del número de columnas que haya que desplazar. Se presenta el algoritmo para el caso de un desplazamiento a la derecha (K > 0), siendo similar el otro caso.

Algoritmo 4.1 PDSHIFT *(con* K > 0*)*

1. *Recorrer en todos los procesos de derecha a izquierda los bloques de la submatriz a desplazar (se va a desplazar* K *columnas hacia la derecha). Para cada bloque:*

a) *Calcular el número máximo de columnas del bloque actual que va a desplazarse de modo que se encuentre en una única columna de procesos.*

b) *Calcular el número máximo de columnas del bloque destino actual donde se dejarán los valores desplazados de modo que se encuentre en una única columna de procesos.*

c) *El número de columnas a desplazar en esta iteración se corresponde con el mínimo de los dos valores anteriores.*

d) *Desplazar el bloque* `K` *columnas a la derecha. Para ello:*

1) *Si la columna de procesos donde se encuentra el bloque origen coincide con la columna de procesos donde se deben dejar sus valores, realizar el desplazamiento de forma local.*

2) *Si no, la columna de procesos con el bloque origen lo envía a la columna de procesos donde debe dejarse.*

2. *Si así se ha indicado en el parámetro* `LZERO`*, se rellenan las* `K` *columnas de la submatriz origen del desplazamiento con el valor cero. Esta operación se realiza de forma completamente local en cada proceso, ya que cada uno simplemente tiene que rellenar su parte de la submatriz.*

4.2.3. PSB03OT

```
PSB03OT( DISCR,LTRANS, N,S,IS,JS,DESCS, R,IR,JR,DESCR,
                       SCALE, DWORK, INFO )
```

`PSB03OT` permite resolver las versiones reducidas de ecuaciones de Lyapunov estables de tiempo continuo o tiempo discreto tanto en forma traspuesta como no traspuesta. Es una versión paralela de la rutina `SB03OT` de SLICOT.

La ecuación de Lyapunov de tiempo continuo en forma reducida y su versión traspuesta presentan la forma

$$S^T X + XS = -\alpha^2 R^T R,$$
$$SX + XS^T = -\alpha^2 RR^T,$$

mientras que en su forma reducida en tiempo discreto sin trasponer y traspuesta tienen la forma

$$S^T XS - X = -\alpha^2 R^T R,$$
$$SXS^T - X = -\alpha^2 RR^T.$$

El valor α de estas ecuaciones se corresponde con el parámetro `SCALE` de la rutina.

Al tratarse de ecuaciones de Lyapunov *en forma reducida*, la matriz R es una matriz triangular superior y la matriz S está en forma real de Schur.

Al igual que en el caso de la rutina `PSB03OU`, no se desea obtener la solución X de las anteriores ecuaciones sino su factor de Cholesky U. En la figura 4.2 puede

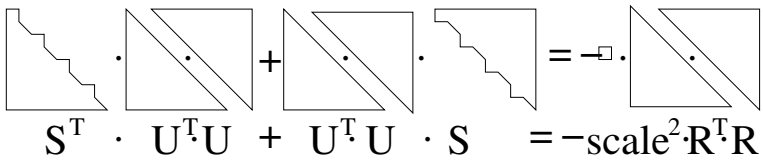

Figura 4.2: Forma de las matrices en la ecuación de Lyapunov reducida de tiempo continuo no traspuesta

observarse a modo de ejemplo la forma que presentan las matrices de la ecuación en el caso continuo no traspuesto.

El método utilizado es una versión paralela del método de Hammarling que se explicó para el caso generalizado en la sección 4.1.3.

De esta rutina se han desarrollado varias versiones, aunque sólo van a mencionarse las dos más importantes, denominadas aquí *versión 1* y *versión 2*. La *versión 2* es la final, con mejores características que la *versión 1*.

Ambas versiones permiten resolver los cuatro casos posibles de ecuaciones de Lyapunov reducidas: tiempo continuo no traspuesta, tiempo continuo traspuesta, tiempo discreto no traspuesta y tiempo discreto traspuesta. En la implementación se distinguen dos grandes bloques (con una sentencia condicional) que se corresponden con el caso traspuesto y el caso no traspuesto de la ecuación. Las diferencias entre ellos son muy notables ya que afectan al orden en que se irá calculando la solución y con ello también al sentido y la forma de las comunicaciones. Las diferencias entre el caso de tiempo continuo y el caso de tiempo discreto son menores y se han implementado distinguiendo uno u otro caso allá donde implican realizar operaciones diferentes, añadiendo algunas operaciones extra en el caso discreto.

Esta multiplicidad del código ya se daba en la versión secuencial desarrollada en la rutina de SLICOT `SB03OT`. A ella se unía otra sentencia condicional más para diferenciar el caso de resolver un bloque 1×1 de la ecuación, correspondiente a un valor propio real de la matriz S, del caso de resolver un bloque 2×2 de la ecuación, correspondiente a un par de valores propios complejos de S. Hay que recordar que, debido al método utilizado, la ecuación se va resolviendo en bloques 1×1 ó 2×2 en función de si se van encontrando valores propios reales o complejos. Cuando se halla un bloque 2×2, se utilizan expresiones diferentes que deben usarse para todo el bloque, no pudiendo cortarlo.

Este hecho en el caso secuencial tiene fácil solución sin más que aumentar a 2 el tamaño de bloque cuando se encuentra un par de valores propios complejos en S. En el caso paralelo hay que hacer lo mismo. Pero aumentar en 1 el tamaño del bloque que se está resolviendo en un momento dado puede convertirse en un problema, si esa fila y columna que se añade no se encuentra físicamente distribuida en el proceso que está resolviendo el bloque actual. En estos casos se requiere realizar comunicaciones para traer esa fila y columna al proceso adecuado y más comunicaciones posteriormente para devolver la solución a su lugar correspondiente. Esto es lo que

se ha implementado, logrando así que el algoritmo funcione bien incluso cuando la distribución utilizada separa en procesos diferentes los bloques 2×2 correspondientes a pares de valores propios complejos de la matriz S.

En versiones anteriores [GHRV97b, GHRV97a] no se permitía el corte de estos bloques. Puede garantizarse que estos bloques no se cortan si en el paso previo de factorización de la matriz a forma real de Schur se reordena la matriz dejando los bloques 2×2 de valores propios complejos en la parte superior y se utiliza un tamaño de bloque par.

En esta ocasión sí que se permiten esos cortes en la distribución gracias a que son tratados de forma explícita en el código. Hay que mencionar que el trabajo realizado para solventar este problema ha sido importante. Nótese que en el caso de una distribución *bidimensional*, las comunicaciones necesarias en el caso de un corte implican a cuatro procesos, ya que se tendría cada elemento del bloque 2×2 en un proceso diferente. Además, no sólo hay que llevar estos cuatro elementos al proceso que va a resolver esta porción de la ecuación, sino toda la fila/columna correspondiente para poder actualizar en consecuencia y seguir con la resolución del resto de la ecuación.

Este problema de los cortes también se da en la resolución de la ecuación de Sylvester que aparece en cada etapa de resolución de la ecuación reducida. Esta operación, que se realiza en la rutina `PSB03OR` o `PSB03OR2`, también ha sido implementada prestando especial cuidado para tratar los casos en que aparecen estos cortes en la distribución.

`PSB03OT` *versión 1*

Esta versión de la rutina `PSB03OT` resuelve las ecuaciones de Lyapunov reducidas mediante una implementación paralela del método de Hammarling, tras haberlo orientado a bloques. Ahora se aplican las expresiones y pasos explicados en la sección 4.1.3, pero los tamaños de las matrices más pequeñas involucradas pasan a ser $nb \times nb$, siendo nb el tamaño de bloque utilizado en la distribución. En la obtención del primero y el último bloque de la matriz U solución puede ocurrir que se utilice un tamaño inferior.

Con la orientación a bloques se consigue, por una parte, reducir el número de comunicaciones necesarias, puesto que se realizarán menos pasos para resolver la ecuación. Y por otra parte, el agrupar las operaciones sobre bloques de elementos favorece la localidad en el acceso a memoria, reduciendo así el trasiego de datos entre las diferentes jerarquías de memoria y mejorando las prestaciones.

En lo que sigue, van a centrarse las explicaciones en el caso de la ecuación de Lyapunov de tiempo continuo no traspuesta.

Antes de pasar a explicar el algoritmo orientado a bloques y después el paralelo, se va a recordar mínimamente el método de Hammarling (puede verse con más detalle en la sección 4.1.3).

Inicialmente se particionan las matrices involucradas S, R y U en cuatro bloques, que en el caso de la matriz S serían S_{11}, S_{12}, S_{21} y S_{22}. S_{11} es de tamaño 1×1 ó 2×2 en función de si en S se corresponde con un valor propio real o un par

de valores propios complejos, respectivamente. Este mismo particionado se utiliza para el resto de matrices, de tal forma que las submatrices con idénticos subíndices tienen igual tamaño. Las submatrices con subíndices 21 son nulas (recuérdese que se está explicando el caso no traspuesto).

Una vez realizado este particionado, se resuelve una ecuación de Lyapunov como la que se quiere resolver pero de un menor tamaño, para obtener U_{11}. Esta ecuación es más sencilla de resolver por involucrar matrices de a lo sumo 2×2.

Una vez que se tiene U_{11}, se calculan las matrices auxiliares M_1 y M_2 con igual tamaño que U_{11}. Con estas tres matrices se calculan los coeficientes y la parte derecha de una ecuación reducida de Sylvester cuya solución es U_{12}. Es una ecuación de Sylvester *reducida* porque las matrices que multiplican a las incógnitas están en forma real de Schur. Al resolver esta ecuación se obtendrá U_{12}.

Operando con U_{12} se obtiene un conjunto de vectores que se agrupan en Y. Entonces se calcula la descomposición QR de una matriz formada por R_{22} e Y^T. La matriz R obtenida en esta descomposición se corresponde con la matriz de la parte derecha de una ecuación de Lyapunov reducida cuya matriz de coeficientes es S_{22} y cuya solución es U_{22}.

Se ha resuelto U_{11} y U_{12} y queda una ecuación de Lyapunov reducida para obtener U_{22}. Nótese que esta ecuación es como la original que se está resolviendo pero con un menor tamaño, ya que se ha reducido el orden de las matrices involucradas en 1 ó 2 en función del tamaño de U_{11}. Ahora se pasa a resolver esta ecuación de Lyapunov reducida en U_{22} realizando otra vez todo el proceso explicado.

De esta forma se va obteniendo la matriz U, solución de la ecuación original que se desea resolver, por filas de arriba abajo. Cada vez se obtiene 1 ó 2 filas según en S haya un valor propio real o un par de valores propios complejos.

En la versión del algoritmo orientada a bloques se procede de igual manera, salvo que en los particionados se toma S_{11} de $nb \times nb$, siendo nb el tamaño de bloque deseado. Ahora las matrices auxiliares M_1, M_2 y todas las de subíndices 11 son de ese tamaño y esto influye en el tamaño de las ecuaciones que se van resolviendo en cada etapa. Para resolver la ecuación de Lyapunov necesaria en cada etapa para obtener U_{11} se utiliza el método no orientado a bloques. U se sigue calculando de arriba abajo (en el caso no traspuesto), pero ahora en bloques de nb filas.

Hay que notar que en la resolución mediante el método de Hammarling no se deben partir los bloques de pares de valores propios complejos de la matriz S. Es por esto que en el caso de que al final de un bloque se tenga sólo uno de un par de valores propios complejos conjugados, se aumenta en uno el tamaño de bloque utilizado en ese caso, de forma que no se corte el bloque 2×2 correspondiente de S.

En el caso paralelo, se parte del algoritmo orientado a bloques y se trabaja con matrices distribuidas en varios procesos. El tamaño de bloque nb a utilizar en el algoritmo va a ser el mismo que el usado en la distribución de las matrices. Esto es así porque es una forma de garantizar que las submatrices con subíndices 11 se encuentren completamente almacenadas en un único proceso. De esta forma, la resolución de la ecuación de Lyapunov reducida para la obtención de U_{11} en cada etapa puede hacerse perfectamente en un único proceso mediante el algoritmo no orientado a bloques. Además, la preparación, resolución y posteriores actualizaciones

del resto de ecuaciones en cada etapa utilizan una distribución acorde a la de tener U_{11} en un único proceso, lo que resulta conveniente.

Merece una mención especial explicar qué sucede en el caso paralelo cuando el uso del tamaño de bloque indicado corta un bloque 2×2 de S correspondiente a un par de valores propios complejos. Como ya se ha comentado, el método de Hammarling requiere que no se corten estos bloques. Por esto, se procede al igual que en el caso orientado a bloques: se aumenta en uno el tamaño de bloque en estos casos. Lo que pasa es que esto, que resulta trivial de implementar en el algoritmo secuencial orientado a bloques, no lo es tanto en el caso paralelo. Para empezar, ahora la última fila y columna del bloque no se tienen en el mismo proceso que el resto. Y además U_{12}, que aumentará en una fila su tamaño, tampoco se tendrá en una única fila de procesos. Lo que se hace en estos casos es comunicar la fila extra de las matrices involucradas a la fila de procesos adecuada, de forma que ahora sí que se tenga todo lo necesario en un único proceso, en el caso de la ecuación de Lyapunov para calcular U_{11}, o en una única fila de procesos, en el caso de la ecuación de Sylvester para calcular U_{12}. Además, tras resolver ambas ecuaciones habrá que devolver la fila extra a los procesos donde deberá estar la solución. En resumen, evitar en el caso paralelo cortes de bloques 2×2 correspondientes a valores propios complejos implica comunicaciones extra y una mayor complejidad en el código, pero resulta necesario para que el método funcione bien en todos los supuestos.

A continuación se expone una descripción no muy detallada del algoritmo paralelo implementado, procurando dar una idea de las operaciones y, sobre todo, las comunicaciones realizadas.

Algoritmo 4.2 `PSB03OT` *v1 (caso continuo no traspuesto)*

Recorrer en todos los procesos las matrices involucradas de arriba abajo, calculando la solución U por bloques de filas. Para cada bloque:

1. *Determinar el tamaño de bloque máximo a utilizar tal que U_{11} esté completamente almacenada en un único proceso, usando como mucho el tamaño de bloque de la distribución.*

2. *Si U_{11} no es el último bloque de U, hay que determinar si con este tamaño de bloque se corta un bloque 2×2 de valores propios complejos de la matriz S. Para que todos los procesos sepan si se da este corte o no, se requiere que el proceso donde se encuentra el elemento de S justo debajo del último elemento de la actual S_{11} envíe este valor a todos los demás. Si este valor es no nulo, utilizar este tamaño de bloque implicaría cortar un bloque 2×2 de S, cosa que no debe ocurrir. Si se da esta situación, aumentar en una fila y columna el bloque U_{11} con el que se va a trabajar. Esto implica:*

   a) *Traer a la fila de procesos donde se encuentra U_{11} la última fila de las matrices S_{12}, R_{12}.*

   b) *Traer al proceso que tiene U_{11} la última columna de las matrices S_{11}, R_{11}.*

3. *Resolver localmente en el proceso donde está U_{11} la ecuación de Lyapunov de la que es solución. Utilizar para ello el algoritmo secuencial.*

4. *Si se amplió el tamaño de bloque para no cortar un bloque* 2×2 *de* S*, entonces llevar la última columna de la recién calculada* U_{11} *al proceso donde debe estar y si no quedan más bloques de* U *por calcular, enviar el último elemento de* U_{11} *a donde corresponde.*

5. *Enviar a todos los procesos la indicación de si se ha producido algún error en la obtención de* U_{11}*, así como el factor de escalado utilizado.*

6. *Si ha habido algún error, abortar.*

7. *Si se ha producido escalado, escalar acordemente la solución previamente calculada de* U *y la porción de la matriz* R *a utilizar posteriormente para seguir calculando* U *y acumular el escalado en el factor de escala global.*

8. *Si quedan bloques de* U *por resolver, entonces:*

   a) *Preparar y resolver la ecuación de Sylvester de la que se obtendrá* U_{12}*:*

      1) *En el proceso donde está* U_{11}*, calcular las matrices auxiliares* M_1 *y* M_2*, y mandarlas junto con* U_{11} *al resto de procesos de la misma fila.*
      2) *En esta fila de procesos, calcular la parte derecha de la ecuación (sumas y productos matriciales que pueden hacerse de forma local).*
      3) *Resolver la ecuación de Sylvester en paralelo utilizando la rutina* `PSB03OR`*.*
      4) *Si se ha producido escalado en la resolución, escalar acordemente tanto la porción de* U *ya resuelta como la parte de* R *que se usará después y acumular el escalado en el factor de escala global.*
      5) *Si se amplió el tamaño de bloque para no cortar un bloque* 2×2 *de* S*, entonces enviar la última fila de la recién calculada* U_{12} *a la fila de procesos donde debe estar.*

   b) *Actualizar* R_{22} *para continuar con la resolución:*

      1) *En la fila de procesos de* U_{12}*, calcular el conjunto de vectores auxiliares y (sumas y productos matriciales que pueden hacerse de forma local) (en el caso discreto aquí aparece una descomposición QR, también local).*
      2) *Actualizar* R_{22} *mediante una descomposición QR que usa los vectores* Y*. Este proceso se hace en la rutina* `PMB04OD`*.*

Puede verse un esquema gráfico de las operaciones realizadas en este algoritmo en la figura 4.3. Aquí se ilustra la sucesión de operaciones y comunicaciones suponiendo una distribución no cíclica de las matrices en la malla de procesadores.

Es necesario indicar que los pasos de "Preparar y resolver Sylvester" y "Actualizar mediante QR" son bastante más complejos que lo mostrado en la figura. Ambos requieren un proceso más complejo que involucra comunicaciones y operaciones extra. Están explicados con mayor detalle en los apartados correspondientes a

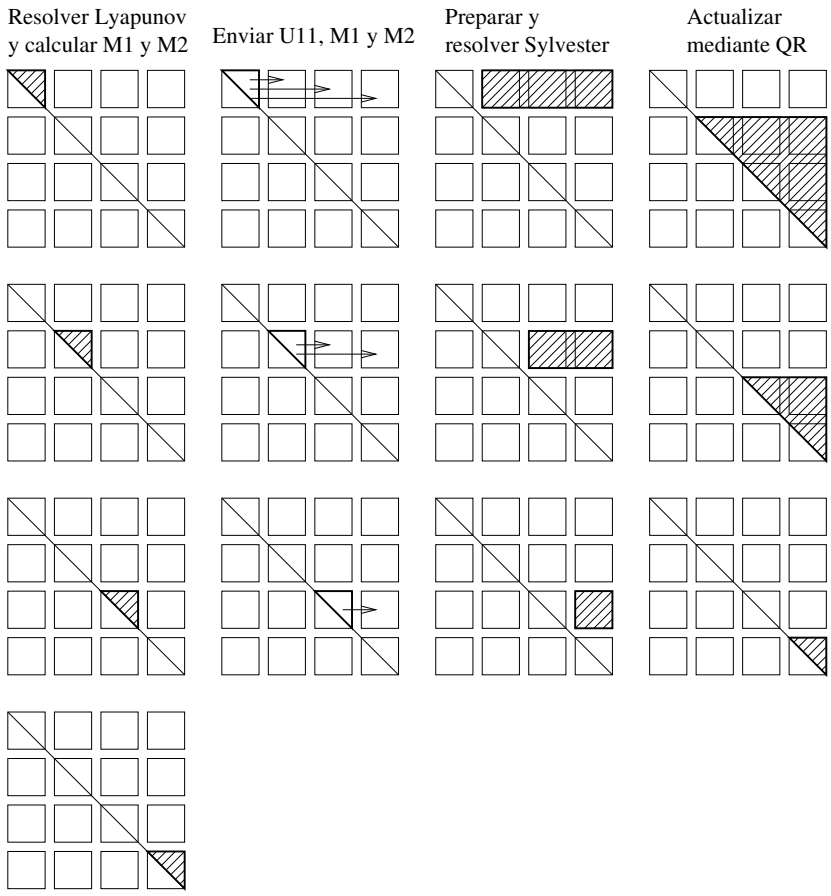

Figura 4.3: Esquema de operaciones en `PSB03OT` (v1) caso continuo no traspuesto con una distribución no cíclica

las rutinas donde están implementados: `PSB03OR`, `PMB04OD` y `PMB04ND`. En esta ocasión no se han incluido con detalle para no complicar excesivamente la descripción de la labor realizada en la rutina `PSB03OT`.

A pesar de que en la figura se ha ilustrado el caso de una distribución no cíclica en aras de una mayor claridad, el algoritmo implementado permite tanto distribuciones cíclicas como no cíclicas. Además, lo habitual será encontrarse una distribución cíclica de las matrices involucradas, ya que esto suele ser más eficiente.

En el caso de una distribución cíclica, la comunicación de las matrices U_{11}, M_1 y M_2 se realizaría en ambas direcciones y no sólo hacia la derecha. En el proceso de preparación y resolución de la ecuación de Sylvester intervendrían todos los procesos de la fila y no sólo los de la derecha de la diagonal. Y en la actualización mediante la QR, se usarían todos los procesos, ya que en todos habrían partes de la matriz

R_{22}.

Esto ocurrirá así en las primeras iteraciones del algoritmo, mientras quede parte de las submatrices con las que se opera en todos los procesos. En las últimas iteraciones de la distribución cíclica, el algoritmo se comportaría de la forma descrita para el caso no cíclico. Pero hasta llegar a ese punto, con la distribución cíclica se consigue mantener más procesos trabajando, logrando así unas mejores prestaciones.

En la figura 4.4 se ilustra el comportamiento en el caso de una distribución cíclica sencilla. Como se trabaja en estos ejemplos con una malla de 4×4 procesos, se ha utilizado una distribución para las matrices donde el tamaño de bloque es $1/8$ del tamaño de las matrices, tanto en filas como en columnas. De esta manera se tiene una distribución cíclica, pero que sólo mantiene 4 bloques de cada matriz en cada proceso. Para una distribución "más cíclica", esto es que divida las matrices involucradas en un mayor número de bloques, la carga se mantendrá más equilibrada y además lo hará durante un tiempo mayor.

Puede observarse como la distribución cíclica afecta al proceso de la forma recientemente descrita. En esta figura sólo se muestran las primeras iteraciones del algoritmo. Para este tamaño de bloque, coincide que lo que queda por mostrar en la figura para terminar el proceso de resolución es equivalente al caso no cíclico que se muestra en la figura 4.3.

`PSB03OT` utiliza las nuevas rutinas `PDTRSCAL` (para el escalado cuando es necesario), `PSB03OR` (para resolver la ecuación reducida de Sylvester) y `PMB04OD`/`PMB04ND` (para las actualizaciones mediante la descomposición QR o RQ).

`PSB03OT` *versión 2*

La versión paralela orientada a bloques de la rutina de SLICOT `SB03OT` presentada en el apartado anterior tiene buenas prestaciones. Sin embargo, pueden aparecer problemas de precisión que afectan a la solución calculada.

En el método de Hammarling hay un paso del algoritmo más sensible al condicionamiento de las matrices implicadas, que puede llevar a que se inestabilice el método. Se trata del cálculo de las matrices auxiliares M_1 y M_2, donde aparece la inversa del fragmento de la matriz solución recientemente calculado U_{11}. Cuando la matriz solución U de la ecuación de Lyapunov es única, que es cuando normalmente se resuelve esta ecuación, U resulta ser una matriz *semidefinida positiva*. El problema viene causado por la posibilidad de que no sea *definida positiva* o que, aún siéndolo, esté próxima a ser una matriz singular. En estos casos, el cálculo de la inversa o no es posible o sufrirá de grandes errores de redondeo.

La implementación en la librería SLICOT del algoritmo original de Hammarling [Ham82] realiza los cálculos de las matrices M_1 y M_2 de una manera muy eficaz (evitando las inversas) para salvaguardar los problemas de precisión que aparecen cuando U_{11} es casi singular. Pero esta forma de proceder resulta eficiente y adecuada sólo al trabajar con matrices M_1 y M_2 de tamaños 1×1 ó 2×2, que son los tamaños con los que se enfrenta el algoritmo secuencial presente en SLICOT. Generalizar a un tamaño $nb \times nb$ esa forma de calcular las matrices auxiliares puede resultar complejo además de poco eficiente. Es por esto que en lo que se ha llamado `PSB03OT` *versión*

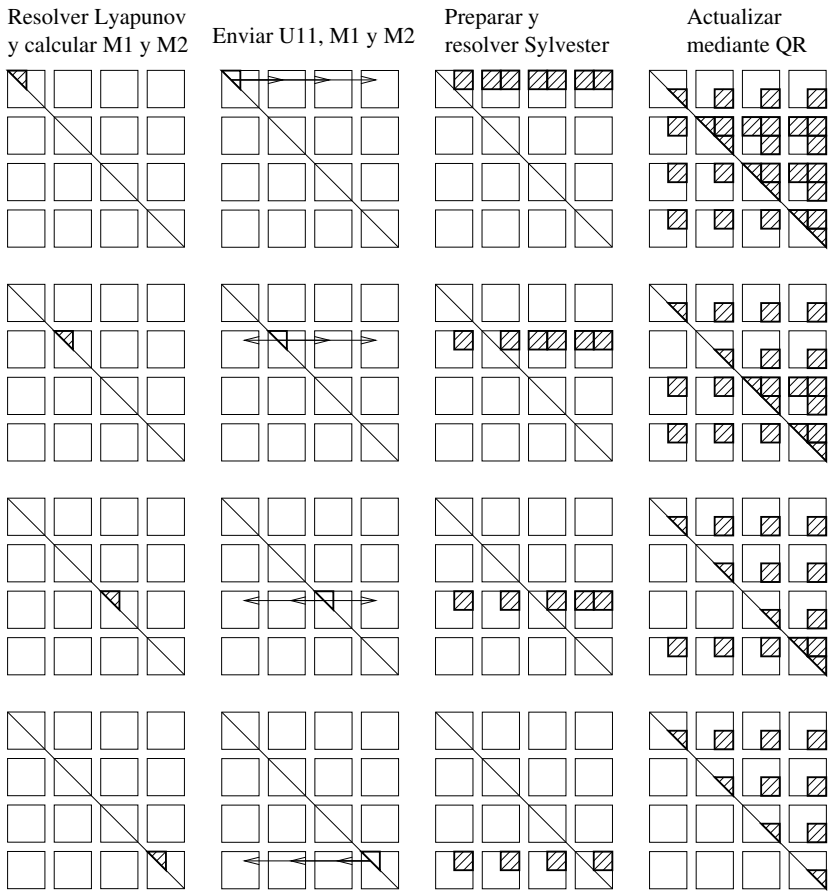

Figura 4.4: Esquema de operaciones en `PSB03OT` (v1) caso continuo no traspuesto con una distribución cíclica

1, presentado en el apartado anterior, no se ha podido prestar un gran cuidado en el cálculo de estas matrices. Se han tomado algunas pequeñas precauciones, como sustituir inversion de matrices por resolución de sistemas triangulares, pero esto no es suficiente. Cuando se utilizan tamaños de bloque reducidos, ese algoritmo funciona bien. Pero a medida que se trabaja con tamaños de bloque mayores, si la solución está próxima a ser una matriz singular, la precisión de la solución calculada va empeorando.

En principio no parece grave esta situación, puesto que se soluciona utilizando tamaños de bloque pequeños. Pero debe notarse que al ejecutar en paralelo suele convenir aumentar en lo posible el tamaño de bloque, ya que así se reducirá el número de comunicaciones necesarias. En realidad existe un compromiso, ya que interesa tamaño de bloque grande por esta razón, pero al mismo tiempo interesa

pequeño para que todas las matrices involucradas en una operación sobre un bloque quepan en memoria caché.

Aunque se pueda trabajar con tamaños de bloque más pequeños, esa implementación no puede ser considerada una buena implementación paralela. La limitación que impone al tamaño de bloque para ofrecer resultados suficientemente precisos no es deseable.

Se ha continuado revisando el diseño del algoritmo paralelo con el objetivo de paliar este problema. Y se ha podido resolver realizando el cálculo de las matrices M_1 y M_2 "exactamente igual" que en el caso secuencial, esto es utilizando matrices M_1 y M_2 de a lo sumo tamaño 2×2.

Con esta nueva versión paralela del método de Hammarling para resolver ecuaciones de Lyapunov, se consiguen las buenas prestaciones obtenidas con la *versión 1*, presentada anteriormente, y precisiones en la solución tan buenas como las del algoritmo secuencial de SLICOT, esto último "independientemente del tamaño de bloque utilizado en la resolución".

Al trabajar con matrices tan pequeñas se perdería la orientación a bloques del algoritmo y con ello las buenas prestaciones. Pero, para evitar esto, se han agrupado las diversas y "diferentes" operaciones que se realizan en el algoritmo secuencial sobre un bloque del tamaño deseado. Se realizan las mismas operaciones que en el caso secuencial, pero en otro orden. Un orden que agrupa las operaciones a realizar según los bloques que se tengan de la matriz.

Para cada elemento de las matrices se consigue que las diferentes operaciones que hay que hacer sobre él se hagan en el mismo orden que en el caso secuencial, es decir en el orden correcto (con otro orden los cálculos no obtendrían la solución buscada). Sin embargo, el orden en que las distintas operaciones se efectúan sobre elementos diferentes no se conserva con respecto al algoritmo secuencial.

En la figura 4.5 se muestra el orden en que se van calculando los elementos de la matriz solución U (de tamaño 8×8 en la figura) con este algoritmo para diferentes tamaños de bloque (8, 4 y 2), suponiendo su ejecución en un único proceso. Por simplicidad, en este ejemplo se supone que todos los valores propios de la matriz S son reales.

El primer orden mostrado, con un tamaño de bloque igual al tamaño de las matrices, se corresponde con el orden seguido en el algoritmo secuencial básico. Como puede apreciarse, este orden es por elementos (suponiendo sólo valores propios reales) de izquierda a derecha y luego de arriba abajo.

El orden utilizado en los algoritmos orientados a bloques propuestos (ambas versiones) sigue este mismo criterio dentro de cada bloque, pero también se aplica en el orden para resolver los distintos bloques al ser ejecutados en un único proceso.

En este problema en concreto se podría usar cualquier orden en el que se cumpla que no se calcula un elemento hasta haber calculado el de su izquierda y el de encima de él. Esto marca las dependencias del problema.

Por ejemplo, si se tuvieran suficientes elementos computacionales, se podrían ejecutar en paralelo muchas de esas operaciones, siempre que se respete el orden marcado por las dependencias existentes entre ellas. El orden al que se llegaría puede verse en la figura 4.6.

nb=8

1	2	3	4	5	6	7	8
	9	10	11	12	13	14	15
		16	17	18	19	20	21
			22	23	24	25	26
				27	28	29	30
					31	32	33
						34	35
							36

nb=4

1	2	3	4	11	12	13	14
	5	6	7	15	16	17	18
		8	9	19	20	21	22
			10	23	24	25	26
				27	28	29	30
					31	32	33
						34	35
							36

nb=2

1	2	4	5	8	9	12	13
	3	6	7	10	11	14	15
		16	17	19	20	23	24
			18	21	22	25	26
				27	28	30	31
					29	32	33
						34	35
							36

Figura 4.5: Orden en el cálculo de la solución en `PSB03OT` ejecutada en un solo proceso para diferentes tamaños de bloque nb

Volviendo al orden realmente implementado, el de la figura 4.5, nótese que cumple las restricciones en el orden de ejecución que acaban de mencionarse. Las diferencias con respecto al código secuencial básico son para agrupar el trabajo por bloques y así minimizar los fallos de caché y el número de comunicaciones a realizar.

Este es el orden que seguirían las dos versiones comentadas de esta rutina al ser ejecutadas en un solo proceso.

La diferencia entre ambas versiones no viene dada por tanto por el orden en el cálculo de la solución, que se mantiene orientado a bloques en ambas, sino por la forma en que esta se calcula dentro de cada bloque no diagonal.

En el caso de la *versión 1*, en los bloques no diagonales la solución se obtiene mediante la resolución de una única ecuación de Sylvester reducida del tamaño dado por el tamaño de bloque, tras haber actualizado mediante sumas y productos de matrices.

En la *versión 2*, el orden utilizado también es el indicado en la figura 4.5. Los

1	2	3	4	5	6	7	8
	3	4	5	6	7	8	9
		5	6	7	8	9	10
			7	8	9	10	11
				9	10	11	12
					11	12	13
						13	14
							15

Figura 4.6: Orden necesario en el cálculo de la solución en `PSB03OT`

bloques no diagonales se obtienen también resolviendo ecuaciones reducidas de Sylvester, pero no con una única ecuación por cada bloque. Ahora se hace exactamente igual que en el caso secuencial (aunque no en el mismo orden para fragmentos de la solución separados en bloques diferentes). En cada bloque, por cada fila a calcular (cada dos filas si aparecen valores propios complejos en S) se resuelve una ecuación reducida de Sylvester y se actualiza hacia abajo (en el mismo bloque) mediante una factorización QR y hacia la derecha (otros bloques) mediante sumas y productos. Procediendo de esta manera, las matrices M_1 y M_2 son a lo sumo de 2×2 evitando así el problema numérico que aparecía en su cálculo para tamaños de bloque grandes. Pero ahora, donde en la *versión 1* simplemente se resolvían ecuaciones reducidas de Sylvester, en la *versión 2* se están realizando múltiples operaciones numéricas: se resuelven ecuaciones reducidas de Sylvester y se hacen descomposiciones QR (o RQ para el caso traspuesto) para ir actualizando las ecuaciones.

En esta nueva versión, en lugar de usar matrices M_1 y M_2 de tamaño $nb \times nb$ (suponiendo nb el tamaño de bloque utilizado), se usan de tamaño 1×1 ó 2×2. Resulta necesario mantener todas estas matrices utilizadas en el cálculo de un bloque diagonal, para ser usadas en el cálculo de los bloques no diagonales de la solución en esa misma fila de bloques. Esto se hace "empaquetando" todas estas matrices juntas para su posterior uso. Esta es la razón por la que se ha tenido que cambiar la rutina `SB03OT`, sustituyéndola por la nueva rutina `SB03OT2` que hace lo mismo pero además guarda todas estas matrices de la manera adecuada, como luego se explicará.

Por esto mismo, se ha cambiado también la rutina `PSB03OR` (a cargo de la resolución de la ecuación reducida de Sylvester en paralelo), que ahora es `PSB03OR2`, para que reciba como argumentos las matrices M_1 y M_2 "empaquetadas". Además, internamente esta rutina ha cambiado bastante, ya que ahora hay que ir resolviendo cada bloque de arriba abajo, actualizando mediante descomposiciones QR (o RQ) como se ha comentado.

Finalmente el código de la rutina `PSB03OT` *versión 2* es muy similar al de la *versión 1*. Los cambios propuestos afectan mucho más a la rutina `PSB03OR`/`PSB03OR2`, donde se debe resolver la ecuación de Sylvester de otra forma para cada bloque. El algoritmo de esta versión coincide con el mostrado anteriormente para la rutina `PSB03OT` *versión 1*, pero llamando ahora a las nuevas rutinas `SB03OT2` y `PSB03OR2`.

Es importante notar que en la ejecución en paralelo de esta rutina, el orden

mostrado en la figura 4.5 no se sigue de forma estricta. Esto es obvio, puesto que si no, se tendría una serialización de las operaciones resultando en un coste similar al secuencial. Cada proceso sí que va calculando las porciones de la solución que posee siguiendo el orden indicado en esa figura. Pero el orden del cálculo de porciones de la solución que se encuentran en diferentes procesos no tiene por qué seguir ese orden, salvo que siempre se van a cumplir las dependencias implícitas en el algoritmo y que se han mostrado en la figura 4.6.

En la implementación de esta y de todas las demás rutinas no se han forzado sincronizaciones adicionales allí donde no son necesarias. Lo que ocurre es que cada proceso va a ir calculando sus fragmentos de la solución siguiendo ese orden de forma local y se sincronizará con otros procesos cuando deba enviarles algún fragmento de la solución o recibirlo para continuar procesando. Ciertamente no sólo se envían porciones de la matriz solución, sino de todas las matrices involucradas en las operaciones, con lo que la sincronización es algo mayor de lo que a priori pudiera parecer, como se verá después en las explicaciones de las rutinas a cargo de la resolución de la ecuación reducida de Sylvester y de la actualización mediante QR.

Esta versión de `PSB03OT` utiliza las nuevas rutinas `SB03OT2` (para resolver la ecuación estándar reducida de Lyapunov guardando las matrices M_1 y M_2), `PDTRSCAL` (para el escalado cuando resulta necesario), `PSB03OR2` (para resolver la ecuación reducida de Sylvester) y `PMB04OD`/`PMB04ND` (para las actualizaciones mediante la descomposición QR o RQ).

4.2.4. `SB03OT2`

```
SB03OT2( DISCR, LTRANS, N, S,LDS, R,LDR, SCALE,
                        M1,LDM1, M2,LDM2, VTAU,LDVTAU,
                        DWORK, INFO )
```

La rutina `SB03OT2` permite resolver la ecuación de Lyapunov estándar en forma reducida tanto en tiempo continuo como en tiempo discreto, ya sea en forma traspuesta o no traspuesta, siempre que sea estable. Es una rutina secuencial.

Esto es lo mismo que hace la rutina de SLICOT `SB03OT`. La diferencia entre ambas rutinas reside en que en `SB03OT2` además se almacenan las matrices intermedias M_1 y M_2 (en `M1` y `M2`) y las transformaciones ortogonales realizadas (en `VTAU`), para su uso posterior según se ha explicado ya anteriormente. Por lo demás, las rutinas se comportan de igual forma.

Los parámetros extra que tiene esta rutina con respecto a `SB03OT` son precisamente para devolver estos resultados intermedios. El código interno es muy similar. Tan sólo se ha añadido lo necesario para almacenar en estos nuevos parámetros los datos que se quieren conservar.

Las matrices intermedias M_1 y M_2 irán siendo de tamaño 1×1 ó 2×2 en función de que se vaya encontrando en la matriz S un valor propio real o un par de valores propios complejos conjugados, respectivamente. Recuérdese que en el caso de ser matrices de 2×2, M_1 es una matriz en forma real de Schur con 2 valores propios complejos conjugados y M_2 es una matriz triangular superior.

Dado que el orden de la matriz S es `N`, para almacenar todas estas matrices basta con tener sendos espacios de `N`×2 ó 2×`N` para los parámetros de salida `M1` y `M2`. En los casos en que aparezca un valor propio real se estará desaprovechando una posición de estas matrices, pero esta posición resultará necesaria cuando sean valores propios complejos.

A la hora de escoger una de las dos posibles configuraciones para los parámetros (`N`×2 ó 2×`N`), en principio daría igual una que otra. Sin embargo, hay más información que resulta conveniente almacenar aquí y que aconseja el utilizar una u otra forma.

Cuando posteriormente se utilicen estas matrices para resolver otras partes de la ecuación, obviamente se tendrá que saber para cada bloque si se trata de un bloque correspondiente a un valor propio real de S (1×1) o correspondiente a un par de valores propios complejos de S (2×2). Esta información puede obtenerse volviendo a examinar los bloques diagonales de la matriz S. Pero, aunque los procesos que necesitarán esta información van a tener copia de las estructuras `M1` y `M2`, en el caso paralelo no tendrán fácil acceso a la parte correspondiente de la matriz S. Es por esto que resulta conveniente incluir de algún modo esta información en estas estructuras de datos, ahorrando así la necesidad de tener que ir a buscarla a la matriz S.

A la hora de incluir esta información, se ha optado por utilizar los huecos existentes en `M2`. En `M2` seguro que hay huecos (espacio no utilizado para almacenar elementos de las matrices M_2). Para cada valor propio real de S habrá una posición no usada. Y para cada par de valores propios complejos también, ya que entonces se tiene una matriz *triangular superior* de 2×2. En `M1` no hay huecos que puedan usarse para otras cosas en el caso de bloques 2×2 correspondientes a pares de valores propios complejos de la matriz S.

La forma de indicar si el bloque actual de S para cada iteración era 1×1 ó 2×2 es la siguiente: En los casos en que M_2 sea una matriz triangular superior de 2×2, se pone el elemento de la fila 2 y columna 1 de M_2 a cero. En los casos en que M_2 conste de un único elemento, se pone a uno el elemento de `M2` que queda libre.

Además, es conveniente que las matrices almacenadas en `M1` y `M2` se encuentren sin trasponer, de forma que puedan utilizarse directamente en otras rutinas. Esto, unido a la forma recién explicada de indicar internamente el tamaño de estas matrices en cada iteración, hace que la estructura donde se guardan las matrices deba tener una forma concreta. En el caso no traspuesto de la ecuación la configuración adecuada resulta ser 2×`N`, siendo `N`×2 en el caso traspuesto.

Para resolver la ecuación de Sylvester posterior, en el caso no traspuesto se recorre la diagonal de (parte de) la matriz S de arriba abajo (de izquierda a derecha), mientras que en el caso traspuesto se recorre la diagonal de abajo arriba (de derecha a izquierda). Es importante que la forma de indicar el tipo de cada bloque en `M2` sea compatible con este orden de recorrido.

Por esto, en el caso no traspuesto se almacenan los bloques en una matriz `M2` de 2×`N`. Posteriormente, a medida que vaya avanzando el algoritmo, se mirará la segunda fila de la columna actual. Si su valor es 1, se trata de un bloque 1×1 (en la primera fila de la columna actual) y si su valor es 0, es una matriz triangular superior 2×2 (la columna actual más la siguiente).

En el caso traspuesto, la estructura usada es de `N`×2. Ahora cuando el algoritmo posterior vaya avanzando de abajo a arriba en esta estructura, se mirará el elemento de la primera columna de la fila actual. Si vale 1, se trata de un elemento (que se encuentra en la segunda columna) y si vale 0, se trata de una matriz triangular superior 2×2 (añadiendo a la fila actual la anterior).

En la figura 4.7 se muestra un ejemplo de estas estructuras para una matriz S de 5×5. Nótese como el utilizar un formato diferente al presentado implicaría una mayor dificultad a la hora de detectar el tamaño del bloque actual (sin acceder a la matriz S).

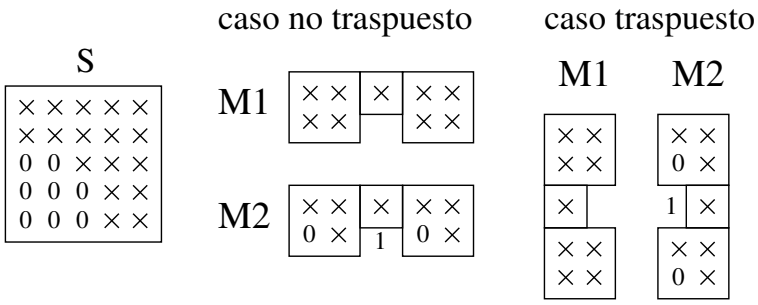

Figura 4.7: Ejemplo de estructura de los bloques `M1` y `M2` para una matriz S de muestra

Aparte de las matrices intermedias M_1 y M_2, en cada paso de resolución de la ecuación se triangulariza la matriz de la parte derecha de la ecuación tras añadirle algunas filas extras (1 ó 2 en función de si se está en un bloque 1×1 ó 2×2). Esta triangularización se corresponde con una descomposición QR, aunque la matriz a transformar ya es muy triangular, con lo que basta con aplicar unas cuantas transformaciones de Householder. Esto ya se vio en la descripción del método de Hammarling, donde se ilustraba esta operación en la ecuación (4.22).

Las transformaciones realizadas deben devolverse al finalizar esta rutina, puesto que se necesitarán para ser aplicadas a los bloques a la derecha del actual (encima de él en el caso traspuesto), en el proceso global de resolución de una ecuación mayor. Estas transformaciones se devuelven en la forma de reflectores de Householder, almacenados de la forma habitual en LAPACK/ScaLAPACK: para cada reflector se almacena un valor `TAU` y un vector `V`.

Se realizan tantas descomposiciones QR como bloques 1×1 ó 2×2 aparezcan en la ecuación a resolver. Para cada descomposición se aplican tantos reflectores como filas/columnas queden en la matriz parte derecha de la porción de ecuación que reste por resolver. Y el vector `V` de cada reflector constará de uno o dos elementos (sin contar el valor 1 que queda fijo al principio del vector) en función del tamaño de cada bloque de S. Este número de reflectores y su tamaño permiten que los diferentes reflectores de Householder se puedan almacenar de forma empaquetada en una matriz con `N`-1 filas y `N` columnas: `VTAU`.

Nuevamente, hay que diferenciar el caso traspuesto de la ecuación del caso no traspuesto. El formato utilizado en `VTAU` es diferente en función del tipo de ecuación, simplemente por mantener los vectores de forma que puedan ser usados posteriormente por las rutinas habituales sin tener que perder tiempo trasponiéndolos. En la figura 4.8 se muestra un ejemplo de cómo quedarían los diferentes reflectores para la matriz S mostrada en el ejemplo anterior, tanto en el caso no traspuesto de la ecuación como en el caso traspuesto. En la figura, $\mathtt{T}_i$ representa el valor de `TAU` para el reflector i y $\mathtt{V}_{ij}$ representa el elemento j del vector `V` usado para el reflector i.

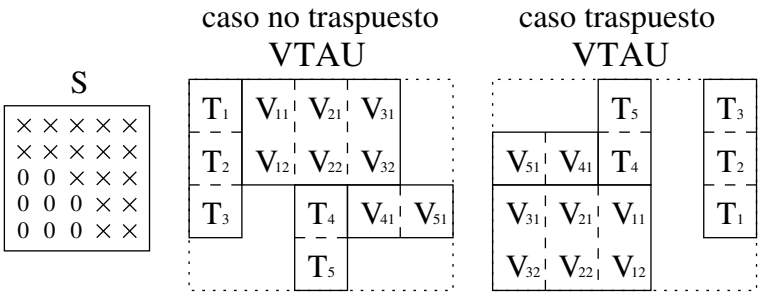

Figura 4.8: Ejemplo de estructura del bloque `VTAU` para una matriz S de muestra

La figura 4.8 puede dar la falsa impresión de que se está desaprovechando espacio en el empaquetamiento que se hace en `VTAU`. Hay huecos sin usar porque aparecen bloques 2×2 de valores propios complejos en la matriz S. Pero `VTAU` tiene que estar preparada para el peor caso, que es cuando todos los valores propios de S son reales. En este caso, se ocuparía todo el espacio disponible, como puede apreciarse en la figura 4.9.

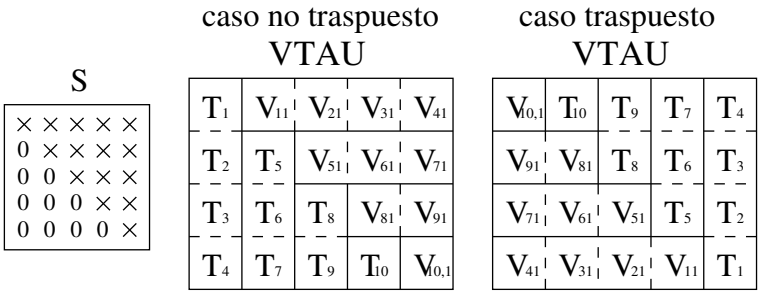

Figura 4.9: Ejemplo de estructura del bloque `VTAU` para una matriz S con todos sus valores propios reales

Aparte de la información que ya se ha comentado, en el caso discreto de la ecuación todavía se guardan más datos.

En el caso discreto, en cada paso de resolución de la ecuación reducida, previamente a la actualización de la parte derecha, se requiere aplicar unas transformaciones de Householder cuando el bloque utilizado de S es de 2×2. Los reflectores correspondientes a estas transformaciones son los que triangularizarían una matriz formada por las mencionadas matrices M_1 y M_2.

Estos reflectores deben aplicarse a un bloque matricial cuyo tamaño (y distribución) coincide con el del bloque de S que resta a la derecha (encima en el caso traspuesto) del bloque diagonal de S utilizado en cada iteración. Esto implica que en el caso paralelo (con matrices distribuidas) se tendrá que aplicar a fragmentos de matrices presentes en otros procesos.

Para evitar la necesidad de recalcular los reflectores (en todos los procesos incluido el que ya los ha calculado, ya que se necesitarían fuera de esta rutina), se ha optado por almacenarlos una vez calculados en un espacio extra en los bloques `M1` y `M2`. Con esto, los bloques `M1` y `M2` duplican su tamaño en el caso discreto.

En la rutina original de SLICOT el cálculo de los reflectores necesarios sólo se hace cuando queda parte derecha por actualizar. En esta nueva versión se hace siempre. Desde dentro de la rutina no se sabe si hay o no más parte derecha por actualizar. Lo que se está resolviendo puede ser (y así será normalmente) un fragmento de la ecuación real y por tanto, muy probablemente, quedará parte derecha (externa a esta rutina) que actualizar.

4.2.5. PDTRSCAL

```
PDTRSCAL( M,N, ALPHA, A,IA,JA,DESCA )
```

La rutina `PDTRSCAL` se utiliza para multiplicar por un escalar una submatriz trapezoidal superior dentro de una matriz distribuida. Es una rutina auxiliar que no se corresponde con la paralelización de ninguna rutina secuencial de BLAS, LAPACK o SLICOT.

Los parámetros `M,N` indican el tamaño de la submatriz trapezoidal a escalar. En función de cuál de los dos parámetros es mayor, se trabaja con una u otra forma para la matriz trapezoidal superior, como puede apreciarse en la tabla 4.1.

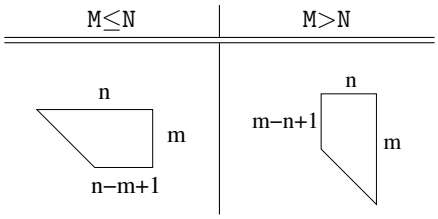

Tabla 4.1: Matrices trapezoidales superiores

Esta operación puede realizarse sin necesidad de comunicar ningún dato y así se ha implementado. Cada proceso escala el fragmento de submatriz que posee.

4.2.6. PSB03OR2

Las rutinas PSB03OR/PSB03OR2 son las encargadas de resolver la ecuación *reducida* de Sylvester que aparece en el paso 2 de la resolución de la ecuación reducida de Lyapunov por el método de Hammarling (ver sección 4.1.3). Se trata de la ecuación *reducida* de Sylvester porque las matrices que aparecen como coeficientes a la izquierda de la ecuación están en forma real de Schur.

Estas rutinas permiten resolver versiones continuas y discretas de la ecuación, traspuestas o sin trasponer. Son una aproximación paralela de la rutina SB03OR de SLICOT, pero no van enfocadas a la resolución genérica de ecuaciones con esta forma. Van dirigidas a la resolución de este tipo de ecuaciones dentro del contexto de la resolución de ecuaciones de Lyapunov por el método de Hammarling. A este respecto, sólo una de las matrices de entrada tiene distribución genérica. El resto de matrices ya se esperan con una distribución concreta, que se corresponde con la que tienen en la rutina que llama a esta en el proceso de resolución de ecuaciones de Lyapunov (PSB03OT).

La ecuación reducida de Sylvester en tiempo continuo y su versión traspuesta tienen la forma

$$S^T X + XA = \alpha \cdot C,$$
$$SX + XA^T = \alpha \cdot C,$$

y en tiempo discreto y su versión traspuesta, la forma

$$S^T XA - X = \alpha \cdot C,$$
$$SXA^T - X = \alpha \cdot C.$$

El valor α se corresponde con el parámetro SCALE de la rutina, utilizado para evitar un posible desbordamiento en la solución. S y A son matrices en forma real de Schur (por esto son versiones reducidas de la ecuación de Sylvester). La matriz S es una matriz cuadrada habitualmente bastante más grande en tamaño que la matriz A, también cuadrada. De hecho, la matriz S va a estar distribuida entre los distintos procesos, mientras que la matriz A se encontrará replicada en los procesos involucrados, gracias a su menor tamaño.

Esta rutina es un paso importante dentro de la resolución de la ecuación reducida de Lyapunov. Además, se ve muy influenciada por cuál de las dos versiones explicadas de la rutina PSB03OT se utilice. Por esto se van a describir dos rutinas PSB03OR y PSB03OR2, que son las encargadas de realizar esta tarea con la versión 1 de PSB03OT y con la versión 2 de esta misma rutina, respectivamente.

Igual que ocurría con PSB03OT, al trabajar con matrices en forma real de Schur que se encuentran distribuidas vuelven a aparecer problemas de cortes en bloques 2×2 que deben resolverse sin cortar. De la misma forma como se hizo en PSB03OT,

en estas rutinas se ha prestado una especial atención a la aparición de estos casos resolviéndolos adecuadamente.

A continuación se pasa a describir por separado las dos versiones de esta rutina encargada de la resolución de ecuaciones reducidas de Sylvester.

`PSB03OR` ***versión 1***

```
PSB03OR( DISCR, LTRANS, N, M, S,IS,JS,DESCS, A,LDA, C,LDC, Y,LDY,
         SCALE, DWORK, INFO )
```

La rutina `PSB03OR` es la encargada de resolver las ecuaciones reducidas de Sylvester que aparecen en la resolución de la ecuación reducida de Lyapunov que se efectúa en la rutina `PSB03OT` *versión 1*.

En la figura 4.10 puede verse la forma que tienen las matrices involucradas en la ecuación para el caso continuo no traspuesto.

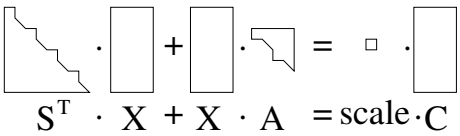

Figura 4.10: Forma de las matrices en la ecuación reducida de Sylvester de tiempo continuo no traspuesta

Esta ecuación (caso continuo no traspuesto) se resuelve procediendo de arriba abajo por bloques de filas de X, resolviendo para cada bloque la siguiente ecuación reducida de Sylvester:

$$S_{kk}^T X_k + X_k A = C_k - R_k,$$
$$R_k = \sum_{j=1}^{k-1} S_{jk}^T X_j.$$

Se utiliza un particionado en bloques tal que la ecuación reducida de Sylvester a resolver para cada bloque tenga todas las matrices involucradas en un único proceso.

La matriz S que aparece en la ecuación tiene un tamaño genérico. Se encuentra normalmente distribuida entre múltiples procesos. Sin embargo, la matriz A, que habitualmente es de un tamaño mucho más reducido, se encuentra replicada en los procesos que la van a necesitar. Anteriormente a la llamada a esta función ya se habrá difundido a este conjunto de procesos: una fila de la malla de procesos en el caso no traspuesto o una columna en el caso traspuesto. De esta manera, para actualizar sólo resulta necesario ir comunicando la matriz S y fragmentos de la solución que se va calculando X. La resolución de cada bloque se hace de forma local en el proceso que lo tiene.

La matriz C, parte derecha de la ecuación, se sobrescribe con la solución. Esta matriz, y por tanto también la solución, se almacena traspuesta en el caso no traspuesto de la ecuación. En ambos casos de la ecuación, esta matriz es de tamaño `N` por `M`, con `M` habitualmente pequeña ya que es del orden del tamaño de bloque usado en la distribución de las matrices. Sin embargo, por la forma en que se resuelve la ecuación en cada caso, conviene trabajar con una versión traspuesta de C en el caso no traspuesto. Con esto se consigue que en ambos casos la matriz C quede distribuida coherentemente con la distribución de los bloques de S que se necesitarán en cada paso de resolución, minimizando así las comunicaciones necesarias. Afortunadamente esta misma distribución es también la que resulta más adecuada en el trabajo posterior con la solución de la ecuación de Sylvester en la rutina `PSB03OT`, donde conviene que coincida con la distribución de su submatriz R_{12}.

El parámetro `Y`, con igual distribución que `C`, se utiliza solamente en el caso discreto. En este caso, en la rutina `PSB03OT` se necesita posteriormente el resultado de la expresión $X^T \cdot S$ (sumado a otra expresión que se calcula previamente en `PSB03OT` y se pasa a esta rutina en `Y`). Esta expresión es más eficiente calcularla dentro de esta rutina. Aquí se puede hacer todo mediante operaciones locales, sin comunicaciones extra. Se aprovecha que la matriz S se va recibiendo en una fila de procesos (columna en el caso traspuesto) durante la labor de resolución de la ecuación de Sylvester. Además es una operación que se realizaría de todas formas, al ser necesaria para ir actualizando la parte derecha de la ecuación de Sylvester. Se ha añadido lo necesario para ir almacenando el resultado y así evitar recalcularla en el futuro, lo que supondría repetir comunicaciones y cálculos.

A continuación, se ilustra en forma de algoritmo las operaciones que se realizan en esta rutina para el caso continuo no traspuesto. El caso traspuesto es muy similar al presentado, salvo por el orden en que se va calculando la solución y por lo tanto la forma de realizar las comunicaciones.

Algoritmo 4.3 `PSB03OR` *(caso continuo no traspuesto)*

1. *Determinar la* **fila de trabajo**. *La* **fila de trabajo** *es la fila de la malla de procesos en la que se van a hacer todos los cálculos. Se corresponde con la fila de procesos donde se acaba de resolver un fragmento de la ecuación de Lyapunov reducida. Además es en esta fila donde se dejará la solución.*

2. *Recorrer en todos los procesos la diagonal de la matriz S de arriba abajo (de izquierda a derecha), calculando la solución X por bloques de filas. Para cada bloque:*

   a) *Determinar el tamaño de bloque máximo a utilizar tal que el bloque diagonal correspondiente de S esté completamente almacenado en un único proceso, usando como mucho el tamaño de bloque de la distribución.*

   b) *Llevar a la* **fila de trabajo** *el bloque de filas actual de S.*

   c) *Si este bloque de S no es el último, hay que determinar si con este tamaño de bloque se corta un bloque 2×2 de valores propios complejos de la matriz*

S. Para que todos los procesos sepan si se da este corte o no, se requiere que el proceso donde se encuentra el elemento de S justo debajo del último elemento del bloque diagonal actual envíe este valor a todos los demás. Si este valor es no nulo, utilizar este tamaño de bloque implicaría cortar un bloque 2×2 de S, cosa que no debe ocurrir. Si se da esta situación, aumentar en una fila los bloques de X y S con los que se va a trabajar. Esto implica:

1) *Traer a la* **fila de trabajo** *la última fila del bloque actual de S.*
2) *Traer al proceso donde se va a calcular un bloque de la solución X la última columna del correspondiente bloque diagonal S y la última fila del bloque equivalente de C. Ambas están en el mismo proceso, ya que se está trabajando con una versión traspuesta de C y de X.*

d) *Resolver el bloque actual de la ecuación de Sylvester. Para calcular este bloque se tiene que resolver una ecuación reducida de Sylvester completamente almacenada en un único proceso. Se sigue un esquema similar al utilizado para toda la ecuación, pero ahora en un único proceso, resolviendo el bloque por columnas y así utilizando la rutina auxiliar* `SB03OR` *al igual que en el caso secuencial. Se va calculando de 1 en 1 columna o de 2 en 2, según se vaya encontrando un valor propio real o un par de valores propios complejos en la matriz A.*

e) *Difundir el bloque de solución recién calculado junto al factor de escalado utilizado a toda la* **fila de trabajo**.

f) *Si se aumentó en una fila/columna el bloque diagonal de S para evitar un corte de un bloque 2×2, ahora se debe llevar la última fila del fragmento de la solución X recién calculada a su lugar correspondiente.*

g) *Si se ha producido escalado, escalar la solución previamente calculada de X y la porción de la matriz C a utilizar posteriormente para seguir calculando X y acumular el escalado en el factor de escala global.*

h) *Actualizar la parte derecha de la ecuación utilizando el bloque recién calculado de la solución. Esto implica la resta de un producto de matrices, operaciones que pueden hacerse de forma local en cada proceso de la* **fila de trabajo**, *aprovechando que ya se ha traido aquí la parte necesaria de la matriz S.*

En la figura 4.11 se muestra el esquema de operaciones utilizado en la rutina, prestando especial interés a las comunicaciones que se efectúan. En esta figura se utiliza una distribución de datos no cíclica para que resulte más sencilla su interpretación.

En la siguiente figura, la 4.12, se muestra el mismo caso con una distribución cíclica. Aquí puede apreciarse como la carga se mantiene mejor equilibrada durante más tiempo al involucrar el trabajo a más procesos. Al final del esquema de esta figura, quedaría por hacer el esquema mostrado en la figura 4.11 que también se corresponde con las últimas etapas del caso cíclico.

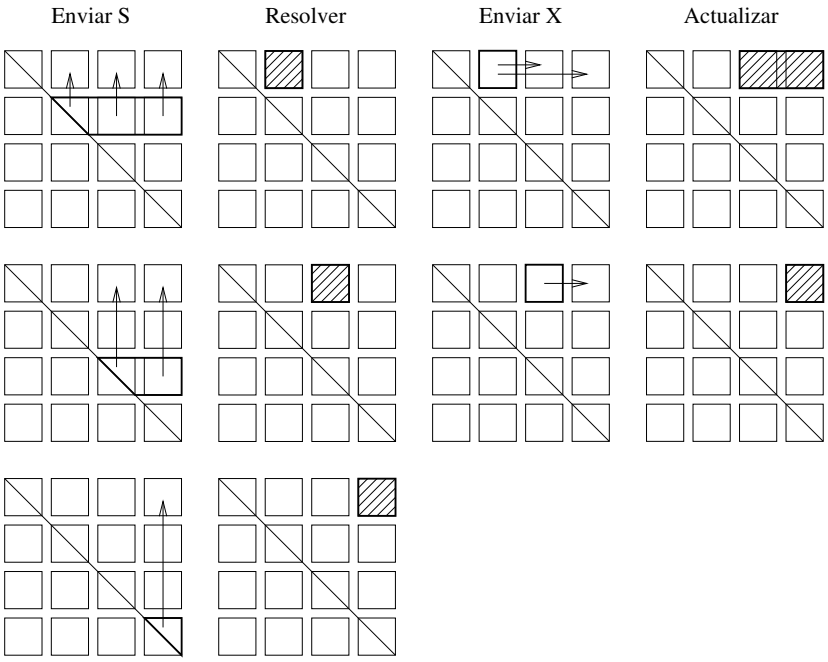

Figura 4.11: Esquema de operaciones en PSB03OR caso continuo no traspuesto con una distribución no cíclica

En cualquier caso, en ambos esquemas puede apreciarse que esta rutina tiene un grado de paralelismo importante pero concentrado en una única fila de procesos (una columna en el caso traspuesto). Todas las operaciones numéricas se realizan en esta fila. El resto de procesos sólo realizan comunicaciones de los datos necesarios de la matriz S. El mayor paralelismo se alcanza en la fase de actualización de la ecuación, que afortunadamente ocupa la mayor parte del tiempo.

PSB03OR2 ***versión 2***

```
PSB03OR2( DISCR, LTRANS, N, M, S,IS,JS,DESCS, M1,LDM1, M2,LDM2,
          VTAU,LDVTAU, C,LDC, Y,LDY, SCALE, DWORK, INFO )
```

La rutina PSB03OR2 es la encargada de resolver las ecuaciones reducidas de Sylvester que aparecen en la resolución de la ecuación reducida de Lyapunov que se efectúa en la rutina PSB03OT *versión 2*.

Esta rutina comparte la mayoría de las características explicadas para su versión anterior: la rutina PSB03OR. Sirve para resolver el mismo tipo de ecuaciones dentro del mismo ámbito de aplicación.

En la presentada como segunda versión de las rutinas PSB03OT y PSB03OR se ha vuelto a los inicios. Se resuelven muchas ecuaciones de Sylvester de pequeño

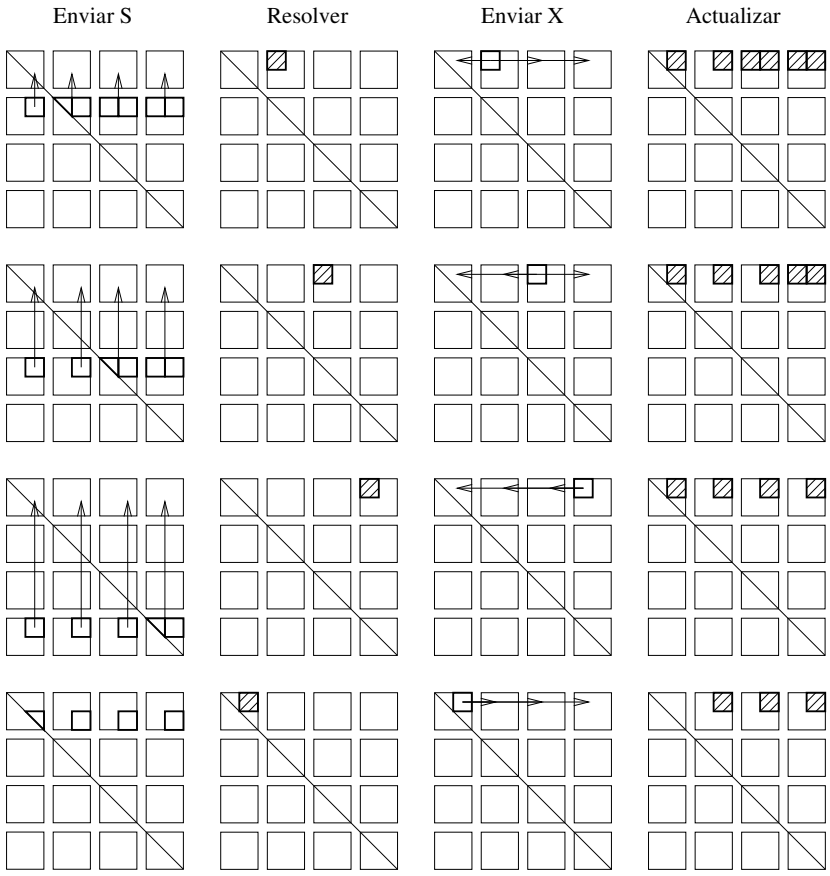

Figura 4.12: Esquema de operaciones en `PSB03OR` caso continuo no traspuesto con una distribución cíclica

tamaño. Pero las múltiples resoluciones de estas ecuaciones se agrupan de forma que se consigue un algoritmo orientado a bloques y el número de comunicaciones requerido es menor.

Para realizar esto, se ha tenido que trasladar las operaciones de preparación de cada una de estas ecuaciones de Sylvester de menor tamaño a dentro de esta rutina. Antes estaban en la rutina que resolvía la ecuación de Lyapunov. Es por esto que a esta nueva versión de la rutina para resolver ecuaciones de Sylvester se le pasan más argumentos con información proveniente de la resolución de los bloques diagonales de la ecuación de Lyapunov. Con estos datos y el resultado de cada ecuación de Sylvester, esta rutina va actualizando el resto de ecuaciones de Sylvester que va resolviendo. Desde fuera, aparenta que se está resolviendo una única ecuación de Sylvester del tamaño de un bloque de la matriz, pero internamente esto se traduce

en muchas resoluciones de ecuaciones de Sylvester de a lo sumo orden 2. Ahora se hace igual que en la rutina secuencial.

Al final, el algoritmo obtenido es muy similar, si se mira a grandes rasgos, al usado en la rutina PSB03OR *versión 1*. Lo mismo sucede con el esquema de operaciones y comunicaciones (figuras 4.11 y 4.12). El cambio más notable afecta a la forma en que se preparan y posteriormente se actualizan las partes derechas de las sucesivas ecuaciones de Sylvester que se van resolviendo con llamadas a SB03OR.

La parte derecha de la ecuación de Sylvester que se resuelve en esta rutina ahora no se pasa completamente calculada. Esto sí se hacía en la versión anterior. Pero ahora aún le falta ser actualizada de forma previa a la resolución. Esta actualización se realiza inmediatamente antes de llamar a la rutina SB03OR para ir resolviendo fragmentos de la ecuación. Se utilizan las matrices M_1 y M_2 que se obtuvieron en la resolución de la ecuación reducida de Lyapunov mediante la rutina SB03OT2 y que están convenientemente empaquetadas en los argumentos M1 y M2. De esta manera se consigue que el proceso sea mucho más parecido al de la implementación secuencial.

De igual modo, también se ha modificado la actualización justo después de la llamada a SB03OR. Se ha añadido lo necesario para ir calculando el argumento Y que se utiliza posteriormente en la rutina PSB03OT que ha llamado a esta. También se actualiza un fragmento de la parte derecha de la ecuación, utilizando para ello la información contenida en el argumento VTAU rellenada también en SB03OT2.

Por último, también se ha visto afectada la actualización de la parte derecha que se efectúa tras resolver por completo un bloque de la solución (mediante varias llamadas a SB03OR). Esta actualización es ahora idéntica para el caso discreto y para el caso continuo, incluso más simple en el caso discreto que la que se hacía en PSB03OR *versión 1*. Esto es porque en el caso discreto parte de la actualización se ha tenido que trasladar a justo antes de llamar a SB03OR, como se acaba de comentar, en pro de asemejarse al caso secuencial. A pesar del cambio, se sigue concentrando un alto grado de paralelismo en esta actualización de la parte derecha de la ecuación, como ya se comentó en la descripción de la rutina PSB03OR *versión 1*.

Con todo esto, al final se tiene una rutina que realiza su cometido en cuanto a resolver la ecuación de Sylvester necesaria para resolver ecuaciones de Lyapunov, pero haciéndolo mediante una secuencia de operaciones muy similar a la utilizada en las rutinas secuenciales de SLICOT para este proceso. Así se consigue obtener unos resultados similares en cuanto a precisión. Pero además se ha reestructurado todo tratando de fomentar unas buenas prestaciones paralelas.

4.2.7. PMB04OD

```
PMB04OD( UPLO, N, M, P, R,IR,JR,DESCR,
         A,LDA, B,LDB, C,LDC, TAU, DWORK )
```

La rutina PMB04OD calcula la descomposición QR de una matriz rectangular

$$\begin{pmatrix} R \\ A \end{pmatrix}$$

de tamaño $(n + p) \times n$ cuya submatriz R de $n \times n$ ya está en forma triangular superior (ver figura 4.13). Para ello, aplica n transformaciones de Householder que van anulando cada una de las n columnas de A.

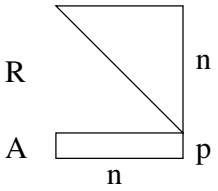

Figura 4.13: Forma de las matrices de entrada de `PMB04OD`

Esta rutina es una versión paralela simplificada de la rutina de SLICOT `MB04OD`. Es una versión *simplificada* en el sentido de que ofrece menos funcionalidad que la de la rutina secuencial. Esto es así porque se ha implementado sólo lo necesario para su uso por parte de la rutina `PSB03OT`.

La forma en que se anula la matriz A preservando la triangularidad existente en la matriz R es mediante la sucesiva aplicación de n transformaciones de Householder. Cada transformación de Householder anula los elementos de una columna de A y altera el resto de elementos de A y una fila de R.

Estas operaciones se han agrupado por bloques de ambas matrices para la obtención de un algoritmo paralelo eficiente. El algoritmo resultante se resume a continuación:

Algoritmo 4.4 `PMB04OD`

1. *Determinar la fila de procesos donde está la primera fila de la matriz R. Esta fila de procesos, que denominaremos* **primera fila***, realizará las comunicaciones de forma diferente al resto.*

2. *Enviar a esta fila la matriz A, si no está ya en ella.*

3. *Recorrer en todos los procesos la matriz R por bloques diagonales de izquierda a derecha (de arriba abajo). Para cada bloque de R, anular el bloque de A en sus mismas columnas y actualizar consecuentemente el resto de R y A:*

   a) *Determinar la* **fila de trabajo***, que será la fila de procesos en la que se va a trabajar para este bloque de R. Es la fila donde está el bloque diagonal actual.*

   b) *Si la* **fila de trabajo** *coincide con la* **primera fila***, recibir en ella toda la matriz A que queda por anular, a menos que ya se tenga aquí.*

   c) *Recorrer la matriz A por bloques de izquierda a derecha haciendo para cada bloque:*

      1) *Si no se tiene el bloque actual de A, recibirlo de la fila anterior de procesos.*

2) Si es el primer bloque de A y por tanto hay que anularlo:

a′ *Anular el bloque actual de A mediante transformaciones de Householder, que también modifican el bloque actual de R.*

b′ *Difundir los reflectores de Householder utilizados a toda la* **fila de trabajo**.

3) Si no:

a′ *Actualizar el bloque actual de A utilizando los reflectores recibidos, actualizando al mismo tiempo el correspondiente bloque de la matriz R.*

b′ *Enviar hacia abajo, a la siguiente fila de procesos, el bloque actual de A ya actualizado. Este envío no se hace si la fila siguiente es la* **primera fila**.

d*) En la fila de procesos anterior a la* **primera fila**, *enviar a la* **primera fila** *toda la parte de A que queda por anular y ha sido actualizada.*

Puede verse un esquema de las operaciones efectuadas suponiendo una distribución no cíclica en la figura 4.14.

Se presenta el caso no cíclico porque permite apreciar mejor las distintas operaciones a realizar. Sin embargo, en el caso no cíclico no se obtiene un correcto solapamiento de operaciones que permitiría mejorar más las prestaciones.

Si se trabaja con una distribución de matrices cíclica, a cada proceso le corresponden múltiples bloques de cada matriz. Esto permite que tras operar con un bloque se pueda comunicar inmediatamente el resultado obtenido a la siguiente fila de procesos, que podrá realizar operaciones mientras se continúa con el resto de bloques. La siguiente fila procederá de igual modo y así pueden llegar a tenerse múltiples filas de procesos calculando simultáneamente.

En la figura 4.15 puede verse el esquema de operaciones, ahora aplicado en un caso con distribución cíclica. Se trata de un caso "poco cíclico" en el que cada proceso tiene cuatro bloques (2×2 bloques) de cada matriz. Después de las operaciones mostradas en el esquema, se continuaría con la figura presentada para el caso no cíclico (figura 4.14) para terminar el proceso.

Puede observarse que en la parte descrita como "Calcular reflectores" se solapan en el tiempo la operación de "cálculo de reflectores" por parte de un proceso con la operación de "aplicación de reflectores" de procesos de filas anteriores. Esto aumenta el número de procesos trabajando a la vez, mejorando el grado de paralelismo del algoritmo.

Además, esta mejora aumenta a medida que se trabaja con casos más reales, es decir con una distribución más cíclica. Cuantos más bloques de las matrices tenga cada proceso, más trabajo se hará en paralelo con el resto de procesos.

Para ilustrar esto, se ha añadido la figura 4.16 que representa el caso en que cada proceso tiene 4×4 bloques de cada matriz.

Puede apreciarse un mayor número de procesos trabajando a la vez. Cuantos más bloques tenga cada proceso, más paralelismo podrá explotarse en esta rutina. Tal como se ha implementado, si se tienen bloques suficientes en cada proceso,

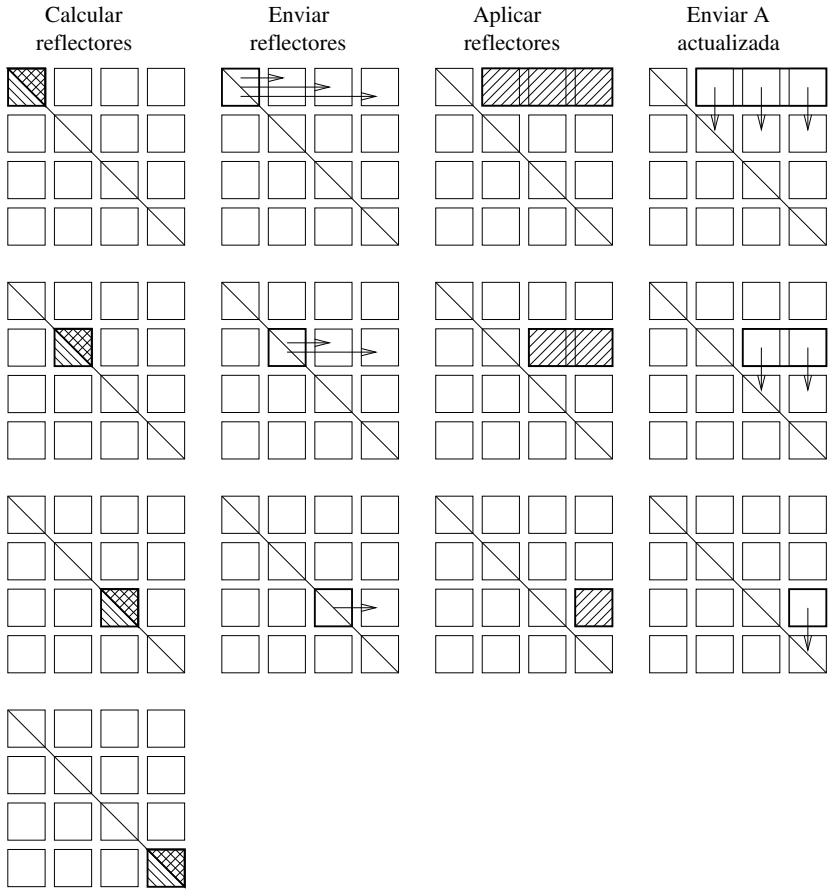

Figura 4.14: Esquema de operaciones en `PMB04OD` con una distribución no cíclica

múltiples filas de procesos podrán empezar a aplicar reflectores mientras la primera no habrá terminado con los suyos.

En las figuras se han agrupado las fases de comunicación y de cálculo de diferentes procesos en una misma columna, representando que suceden al mismo tiempo. Sin embargo, en la realidad no será exactamente así. No se ha incluido ningún tipo de sincronización para que ocurra esto, ya que sería contraproducente. En realidad cada proceso va a ir realizando sus operaciones independientemente del resto, hasta que necesite comunicarse con algún otro. En ese momento si debe recibir algo, necesariamente quedará a la espera de que se le envíe.

Esto implica que el esquema real de operaciones será similar pero no idéntico al mostrado en las figuras. Será fácil que se solapen comunicaciones de unos procesos con cálculos de otros. Tal como está implementado, cada proceso irá tan rápido como

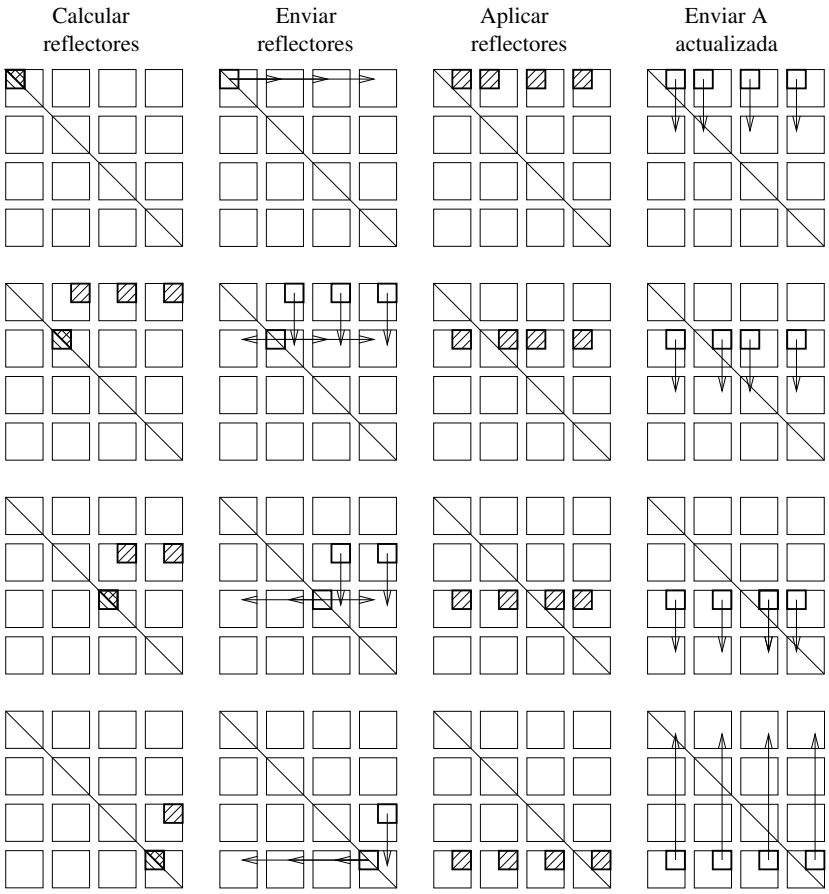

Figura 4.15: Esquema de operaciones en `PMB04OD` con una distribución cíclica

pueda realizando sus cálculos y comunicaciones, tan sólo parando en las recepciones de datos y en los envíos si son síncronos.

La implementación realizada favorece el solapamiento entre cálculo y comunicaciones, pero garantiza el correcto funcionamiento incluso si algunos envíos se hicieran síncronos.

Esto guarda relación con lo que en el algoritmo se ha presentado como *primera fila*, de la que se ha dicho que realiza las comunicaciones de forma diferente al resto. Es precisamente para evitar un interbloqueo que podría ocurrir en el caso de ejecutar el algoritmo en un sistema con comunicaciones síncronas, ya sea porque realmente sean síncronas o porque en un determinado momento se comporten así por llenarse los *buffers* de envío de mensajes.

Si las comunicaciones se comportasen de forma síncrona y no se hubiera tenido

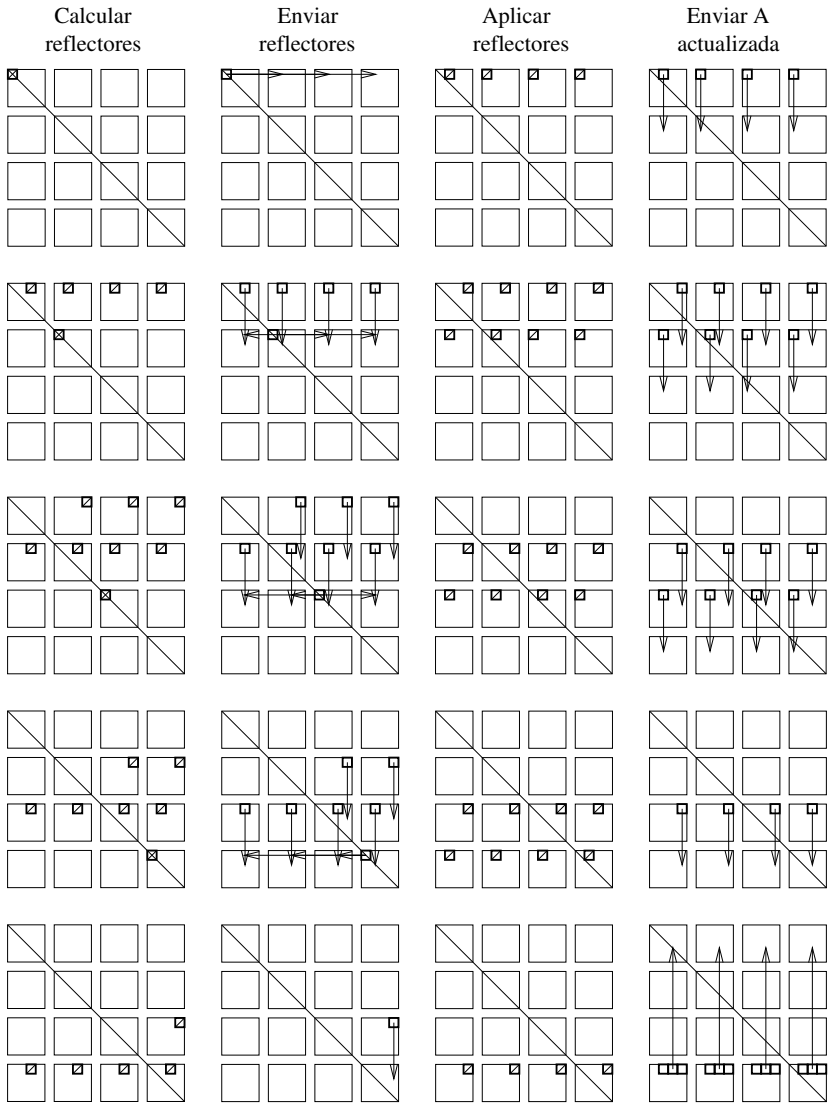

Figura 4.16: Esquema de operaciones en `PMB04OD` con una distribución cíclica que deja más bloques de cada matriz en cada proceso

en cuenta, el sistema podría llegar a un interbloqueo cuando se trabajase con menos filas de procesos que bloques de cada matriz se tienen en un proceso. Esto puede verse en la figura 4.17. Esta figura representa lo que pasaría en una malla de 3×3 procesos con las matrices distribuidas cíclicamente de forma que se tengan 4×4 bloques de cada matriz en cada proceso. Con esta configuración, si las comunicaciones se realizasen de igual forma en todos los procesos, se llegaría a una situación de interbloqueo. Se corresponde con la última viñeta de la figura. En ella todos los procesos quieren enviar y si las comunicaciones son síncronas, se quedarían bloqueados a la espera de que el proceso destino de sus comunicaciones empiece la recepción. Esto no pasará porque el proceso destino también está esperando a otro proceso para enviarle datos.

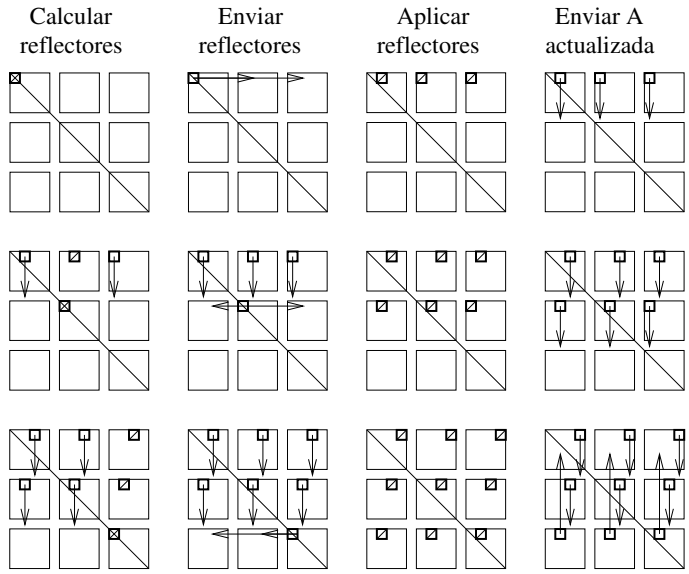

Figura 4.17: Interbloqueo a evitar en `PMB04OD`

Sin embargo, esta situación no va a darse con el algoritmo implementado. Se ha tenido en cuenta su existencia y se ha optado por realizar las comunicaciones de forma asimétrica para evitar que pueda producirse este interbloqueo. Como se ha mostrado en el algoritmo, lo que se hace es que una de las filas de procesos no realice sus envíos hasta haber terminado por completo de trabajar con la fila de bloques actual de las matrices. De esta manera, esta fila nunca quedará a la espera de enviar algo, permitiendo que el resto de filas puedan avanzar y nunca queden bloqueadas.

A modo de ejemplo, en la figura 4.18 se muestra cómo se comportaría el algoritmo implementado en la misma situación que se ha ilustrado en la figura 4.17. El algoritmo empieza de la forma mostrada en la figura 4.17, pero la última viñeta de esta figura no llega a producirse. En su lugar se proseguiría de la forma mostrada

en la figura 4.18. Luego se continuaría con normalidad con la anulación del resto de bloques de la matriz A.

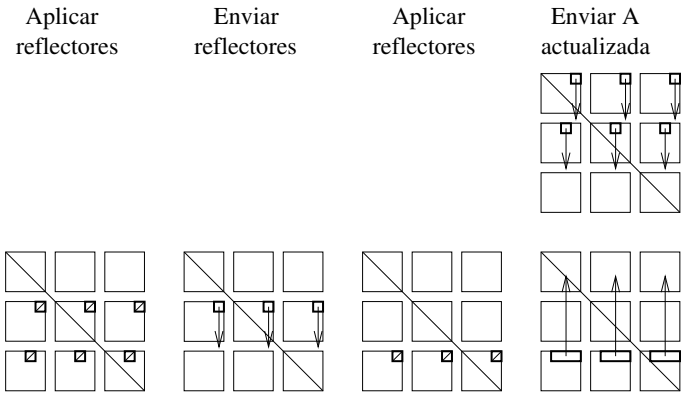

Figura 4.18: Esquema de operaciones de `PMB04OD` con comunicaciones síncronas

Por tanto, aún suponiendo comunicaciones síncronas, el algoritmo no llegará a interbloquearse. Con comunicaciones asíncronas, además de no bloquearse, se tendrán mejores prestaciones al tener a más procesos trabajando a la vez.

4.2.8. `PMB04ND`

```
PMB04ND( UPLO, N, M, P, R,IR,JR,DESCR,
         A,LDA, B,LDB, C,LDC, TAU, DWORK )
```

La rutina `PMB04ND` calcula la descomposición RQ de una matriz rectangular $(A\ R)$ de tamaño $n \times (p+n)$ cuya submatriz R de $n \times n$ ya está en forma triangular superior (ver figura 4.19). Para ello, se aplican n transformaciones de Householder que van anulando cada una de las n filas de A.

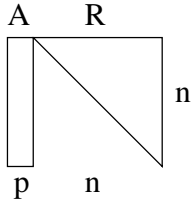

Figura 4.19: Forma de las matrices de entrada de `PMB04ND`

Esta rutina es parecida a lo que se obtendría con una versión traspuesta de la rutina anterior (`PMB04OD`). De hecho, el algoritmo que se ha implementado está fuertemente basado en el anterior. Tanto el algoritmo como los esquemas presentados

son válidos para esta rutina, si se intercambian filas por columnas en las operaciones y comunicaciones.

También se da la misma problemática en el caso de trabajar con operaciones de comunicación síncronas. Y se ha resuelto del mismo modo.

El algoritmo de esta rutina queda así:

Algoritmo 4.5 `PMBO4ND`

1. *Determinar la columna de procesos donde está la última columna de la matriz R. Esta columna de procesos, que denominaremos* **última columna***, realizará las comunicaciones de forma diferente al resto.*

2. *Enviar a esta columna la matriz A, si no está ya en ella.*

3. *Recorrer en todos los procesos la matriz R por bloques diagonales de derecha a izquierda (de abajo arriba). Para cada bloque de R, anular el bloque de A en sus mismas filas y actualizar consecuentemente el resto de R y A:*

   a) *Determinar la* **columna de trabajo***, que será la columna de procesos en la que se va a trabajar para este bloque de R. Es la columna donde está el bloque diagonal actual.*

   b) *Si la* **columna de trabajo** *coincide con la* **última columna***, recibir en ella toda la matriz A que queda por anular, a menos que ya se tenga aquí.*

   c) *Recorrer la matriz A por bloques de abajo arriba haciendo para cada bloque:*

      1) Si no se tiene el bloque actual de A, recibirlo de la columna siguiente de procesos.

      2) Si es el bloque inferior de A y por tanto hay que anularlo:

         a′ *Anular el bloque actual de A mediante transformaciones de Householder, que también modifican el bloque actual de R.*

         b′ *Difundir los reflectores de Householder utilizados a toda la* **columna de trabajo**.

      3) Si no:

         a′ *Actualizar el bloque actual de A utilizando los reflectores recibidos, actualizando al mismo tiempo el correspondiente bloque de la matriz R.*

         b′ *Enviar hacia la izquierda, a la columna anterior de procesos, el bloque actual de A ya actualizado. Este envío no se hace si la columna anterior es la* **última columna**.

   d) *En la columna de procesos siguiente a la* **última columna***, enviar a la* **última columna** *toda la parte de A que queda por anular y ha sido actualizada.*

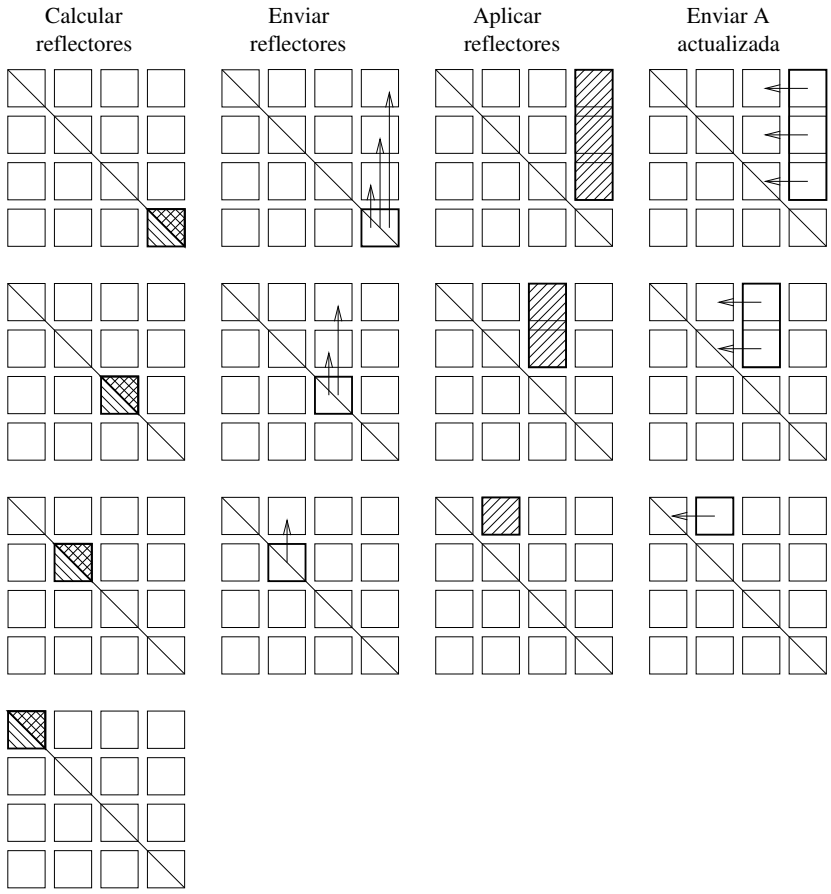

Figura 4.20: Esquema de operaciones en `PMB04ND` con una distribución no cíclica

Puede verse un esquema de las operaciones efectuadas suponiendo una distribución no cíclica en la figura 4.20.

En la figura 4.21 puede verse el correspondiente esquema de operaciones en el caso de una distribución cíclica. Al finalizar estas operaciones se continuaría de la forma mostrada en la figura 4.20, que muestra el caso no cíclico.

Es fácil observar la gran semejanza entre estas gráficas y las correspondientes versiones de la rutina `PMB04OD`: las figuras 4.14 y 4.15.

Lo mismo sucede con el resto de gráficas mostradas para la rutina `PMB04OD`. Es trivial aplicar a ellas los cambios notados en las dos gráficas anteriores, de forma que el lector puede utilizarlas también para el caso de esta rutina `PMB04ND`.

Como ya se ha indicado, las diferencias entre ambas rutinas pasan por ser "casi" sólo una trasposición. Precisamente por esto, todo lo dicho para la rutina `PMB04OD`

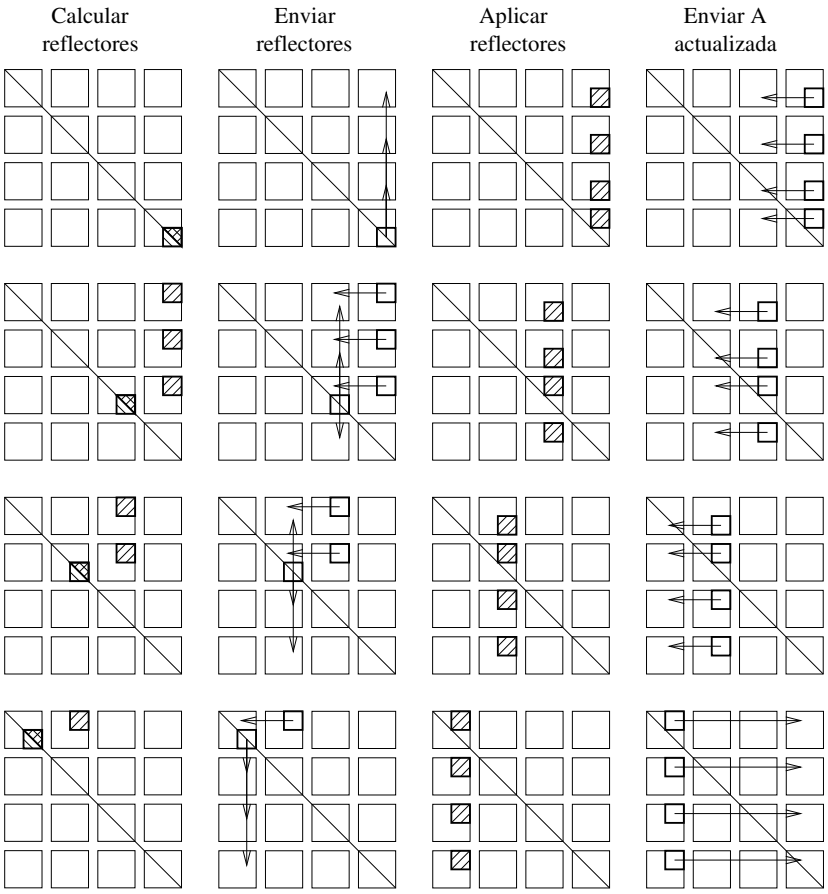

Figura 4.21: Esquema de operaciones en PMBO4ND con una distribución cíclica

sigue siendo cierto también para esta rutina, salvo por el orden de evaluación de las operaciones y el sentido de las comunicaciones.

4.2.9. PDDIAGZ

```
PDDIAGZ( SCOPE, M,N, A,IA,JA,DESCA, WORK,LWORK )
```

PDDIAGZ hace no negativos los elementos de la diagonal de la submatriz trapezoidal superior especificada. Para lograr esto, multiplica por -1 las filas o las columnas de la submatriz cuyo elemento diagonal es negativo, según indique el parámetro SCOPE (‘R’ para filas o ‘C’ para columnas).

En general, los elementos diagonales no van a estar en todos los procesos en los que se tiene porciones de las filas o las columnas que se van a escalar. Esto implica

que será necesario comunicar los valores de la diagonal entre los diferentes procesos, para que sepan si deben multiplicar o no cada fila/columna.

Para minimizar el impacto de las comunicaciones, la forma de proceder consiste en almacenar los elementos de la diagonal en el `WORK` de la rutina y luego hacer una suma colectiva por filas o columnas, logrando así que todos los procesos de cada fila/columna de la malla tengan todos los elementos diagonales correspondientes a sus filas/columnas de la submatriz. Con esto, luego (sin más comunicaciones) cada proceso puede cambiar el signo de aquellas filas/columnas cuyas diagonales sean negativas.

La implementación desarrollada puede resumirse en el siguiente algoritmo, donde cada vez que se menciona fila(s)/columna(s) se está refiriendo a la(s) fila(s) o a la(s) columna(s) en función del ámbito especificado en el parámetro `SCOPE`.

Algoritmo 4.6 `PDDIAGZ`

1. *Cada proceso recorre todas las filas/columnas que posee (parcialmente) de la submatriz trapezoidal superior y para cada una:*
   a) *si posee la diagonal de esa fila/columna, almacena su valor en la posición correspondiente del* `WORK`*,*
   b) *si no posee la diagonal de esa fila/columna, almacena el valor cero en la posición correspondiente del* `WORK`*.*
2. *Se realiza una operación de comunicación de suma colectiva del* `WORK` *por filas/columnas en la malla de procesos, dejando el resultado en todos los procesos (se realiza una operación de comunicación colectiva por cada fila/columna de la malla de procesos, todas ellas al mismo tiempo).*
3. *Cada proceso recorre todas las filas/columnas que posee (parcialmente), examinando en el* `WORK` *si el elemento de la diagonal es negativo o no. Si es negativo, se multiplica por* `-1` *todos los elementos que se tengan de la correspondiente fila/columna y si no, no se hace nada.*

4.3. Resultados experimentales

Nota: Todos los índices de prestaciones mostrados en este apartado se corresponden con ejecuciones realizadas en el cluster *Tirant*, cuyas características pueden verse en la sección 3.4.1.

En este apartado se van a mostrar diferentes resultados obtenidos sobre la rutina `PSB03OT`. Esta es la rutina a cargo de la resolución de la ecuación de Lyapunov en forma reducida. En el caso de resolver la ecuación sin reducir, faltaría sumar al tiempo de resolver la ecuación reducida el tiempo necesario en la transformación a forma reducida. Este tiempo se correspondería aproximadamente al tiempo de realizar el paso de transformar a forma real de Schur la matriz coeficiente de la ecuación más el tiempo de calcular la descomposición QR (o RQ para el caso de la ecuación traspuesta) de la parte derecha de la ecuación (esto último estaría incluido si mostrásemos

los resultados de la rutina `PSB03OU`). Sin embargo, en este capítulo es más interesante centrarse en el coste de la solución de la ecuación reducida. Para el coste del paso de la transformación a forma real de Schur puede consultarse el capítulo 3. El coste extra de la descomposición QR (o RQ) sería el de la correspondiente rutina de ScaLAPACK, ya que esta operación no se ha desarrollado al estar disponible en esta librería.

De las cuatro variantes de posibles ecuaciones que resuelve esta rutina (caso continuo y caso discreto, traspuestos y sin trasponer), se muestran sólo resultados del caso de la ecuación continua no traspuesta. El resto de casos se han paralelizado de forma similar. Se ha apreciado una diferencia en los resultados para las ecuaciones traspuestas, que luego se comentará (con respecto a la configuración de malla de procesos utilizada).

Se muestran primero las prestaciones obtenidas al resolver ecuaciones de Lyapunov en su uso dentro del problema de la reducción de modelos.

Considerando que el tamaño utilizado en estas pruebas no es especialmente grande, se utilizan después ecuaciones de mayor tamaño para analizar la escalabilidad de las nuevas rutinas desarrolladas.

4.3.1. Reducción de modelos

En un primer análisis se muestran en la figura 4.22 los tiempos de ejecución para resolver la ecuación de Lyapunov no traspuesta que aparece en las pruebas de los métodos de reducción de modelos del capítulo 5. En los métodos de reducción de modelos implementados, se resuelven cada vez dos ecuaciones de Lyapunov: la traspuesta y la no traspuesta, para el tipo de sistema que se esté usando (de tiempo continuo o de tiempo discreto).

En la sección 5.4 pueden consultarse las características del problema utilizado para la medida de prestaciones. En este momento, basta con conocer que se trata de un problema con matrices de orden $n = 3000$ y que se ha utilizado un tamaño de bloque en la distribución de 32 (en esa misma sección se muestra el proceso de selección de este tamaño de bloque).

En la figura se muestran los tiempos de ejecución de la resolución de la ecuación de Lyapunov reducida utilizando desde 1 hasta 16 procesos. Para cada número de procesos se muestra el mejor tiempo obtenido con todas las posibles configuraciones de malla de procesos.

En la figura 4.23 se muestran los tiempos de esas mismas ejecuciones (aunque sólo para un subconjunto de procesos), pero ahora especificando la malla de procesos utilizada.

Se aprecia fácilmente que esta rutina funciona mejor en mallas horizontales de procesos. Esto es debido a la forma en que se realizan las operaciones en el método de Hammarling para resolver la ecuación reducida. Para cada bloque diagonal de la matriz solución, cuando se está calculando la submatriz U_{12} (mediante la resolución de una ecuación reducida de Sylvester), van a estar activos sólo los procesos de una fila de la malla de procesos, aunque enseguida todos actualizan el resto de la ecuación. Por tanto, cuando queramos obtener las mejores prestaciones al resolver

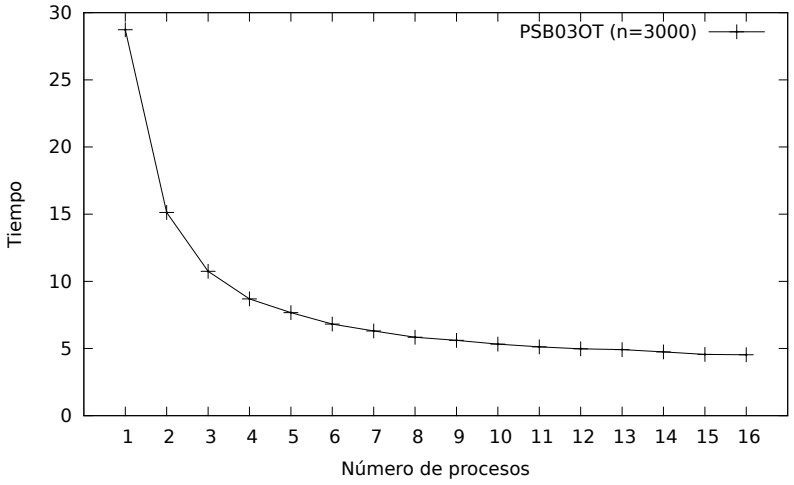

Figura 4.22: Tiempo de ejecución paralelo para la rutina `PSB03OT` con $n = 3000$

una ecuación reducida de Lyapunov con esta rutina, convendrá trabajar con una malla horizontal de procesos. En el caso de resolver la ecuación traspuesta, ocurre al revés: conviene trabajar en una malla vertical de procesos, por la misma razón.

En la tabla 4.2 se muestran los speed-ups y eficiencias (en tanto por cien) obtenidos para este problema. Se ha utilizado como tiempo secuencial para su cálculo el tiempo del código paralelo ejecutado en 1 proceso, que luego se verá que es una buena aproximación para "el mejor algoritmo secuencial".

Procesos	1	2	4	8	12	16
Speed-up	1.00	1.90	3.31	4.92	5.77	6.34
Eficiencia	100.00	94.94	82.65	61.49	48.08	39.64

Tabla 4.2: Speed-ups y eficiencias para la rutina `PSB03OT` con $n = 3000$

A pesar de que el tamaño del problema no es muy grande, los resultados obtenidos indican una importante reducción del tiempo de ejecución gracias a la resolución en paralelo de la ecuación. Por ejemplo, para 8 procesos aún se mantiene una eficiencia por encima del 60 %. Es conveniente probar problemas de mayor tamaño para ver si se mantienen las buenas características de la implementación.

4.3.2. Problemas de mayor tamaño

Hasta aquí se han mostrado resultados de la rutina paralela para resolver la ecuación de Lyapunov reducida, dentro del problema de la reducción de modelos que se verá en el capítulo 5. Como puede apreciarse en la figura 4.22, no son tiempos

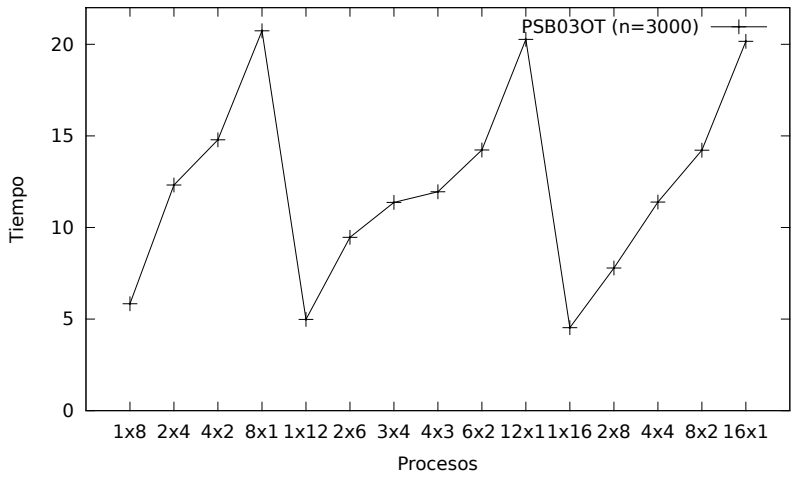

Figura 4.23: Tiempo de ejecución con malla de procesos variable para la rutina `PSB03OT`

muy elevados para esta operación (unos 30 segundos en 1 proceso). Esto es así porque se ha utilizado un tamaño de problema que suponga un tiempo razonable con todo el proceso de la reducción de modelos, donde aparecen muchas otras operaciones a sumarse a este tiempo. Pero si se quiere analizar el comportamiento de esta parte del proceso con vistas a comprobar que se van a poder resolver ecuaciones de Lyapunov de mucho mayor tamaño, conviene trabajar con problemas más grandes.

Se han probado diferentes tamaños y se ha optado por usar un problema con las matrices de tamaño $n = 10000$, que ya tiene un coste más adecuado en 1 proceso: unos 1000 segundos.

Dado que el tiempo de resolución de esta ecuación usando esta rutina no depende de los datos, al ser un método directo, los problemas utilizados han sido generados de forma aleatoria. Se han generado matrices con valores aleatorios con distribución uniforme en el rango $[-1, 1]$, cumpliendo las restricciones necesarias para que la ecuación tenga solución. En el caso de la ecuación continua no traspuesta, estas condiciones son que la matriz coeficiente de la ecuación debe tener todos sus valores propios con parte real negativa y que la parte derecha debe ser simétrica semidefinida positiva.

En el estudio de escalabilidad débil que se muestra luego, se ha trabajado con problemas de mayor tamaño. En los problemas de tamaño moderado, esto es hasta $n = 20000$, la parte derecha no ha sido generada de forma aleatoria. Para estos problemas se ha calculado la parte derecha tal que la solución de la ecuación fuera la matriz identidad. De esta forma, para estos problemas se ha podido comprobar que al final de la ejecución (en todas las configuraciones de procesos probadas) la solución era la correcta: la 1-norma de la diferencia entre la solución obtenida y la

solución conocida ha sido siempre del orden de 10^{-15}, como luego se verá.

Tamaño de bloque

Para analizar solamente la rutina `PSB03OT`, se ha empezado por buscar un tamaño de bloque adecuado para usarlo en los experimentos. Se ha ejecutado un problema de tamaño medio, $n = 5000$, en 1 proceso con diferentes tamaños de bloque. Pueden verse sus tiempos de ejecución en la figura 4.24.

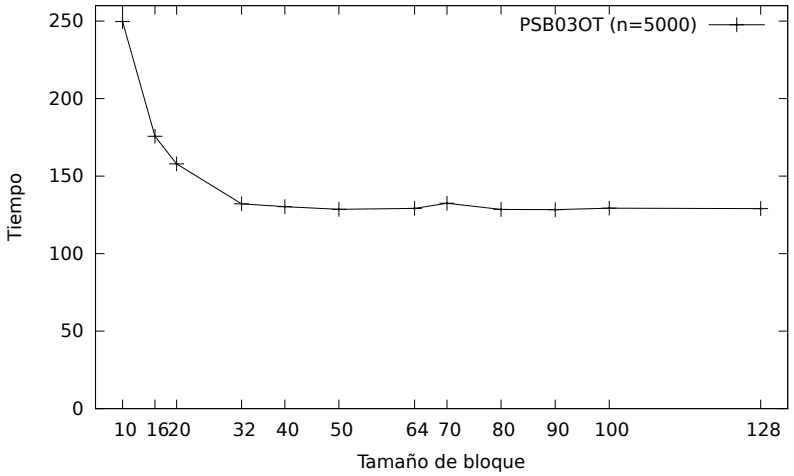

Figura 4.24: Tiempo de ejecución con tamaño de bloque variable para la rutina `PSB03OT`

Puede apreciarse que los tiempos son peores para tamaños de bloque inferiores a 32. Se ha elegido un tamaño de bloque de 50 para hacer las pruebas, ya que se aprecia una leve mejoría del tiempo de ejecución para ese valor. En realidad, las diferencias a partir de 32 son mínimas y hubiera podido utilizarse otro valor.

Escogiendo así el tamaño de bloque se procura optimizar la ejecución del código, ya que esta va a depender del tamaño de bloque elegido. Sin embargo, podría ocurrir que este no sea el tamaño de bloque ideal para todas las posibles ejecuciones. Debe tenerse en cuenta que el tamaño de bloque no sólo afecta a la ejecución de las operaciones numéricas, sino también a las comunicaciones a realizar, que habitualmente van a ser en bloques de múltiplos del tamaño escogido. Eligiéndolo de esta manera se tendrá una buena aproximación para el tamaño óptimo en una ejecución secuencial. Esta es la razón por la que el tiempo del algoritmo paralelo ejecutado en 1 proceso se ha usado anteriormente y se va a usar ahora como el punto de referencia para el cálculo de speed-ups. Es una buena aproximación a un buen algoritmo secuencial para resolver el problema. Se ha probado la rutina secuencial equivalente de SLICOT para este tamaño de $n = 5000$ y el tiempo que ha necesitado para resolver la

ecuación ha sido de 1380.82 segundos, mientras que el código paralelo ejecutado en 1 proceso con tamaño de bloque 50 ha tardado 128.60 segundos. Esto supone que para este tamaño de problema el código paralelo ejecutado en 1 proceso es más de 10 veces más rápido que el código secuencial existente. Esta gran diferencia de tiempos es debida a que el algoritmo paralelo está orientado a bloques para minimizar las comunicaciones necesarias. Y esta orientación a bloques hace que al ejecutarlo en 1 proceso se reduzcan los fallos de memoria caché, aumentando con ello las prestaciones. Si se usara como tiempo de referencia el del algoritmo secuencial, los speed-ups y eficiencias serían artificialmente elevados.

En [Kre08] puede consultarse una implementación secuencial orientada a bloques para resolver la ecuación de Lyapunov. Se ha comparado esta versión con la ejecución del nuevo código desarrollado ejecutado en 1 proceso. El código de este artículo resuelve la ecuación partiendo de una forma a mitad de camino entre la estándar y la reducida: con la matriz coeficiente ya en forma real de Schur, pero la parte derecha aún sin estar en forma triangular. Esta misma forma es la que espera la rutina `PSB03OU`, así que se ha utilizado esta rutina (que internamente llama a `PSB03OT`) para hacer una comparación justa. En el caso de trabajar con matrices cuadradas para la parte derecha de la ecuación no se han apreciado diferencias notables de prestaciones entre ambos códigos. Aunque sí que ha resultado más eficiente el código del artículo en el caso de ecuaciones con la parte derecha siendo una matriz muy alejada de ser cuadrada, ya que se explota específicamente esta característica en su código.

Prestaciones paralelas

En la figura 4.25 se muestran los tiempos de ejecución de la rutina `PSB03OT` para problemas de tamaño $n = 10000$ con tamaño de bloque 50 y ejecutando desde 1 hasta 16 procesadores. Al igual que anteriormente, se muestra el mejor tiempo obtenido para cada posible configuración de malla para cada número de procesos, que vuelve a ser para las mallas horizontales.

Procesos	1	2	4	8	12	16
Speed-up	1	1.90	3.44	6.02	8.31	9.40
Eficiencia	100.0	95.0	86.0	75.3	69.3	58.8

Tabla 4.3: Speed-ups y eficiencias para la rutina `PSB03OT` con $n = 10000$

En la tabla 4.3 se muestran los speed-ups y eficiencias obtenidos para este problema. Como puede apreciarse al comparar estos datos con los de la tabla 4.2, los speed-ups y eficiencias son mejores ahora que el problema es más grande.

Puede decirse que la nueva rutina paralela desarrollada es escalable, en el sentido de que se consiguen mejorar las prestaciones a costa de aumentar el tamaño del problema. Al menos desde el punto de vista de la escalabilidad fuerte.

Se está obteniendo una eficiencia superior al 75 % al ejecutar en 8 procesos para tamaños de problema con $n = 10000$.

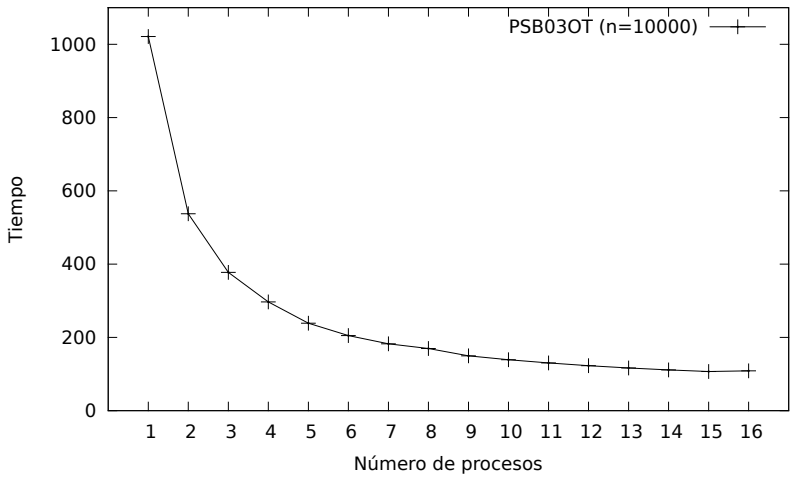

Figura 4.25: Tiempo de ejecución paralelo para la rutina `PSB03OT` con $n = 10000$

Escalabilidad

Puede ser interesante realizar un estudio de lo escalable que es la rutina desde el punto de vista de la escalabilidad débil. En este caso, en lugar de tener un problema de tamaño total fijo y ver cómo se comporta al resolverlo con un número creciente de procesos, lo que se hace es mantener constante la talla del subproblema que resuelve localmente cada proceso a medida que se va aumentando el número de procesos.

En el caso de este problema, al tratarse de un problema matricial denso, el tamaño del problema es $O(n^2)$, siendo n el orden de las matrices involucradas. Esto es lo mismo que decir que el tamaño es $k \cdot n^2$, siendo k una constante. Entonces, el tamaño del subproblema que corresponde a cada proceso al usar p procesos es $t = k \cdot n^2/p$. Si se pretende mantener constante esta cantidad para cualquier número de procesos p, se tendría que $n = \sqrt{t \cdot p/k}$. Si se utiliza un tamaño n_1 para la ejecución en 1 proceso, el tamaño n_p a utilizar en p procesos manteniendo constante el tamaño del subproblema de cada uno al valor $t = k \cdot n_1^2/1$ sería $n_p = n_1\sqrt{p}$.

Se ha realizado un estudio de la escalabilidad débil usando hasta 256 procesos, partiendo de un tamaño $n_1 = 5000$. En la tabla 4.4 se muestran los diferentes valores de n_p utilizados para cada número de procesos, cumpliendo la propiedad de mantener constante el tamaño de subproblema a resolver en cada proceso.

p	1	2	4	8	12	16	32	64	128	256
n_p	5000	7071	10000	14142	17321	20000	28284	40000	56569	80000

Tabla 4.4: Tamaños utilizados para el estudio de escalabilidad débil de la rutina `PSB03OT`

Como se ha comentado anteriormente, hasta tamaño 20000 se han generado los problemas con solución conocida para poder ir verificando la corrección del resultado. Para tamaños superiores, el problema empezaba a ser demasiado grande como para hacer esto. Para tamaños a partir de 20000, la generación de los problemas se ha efectuado de forma distribuida directamente dentro del programa de prueba. Esto es que cada proceso rellena los fragmentos de matrices del problema que le corresponden.

En la tabla 4.5 se muestra la 1-norma de la diferencia entre la solución obtenida y la solución conocida (la matriz identidad) para las ejecuciones en paralelo con los tamaños en los que se ha usado una solución conocida. Los errores han sido todos del orden de 10^{-15}. No se han apreciado diferencias al utilizar diferentes mallas para un número fijo de procesos.

p	2	4	8	12	16
n_p	7071	10000	14142	17321	20000
$\|U - I\|_1$	$4.90{\cdot}10^{-15}$	$6.70{\cdot}10^{-15}$	$7.60{\cdot}10^{-15}$	$8.89{\cdot}10^{-15}$	$7.68{\cdot}10^{-15}$

Tabla 4.5: Errores obtenidos al resolver la ecuación de Lyapunov (rutina `PSB03OT`) para diferentes tamaños y en diferente número de procesos

Si para la escalabilidad débil se utiliza un tamaño tal que se mantenga la talla del subproblema a resolver en cada proceso, al hacer las pruebas se tienen tiempos de ejecución que se corresponden a problemas de diferentes tamaños para cada número de procesos. En el cálculo del speed-up no se puede utilizar el tiempo de resolver cada uno de estos problemas en 1 solo proceso, ya que no se tiene. Hay que obtener una aproximación de este tiempo a partir del tiempo del problema que realmente se ha ejecutado en 1 proceso y el coste teórico del problema que se está estudiando. Como en este caso el problema de resolver la ecuación de Lyapunov reducida tiene un coste teórico $O(n^3)$, los tiempos aproximados en 1 proceso para cualquier tamaño n del problema serían $T_1(n) \approx T_1(n_1) \cdot (n/n_1)^3$.

Con estos tiempos aproximados para cualquier tamaño de problema en 1 proceso, ya se pueden calcular los speed-ups aproximados y con ellos las eficiencias. Para distinguirlos de los valores calculados usando un tiempo real en 1 proceso, se les va a llamar speed-up y eficiencias *escalados*. Resumiendo, estos índices de prestaciones *escalados*, que son los que se muestran en los resultados del estudio de la escalabilidad débil, se han calculado siguiendo las expresiones

$$\begin{aligned} n_p &= n_1\sqrt{p}\ , \\ T_1(n_p) &\approx T_1(n_1)\cdot(n_p/n_1)^3\ , \\ S_p &= T_1(n_p)/T_p\ , \\ E_p &= S_p/p\ . \end{aligned}$$

En la figura 4.26 pueden verse los speed-ups y eficiencias obtenidos tanto en el estudio de la escalabilidad fuerte como en el de la escalabilidad débil para un

número de procesos desde 1 hasta 16. Se muestra también el speed-up ideal (p) y la eficiencia ideal (1, 100 %). Recuérdese que para la escalabilidad fuerte se ha usado un problema de tamaño fijo $n = 10000$, mientras que para la escalabilidad débil se ha usado un tamaño variable.

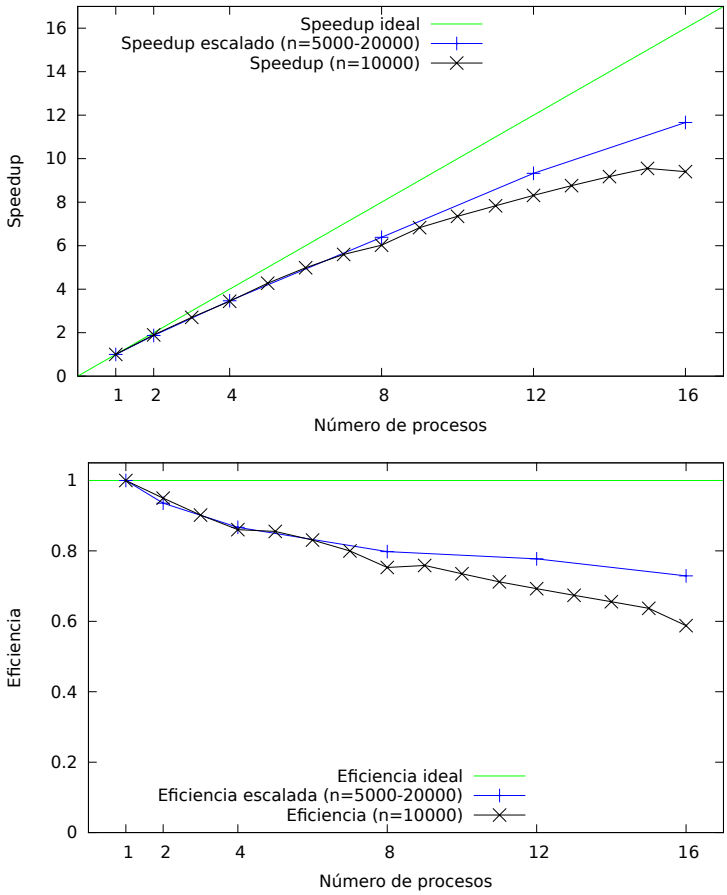

Figura 4.26: Speed-ups y eficiencias para la rutina `PSB03OT`

Se observa que, aunque a cierta distancia de los valores ideales, los índices de prestaciones escalados resultan mejores que los del problema con tamaño constante.

En la figura 4.27 y la tabla 4.6 se muestran los índices de prestaciones escalados desde 1 a 256 procesos. Se observa cómo el speed-up sigue creciendo al aumentar el número de procesos, gracias al aumento en el tamaño del problema. Esto indica que el algoritmo paralelo escala razonablemente bien, siendo factible resolver problemas de gran tamaño si se dispone de un número suficiente de procesadores.

Sin embargo, la eficiencia va disminuyendo al aumentar el número de procesos y

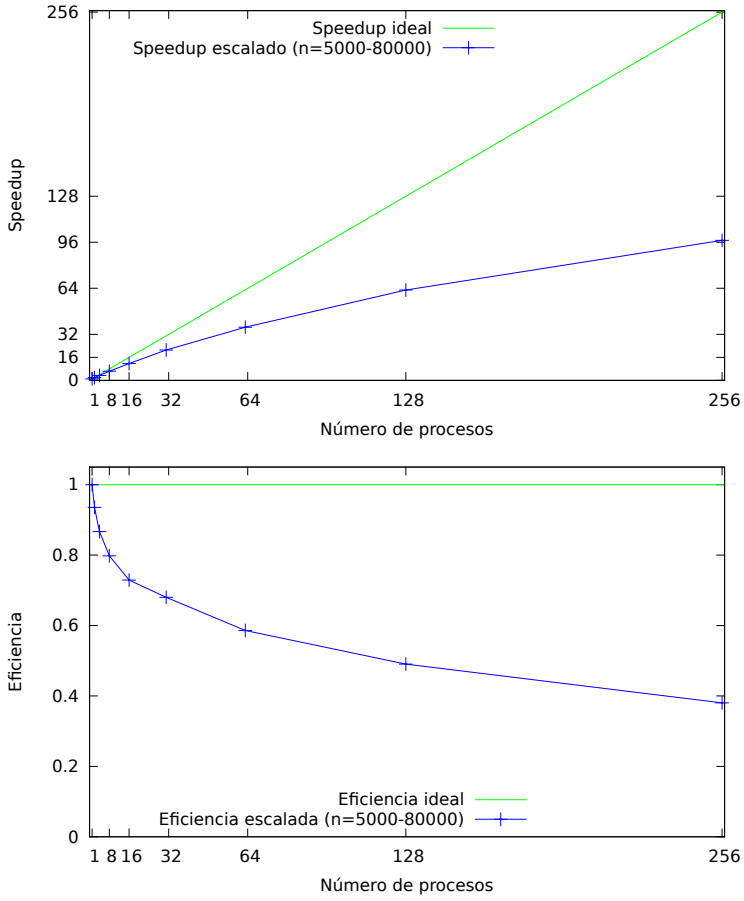

Figura 4.27: Speed-ups y eficiencias escalados para la rutina `PSB03OT`

ambos índices de prestaciones se alejan de sus valores ideales.

Una posible causa para este comportamiento es el hecho de que este algoritmo funcione de forma óptima en mallas horizontales de procesos y no en mallas cuadradas. En mallas horizontales, aunque se conserve el número total de elementos de la submatriz de cada proceso, cambia su forma. Al aumentar el número de procesos y el tamaño de la matriz, la submatriz de cada proceso se vuelve más estrecha y larga. Esto puede afectar negativamente a los cálculos que se realizan, que no serán tan homogéneos como en el caso de trabajar para el mismo problema en una malla cuadrada. Por ejemplo, para los tamaños que se han empleado (5000-80000 en 1-256 procesos), con una malla de procesos cuadrada en todos los casos todos los procesos tendrían una submatriz de 5000×5000. Mientras que con mallas horizontales se iría desde una (sub)matriz de 5000×5000 para 1 proceso hasta submatrices de

Procesos	1	8	16	32	64	128	256
Speed-up	1	6.38	11.66	21.08	36.91	62.78	97.36
Eficiencia	100	79.79	72.89	68.01	58.58	49.04	38.03

Tabla 4.6: Speed-ups y eficiencias escalados para la rutina `PSB03OT`

80000×313 (aproximadamente) para 256 procesos.

4.4. Conclusiones

Se han implementado rutinas de altas prestaciones para resolver la ecuación estándar de Lyapunov en paralelo. Se permite resolver las versiones de tiempo continuo y de tiempo discreto de la ecuación, en sus versiones normal y traspuesta.

Para el problema de reducción de modelos resulta conveniente resolver la ecuación de Lyapunov con su parte derecha en la forma del producto de una matriz por su traspuesta y tal que se desea el factor de Cholesky de la solución. Por ello, el método paralelizado ha sido el debido a Hammarling, que se caracteriza por aprovechar estas dos características.

Las pruebas prácticas realizadas confirman el buen comportamiento paralelo de las rutinas desarrolladas. Incluso ejecutadas en 1 solo procesador resultan mucho más eficientes que el algoritmo secuencial básico de la librería SLICOT, siendo comparables a implementaciones secuenciales orientadas a bloques de otros autores.

Sin embargo, el comportamiento paralelo óptimo de las nuevas rutinas es con mallas horizontales (o verticales en el caso de la ecuación traspuesta), lo que muy posiblemente sea la causa de que la escalabilidad del algoritmo, aún siendo buena, esté lejos de la ideal.

En el caso de usar estas rutinas únicamente para resolver ecuaciones de Lyapunov, se aconseja utilizar la configuración de malla de procesos óptima (horizontal para la ecuación no traspuesta o vertical para la ecuación traspuesta).

En el caso de usar estas rutinas para el problema de reducción de modelos que se presenta en el próximo capítulo, donde las partes más costosas del proceso son otras operaciones, se aconseja utilizar la configuración de malla de procesos más adecuada para estas otras operaciones. Las nuevas rutinas para resolver la ecuación de Lyapunov funcionarán correctamente para cualquier configuración de malla, aunque no sea la que consiga mejores prestaciones.

Capítulo 5

Reducción de modelos

En este capítulo se explica en qué consiste el problema de la reducción de modelos, por qué es conveniente disponer de implementaciones de altas prestaciones para resolverlo y cómo puede hacerse.

Se empieza describiendo las ideas básicas fundamentales sobre sistemas lineales de control y la reducción de modelos. Para ello, primero se recuerda qué es un sistema lineal de control y cuáles son las propiedades más importantes de estos sistemas, para después explicar en qué consiste el proceso de reducir un sistema lineal de control y por qué es necesario. Para terminar la descripción del problema, se presentan algunos métodos habituales de reducción de modelos.

A continuación, como se ha hecho también en los capítulos precedentes, se describen las rutinas paralelas de altas prestaciones que se han implementado para poder reducir modelos de sistemas lineales de control de gran tamaño.

Por último, se detallan las pruebas prácticas que se han llevado a cabo sobre las rutinas desarrolladas. Por una parte, se trata de comprobar que su funcionamiento es correcto. Y por otra parte, hay que ver que las prestaciones paralelas obtenidas resultan adecuadas.

5.1. Sistemas lineales de control

En este apartado se describe qué se entiende por un sistema lineal de control y cómo es su representación en el espacio de estados, además de presentar algunas propiedades importantes de los sistemas lineales de control. El lector interesado puede encontrar más detalles sobre los fundamentos de los sistemas lineales de control en [GL95, Lev96, Mut99, Son98, ZDG96, Dat04].

Un sistema de control viene dado por cualquier proceso físico o problema numérico en el que se tenga un sistema que ofrece unas *variables de salida* que cambian en el tiempo condicionadas por las actuaciones realizadas sobre unas *variables de entrada*.

Existen dos formas comunes de describir estos sistemas, dependiendo de si se ex-

presan en el dominio del tiempo o en el dominio de la frecuencia. En este documento se utiliza la representación en el dominio del tiempo, más conocida como *representación en el espacio de estados*, que ahora se presentará. En esta representación, las variables de salida se muestran como una función de las variables de entrada y unas nuevas variables llamadas *variables de estado*, mostrando también la relación existente entre las variables de estado y las variables de entrada.

Cuando todas estas relaciones vienen dadas por operaciones matriciales lineales (aparte de aparecer la derivada de las variables de estado en el caso de sistemas continuos), se habla de *sistemas lineales de control*. En el caso en el que las matrices involucradas son de valores constantes, se habla de *sistemas lineales de control invariantes en el tiempo*. Estos últimos son el tipo de sistemas utilizados en esta tesis.

Además, en función de si las variables son estudiadas de forma continua en el tiempo o tan sólo en instantes de tiempo concretos, se distinguen dos tipos de sistemas conocidos como *sistemas de tiempo continuo* o *sistemas de tiempo discreto*, respectivamente.

Un sistema lineal de control de tiempo continuo e invariante en el tiempo viene representado por el siguiente conjunto de ecuaciones:

$$\dot{x}(t) = Ax(t) + Bu(t), \quad t > t_0,\ x(t_0) = x_0, \tag{5.1}$$

$$y(t) = Cx(t), \quad t \geq t_0, \tag{5.2}$$

donde $x(t)$ es el *vector de estados* de dimensión n ($\dot{x}$ representa su derivada en función del tiempo t), $u(t)$ es el *vector de entradas o controles* de dimensión m e $y(t)$ es el *vector de salidas* de dimensión p. La matriz A de orden $n \times n$ es la *matriz de estados*, la matriz B de tamaño $n \times m$ es la *matriz de entradas o de control*, y la matriz C de orden $p \times n$ es la *matriz de salidas*. Son matrices de números reales, invariantes en el tiempo. El orden del sistema viene dado por n, la dimensión del vector de estados. Además, normalmente se cumple que $m \leq n, p \leq n$.

Estas dos ecuaciones, la *ecuación de estado* (5.1) y la *ecuación de salida* (5.2), son la *representación en el modelo de espacio de estados* de un sistema lineal de control. En este caso la ecuación de estado es una ODE (*Ordinary Differential Equation*), aunque en el caso más general (que se verá después) es una DAE (*Differential Algebraic Equation*). Cuando aparece sólo la primera derivada del vector de estados x, se habla de modelos de primer orden en contraposición con los modelos de segundo orden, en los que aparece además la segunda derivada.

En la figura 5.1 puede verse el esquema general de un sistema lineal de control.

Se denomina *función de transferencia* de este sistema de control a la función

$$M(s) = C(sI - A)^{-1}B.$$

La representación en el espacio de estados no es única. Un mismo sistema tiene múltiples representaciones diferentes. Por ejemplo, si se cambia las variables de estado del sistema definido por (5.1) y (5.2), mediante una transformación no singular

$$x(t) = S\tilde{x}(t)$$

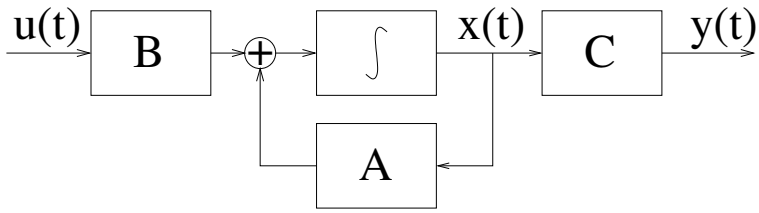

Figura 5.1: Esquema general de un sistema lineal de control

se obtiene un nuevo modelo (una nueva representación en el espacio de estados) del mismo sistema, esto es un *sistema algebraicamente equivalente*

$$\begin{aligned} \dot{\tilde{x}}(t) &= \tilde{A}\tilde{x}(t) + \tilde{B}u(t), \\ y(t) &= \tilde{C}\tilde{x}(t), \end{aligned}$$

donde

$$\tilde{A} = S^{-1}AS, \qquad \tilde{B} = S^{-1}B, \qquad \tilde{C} = CS.$$

El uso de uno u otro modelo puede simplificar la resolución del correspondiente problema de control. En particular, existen unas representaciones especiales, caracterizadas por un gran número de elementos nulos en unas posiciones concretas de las matrices del sistema, que se conocen como *formas canónicas* del sistema. Existen múltiples formas canónicas, diferenciadas por las disposiciones de los elementos nulos de las matrices del sistema. Un sistema lineal que sea controlable y observable (luego se verá lo que esto significa), permite transformaciones en el espacio de estados que lo reducen a una forma canónica deseada.

En algunas ocasiones, se trabajará con sistemas en los que sólo se dispone de las entradas, salidas y estados en instantes de tiempo determinados. Estos sistemas son los llamados sistemas lineales de *tiempo discreto*. La representación en el espacio de estados de estos sistemas viene dada por

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k, \qquad k \geq 0, \\ y_k &= Cx_k, \end{aligned}$$

donde, al igual que en el caso continuo, x_k,u_k e y_k son los vectores de estados, entradas o controles y salidas, pero ahora en instantes discretos del tipo $k \cdot \Delta t$. Las matrices A, B y C son, igual que en el caso continuo, matrices constantes de números reales. x_0 es el vector de estados para el tiempo inicial.

A veces aparece una cuarta matriz D en estas representaciones (tanto en tiempo continuo como en tiempo discreto) en la forma de un término extra en la ecuación de salida, relacionando las salidas con las entradas. Por ejemplo, en el caso continuo la ecuación de salida pasaría a ser $y(t) = Cx(t) + Du(t)$.

Además, ambos tipos de sistemas (con o sin matriz D) pueden generalizarse a los casos en que aparece otra matriz E en la ecuación de estado. En el caso continuo el sistema generalizado viene representado por las ecuaciones

$$\begin{aligned} E\dot{x}(t) &= Ax(t) + Bu(t), \quad t > t_0,\ x(t_0) = x_0, \\ y(t) &= Cx(t), \qquad\qquad\quad t \geq t_0. \end{aligned}$$

En el caso discreto las ecuaciones son

$$\begin{aligned} Ex_{k+1} &= Ax_k + Bu_k, \quad k \geq 0, \\ y_k &= Cx_k. \end{aligned}$$

Cuando la matriz E es singular, a estos sistemas se los conoce como *sistemas en forma descriptor* (*descriptor systems*).

El problema de control de un sistema lineal consiste en obtener las entradas adecuadas para provocar en las salidas un comportamiento deseado. El *controlador* es el encargado de calcular el valor a aplicar en las entradas del sistema para tratar de obtener ese comportamiento deseado.

Si para calcular las entradas el controlador utiliza una función que sólo depende del tiempo, es decir no tiene en cuenta el efecto que su acción provoca en las salidas, se hablará de *control en bucle abierto*. Sin embargo, los sistemas normalmente se ven sometidos a perturbaciones inesperadas, lo que hace necesario observar las salidas para actuar de una u otra manera en función de cómo las salidas se ven afectadas. Los sistemas en los que el controlador tiene en cuenta las salidas para calcular las entradas se conocen como *sistemas de control en bucle cerrado* (sistemas con realimentación de las salidas).

En ocasiones se realiza un proceso de análisis de un sistema lineal de control, con la intención de conocer propiedades importantes del sistema que ayuden a realizar un mejor control.

Algunas de las propiedades más importantes de los sistemas lineales de control se presentan a continuación [Moo81].

5.1.1. Propiedades

Estabilidad

Un sistema lineal invariante en el tiempo, descrito por la ecuación diferencial homogénea

$$\dot{x}(t) = Ax(t), \qquad x(0) = x_0 \tag{5.3}$$

se dice que es *asintóticamente estable*, si $x(t)$ tiende a cero cuando t tiende a infinito, para cualquier x_0.

El sistema es *estable*, si para cada valor de x_0 existe una constante c tal que $\|x(t)\| < c$ cuando t tiende a infinito.

El sistema es *inestable*, si hay algún x_0 tal que $\|x(t)\|$ tiende a infinito a medida que t tiende a infinito.

La propiedad de *estabilidad asintótica* es importante para el sistema lineal de control descrito por las ecuaciones (5.1) y (5.2), ya que garantiza salidas $y(t)$ acotadas siempre que se proporcionen entradas $u(t)$ acotadas.

Para analizar la estabilidad de un sistema no es necesario resolver la ecuación de estado, basta con estudiar algunas propiedades de la matriz de estados del sistema.

Teorema 5.1

El sistema (5.3) es asintóticamente estable si y sólo si todos los valores propios λ_k de A tienen parte real negativa, es decir

$$\mathrm{Re}(\lambda_k) < 0, \qquad \forall\ \lambda_k \in \lambda(A).$$

Puede consultarse la demostración del teorema en la sección 2.3 de [PCK91].

Si una matriz A cumple la condición dada en este teorema, se dice que es una *matriz estable.*

Teorema 5.2

El sistema (5.3) es asintóticamente estable si y sólo si para cualquier matriz Q simétrica y definida positiva, la solución P de la ecuación matricial de Lyapunov

$$A^T P + PA + Q = 0$$

es también simétrica y definida positiva.

La demostración puede verse en el capítulo 8 de [LT85].

Los resultados anteriores pueden ser formulados en términos de los sistemas lineales de control en tiempo discreto.

Sea el sistema lineal de tiempo discreto

$$x_{k+1} = Ax_k. \tag{5.4}$$

Este sistema es *asintóticamente estable* si x_k tiende a cero cuando k tiende a infinito para cualquier x_0.

El sistema es *estable* si para cada valor de x_0 existe una constante c tal que $\|x_k\| < c$ cuando k tiende a infinito.

El sistema es *inestable* si hay algún x_0 tal que $\|x_k\|$ tiende a infinito a medida que k tiende a infinito.

Teorema 5.3

El sistema (5.4) es asintóticamente estable si y sólo si todos los valores propios de A tienen módulo menor que la unidad, es decir

$$|\lambda_k| < 1, \qquad \forall\ \lambda_k \in \lambda(A).$$

Puede consultarse la demostración del teorema en la sección 2.3 de [PCK91].

Si una matriz A cumple la condición dada en este teorema, se dice que es una *matriz convergente.*

Teorema 5.4

El sistema (5.4) es asintóticamente estable si y sólo si para cualquier matriz Q simétrica y definida positiva, la solución P de la ecuación discreta de Lyapunov

$$A^T P A - P + Q = 0$$

es también simétrica y definida positiva.

La demostración de este teorema puede consultarse en el capítulo 4 de [BC85].

Controlabilidad

Se dice que el sistema definido por

$$\dot{x}(t) = Ax(t) + Bu(t) \qquad x(0) = x_0 \tag{5.5}$$

es *(completamente) controlable* si y sólo si para cualquier estado inicial x_0 y para cualquier estado final x_f, existe un tiempo finito t_f y un control $u(t), 0 \leq t \leq t_f$, tal que $x(t_f) = x_f$.

La controlabilidad de un sistema supone la posibilidad de llevarlo de un estado inicial cualquiera a un estado final cualquiera en un tiempo finito, sin más que elegir adecuadamente el control.

Si el sistema (5.5) es controlable, se dice que el par (A, B) es controlable.

El siguiente teorema da una condición necesaria y suficiente para que un sistema sea controlable.

Teorema 5.5 (Kalman)

El sistema (5.5) es completamente controlable si y sólo si la **matriz de controlabilidad** *de orden $n \times nm$*

$$\left[B, AB, \ldots, A^{n-1}B\right]$$

tiene rango n (rango completo).

Puede consultarse la demostración del teorema en el capítulo 2 de [KFA69].

Si el sistema (5.5) no es controlable, es posible reducirlo mediante una transformación de estado no singular a la forma

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} B_1 \\ 0 \end{bmatrix} u(t)$$

donde el par (A_{11}, B_1) es controlable, y el vector $x_1(t)$ tiene dimensión

$$d = \text{rango}\left[B, AB, \ldots, A^{n-1}B\right] < n.$$

El vector $x_2(t)$ contiene las componentes del vector de estados que no son controlables. Si A_{22} es una matriz estable, se dice que el sistema es *estabilizable*. Si el sistema inicial es asintóticamente estable o controlable, entonces es estabilizable.

Observabilidad

Se dice que el sistema lineal

$$\begin{array}{l} \dot{x}(t) = Ax(t), \qquad x(0) = x_0 \\ y(t) = Cx(t) \end{array} \tag{5.6}$$

es *(completamente) observable* si para cualquier estado inicial x_0 existe un tiempo finito t_f para el cual el conocimiento de las salidas $y(t)$ para $0 \leq t \leq t_f$ permite determinar x_0.

Si un sistema es observable, puede obtenerse el estado inicial del sistema con sólo conocer sus salidas. Si el sistema (5.6) es observable, se dice que el par (C, A) es observable.

Teorema 5.6

El sistema (5.6) es completamente observable si y sólo si la **matriz de observabilidad** *de orden* $rn \times n$

$$\begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

tiene rango n (rango completo).

Puede consultarse la demostración del teorema en el capítulo 2 de [KFA69].

Si el sistema (5.6) no es observable, es posible reducirlo mediante una transformación de estado no singular a la forma

$$\begin{aligned} \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} &= \begin{bmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix}, \\ y(t) &= [C_1\ 0] \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \end{aligned}$$

donde el par (C_1, A_{11}) es observable, y el vector $x_1(t)$ tiene dimensión igual al rango de la matriz de observabilidad del sistema original.

Si A_{22} es una matriz estable, se dice que el sistema es *detectable*.

5.1.2. Realización minimal

La *realización minimal* de un sistema lineal de control es la parte de éste que es controlable y observable.

Si el sistema (5.1),(5.2) no es controlable y observable, es posible reducirlo mediante una transformación de estado no singular a la forma

$$\begin{aligned} \begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{bmatrix} &= \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} u(t), \\ y(t) &= [C_1, C_2] \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} \end{aligned}$$

donde el sistema (A_{11}, B_1, C_1) es un sistema minimal (controlable y observable). Además, las funciones de transferencia de los sistemas (A, B, C) y (A_{11}, B_1, C_1) son iguales, es decir:

$$C_1(sI - A_{11})^{-1}B_1 = C(sI - A)^{-1}B.$$

El problema de calcular la realización minimal de un sistema aparece con mucha frecuencia, ya que obtener el subsistema totalmente controlable y observable suele ser el primer paso a realizar en muchos algoritmos de control.

5.1.3. Gramianos

Sea el sistema lineal descrito por las ecuaciones (5.1) y (5.2) una realización minimal (controlable y observable). Además sea $\lambda_i + \lambda_j \neq 0, i \neq j$ para todos los valores propios λ_i, λ_j de A (el sistema puede ser inestable).

Bajo esta hipótesis, el *gramiano de controlabilidad* W_c y el *gramiano de observabilidad* W_o pueden definirse como las soluciones respectivas de las ecuaciones de Lyapunov

$$\begin{aligned} AW_c + W_cA^T &= -BB^T, \\ A^TW_o + W_oA &= -C^TC. \end{aligned}$$

Si los valores propios de A se encuentran estrictamente en el semi-plano izquierdo (su parte real es negativa), entonces pueden definirse el gramiano de controlabilidad y el gramiano de observabilidad como

$$W_c = \int_0^\infty e^{At}\ BB^T\ e^{A^Tt}\ dt, \tag{5.7}$$

$$W_o = \int_0^\infty e^{A^Tt}\ C^TC\ e^{At}\ dt. \tag{5.8}$$

Teorema 5.7

El sistema (5.5) es completamente controlable si y sólo si el gramiano de controlabilidad

$$W_c(t) = \int_0^t e^{As}\ BB^T\ e^{A^Ts}\ ds$$

es definido positivo para $t > 0$.

La demostración del teorema puede verse en [Kal63].

Teorema 5.8

El sistema (5.6) es completamente observable si y sólo si el gramiano de observabilidad

$$W_o(t) = \int_0^t e^{A^Ts}\ C^TC\ e^{As}\ ds$$

es definido positivo para $t > 0$.

Proposición 5.1
Para dos sistemas algebraicamente equivalentes con vectores de estado x y $\tilde{x}$, se verifica que

$$\hat{W}_c = S^{-1} W_c S^{-T} \qquad \hat{W}_o = S^T W_o S,$$

donde S es la matriz no singular de orden $n \times n$ tal que $x = S\tilde{x}$.

Por tanto, los valores propios de los gramianos son dependientes de la representación utilizada en el espacio de estados. Sin embargo, los valores propios del producto de los gramianos son independientes de la representación empleada.

Proposición 5.2
Los valores propios λ_i del producto de matrices $W_c W_o$ son invariantes bajo transformaciones de coordenadas en el espacio de estados ($x = S\tilde{x}$).

Si el sistema descrito por las ecuaciones (5.1) y (5.2) es asintóticamente estable, entonces los *valores singulares de Hankel* de su función de transferencia $M(s) = C(sI - A)^{-1}B$ se definen como

$$\sigma_v(M(s)) = \{\lambda_i^{1/2} : \lambda_i \in \lambda(W_c W_o), i = 1, 2, \ldots, n\}.$$

Generalmente, se suelen ordenar de modo decreciente para $i = 1, \ldots, n$. Estos valores son lo que Moore [Moo81] llamó los *modos de segundo orden* del sistema.

5.2. El problema de la reducción de modelos

El problema de la reducción de modelos suele aparecer en aplicaciones de simulación y control de procesos físicos complejos. Es común encontrarse con sistemas demasiado grandes como para permitir un diseño interactivo, una optimización o un control en tiempo real. La reducción de modelos pretende obtener representaciones de un menor tamaño para estos sistemas de forma que sea posible realizar estas operaciones con ellos.

El propósito de la reducción de modelos es permitir reemplazar un sistema de ecuaciones diferenciales o en diferencias de gran tamaño por uno de menores dimensiones, pero que conserve sus características.

Aparecen grandes modelos, por ejemplo, al requerir mayor resolución en discretizaciones espaciales provenientes de problemas de control para fluidos o estructuras.

La reducción de modelos tiene aplicaciones en muy diversas áreas [BMS05]:

- Simulación de sistemas conservativos. Ejemplo: en dinámica molecular.
- Control y regulación de flujos en aplicaciones de dinámica de fluidos computacional (*Computational Fluid Dynamics*).
- Simulación y estabilización de grandes estructuras.

- Diseño de controles para vehículos (por tierra, mar y aire).
- Diseño de chips VLSI (*Very Large Scale Integration*).
- Simulación de sistemas micro-electro-mecánicos (MEMS).
- Simulaciones de semiconductores.
- Procesado de imágenes.

El problema de la reducción de modelos aparece allí donde se necesita obtener un modelo "equivalente" a uno dado, pero con un menor tamaño. En el caso de usar un modelo representado en el espacio de estados, esto se traduce directamente en un menor orden de las matrices que aparecen en su representación. Generalmente lo que se pretende es obtener un modelo más pequeño pero que siga siendo válido en cuanto a sus propiedades de control. Se tendrá una representación en la que el menor orden de las matrices involucradas permitirá reducir el coste de cualquier procesado posterior de ese modelo, pero en la que no se pierde precisión con respecto a la utilización del modelo original (o al menos la perdida de precisión se encuentra dentro de unos márgenes aceptables).

Dado un sistema de control de orden elevado, lineal e invariante en el tiempo, con la siguiente representación en el modelo de espacio de estados:

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ y &= Cx, \end{aligned} \tag{5.9}$$

donde $x \in \mathbb{R}^n, y \in \mathbb{R}^p, u \in \mathbb{R}^m$ y A, B, C son matrices constantes de las dimensiones adecuadas, un modelo de orden reducido de este sistema tiene la forma

$$\begin{aligned} \dot{x}_r &= A_r x_r + B_r u, \\ y_r &= C_r x_r, \end{aligned} \tag{5.10}$$

donde $x_r \in \mathbb{R}^r, y_r \in \mathbb{R}^p, u \in \mathbb{R}^m, r < n$ y A_r, B_r, C_r son de las dimensiones adecuadas.

Se pretende obtener un modelo de orden reducido, tal que el vector de salida y_r mejor se aproxime (de acuerdo con algún criterio) al vector de salida y del sistema original, para todos los vectores de control u dentro de la clase de funciones de entrada admisibles.

Esta descripción del proceso de reducción de modelos resulta un tanto ambigua ('algún criterio', 'funciones de entrada admisibles'). Esto es porque hay muchas formas de orientar un problema de reducción de modelos, todas ellas altamente dependientes de la aplicación concreta de la que se trate. En este trabajo se procura utilizar métodos de reducción de modelos genéricos que permitan su aplicación al mayor número de problemas de control posible.

Aunque se ha definido la reducción de modelos en términos de un sistema descrito en su forma en el espacio de estados, la reducción de modelos es igualmente aplicable a sistemas en el dominio de la frecuencia. Sin embargo, en esta memoria se opera

únicamente con la representación en el espacio de estados por su mayor carácter algebraico.

Las técnicas de reducción de modelos son un procedimiento natural en la ingeniería práctica actual, ya que hay muchas razones que justifican la necesidad de reducir el orden de los modelos. Fundamentalmente se busca alcanzar alguno de estos objetivos [FGG92]:

- Simplificar el modelo de funcionamiento de un sistema y facilitar su comprensión.
- Reducir los esfuerzos computacionales en problemas de simulación.
- Disminuir el coste computacional para hacer más eficiente el diseño numérico de un controlador.
- Obtener leyes de control más simples.

Particularmente dentro del diseño de controladores, la reducción de modelos supone reducir también el coste computacional del control necesario, lo que en situaciones como control óptimo o control adaptativo puede suponer la viabilidad del diseño del controlador. Además, el trabajo con modelos reducidos supone un menor requerimiento *hardware* para los controladores, lo que facilitará la implementación física.

En cualquier caso, siempre existirá un compromiso entre la precisión del modelo obtenido y su simplicidad. Será en cada caso concreto en el que se podrá optar por una mayor o menor reducción, en función de si se permite mayor o menor pérdida de propiedades con respecto al modelo original.

A continuación van a describirse los tipos más habituales de métodos de reducción de modelos, para enseguida centrarse en los métodos de truncamiento de componentes, explicando brevemente algunos de ellos. Los métodos de reducción de modelos que se han paralelizado en esta tesis pertenecen a estos últimos tipos de métodos.

Por último, se explican las formas más habituales de proceder en el caso de querer reducir sistemas inestables. Aunque todo el trabajo realizado va orientado a sistemas estables, la reducción de sistemas inestables suele basarse en reducir la parte estable del sistema, con lo que se tienen que usar métodos enfocados a sistemas estables, tras separar la parte inestable del sistema. En la sección 5.2.5 se mencionan dos técnicas comunes para reducir sistemas inestables mediante esta separación en parte estable e inestable.

5.2.1. Métodos de reducción de modelos

No existe un esquema de reducción de modelos universal. Hay muchos métodos para la reducción del orden de los modelos. A continuación se mencionan los más conocidos, para después profundizar en los que más interesan desde el punto de vista matricial, con vistas a una implementación de altas prestaciones.

Existen varias clasificaciones de los métodos de reducción de modelos. Una primera clasificación podría hacerse conforme al dominio de los modelos en que se aplica el método, ya sea el dominio del tiempo o el dominio de la frecuencia. Sin embargo, debido a la posibilidad de cambiar la representación de un sistema de uno a otro dominio, esto no supone ningún impedimento en su utilización.

Desde un punto de vista operativo, lo que parece más interesante, puede hablarse de otra clasificación sugerida por Skelton [Ske80], en la que se mencionan tres categorías de procedimientos para la reducción de modelos:

- Métodos basados en aproximaciones polinomiales (normalmente aplicables en el dominio de la frecuencia).
- Técnicas de optimización paramétricas (aplicables en ambos dominios).
- Procedimientos de truncamiento de componentes, basados en transformaciones en el espacio de estados (dominio del tiempo).

Los métodos no son exclusivos entre sí. Incluso pueden obtenerse métodos adecuados mediante combinación de varias aproximaciones [Sha75].

Los métodos de reducción polinomial son aplicados en el dominio de la frecuencia y no suelen ser computacionalmente intensivos. Están basados en relacionar los momentos y los parámetros de Markov entre el modelo original y el modelo reducido.

Las aproximaciones de optimización son generalmente procedimientos secuenciales de optimización paramétrica, basados en la minimización de algún índice definido apropiadamente. Este índice mide el 'error' entre los modelos original y reducido. Estos métodos suelen requerir grandes esfuerzos computacionales que, cuando el modelo original es de un orden elevado, pueden hacer el coste prohibitivo.

Los procedimientos de truncamiento de componentes se basan en transformaciones sobre la representación original del modelo en el espacio de estados, para obtener un modelo reducido que, en lo posible, conserve las propiedades que interesen del modelo original. Estas propiedades pueden ser: tiempo de respuesta, observabilidad, controlabilidad, prestaciones en bucle cerrado, etc. La forma de proceder en estos métodos es la siguiente.

Sea un sistema lineal invariante en el tiempo con un vector de estados x n-dimensional

$$\begin{aligned} \dot{x} &= Ax + Bu, \\ y &= Cx. \end{aligned}$$

Mediante una transformación de estados adecuada S, tal que $x = S\tilde{x}$, se obtiene un sistema equivalente:

$$\begin{aligned} \begin{bmatrix} \dot{\tilde{x}}_1 \\ \dot{\tilde{x}}_2 \end{bmatrix} &= \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \tilde{A}_{22} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix} + \begin{bmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{bmatrix} u, \\ y &= \begin{bmatrix} \tilde{C}_1 & \tilde{C}_2 \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \end{bmatrix}, \end{aligned}$$

siendo $\tilde{x}_1$ un vector de r elementos y $\tilde{x}_2$ de $(n-r)$ elementos.

El nuevo vector de estado se descompone según la filosofía de diseño seleccionada, de forma que las variables de estado 'más importantes' se dejan en $\tilde{x}_1$, y sólo éstas se toman en cuenta en el modelo reducido.

La teoría de realizaciones balanceadas ha supuesto una contribución significativa al campo de la reducción de modelos. En particular, Moore [Moo81] introdujo un conjunto de valores no negativos que eran independientes de la representación usada para el sistema. Conocidos como *modos de segundo orden del sistema* (los valores singulares de Hankel), se trata de las raíces cuadradas de los valores propios del producto de los gramianos del sistema. Estos valores representan el peso de cada variable de estado con respecto a las propiedades de controlabilidad y observabilidad del modelo original. Puede calcularse una transformación de estados tal que, en la nueva representación, los gramianos de controlabilidad y observabilidad sean iguales y diagonales, con sus valores ordenados decrecientemente (que serán los valores singulares de Hankel). De esta forma, es posible obtener un modelo de orden reducido, despreciando las componentes de estado que hacen sólo una pequeña contribución a estas propiedades estructurales.

Estos métodos de reducción de modelos de truncamiento mediante balanceado suelen obtener sistemas reducidos que aproximan mejor al sistema original [Ben06], en parte, porque permiten calcular un modelo que cumpla una determinada cota en el error cometido en la salida del sistema [LS85]. Habitualmente los modelos reducidos obtenidos mediante estos métodos suelen comportarse bien en todo el espectro de frecuencias, mientras que otros métodos tienen un comportamiento mejor sólo en algunas zonas.

Estas buenas características unidas al hecho de que permiten su aplicación a un amplio campo de problemas hace que sean estos métodos a los que se ha dirigido el trabajo presentado aquí. Requieren la resolución de ecuaciones de Lyapunov y otras operaciones, cuya implementación en paralelo forma el núcleo de este trabajo. En la siguiente sección se explican estos métodos con más detalle.

A pesar de que el trabajo central se dedique a la implementación de unos métodos de reducción de modelos específicos, por su dependencia del mismo tipo de operaciones matriciales y su mayor generalidad a la hora de ser aplicados, no hay que olvidar que hay otras técnicas de reducción de modelos [ASG01] [Ben06].

Cuando se trabaja con problemas de dimensión muy grande, es frecuente que las matrices del sistema sean dispersas. Existe una importante tendencia a optimizar todo tipo de métodos de reducción de modelos para aprovechar esta posible dispersidad de las matrices. A este respecto, se están utilizando mucho modernas técnicas algebraicas basadas en subespacios de Krylov [Bai02] [Fre03] [BQO05].

En el caso de los métodos de balanceado (que se explican a continuación), también es posible aprovechar estas modernas técnicas a costa de trabajar con aproximaciones de bajo rango de los gramianos en lugar de con los gramianos completos [Ben04] [GL05] [LW01]. Este tipo de técnicas permiten trabajar con sistemas del orden de millones de variables de estado, sistemas de dimensión muy grande.

5.2.2. Reducción basada en realizaciones balanceadas

Los métodos de reducción de modelos de truncamiento de componentes tienen especial interés, porque pueden ser aplicados a un conjunto muy amplio de sistemas de control. Dentro de esta clase de métodos cabe destacar los métodos de balanceado.

Sea el sistema lineal invariante en el tiempo (5.9), donde se asume que el par (A, B) es controlable y el par (C, A) es observable. Los gramianos de controlabilidad y observabilidad dados por las expresiones (5.7) y (5.8), respectivamente, son definidos positivos (teoremas 5.7, 5.8).

El balanceado de un sistema consiste en la diagonalización simultánea de ambos gramianos. Se trata de obtener una transformación en el espacio de estados que transforme el sistema original en uno equivalente en el que ambos gramianos sean iguales y diagonales.

Si se aplica una transformación no singular de la forma $x = S\tilde{x}$, los gramianos del sistema transformado presentan la forma indicada en la proposición 5.1. La transformación para la cual los nuevos gramianos $\hat{W}_c$ y $\hat{W}_o$ se hacen diagonales, se conoce con el nombre de *transformación contragradiente*.

Existen diferentes transformaciones contragradiente, que pueden obtenerse mediante el siguiente procedimiento.

Como la matriz W_c es simétrica definida positiva, puede reducirse mediante una transformación de semejanza ortogonal a una forma diagonal con todos sus elementos diagonales positivos. Calculando las raíces cuadradas positivas de esta forma diagonal se tendría la descomposición

$$V_c^T W_c V_c = \Sigma_c^2$$

donde V_c es una matriz ortogonal y Σ_c es una matriz diagonal con elementos diagonales positivos. De forma similar, como la matriz $(V_c\Sigma_c)^T W_o (V_c\Sigma_c)$ también es simétrica definida positiva, existe otra matriz ortogonal U tal que

$$U^T (V_c\Sigma_c)^T W_o (V_c\Sigma_c) U = \Sigma^2$$

donde Σ es una matriz diagonal con elementos diagonales positivos. Usando ambas matrices diagonales se tiene que la expresión

$$S_k = V_c\Sigma_c U \Sigma^{-k},\ 0 \leq k \leq 1$$

define una familia de transformaciones contragradiente con

$$S_k^{-1} W_c S_k^{-T} = \Sigma^{2k}, \qquad S_k^T W_o S_k = \Sigma^{2-2k}.$$

Los elementos diagonales $\sigma_i, i = 1, \ldots, n$ de la matriz Σ, son las raíces cuadradas positivas de los valores propios de $W_c W_o$, los valores singulares de Hankel, ya que

$$S_k^{-1} W_c W_o S_k = \Sigma^2.$$

Los siguientes valores de k son de especial interés y reciben un nombre específico:

- $k = 0 \quad \hat{W}_c = I \quad \hat{W}_o = \Sigma^2$: coordenadas de entrada normal.
- $k = 1/2 \quad \hat{W}_c = \Sigma \quad \hat{W}_o = \Sigma$: coordenadas balanceadas internamente.
- $k = 1 \quad \hat{W}_c = \Sigma^2 \quad \hat{W}_o = I$: coordenadas de salida normal.

Tiene especial interés la transformación contragradiente con $k = 1/2$, también conocida como *transformación balanceada*.

A continuación se muestra un método para el cálculo de la transformación balanceada (el algoritmo de [Lau80]).

Se calculan los gramianos mediante la resolución de las ecuaciones de Lyapunov

$$\begin{aligned} AW_c + W_c A^T + BB^T &= 0, \\ A^T W_o + W_o A + C^T C &= 0, \end{aligned} \tag{5.11}$$

que pueden resolverse por el método de Bartels-Stewart [BS72]. Aplicando posteriormente la descomposición de Cholesky al gramiano de controlabilidad se obtiene la matriz triangular inferior L_c que cumple

$$W_c = L_c L_c^T.$$

Entonces, se transforma el gramiano de observabilidad a la matriz $L_c^T W_o L_c$, que será reducida a la forma diagonal

$$V^T (L_c^T W_o L_c) V = \Sigma^2.$$

La reducción a forma diagonal se calcula mediante la descomposición en valores propios, donde V es la matriz ortogonal de vectores propios.

Es fácil comprobar que

$$S = L_c V \Sigma^{-1/2}$$

es una transformación balanceada tal que

$$S^{-1} W_c S^{-T} = S^T W_o S = \Sigma.$$

Las matrices del sistema balanceado vienen dadas por

$$\begin{aligned} \hat{A} &= S^{-1} A S = \Sigma^{1/2} V^T L_c^{-1} A L_c V \Sigma^{-1/2}, \\ \hat{B} &= S^{-1} B = \Sigma^{1/2} V^T L_c^{-1} B, \\ \hat{C} &= C S = C L_c V \Sigma^{-1/2}. \end{aligned}$$

Un problema del algoritmo de balanceado planteado es que el cálculo de los gramianos mediante (5.11) utilizando el método de Bartels-Stewart requiere calcular BB^T y $C^T C$, que puede introducir grandes errores de redondeo en algunos casos.

Alternativamente, es posible resolver las ecuaciones (5.11), obteniendo la descomposición de Cholesky de la solución, que es lo que se busca, sin calcular el producto BB^T y C^TC. Para ello, se utiliza el método propuesto por Hammarling para resolver ecuaciones de Lyapunov, que puede verse en la sección 4.1.3. Pueden entonces expresarse todas las operaciones del balanceado en función de los factores de Cholesky de los gramianos y no de los gramianos completos. Cuando el balanceado se realiza de esta manera, el proceso posterior de reducción de modelos mediante truncamiento de componentes se denomina **método de balanceado y truncamiento de la raíz cuadrada** (*the Square-Root Balance & Truncate model reduction method*) [TP87].

Por ejemplo, un algoritmo de balanceado que se comporta de esta manera sería el siguiente [LHPW87].

En primer lugar, se obtienen los factores de Cholesky de los gramianos

$$\begin{aligned} W_c &= L_c L_c^T, \\ W_o &= L_o L_o^T, \end{aligned}$$

mediante el algoritmo de Hammarling (no llegan a calcularse las matrices W_c, W_o). Los valores singulares de $L_o^T L_c$ son las raíces cuadradas positivas de los valores propios de $W_c W_o$, es decir los modos de segundo orden del sistema. Si se obtiene la descomposición en valores singulares de $L_o^T L_c$

$$L_o^T L_c = U \Sigma V^T,$$

la matriz de transformación balanceada y su inversa se obtendrán mediante

$$\begin{aligned} S &= L_c V \Sigma^{-1/2}, \\ S^{-1} &= \Sigma^{-1/2} U^T L_o^T. \end{aligned}$$

El proceso puede esquematizarse en la forma del siguiente algoritmo.

Algoritmo 5.1 *Balanceado interno de un sistema lineal de control.*

1. *Calcular los factores de Cholesky L_c, L_o de los gramianos W_c,W_o, resolviendo por el método de Hammarling las ecuaciones de Lyapunov*

$$\begin{aligned} AW_c + W_c A^T + BB^T &= 0, \\ A^T W_o + W_o A + C^T C &= 0. \end{aligned}$$

2. *Calcular la descomposición en valores singulares del producto de los factores de Cholesky*

$$L_o^T L_c = U \Sigma V^T.$$

3. *Obtener la matriz de transformación balanceada y su inversa*

$$S = L_c V \Sigma^{-1/2}, \qquad S^{-1} = \Sigma^{-1/2} U^T L_o^T.$$

4. Obtener las matrices del sistema balanceado

$$\begin{aligned}\hat{A} &= S^{-1}AS,\\ \hat{B} &= S^{-1}B,\\ \hat{C} &= CS.\end{aligned}$$

El coste de este algoritmo es aproximadamente de $50n^3$ flops, de los cuales aproximadamente $16n^3$ flops son para el cálculo de los factores de Cholesky de los gramianos.

Una operación de este algoritmo que puede dar lugar a grandes errores de redondeo es la formación del producto $L_o^T L_c$. Si este producto tiene valores singulares de muy diferentes magnitudes, los menores de ellos pueden verse afectados por errores de redondeo significativos al calcular el producto $L_o^T L_c$. Para evitar esto, es recomendable utilizar el algoritmo de [HLPW86] para el cálculo de los valores singulares de un producto de matrices sin formar explícitamente el producto.

También hay que tener cuidado con sistemas que estén cerca de ser no controlables o no observables. Esto puede provocar resultados muy poco precisos. Sin embargo, hay que mencionar que esta característica no es propia de este algoritmo, sino que es una propiedad del balanceado de sistemas en general.

El algoritmo puede adaptarse para calcular transformaciones balanceadas de entrada normal o de salida normal.

En el caso de sistemas discretos

$$\begin{aligned}x_{k+1} &= Ax_k + Bu_k,\\ y_k &= Cx_k,\end{aligned}$$

el algoritmo es similar, pero trabajando con las ecuaciones de Lyapunov discretas asociadas a los gramianos de controlabilidad y observabilidad:

$$\begin{aligned}AW_cA^T - W_c + BB^T = 0,\\ A^TW_oA - W_o + C^TC = 0.\end{aligned}$$

A continuación se explica cómo puede aplicarse el balanceado de sistemas para obtener un modelo de orden reducido de un sistema estable, completamente controlable y completamente observable. Si el sistema no fuera controlable u observable, se podría extraer la parte controlable y observable del mismo [PCK91] y obtener un sistema balanceado de esta.

La forma de obtener un modelo de orden reducido de un sistema dado al que se le ha practicado el balanceado interno, es eliminando los estados 'menos controlables/observables' del sistema.

Si el sistema balanceado internamente es

$$\begin{aligned}\dot{\hat{x}} &= \hat{A}\hat{x} + \hat{B}u,\\ y &= \hat{C}\hat{x},\end{aligned}$$

y existe un q, $1 \leq q < n$, para el cual los modos de segundo orden verifican

$$\sigma_q \gg \sigma_{q+1}, \tag{5.12}$$

entonces el sistema puede particionarse de la forma

$$\hat{A} = \begin{bmatrix} F & F_1 \\ F_2 & F^* \end{bmatrix} \begin{matrix} \} \, q \\ \} \, n-q \end{matrix} \quad \hat{B} = \begin{bmatrix} G \\ G^* \end{bmatrix} \begin{matrix} \} \, q \\ \} \, n-q \end{matrix} \quad \hat{C} = \underbrace{\begin{bmatrix} H \end{bmatrix}}_{q} \; \underbrace{\begin{bmatrix} H^* \end{bmatrix}}_{n-q}$$

y así obtener un modelo de orden reducido

$$\begin{aligned} \dot{w} &= Fw + Gu, \\ z &= Hw. \end{aligned} \tag{5.13}$$

El vector w es una aproximación de las primeras q componentes de $\hat{x}$ y el vector z es una aproximación de y.

Puede probarse que el modelo de orden reducido (5.13) está balanceado con los gramianos de controlabilidad y observabilidad

$$\bar{W}_c = \bar{W}_o = diag(\sigma_1, \sigma_2, \ldots, \sigma_q).$$

La demostración puede consultarse en [TP87].

Para determinar el orden q del modelo reducido, pueden utilizarse otros criterios diferentes al mostrado en (5.12). Por ejemplo, puede utilizarse la condición

$$\left(\sum_{i=1}^{q} \sigma_i^2 \right)^{1/2} \gg \left(\sum_{i=q+1}^{n} \sigma_i^2 \right)^{1/2}.$$

Ni en esta ni en la condición (5.12) se garantiza una proximidad entre la salida z del modelo reducido y la salida y del sistema original. Una condición basada en el análisis del error de la salida puede verse en [LS85].

La búsqueda de un modelo adecuado de orden reducido puede requerir realizar varios modelos de distinto orden y después comparar cada uno con el sistema original, fijándose en unas propiedades concretas.

5.2.3. Reducción libre de balanceado

En el proceso de balanceado de un sistema, las matrices de transformación balanceada y su inversa pueden estar mal condicionadas y dificultar su cálculo cuando el sistema original está muy desbalanceado. Cuando esto ocurre, el modelo reducido puede no resultar adecuado. Una alternativa es utilizar métodos basados en algoritmos libres de balanceado (*Balancing Free algorithms*).

Al igual que pasaba con los algoritmos de balanceado, los algoritmos libres de balanceado tienen una versión que utiliza los gramianos completos y otra que utiliza los factores de Cholesky de los gramianos (la versión de raíz cuadrada), que resulta

más adecuada tanto por su menor coste computacional como por sus mejores características numéricas. A continuación se indica cómo se procede en la versión de raíz cuadrada de esta clase de métodos. Se trata del conocido como **método de reducción de modelos de balanceado y truncamiento de la raíz cuadrada sin balanceado** (*the Balancing-Free Square-Root Balance & Truncate model reduction method*) [Var91b].

Se comienza por los mismos dos primeros pasos que se hacen en el balanceado: obtener los factores de Cholesky de los gramianos y luego calcular la descomposición en valores singulares de su producto.

Entonces se escoge el orden del sistema reducido r de igual modo que en los métodos de truncamiento basados en balanceado. Puede utilizarse el criterio que se desee para despreciar a partir de un determinado valor singular de la descomposición recién calculada. Si se particionan las matrices de la descomposición en valores singulares según el orden r del modelo reducido, se tiene que

$$L_o^T L_c = [U_1\ U_2] \begin{bmatrix} \Sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix} \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix},$$

siendo U_1 y V_1 de $n \times r$, U_2 y V_2 de $n \times (n-r)$, Σ_1 de $r \times r$ y Σ_2 de $(n-r) \times (n-r)$ (n es el orden del sistema sin reducir).

A continuación se calculan dos descomposiciones QR

$$L_c U_1 = [P_1\ P_2] \begin{bmatrix} R_c \\ 0 \end{bmatrix}, \qquad L_o V_1 = [Q_1\ Q_2] \begin{bmatrix} R_o \\ 0 \end{bmatrix},$$

donde P_1 y Q_1 son de $n \times r$. A partir de estas matrices se obtienen las matrices de transformación del sistema con las expresiones

$$\begin{aligned} S &= P_1, \\ S^{-1} &= (Q_1^T P_1)^{-1} Q_1^T. \end{aligned}$$

Nótese que P_2 y Q_2 no se utilizan, con lo que no es necesario calcularlas.

Utilizando estas matrices de transformación sobre las matrices del sistema original se obtiene el modelo reducido mediante este método de reducción de modelos.

5.2.4. Fórmulas de aproximación de perturbación singular

El método de reducción de modelos basado en las fórmulas de aproximación de perturbación singular (*Singular Perturbation Approximation method*) [LA89, Var91a] escoge un determinado orden n_r para el sistema reducido y entonces particiona las matrices del sistema (A, B, C, D) en

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},\ B = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix},\ C = (C_1\ C_2),$$

siendo A_{11} de $n_r \times n_r$, B_1 de n_r filas, C_1 de n_r columnas y manteniendo el resto de dimensiones coherentes con el sistema original.

Entonces se calculan las matrices del sistema de orden reducido (A_r, B_r, C_r, D_r) según estas expresiones:

$$\begin{aligned} A_r &= A_{11} + A_{12}(g \cdot I - A_{22})^{-1} A_{21}, \\ B_r &= B_1 + A_{12}(g \cdot I - A_{22})^{-1} B_2, \\ C_r &= C_1 + C_2(g \cdot I - A_{22})^{-1} A_{21}, \\ D_r &= D + C_2(g \cdot I - A_{22})^{-1} B_2 \end{aligned}$$

siendo $g = 0$ en el caso continuo y $g = 1$ en el caso discreto.

Normalmente este método se usa en combinación con otros que permiten obtener de forma más o menos automática el tamaño del modelo reducido n_r. Por ejemplo, puede usarse un método que calcule una realización balanceada, escoger en él el valor adecuado para n_r y posteriormente ajustar las matrices del modelo reducido utilizando estas fórmulas. Una de las principales características que se consiguen mediante este método es una mejor aproximación en el comportamiento del modelo reducido para bajas frecuencias.

5.2.5. Reducción de modelos de sistemas inestables

La mayoría de métodos de reducción de modelos exigen que el sistema a reducir sea un sistema estable para poder ser aplicados. Sin embargo, no siempre va a ser así. En ocasiones se plantea la necesidad de obtener un modelo reducido de un sistema que no es estable. En estos casos lo más habitual es proceder mediante la descomposición del sistema en subsistemas, reducir en lo posible estos subsistemas y formar un modelo reducido del sistema original a partir de los modelos reducidos de los subsistemas. Esto es lo que se hace en las dos técnicas más comúnmente utilizadas y que se describen brevemente a continuación.

Descomposición aditiva

La técnica conocida como *descomposición aditiva* aplicada a la reducción de modelos de sistemas inestables se basa en reducir la parte estable de un sistema lineal de control inestable. Para ello, se descompone la función de transferencia del sistema en la forma $G = G1 + G2$, tal que $G1$ conserva la parte estable del sistema y $G2$ la parte inestable. Entonces se obtiene un modelo reducido $G1_r$ de la parte estable $G1$ del sistema original. Dado que $G1$ es un sistema estable, para reducir su modelo puede utilizarse cualquiera de los métodos conocidos para la reducción de sistemas estables. Una vez que se tiene una versión reducida del subsistema $G1$, se obtiene el modelo reducido del sistema original G como la suma $G1_r + G2$.

Con la técnica de descomposición aditiva se simplifica la parte estable del sistema conservando intacta la parte inestable del mismo.

Factorización racional coprima estable

La técnica denominada *factorización racional coprima estable* (RCF estable) aplicada a la reducción de modelos consiste en obtener una descomposición de un sistema

inestable de forma que quede expresado en función de sistemas estables y reducir estos. Se empieza representando la función de transferencia del sistema original en la forma $G = M^{-1}N$, donde M y N sean sistemas estables. Entonces se calcula un sistema reducido de $[N, M]$, que será $[N_r, M_r]$, utilizando el método para la reducción de sistemas estables que se desee. El modelo reducido del sistema original viene dado por $G_r = M_r^{-1}N_r$.

5.3. Rutinas de altas prestaciones desarrolladas

En este apartado se presentan aquellas rutinas de altas prestaciones desarrolladas que están más directamente relacionadas con la reducción de modelos. Básicamente se trata de las versiones paralelas de las rutinas de reducción de modelos de SLICOT `AB09AD`, `AB09BD` y `AB09DD`, además de las rutinas auxiliares que utilizan.

También se han tenido que desarrollar nuevas rutinas correspondientes a versiones paralelas de algunas operaciones que en el caso secuencial no tienen rutinas concretas porque son operaciones sencillas, pero que ya no lo son tanto en el caso paralelo.

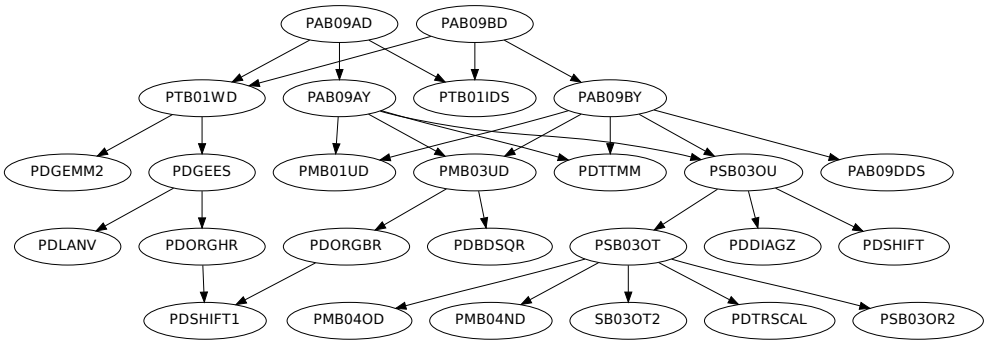

Figura 5.2: Rutinas de altas prestaciones desarrolladas para la reducción de modelos

En la figura 5.2 se muestran todas las rutinas de altas prestaciones desarrolladas para la reducción de modelos. Todas son nuevas rutinas que se han implementado para realizar operaciones necesarias en la reducción de modelos.

Se incluyen también todas las rutinas necesarias para el problema de valores propios que aparece (`PDGEES` y todas las que tiene por debajo) y para la resolución de ecuaciones de Lyapunov (`PSB03OU` y todas las que tiene por debajo), que ya se describieron en capítulos anteriores. Se han incluido para tener en una misma figura todas las rutinas necesarias juntas.

A continuación se describen las nuevas rutinas que no han sido ya descritas en capítulos anteriores.

5.3.1. PAB09AD

```
PAB09AD( DICO,JOB,EQUIL,ORDSEL, N,M,P,NR,A,IA,JA,DESCA,
         B,IB,JB,DESCB, C,IC,JC,DESCC, HSV, TOL,
         IWORK,LIWORK, DWORK,LDWORK, IWARN,INFO )
```

La rutina `PAB09AD` permite calcular un modelo reducido a partir de un modelo original estable mediante uno de estos dos métodos de reducción de modelos:

- SR B&T (*the Square-Root Balance & Truncate model reduction method*): el método de balanceado y truncamiento de la raíz cuadrada [TP87].
- BFSR B&T (*the Balancing-Free Square-Root Balance & Truncate model reduction method*): el método de balanceado y truncamiento de la raíz cuadrada sin balanceado [Var91b].

Esta rutina opera con los modelos en su representación en el espacio de estados. Es una versión paralela de la rutina `AB09AD` de SLICOT.

Conserva la funcionalidad de la correspondiente rutina secuencial de SLICOT. Permite trabajar con modelos de tiempo continuo y discretos. Se puede indicar que se realice un equilibrado (escalado) previo sobre el sistema. Se debe indicar si se desea un orden fijo para el sistema reducido o bien que se determine de forma automática a partir de una tolerancia dada.

A continuación se muestra el algoritmo implementado en esta rutina.

Algoritmo 5.2 `PAB09AD`

1. *Si se ha especificado que hay que hacer equilibrado previo en el sistema, se hace utilizando la rutina* `PTB01IDS`.
2. *Reducir el sistema a uno equivalente en el que la matriz de estados A esté en forma real de Schur. Esto se hace mediante una llamada a la rutina* `PTB01WD`.
3. *Calcular la reducción de modelos deseada sobre el sistema con la matriz de estados en forma real de Schur. Esta operación se realiza en la rutina* `PAB09AY`.

Como puede apreciarse por el algoritmo, esta rutina es básicamente una rutina *puente* que se apoya en otras rutinas para realizar su función.

5.3.2. PAB09BD

```
PAB09BD( DICO,JOB,EQUIL,ORDSEL, N,M,P,NR,A,IA,JA,DESCA,
         B,IB,JB,DESCB, C,IC,JC,DESCC, D,ID,JD,DESCD,
         HSV, TOL1, TOL2, IWORK,LIWORK, DWORK,LDWORK,
         IWARN, INFO )
```

Esta rutina es muy parecida a la rutina anterior `PAB09AD` salvo por utilizar métodos diferentes para la reducción de modelos.

La rutina PAB09BD es una versión paralela de la rutina AB09BD de SLICOT. Sirve para obtener un modelo reducido de un sistema lineal de control estable mediante uno de estos dos métodos de reducción de modelos:

- SR SPA (*the Square-Root Singular Perturbation Approximation model reduction method*): el método de aproximación de perturbación singular de la raíz cuadrada [LA89].
- BFSR SPA (*the Balancing-Free Square-Root Singular Perturbation Approximation model reduction method*): el método de aproximación de perturbación singular de la raíz cuadrada sin balanceado [Var91a].

Esta rutina utiliza los modelos en su representación en el espacio de estados.

Presenta la misma funcionalidad que la correspondiente rutina secuencial de SLICOT. Se puede usar para modelos de tiempo continuo y discretos. De forma opcional se puede indicar que se realice un equilibrado (escalado) previo sobre el sistema. Hay que especificar si se desea un orden fijo para el sistema reducido o si se quiere que se determine de forma automática a partir de una tolerancia dada.

El algoritmo implementado para esta rutina es prácticamente idéntico al utilizado para la rutina anterior PAB09AD (algoritmo 5.2), salvo por llamar ahora a la rutina PAB09BY en lugar de a PAB09AY. Al igual que aquella rutina, esta también es una rutina *puente* que se apoya en otras rutinas para realizar su función.

5.3.3. PTB01IDS

```
PTB01IDS( JOB, N,M,P, MAXRED, A,IA,JA,DESCA,
          B,IB,JB,DESCB, C,IC,JC,DESCC,
          SCALE, DWORK,LDWORK, INFO )
```

La rutina PTB01IDS sirve para reducir la 1-norma de *la matriz del sistema* de un sistema lineal de control expresado en su representación en el espacio de estados. Es la versión paralela de la rutina de SLICOT TB01ID.

Dado un triplete de matrices (A, B, C) correspondientes a un sistema lineal de control, la *matriz del sistema* S asociada a este sistema es

$$S = \begin{pmatrix} A & B \\ C & 0 \end{pmatrix}.$$

La 1-norma de esta matriz es reducida en esta rutina mediante balanceado. Esto implica la aplicación sucesiva de una transformación de semejanza diagonal.

El argumento JOB se utiliza para indicar si balancear toda la matriz del sistema o sólo una parte de ella.

En SCALE se devuelven los factores de escala utilizados (la diagonal de la matriz diagonal utilizada en la transformación de semejanza aplicada).

MAXRED es un valor máximo de reducción a realizar en cada iteración.

El algoritmo utilizado en esta rutina puede describirse de la siguiente manera:

Algoritmo 5.3 `PTB01IDS`

1. *Calcular la 1-norma de la* matriz del sistema *S y finalizar si da cero.*

2. *Repetir mientras se realice algún escalado:*

   a) *Preparar datos para ver si escalar o no. Esto supone calcular para la matriz del sistema la suma de los valores absolutos de cada fila (quitando el elemento diagonal) y el máximo valor absoluto de cada fila. Lo mismo para cada columna.*

   b) *Recorrer la diagonal de la matriz comprobando si conviene balancear y, de ser así, almacenar el escalado a realizar en la fila/columna correspondientes.*

   c) *Comunicar de todos a todos para saber si hay escalados que realizar (se hace una reducción de una variable dejando el resultado en todos).*

   d) *Si hay algún escalado que realizar:*

      1) *Recoger en todos los procesos el escalado a realizar en sus filas y realizarlo.*
      2) *Recoger en todos los procesos el escalado a realizar en sus columnas y realizarlo.*

3. *Calcular la 1-norma de la matriz del sistema balanceado, para devolver el ratio entre la 1-norma del sistema original y la del sistema balanceado.*

La suma de valores absolutos de cada fila/columna así como el cálculo de su máximo en valor absoluto son valores que se necesitarán para cada fila/columna solamente en el proceso en el que se encuentre el elemento diagonal. Sin embargo, para no complicar las comunicaciones, estos valores se dejan en todos los procesos de la fila/columna involucrada en su cálculo. Para su cálculo se requiere al final una operación de reducción (ya sea calculando el máximo o la suma). En esta operación se podría indicar como destino al único proceso que vaya a necesitar estos datos en cada caso. Pero determinar este proceso para cada elemento de la diagonal de la matriz y especificarlo supondrá un sobrecoste en su cálculo y también en las comunicaciones, que quedarán más fragmentadas. Por esta razón se ha optado por indicar que el resultado de la operación de reducción se deje en todos los procesos involucrados, consiguiéndose así cada resultado en el proceso que los va a necesitar (además de en el resto de procesos de la fila/columna).

5.3.4. `PTB01WD`

```
PTB01WD( N,M,P, A,IA,JA,DESCA, B,IB,JB,DESCB,
         C,IC,JC,DESCC, U,IU,JU,DESCU,
         WR,WI, DWORK,LDWORK, INFO )
```

La rutina PTB01WD permite obtener una nueva representación en el espacio de estados para el sistema de control suministrado, en la que la matriz de estados (A) esté en forma real de Schur. Es la versión paralela de la rutina de SLICOT TB01WD.

Esta rutina realiza su función con facilidad mediante el uso de otras rutinas auxiliares: PDGEES y PDGEMM o PDGEMM2.

Su algoritmo es:

Algoritmo 5.4 PTB01WD

1. *Transformar la matriz A a forma real de Schur utilizando la rutina* PDGEES*:* $A \leftarrow U^T AU$.

2. *Aplicar la transformación calculada a las matrices B y C para obtener un sistema equivalente al original. Esto es calcular las nuevas B y C:*

   a) $B \leftarrow U^T B$.

   b) $C \leftarrow CU$.

La aplicación de la transformación calculada sobre las matrices B y C supone un producto matriz por matriz para cada una. Estos productos se hacen con la rutina de PBLAS PDGEMM si hay suficiente *espacio de trabajo* (DWORK). Pero esto supone mucho espacio de trabajo ya que implica tener espacio para una tercera matriz. Para permitir los productos necesarios sin necesitar de una tercera matriz se ha desarrollado la nueva rutina PDGEMM2. Esta nueva rutina permite multiplicar dos matrices dejando el resultado en una de ellas y así no requiriendo tanto espacio de trabajo.

El algoritmo implementado selecciona qué rutina utilizar entre PDGEMM y PDGEMM2 en función de si se dispone de suficiente espacio de trabajo o no. Esto se realiza de forma independiente para los dos productos matriciales a calcular.

La rutina PDGEES se explicó en la sección 3.2.1. La rutina PDGEMM2 se explica a continuación.

5.3.5. PDGEMM2

```
PDGEMM2( SIDE,TRANS, M,N, A,IA,JA,DESCA,
         B,IB,JB,DESCB, DWORK,LWORK)
```

En la versión secuencial de las rutinas de reducción de modelos aparece en ocasiones la necesidad de calcular el producto de dos matrices almacenando el resultado en una de ellas y así no necesitando espacio extra. Esta operación, que es trivial en su versión secuencial, no lo es tanto en su versión paralela.

La rutina PDGEMM2 es la encargada de permitir realizar estos productos de matrices sobre una de ellas sin necesitar espacio extra para toda una matriz. En paralelo, sin embargo, sí que es necesario algo de espacio extra donde ir recibiendo los bloques de las matrices que pertenecen a otros procesos y poder ir realizando los cálculos (esto es el parámetro DWORK).

Se permiten realizar cuatro clases de productos diferentes de matrices genéricas (del tamaño adecuado), en función de los parámetros `SIDE` y `TRANS`. Pueden verse estas cuatro posibles operaciones en la tabla 5.1.

SIDE	TRANS	Operation
Left	No transpose	$B \leftarrow A \times B$
Left	Transpose	$B \leftarrow A^T \times B$
Right	No transpose	$B \leftarrow B \times A$
Right	Transpose	$B \leftarrow B \times A^T$

Tabla 5.1: Posibles productos a realizar en la rutina `PDGEMM2`

Por brevedad, se va a describir solamente la forma de proceder para el primero de los cuatro tipos de productos implementados. No obstante, es conveniente hacer notar que en el caso de los productos con matrices traspuestas las comunicaciones necesarias se complican un poco.

El algoritmo utilizado para el cálculo en paralelo de $B \leftarrow A \times B$ es:

Algoritmo 5.5 `PDGEMM2` *(con* `SIDE='L'` *y* `TRANS='N'`*)*

Para cada bloque de columnas de la matriz B:

1. *Enviar cada sub-bloque de B al proceso de la misma fila de procesos que necesitará ese sub-bloque.*
2. *Difundir estos sub-bloques por columnas, de forma que se tendrán donde se van a necesitar.*
3. *Calcular en paralelo y sin necesidad de ninguna comunicación el producto de la porción de A que tiene cada proceso por la porción de B que ha recibido.*
4. *Reducir sumando por filas los fragmentos recién calculados del producto, dejando el resultado en el bloque actual de columnas de la matriz B.*

En la figura 5.3 se muestra un esquema de las operaciones de este algoritmo para el caso de una distribución no cíclica, donde se aprecia mejor la forma de proceder.

5.3.6. PAB09AY

```
PAB09AY( DICO, JOB, ORDSEL, N, M, P, NR, A,IA,JA,DESCA,
         B,IB,JB,DESCB, C,IC,JC,DESCC, HSV,
         T,IT,JT,DESCT, TI,ITI,JTI,DESCTI, TOL,
         IWORK,LIWORK, DWORK,LDWORK, IWARN, INFO )
```

La rutina `PAB09AY` permite calcular un modelo reducido a partir de un modelo original estable. Implementa los métodos de reducción de modelos de balanceado y truncamiento de la raíz cuadrada (*the Square-Root Balance & Truncate model*

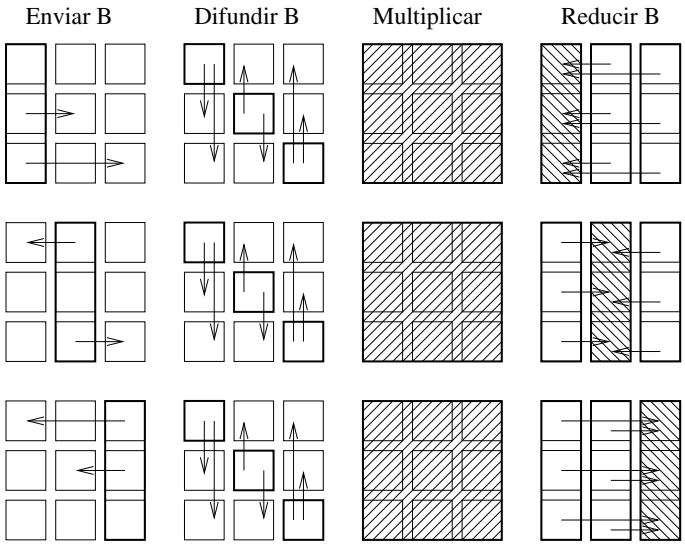

Figura 5.3: Esquema de operaciones en `PDGEMM2` (con `SIDE='L'` y `TRANS='N'`) con una distribución no cíclica

reduction method) y de balanceado y truncamiento de la raíz cuadrada sin balanceado (*the Balancing-Free Square-Root Balance & Truncate model reduction method*).

Esta rutina opera con los modelos en su representación en el espacio de estados, teniendo su matriz de estado A en forma real de Schur. Es una versión paralela de la rutina `AB09AX` de SLICOT.

Es una rutina auxiliar para la rutina `PAB09AD`. Esta otra trabaja con el modelo sin la restricción de tener la matriz A en forma real de Schur.

Aunque la mayoría de rutinas paralelas han heredado su nombre secuencial precedido de una `P` (de paralelo). En el caso de esta rutina y la `PAB09AY` el nombre usado no ha sido el habitual, que hubiera sido `PAB09AX` y `PAB09BX`, debido a que estos nombres ya habían sido utilizados para unas versiones paralelas con una funcionalidad similar, pero implementadas mediante métodos diferentes a los usados en las rutinas secuenciales. Las rutinas con esos nombres utilizan métodos iterativos para la resolución de las ecuaciones de Lyapunov involucradas. Están basadas en la función signo matricial [BCQ98] [BQQ00a].

El algoritmo utilizado en esta rutina es:

Algoritmo 5.6 `PAB09AY`

1. *Calcular el factor de Cholesky de los gramianos de controlabilidad y observabilidad del sistema a reducir mediante sendas llamadas a la rutina* `PSB03OU`*.*

2. *Calcular la descomposición en valores singulares (rutina* `PMB03UD`*) del producto de los factores de ambos gramianos (calculado mediante la rutina* `PDTTMM`*).*

3. *Obtener la matriz de transformación del sistema según el método de reducción de modelos especificado.*

4. *Aplicar la matriz de transformación a las matrices del sistema original para obtener el sistema de orden reducido.*

Es digno de mención el hecho de que tanto en esta rutina como en la rutina `PAB09BY` se utiliza una versión ligeramente modificada del algoritmo de balanceado explicado en la sección anterior. Se utiliza el algoritmo presente en las equivalentes rutinas secuenciales de SLICOT. En esta versión, a la hora de calcular la descomposición en valores singulares del producto de los factores de Cholesky de los gramianos, se utilizan ambos factores en forma triangular superior. Esta característica puede explotarse en el cálculo del producto, usando para ello la nueva rutina `PDTTMM`, que calcula el producto de dos matrices triangulares en paralelo.

5.3.7. PAB09BY

```
PAB09BY( DICO, JOB, ORDSEL, N, M, P, NR, A,IA,JA,DESCA,
         B,IB,JB,DESCB, C,IC,JC,DESCC, D,ID,JD,DESCD,
         HSV, T,IT,JT,DESCT, TI,ITI,JTI,DESCTI, TOL1,
         TOL2, IWORK,LIWORK, DWORK,LDWORK, IWARN,INFO )
```

Esta rutina es muy parecida a la rutina anterior `PAB09AY` salvo por utilizar métodos diferentes para la reducción de modelos.

La rutina `PAB09BY` permite reducir un sistema lineal de control estable mediante los métodos de reducción de modelos de aproximación de perturbación singular de la raíz cuadrada (*the Square-Root Singular Perturbation Approximation model reduction method*) y de aproximación de perturbación singular de la raíz cuadrada sin balanceado (*the Balancing-Free Square-Root Singular Perturbation Approximation model reduction method*).

Utiliza representaciones en el espacio de estados de los modelos, teniendo su matriz de estado A en forma real de Schur. Es una versión paralela de la rutina `AB09BX` de SLICOT, con el mismo ligero cambio en la nomenclatura que se ha explicado para la rutina `PAB09AY`.

Es una rutina auxiliar para la rutina `PAB09BD`. Esta otra no tiene la restricción de tener la matriz A en forma real de Schur.

El algoritmo utilizado en esta rutina es:

Algoritmo 5.7 `PAB09BY`

1. *Calcular el factor de Cholesky de los gramianos de controlabilidad y observabilidad del sistema a reducir mediante sendas llamadas a la rutina* `PSB03OU`.

2. *Calcular la descomposición en valores singulares (rutina* `PMB03UD`*) del producto de los factores de ambos gramianos (calculado mediante la rutina* `PDTTMM`*).*

3. *Obtener la matriz de transformación del sistema según el método de reducción de modelos especificado.*

4. *Aplicar la matriz de transformación a las matrices del sistema original para obtener una realización minimal del sistema.*

5. *Calcular la aproximación de perturbación singular que da el modelo de orden reducido, mediante la rutina* `PAB09DDS`.

5.3.8. PDTTMM

```
PDTTMM( SIDE,UPLO,DIAG, N, A,IA,JA,DESCA,
        B,IB,JB,DESCB, DWORK,LWORK )
```

La rutina `PDTTMM` es una versión de la rutina `PDGEMM2` para matrices triangulares. Es decir, es una rutina para calcular en paralelo el producto de dos matrices triangulares dejando el resultado en una de ellas y por tanto no necesitando espacio extra para otra matriz.

Aunque en el caso secuencial se puede realizar esta operación sin ningún espacio extra, en paralelo es necesario algo de espacio para ir recibiendo y calculando fragmentos de las diferentes matrices involucradas. Este espacio es el `DWORK` usado en la rutina. Obviamente, este espacio sigue siendo mucho menor que si se usase una matriz extra para almacenar el resultado.

Esta rutina tiene tres parámetros para especificar la operación a realizar:

- `SIDE` indica si calcular el producto por la matriz A por la izquierda (cuando vale "Left"): $B \leftarrow A \times B$ o por la derecha ("Right"): $B \leftarrow B \times A$.
- `UPLO` especifica si las matrices son triangulares superiores ("Upper") o triangulares inferiores ("Lower").
- `DIAG` indica si la diagonal de las matrices debe considerarse unitaria ("Unit") o no ("Non-unit").

El algoritmo paralelo utilizado es el mismo que el de la rutina `PDGEMM2` (algoritmo 5.5). Sin embargo, debe notarse que en la implementación se ha tenido en cuenta el hecho de que las matrices involucradas son ahora triangulares. Esto afecta tanto a que no deben comunicarse por completo como a que no deben utilizarse por completo en las operaciones realizadas. Se ha tenido en cuenta en ambas cosas la propiedad de que las matrices son triangulares, de forma que tanto las operaciones como las comunicaciones han sido reducidas significativamente.

En la figura 5.4 se muestra un esquema de las operaciones de este algoritmo para el caso de una distribución no cíclica, ya que con esta distribución se entiende mejor el proceso. Puede compararse este esquema con el similar de la rutina `PDGEMM2` (figura 5.3), con el que comparte las operaciones a realizar, pero ahora se realizan sólo con la parte triangular de las matrices.

5.3.9. PMB03UD

```
PMB03UD( JOBQ, JOBP, N, A,IA,JA,DESCA, Q,IQ,JQ,DESCQ,
         SV, DWORK,LDWORK, INFO )
```

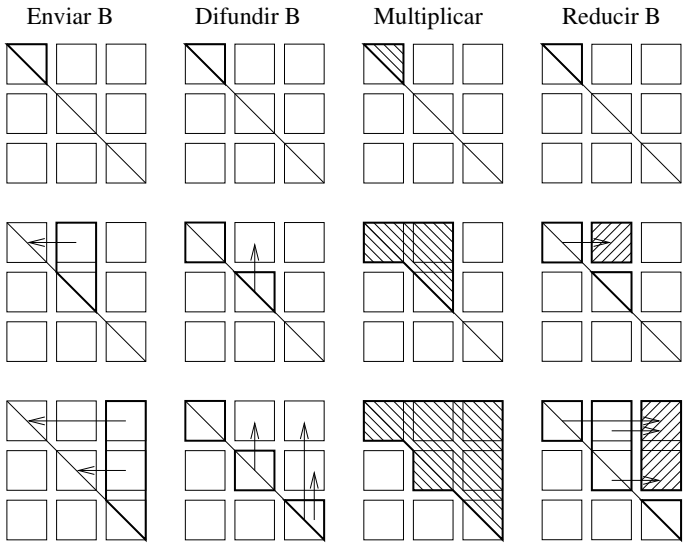

Figura 5.4: Esquema de operaciones en `PDTTMM` (con `SIDE='L'` y `UPLO='U'`) con una distribución no cíclica

La rutina `PMB03UD` calcula la descomposición en valores singulares (SVD: *Singular Value Decomposition*) [Dem97] [Dat95] de una matriz triangular superior A (aunque no se explota el hecho de que sea triangular). Es una versión paralela de la rutina secuencial equivalente de SLICOT `MB03UD`.

Precisamente en la resolución del problema de descomposición en valores singulares se ha trabajado anteriormente en colaboración con el Dr. Carlos Campos y han surgido algoritmos paralelos interesantes [CGHR04, CGHR07, CRHG08], que formaron parte de su tesis doctoral [Cam14]. Sin embargo, para este problema en concreto dentro de la reducción de modelos, las prestaciones ofrecidas por estos nuevos algoritmos no mejoran significamente a las de las rutinas de LAPACK y ScaLAPACK, razón por la que no se han utilizado. Los nuevos algoritmos funcionan mejor para matrices rectangulares, mientras que aquí se calcula la SVD de una matriz cuadrada.

Se ha utilizado también una implementación paralela del algoritmo de Golub–Solna–Van Dooren [GSD01] para calcular la SVD del producto de dos matrices sin el cálculo explícito del producto. Este algoritmo obtiene la SVD con mayor precisión a costa de trabajar con las matrices que se quiere multiplicar y no con su producto, pero para ello tiene un mayor coste computacional. Aunque en el problema de reducción de modelos sí que es cierto que se tiene que calcular la SVD de un producto de matrices, en esta ocasión la precisión obtenida calculando el producto y luego su SVD resulta suficiente, teniendo un coste bastante menor. De hecho, las rutinas secuenciales de SLICOT también lo calculan así. Para reducir aún más el coste, en el producto sí se ha aprovechado (en la rutina `PDTTMM`) el hecho de que ambas

matrices son triangulares superiores.

Al final, esta rutina de cálculo de la SVD de una matriz triangular ha quedado como una rutina puente, que simplemente llama a las funciones encargadas de realizar las diferentes etapas de la descomposición en valores singulares. El proceso que realiza es:

Algoritmo 5.8 `PMB03UD`

1. *Realizar un escalado previo de la matriz A, si su mayor elemento en valor absoluto es muy grande o muy pequeño.*
2. *Transformar la matriz a forma bidiagonal (rutina* `PDGEBRD` *de ScaLAPACK).*
3. *Si se desea, obtener las transformaciones ortogonales utilizadas (vectores singulares izquierdos y/o derechos), que el paso anterior devuelve en forma de reflectores elementales (nueva rutina* `PDORGBR`*, descrita a continuación).*
4. *Obtener los valores singulares a partir de la forma bidiagonal (nueva rutina* `PDBDSQR`*).*
5. *Deshacer el escalado, si se hizo.*

5.3.10. PDORGBR

```
PDORGBR( VECT, M,N,K, A,IA,JA,DESCA, TAU, WORK,LWORK, INFO )
```

`PDORGBR` permite obtener las matrices ortogonales Q o P^T de la transformación a forma bidiagonal ($Q^T AP = B$) realizada en la rutina de ScaLAPACK `PDGEBRD` a partir de los reflectores elementales devueltos por esta rutina. `PDORGBR` es una versión paralela de la rutina `DORGBR` de LAPACK.

Esta rutina guarda relación y cierto parecido con la rutina `PDORGHR` descrita anteriormente (en la sección 3.2.2), sólo que ahora son dos las matrices ortogonales que se deben generar (aunque sólo una en cada llamada) y cada una se genera de una forma.

Nuevamente, en la versión secuencial se realiza un desplazamiento de una submatriz de A, rellenando el resto como la matriz identidad y se llama a otra rutina de LAPACK para realizar la transformación de los reflectores a matriz ortogonal.

En esta ocasión, dependiendo de los argumentos se realiza o no el desplazamiento y se hace en una u otra dirección. Se rellena el resto de la matriz como la identidad sólo cuando hay desplazamiento. La rutina para la transformación también depende de los argumentos, pudiéndo ser `DORGLQ` (cuando se calcula P^T) además de la antes mencionada `DORGQR` (cuando se calcula Q).

Afortunadamente existen versiones paralelas en ScaLAPACK de estas rutinas con los nombres `PDORGLQ` y `PDORGQR`, lo que permite que la versión paralela de `DORGBR` resulte muy similar a la secuencial.

Al igual que en la rutina `PDORGHR`, en los casos en que hay que efectuar desplazamientos de la submatriz se ha optado por realizar estos desplazamientos después de la

transformación, consiguiendo así evitar las comunicaciones necesarias para desplazar coherentemente los vectores con los factores escalares de los reflectores (`TAU`).

Por la misma razón que entonces, ahora, cuando se produce desplazamiento y por tanto hay que rellenar una porción de la matriz como la matriz identidad, también se ha tenido que hacer parte de esto antes y parte después de la transformación y del desplazamiento.

`PDORGBR` también utiliza la rutina `PDSHIFT1` para realizar los desplazamientos de la submatriz, cuando se realizan.

5.3.11. PDBDSQR

```
PDBDSQR( UPLO, N,NCVT,NRU,NCC, D,E,SV, VT,IVT,JVT,DESCVT,
         U,IU,JU,DESCU, C,IC,JC,DESCC, WORK,LWORK, INFO )
```

La rutina `PDBDSQR` calcula la descomposición en valores singulares de una matriz bidiagonal (suministrada como los vectores `D` y `E`).

Su implementación está fuertemente basada en la implementación de esta misma operación dentro de la rutina de ScaLAPACK `PDGESVD` (que resuelve el problema completo de la SVD, por lo que incluye también otras operaciones como el paso previo de transformación de forma general a forma bidiagonal). Aunque se ha cambiado ligeramente la forma de calcular las matrices ortogonales correspondientes a los vectores singulares izquierdos y derechos.

En la rutina de ScaLAPACK se inicializan a la identidad previamente al cálculo de la descomposición en valores singulares de la matriz bidiagonal. Durante esta operación se van acumulando las transformaciones sobre las matrices identidad y, al acabar, son actualizadas con las transformaciones ortogonales obtenidas previamente en el paso de matriz general a matriz bidiagonal.

En esta nueva rutina, las transformaciones ortogonales previas se utilizan como valor inicial de las transformaciones para el cálculo de los valores singulares. De esta manera, durante el cálculo de los valores singulares se van actualizando directamente y no resulta necesaria la actualización posterior.

5.3.12. PMB01UD

```
PMB01UD( SIDE,TRANS, M,N, ALPHA,H,IH,JH,DESCH,
         A,IA,JA,DESCA, B,IB,JB,DESCB,
         DWORK,LWORK, INFO )
```

La rutina `PMB01UD` se corresponde con la paralelización de la rutina secuencial equivalente de SLICOT `MB01UD`. Su propósito es calcular de forma eficiente el producto de una matriz general `A` por una matriz en forma de Hessenberg superior `H`, dejando el resultado en una matriz `B`.

En BLAS y PBLAS hay rutinas optimizadas para el producto de matrices triangulares por generales (`DTRMM`/`PDTRMM`), pero no hay para el caso de matrices en forma de Hessenberg.

En esta rutina se aprovecha el hecho de que una de las matrices involucradas en el producto presenta la forma de Hessenberg superior. Se realizan menos operaciones y menos comunicaciones de las necesarias para el caso general.

La rutina permite especificar la posición izquierda o derecha de la matriz en forma de Hessenberg en el producto (mediante el parámetro `SIDE`) y también si se debe multiplicar por ella o por su traspuesta (parámetro `TRANS`).

El algoritmo utilizado es una versión orientada a bloques y adaptada al caso de una matriz de Hessenberg del algoritmo de producto de matrices SUMMA (*Scalable Universal Matrix Multiplication Algorithm*) [vdGW95].

Para el caso del producto $B \leftarrow \alpha \cdot A \times H$, se procede por columnas de A y filas de H:

Algoritmo 5.9 `PMB01UD` *(con* `SIDE='R'` *y* `TRANS='N'`*)*

1. *Para cada bloque de columnas de A y su correspondiente bloque de filas de H:*
   a) *Difundir por filas el bloque de columnas de A.*
   b) *Difundir por columnas el bloque de filas de H (solamente la parte de Hessenberg superior).*
   c) *Realizar el producto de los dos bloques recibidos actualizando el resultado (proceso totalmente paralelo y sin comunicaciones).*
2. *Si* `ALPHA` *no es uno, escalar el resultado por este valor (en paralelo y sin necesidad de ninguna comunicación).*

En la figura 5.5 se muestra un esquema de las operaciones de este algoritmo para el caso de una distribución no cíclica.

En el caso de la versión cíclica, que es la realmente implementada, las fases del algoritmo en que todos los procesos están involucrados en los cálculos serán más numerosas, logrando así un mayor equilibrio de la carga computacional. En este caso, las comunicaciones de los bloques de columnas de la matriz A se hacen a todos los procesos de cada fila, ya que las van a necesitar.

5.3.13. PAB09DDS

```
PAB09DDS( DICO, N,M,P,NR, A,IA,JA,DESCA, B,IB,JB,DESCB,
          C,IC,JC,DESCC, D,ID,JD,DESCD,
          RCOND, IWORK,LIWORK, DWORK,LDWORK, INFO )
```

La rutina `PAB09DDS` se corresponde con la versión paralela de la rutina de SLICOT `AB09DD`. Esta rutina permite reducir un modelo en su representación en el espacio de estados utilizando el método de fórmulas de aproximación de perturbación singular (*Singular Perturbation Approximation method*). En la sección 5.2.4 puede consultarse las operaciones que se realizan en este método.

Su paralelización es bastante inmediata debido a que su algoritmo es una sucesión de operaciones de BLAS y LAPACK que están presentes en PBLAS y ScaLAPACK.

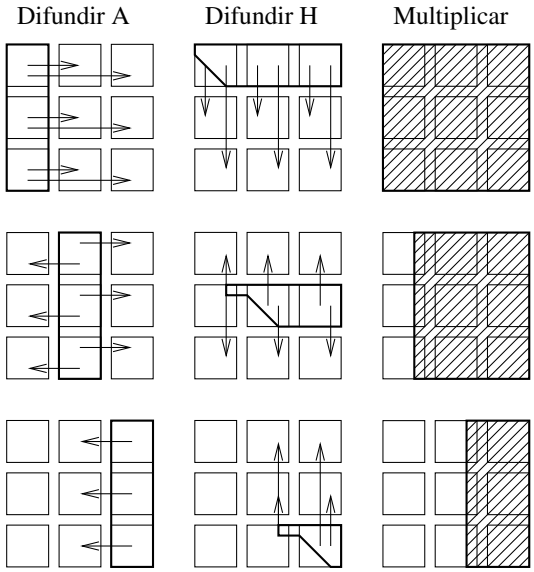

Figura 5.5: Esquema de operaciones en `PMB01UD` (con `SIDE='R'` y `TRANS='N'`) con una distribución no cíclica

Normalmente este método se usa en combinación con otros que indiquen el tamaño a utilizar en el modelo reducido (`NR`). Por ejemplo, la rutina `PAB09BY` (que es llamada por `PAB09BD`) utiliza este método como paso final tras reducir primero el sistema por otros métodos.

5.4. Resultados experimentales

En este apartado se van a mostrar los resultados prácticos obtenidos al ejecutar las nuevas rutinas de altas prestaciones desarrolladas para la reducción de modelos. Primero se muestran los resultados obtenidos al reducir un problema real de pequeño tamaño, para comprobar la corrección de las rutinas desarrolladas. Después se muestran los índices de prestaciones obtenidos al ejecutar las nuevas rutinas con problemas sintéticos de un mayor tamaño, con el propósito de ilustrar cómo las nuevas rutinas pueden utilizarse para reducir sistemas lineales de control de gran tamaño.

Aunque se han paralelizado cinco métodos de reducción de modelos, todos ellos tanto para sistemas de tiempo continuo como para sistemas de tiempo discreto, en las pruebas que siguen se ha utilizado sólo uno de ellos. Los resultados obtenidos son extrapolables al resto de métodos, puesto que los núcleos computacionales en los que se basan son los mismos.

El método utilizado ha sido uno de los dos métodos implementados en la rutina

`PAB09AD` (ver sección 5.3.1): el método de balanceado y truncamiento de la raíz cuadrada (*SR B&T: the Square-Root Balance & Truncate model reduction method*). Se ha utilizado sobre un sistema de tiempo continuo y se ha permitido a la rutina establecer de forma automática el tamaño del modelo reducido.

5.4.1. Validación

Las primeras pruebas que se han realizado han sido reducir sistemas lineales de control reales, tomados de la colección de *benchmarks* de SLICOT [CD02]. Se han probado varios de los problemas de esta colección, obteniéndose similares resultados en todos ellos. Se van a mostrar los resultados del problema conocido como *eady*. Este problema es un modelo de la zona tormentosa de la atmósfera. Tiene 598 estados, 1 entrada y 1 salida.

Estos problemas se han utilizado para comprobar la corrección del algoritmo. Son demasiado pequeños para ser utilizados en la medida de prestaciones.

Para comprobar la corrección de los métodos paralelizados, se han reducido múltiples sistemas utilizando las nuevas rutinas ejecutadas en paralelo con diferentes mallas de procesos y múltiples tamaños de bloque. Los resultados obtenidos se han comparado con los modelos reducidos que devuelven las versiones secuenciales de SLICOT para los mismos métodos.

Esta "comparación" de los resultados no es trivial. Nótese que como para un mismo sistema se pueden tener infinitas representaciones equivalentes en el espacio de estados, no se pueden comparar los resultados simplemente comprobando que las matrices del sistema sean parecidas. Normalmente no lo serán, aunque representasen el mismo sistema.

Una forma de comprobar que los sistemas obtenidos son equivalentes es observar su respuesta en frecuencia, por ejemplo comparando sus diagramas de Bode.

En la figura 5.6 se muestra el diagrama de Bode del sistema lineal de control original para el problema *eady*, junto al del modelo reducido que se ha obtenido con la rutina secuencial y los de los modelos reducidos obtenidos con la nueva rutina paralela para diferentes tamaños de bloque. En todos los casos mostrados el método empleado ha optado por reducir el sistema original de 598 estados a un modelo reducido con 91 estados.

Se ha procurado probar con diferentes tamaños de bloque, porque en las primeras versiones se tenía un algoritmo paralelo que no funcionaba bien con todos los tamaños de bloque. Este problema se describió en la sección 4.2.3 del capítulo anterior al explicar las diferentes versiones de la rutina `PSB03OT`.

En esa figura aparentemente se observa un único sistema. Esto es porque todos los casos son muy similares. En realidad, si se amplia se pueden observar dos sistemas. Uno se corresponde con el sistema original sin reducir. Y el otro, que difiere ligeramente en algunas zonas, es el sistema obtenido al reducir con la rutina secuencial de SLICOT. Todos los demás sistemas, correspondientes a la reducción con las nuevas rutinas paralelas, resultan indistinguibles del modelo reducido obtenido con la rutina de SLICOT. Esto viene a decir que las nuevas rutinas paralelas generan

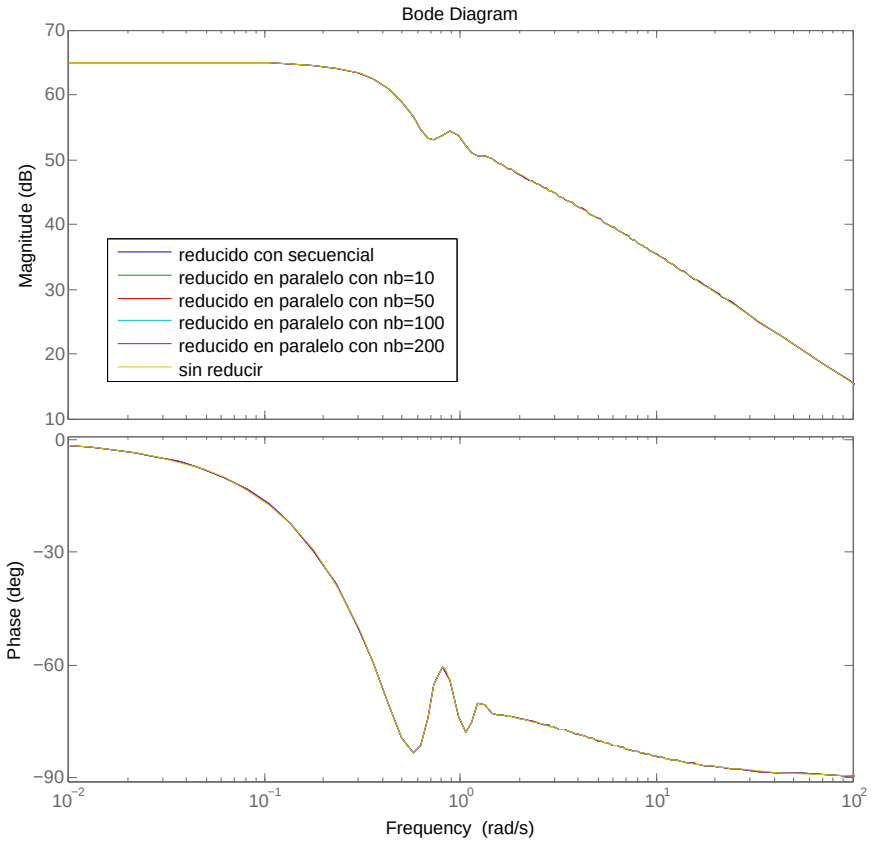

Figura 5.6: Diagrama de Bode comparando diferentes modelos del problema *eady*

sistemas equivalentes a los que se obtienen con las rutinas secuenciales existentes en la librería SLICOT.

En este caso ha sido fácil comparar los diferentes sistemas, porque el problema utilizado tiene una única variable de entrada y una única variable de salida. En los problemas sintéticos que se usan despúes, se trabaja con muchas entradas y salidas (para comprobar que los métodos funcionan con sistemas de gran tamaño). También se ha comprobado la corrección en estos casos y los resultados han sido similares. No se ha hecho con todas las entradas y salidas sino con un subconjunto al azar de ellas. Se han visualizado los diagramas de Bode correspondientes a unas pocas entradas y salidas, comparando los resultados obtenidos con las rutinas secuenciales con aquellos obtenidos con las rutinas paralelas.

5.4.2. Prestaciones paralelas

Para comprobar las buenas prestaciones de las nuevas rutinas desarrolladas y ver que pueden utilizarse para reducir sistemas de gran tamaño, se han generado sistemas sintéticos mucho más grandes. Los sistemas se han generado con valores aleatorios, cumpliendo los requisitos de los métodos implementados. Esto, en el caso continuo, es que la matriz de estados del sistema (A) debe tener todos sus valores propios con parte real negativa. Esta matriz se ha generado de la forma propuesta en [Pen98], como se describe a continuación, que permite fijar los valores propios de la matriz A sin que esta quede en una forma reducida (forma real de Schur).

Se escoge un valor arbitrario para el parámetro t que indicará cuán variados son los valores propios de la matriz y con este valor se genera la matriz A de tamaño $n \times n$:

$$\begin{aligned} A &= V^{-1}\mathrm{diag}(A_i)V,\ i = 1...n/3 \\ V &= E - I \end{aligned}$$

donde E representa una matriz con todos sus elementos a uno y los bloques diagonales A_i vienen dados por:

$$A_i = \begin{pmatrix} s_i & 0 & 0 \\ 0 & t_i & t_i \\ 0 & -t_i & t_i \end{pmatrix}$$

usando $s_i = t_i = -t^i$.

Las matrices B y C del sistema se generan cuadradas con valores aleatorios. El hecho de generarlas cuadradas es para que los requerimientos tanto espaciales como temporales del algoritmo sean mayores. Sin embargo, en problemas reales de gran tamaño no es habitual que estas matrices sean cuadradas. Si bien en problemas provenientes de discretización puede ser común aumentar mucho el tamaño de la matriz A a costa de usar un paso más fino en la discretización, las matrices B y C aumentarían con ello su número de filas y columnas, respectivamente. Pero lo normal es que su otra dimensión no crezca mucho. El número de entradas (columnas de B) y salidas (filas de C) del sistema habitualmente no suele ser superior a unas pocas decenas. Aún así, el utilizar unas matrices mayores permitirá comprobar que el algoritmo se puede aplicar a problemas grandes. Debe notarse que la diferencia entre usar estas matrices cuadradas o rectangulares afecta poco (relativamente) al tiempo total de la reducción de modelos, puesto que las operaciones que más coste tienen (transformación a forma real de Schur y descomposición en valores singulares) operan siempre con matrices cuadradas del tamaño de la matriz A, independientemente de la forma de las matrices B y C.

Se van a mostrar resultados para dos tamaños de problemas: $n = 1500$ y $n = 3000$. Para el primero se ha usado un parámetro $t = 1.02$ y para el segundo $t = 1.01$, que logran mantener un espectro variado en la matriz A sin que la distancia entre los valores propios menor y mayor de la matriz sea desorbitada.

Pruebas en el cluster *Kahan*

Se ha ejecutado el problema de tamaño $n = 3000$ en el cluster *Kahan* con un tamaño de bloque $nb = 50$, que ha resultado el más adecuado en pruebas previas. Para la selección del tamaño de bloque más adecuado se ha ejecutado el programa en 1 proceso con diferentes tamaños de bloque y se ha elegido el que tenía un tiempo menor. Más adelante se muestra más en detalle esta forma de proceder para la selección del tamaño de bloque en el otro cluster utilizado.

En la figura 5.7 se muestran los tiempos de ejecución obtenidos usando desde 1 hasta 16 procesos. Como siempre, para cada número de procesos se han utilizado en la gráfica los mejores tiempos de todas las mallas de procesos posibles con ese número de procesos.

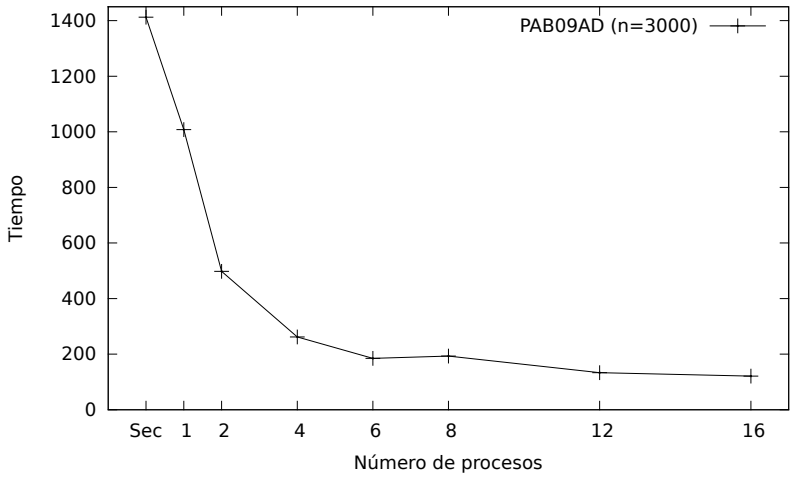

Figura 5.7: Tiempo de ejecución paralelo para la rutina `PAB09AD` con $n = 3000$ en el cluster *Kahan*

Lo primero que llama la atención en la gráfica es que el tiempo de ejecución de la rutina paralela en 1 proceso es bastante mejor que el tiempo de la rutina secuencial de SLICOT. Además, se observa como el tiempo va bajando al aumentar el número de procesos, aunque en 8 empeora un poco.

En la tabla 5.2 se muestran los speed-ups y eficiencias calculados para estos tiempos, usando como tiempo secuencial el tiempo del algoritmo paralelo en 1 proceso, que es mejor que el correspondiente a la rutina secuencial existente.

Se observan speed-ups y eficiencias muy buenos hasta 6 procesos, habiendo una ligera caída en 8 procesos y recuperándose aunque no siendo tan buenos a partir de 12 procesos. Esto cuadra perfectamente con el problema que se explicó al describir el cluster *Kahan* en la sección 3.4.1.

Procesos	1	2	4	6	8	12	16
Speed-up	1	2.02	3.85	5.45	5.22	7.55	8.33
Eficiencia	100.00	101.19	96.26	90.84	65.21	62.95	52.03

Tabla 5.2: Speed-ups y eficiencias para la rutina `PAB09AD` con $n = 3000$ en el cluster *Kahan*

Pruebas en el cluster *Tirant*

Para poder trabajar con más de 6 procesos sin que los resultados se vean afectados por esta característica del cluster *Kahan*, todas las pruebas presentadas de aquí en adelante se han realizado en el cluster *Tirant*, que no presenta este problema, aunque sus unidades de cálculo son algo más lentas que las de *Kahan*.

En algunas de las gráficas que se presentan a continuación, se muestran tiempos muy variados debido al hecho de incluir los tiempos secuenciales y/o los del código paralelo ejecutado en 1 proceso. Esto hace que no se aprecien bien los tiempos más pequeños correspondientes a ejecuciones del algoritmo paralelo en un mayor número de procesos. Allí donde se ha visto esto, se incluyen dos gráficas juntas: una con los tiempos secuenciales, para que se vea cómo bajan rápidamente, y otra sólo con los tiempos paralelos a partir de un determinado número de procesos, para que se pueda apreciar mejor la forma de la gráfica para un mayor número de procesos.

En todas las gráficas, salvo la que se muestra en función de la malla de procesos, se usan los mejores tiempos obtenidos para todas las posibles configuraciones de malla de procesos posibles. Nótese que esto es así también en las últimas gráficas, correspondientes a subrutinas que forman parte de la principal. Es decir, que allí se muestra el mejor resultado para esa subrutina, aunque pueda no ser el que se ha utilizado en los resultados mostrados para la rutina llamante, porque en estos resultaba más conveniente otra configuración. Esto se hace así para analizar cómo sería un hipotético uso individual y por separado de esas subrutinas.

Tamaño de bloque

Lo primero que se ha hecho es buscar el tamaño de bloque a utilizar en las pruebas ejecutadas en *Tirant*. Para ello se ha ejecutado el problema de tamaño $n = 1500$ en 1 proceso con diferentes tamaños de bloques. En la figura 5.8 se muestran los tiempos de ejecución que se han obtenido.

Aunque las diferencias para los diferentes tamaños de bloque no son muy notables, se va a usar un tamaño de bloque de $nb = 32$, que ha presentado (por poco) un menor tiempo de ejecución. Para todas las pruebas que siguen, se va a usar el cluster *Tirant* con problemas de tamaño $n = 3000$ y usando distribuciones con bloques de tamaño $nb = 32$.

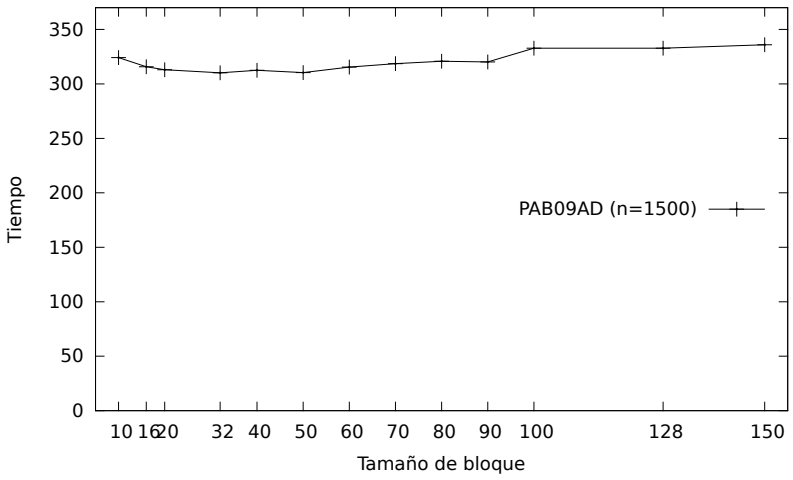

Figura 5.8: Tiempo de ejecución con tamaño de bloque variable para la rutina `PAB09AD` con $n = 1500$ en el cluster *Tirant*

Prestaciones paralelas de la rutina principal

En la figura 5.9 se muestran los tiempos de ejecución de la rutina paralela `PAB09AD` ejecutada desde 1 hasta 16 procesos, junto al tiempo de la rutina secuencial de SLICOT `AB09AD` que ha sido de 3582.16 segundos. Se aprecia como el tiempo necesario baja rápidamente al aumentar el número de procesos, siendo la bajada menor al trabajar con mayor número de procesos. Nótese cómo el tiempo continúa bajando al pasar de 6 a 8 procesos (aunque poco) debido a que ahora no se da el problema que se tenía en el cluster *Kahan*. No se produce bajada de tiempos al pasar a ejecutar a 7, 11 y 13 procesos. Esto son números primos que no permiten más que dos configuraciones de malla de procesos: o todos en horizontal o todos en vertical. El hecho de que los tiempos empeoren en estos casos hace sospechar que la malla óptima va a ser la cercana a una configuración cuadrada de la malla de procesos.

En la figura 5.10 se muestra un subconjunto de los tiempos de ejecución de la rutina paralela, esta vez separados según la configuración usada de la malla de procesos.

Efectivamente se comprueba que la malla que está resultando más adecuada es la más cercana a una malla cuadrada. Esto suele ocurrir cuando hay muchas operaciones y subrutinas auxiliares involucradas. En principio, es probable que las operaciones de más coste funcionen mejor en mallas cuadradas. Pero además, si hay algunas operaciones que tienen un mejor comportamiento en mallas horizontales y otras que lo tienen en mallas verticales, normalmente convendrán las mallas cuadradas porque son un caso en el que ninguna de esas operaciones se encuentra con su configuración más desfavorable (aunque tampoco se encontrará ninguna de ellas en

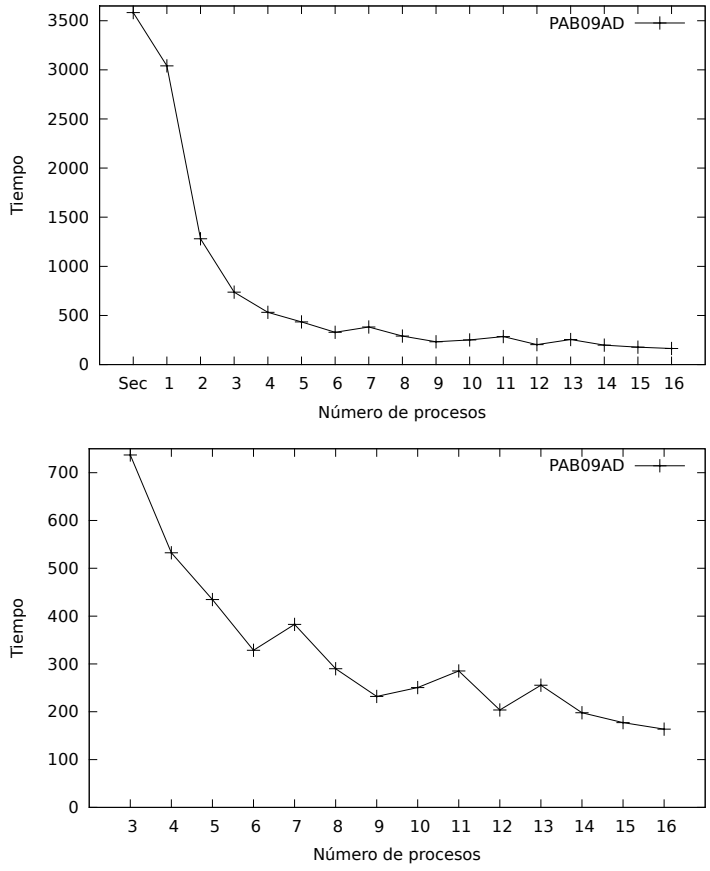

Figura 5.9: Tiempo de ejecución paralelo para la rutina `PAB09AD` con $n = 3000$ en el cluster *Tirant*

la más favorable).

En la tabla 5.3 se muestran los speed-ups y eficiencias de esta rutina. Son valores de speed-ups y eficiencias muy buenos, más si cabe si recordamos que se están calculando tomando como base el tiempo de ejecución del algoritmo paralelo en 1 proceso, que resulta bastante mejor que el del algoritmo secuencial disponible.

Con 16 procesadores se está resolviendo el problema 18.57 veces más rápido que con 1 proceso y si lo comparamos con la rutina secuencial de SLICOT, la ganancia de velocidad es de 21.88. De hecho, si se tuviera que resolver este problema con la rutina secuencial existente, se habrían necesitado 3582.16 segundos (casi 1 hora), mientras que en paralelo con 16 procesos se ha podido resolver en tan sólo 163.69 segundos (algo menos de 3 minutos). Con esta nueva implementación paralela se van a poder reducir sistemas lineales de control de gran tamaño en un tiempo mucho

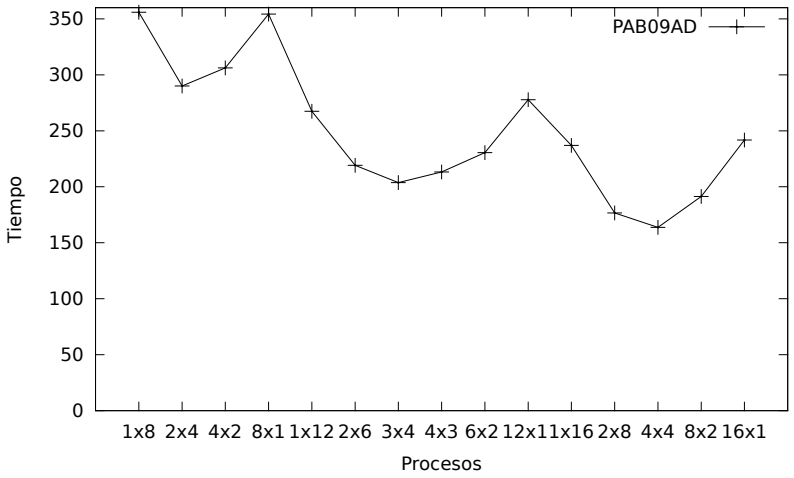

Figura 5.10: Tiempo de ejecución con malla de procesos variable para la rutina `PAB09AD` con $n = 3000$ en el cluster *Tirant*

Procesos	1	2	4	8	12	16
Speed-up	1.00	2.37	5.71	10.48	14.93	18.57
Eficiencia	100.00	118.66	142.76	131.01	124.39	116.08

Tabla 5.3: Speed-ups y eficiencias para la rutina `PAB09AD` con $n = 3000$ en el cluster *Tirant*

menor que el necesario por los códigos secuenciales de SLICOT.

A pesar de que estos datos ilustran unas muy buenas prestaciones del algoritmo paralelo, no es normal que se consiga reducir el tiempo de ejecución más que el factor dado por el número de procesos en que se ejecuta. Se está produciendo speed-up superlineal. Esto suele ser un síntoma de que el algoritmo secuencial usado en el cálculo de los índices de prestaciones no es el óptimo. Hay alguna(s) operación(es) del algoritmo que se podría(n) implementar de forma más eficiente en el caso de ser ejecutada en 1 proceso, quizás sea por una mala o inexistente orientación a bloques de esta operación o tal vez porque el sobrecoste añadido por operaciones necesarias en el código paralelo empeora notablemente el tiempo de ejecución al ejecutar en 1 solo proceso. La existencia de speed-up superlineal suele indicar que se podría optimizar más el algoritmo para hacerlo funcionar mejor cuando es ejecutado en 1 solo proceso.

Prestaciones paralelas de operaciones principales

A continuación se muestran estos mismos tiempos de ejecución desglosados en las operaciones de mayor coste que aparecen en el proceso de reducción de modelos. Analizar el coste desglosado en subrutinas permite identificar la rutina que parece la causante del speed-up superlineal. Al final de esta sección (en la tabla 5.8), se muestran unos nuevos speed-ups y eficiencias 'ficticios' para el proceso completo de reducción de modelos, calculados mediante una simulación en la que se calcula de forma aproximada el tiempo de la rutina problemática, lo que evita el speed-up superlineal.

Una de las operaciones más costosas es la transformación del sistema a una representación equivalente en la que la matriz A esté en forma real de Schur. Esto se hace en la rutina `PTB01WD`, que internamente realiza la transformación de la matriz A a forma de Hessenberg (rutina `PDGEHRD`), la transformación de la matriz usada en ese proceso que es devuelta en forma de reflectores a matriz ortogonal (rutina `PDORGHR`), la transformación de forma de Hessenberg a forma real de Schur (rutina `PDHSEQR`) y una serie de multiplicaciones matriciales para aplicar todas estas transformaciones al resto de matrices del sistema. Nótese que las primeras tres operaciones son las que realiza la rutina `PDGEES`, que se analizó en el capítulo 3.

El tiempo utilizado en todas estas operaciones puede verse en la figura 5.11.

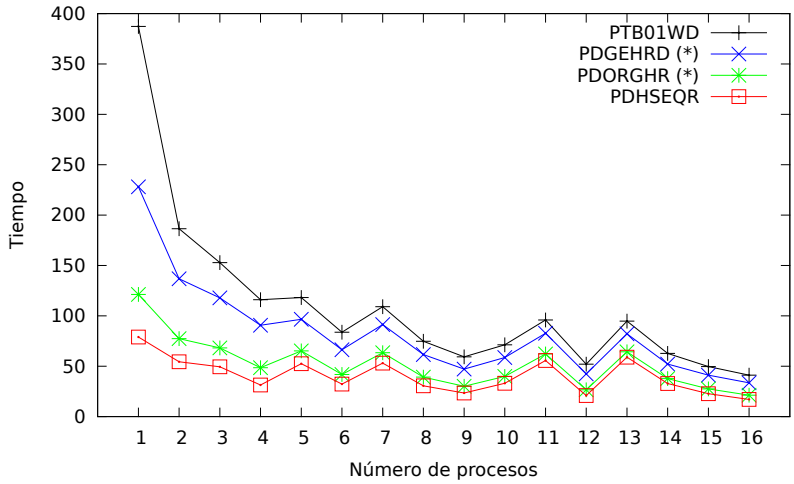

Figura 5.11: Tiempo de ejecución paralelo para la rutina `PTB01WD` con $n = 3000$ en el cluster *Tirant*

En está figura, y también en la figura 5.13, los tiempos de las rutinas que tienen un asterisco están dibujados acumulados sobre los de la rutina que tienen inmediatamente debajo. De esta manera se puede apreciar la porción del tiempo total de la rutina principal en estas gráficas que ha necesitado cada una de las otras rutinas

que aparecen en ellas.

En este caso, en la gráfica no está indicado de forma explícita el tiempo invertido en las actualizaciones del resto de matrices. Este tiempo se corresponde con el espacio entre la curva de `PDGEHRD` y la de `PTB01WD`, ya que este es el tiempo no asignado al resto de operaciones.

Al examinar esta gráfica con los tiempos de la rutina `PTB01WD` se observa que al principio tienen mucho peso las operaciones de transformación a forma de Hessenberg y actualización del resto de matrices del sistema. A medida que se aumenta el número de procesos, el coste relativo de estas operaciones decae rápidamente, síntoma de una buena paralelización, y se hace más importante el coste de la transformación a forma real de Schur, que no se reduce tan bien. Se aprecia que el mal comportamiento en mallas poco cuadradas es principalmente debido también a esta parte de los cálculos.

Procesos	1	2	4	8	12	16
Speed-up	1	2.08	3.34	5.18	7.44	9.41
Eficiencia	100.00	103.81	83.47	64.71	62.04	58.79

Tabla 5.4: Speed-ups y eficiencias para la rutina `PTB01WD` con $n = 3000$ en el cluster *Tirant*

En la tabla 5.4 se muestran los speed-ups y eficiencias de esta rutina `PTB01WD`. Los speed-ups y eficiencias son adecuados. A pesar de que hay un pequeño speed-up superlineal para 2 procesos, el resto de resultados indican que no es en esta parte de los cálculos donde se provoca el speed-up superlineal de la rutina principal.

La siguiente operación costosa es la reducción de modelos del sistema con la matriz de estados en forma real de Schur. Esto se hace en la rutina `PAB09AY`, cuyos tiempos de ejecución pueden verse en la figura 5.12.

Sus correspondientes speed-ups y eficiencias se muestran en la tabla 5.5.

Procesos	1	2	4	8	12	16
Speed-up	1	2.42	6.37	12.32	17.49	22.21
Eficiencia	100.00	121.19	159.28	154.04	145.78	138.84

Tabla 5.5: Speed-ups y eficiencias para la rutina `PAB09AY` con $n = 3000$ en el cluster *Tirant*

Son valores muy buenos que indican un buen comportamiento del algoritmo paralelo. Aunque se vuelve a apreciar el speed-up superlineal que se observó en la rutina principal. Esto indica que alguna de las operaciones aquí presentes es la que se puede mejorar en su ejecución secuencial.

De las operaciones realizadas en esta rutina, la más costosa es la descomposición en valores singulares de una matriz triangular superior, realizada por la nueva rutina paralela `PMB03UD`. Esta rutina primero transforma la matriz a forma bidiagonal (rutina `PDGEBRD`), luego transforma las matrices usadas en ese proceso que

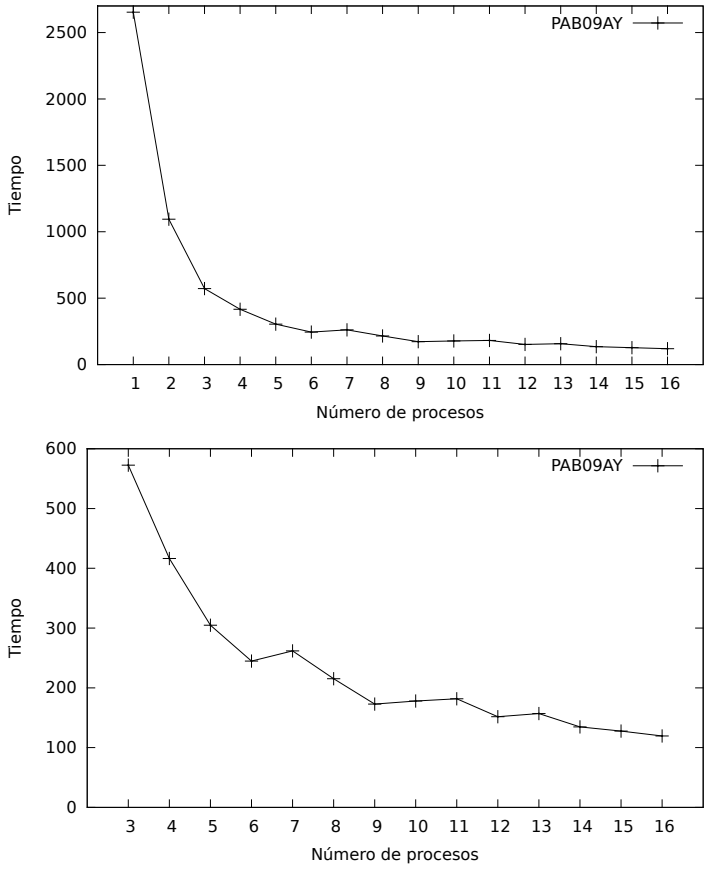

Figura 5.12: Tiempo de ejecución paralelo para la rutina `PAB09AY` con $n = 3000$ en el cluster *Tirant*

son devueltas en forma de reflectores a matrices ortogonales (rutina `PDORGBR`, que es llamada 2 veces) y luego calcula la descomposición en valores singulares de la matriz bidiagonal (rutina `PDBDSQR`). Pueden verse los tiempos de estas operaciones en la figura 5.13.

Los tiempos de la transformación a forma bidiagonal no están indicados explícitamente en la gráfica, pero se corresponden con el breve espacio entre la curva de 2×`PDORGBR` y la de `PMB03UD`, ya que este es el tiempo no asignado al resto de operaciones.

En la tabla 5.6 se muestran los speed-ups y eficiencias de esta rutina. Se observan valores muy buenos tanto para el speed-up como para la eficiencia. De hecho, vuelve a aparecer speed-up superlineal. Es esta parte de los cálculos la causante del speed-up superlineal en las rutinas que llaman a esta.

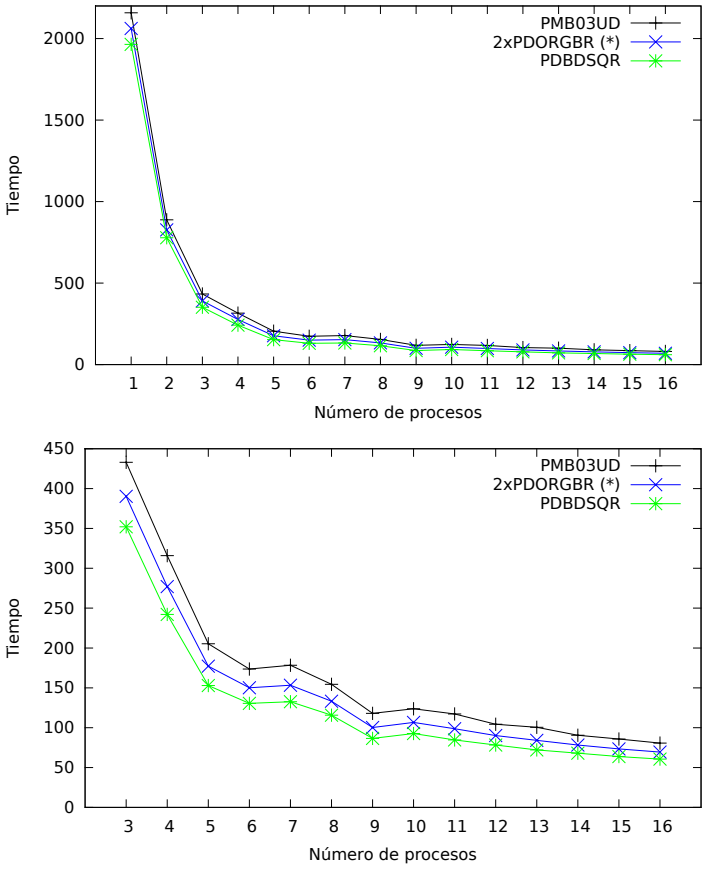

Figura 5.13: Tiempo de ejecución paralelo para la rutina `PMB03UD` con $n = 3000$ en el cluster *Tirant*

Observando los tiempos de ejecución en la figura 5.13, se aprecia que es la operación del cálculo de la descomposición en valores singulares (la SVD) de la matriz bidiagonal la que supone prácticamente todo el tiempo empleado en esta rutina. Es por tanto esta operación, realizada por la rutina `PDBDSQR`, la causante del speed-up superlineal que aparece en múltiples rutinas.

Lo primero que llama la atención es que sea la SVD de la matriz bidiagonal la operación que más tiempo necesita en esta rutina. Esta operación tradicionalmente tiene un coste pequeño, por ejemplo comparada con la operación de transformación de forma general a forma bidiagonal. Lo que ocurre es que en este caso, aparte de actualizar la matriz bidiagonal, se tienen que actualizar las dos matrices ortogonales donde se acumulan las transformaciones realizadas y es aquí donde se dispara el tiempo.

Procesos	1	2	4	8	12	16
Speed-up	1	2.43	6.83	13.98	20.69	26.76
Eficiencia	100.00	121.45	170.80	174.74	172.41	167.25

Tabla 5.6: Speed-ups y eficiencias para la rutina `PMB03UD` con $n = 3000$ en el cluster *Tirant*

Se ha comprobado que es la rutina secuencial de LAPACK `DBDSQR` la causante de este tiempo excesivo. Esto ocurre al actualizar las matrices ortogonales necesarias, porque cuando se ha ejecutado esta rutina `DBDSQR` sin calcular estas matrices (ejecución realizada a propósito para comprobar este hecho), el tiempo de `PDBDSQR` ha sido ridículamente pequeño (1.4 segundos).

Según parece, esta rutina de LAPACK no está bien orientada a bloques y esto causa que se produzca speed-up superlineal en su versión paralela, porque al ejecutar esta en paralelo se comporta como una versión orientada a bloques de la secuencial.

Es curioso que todos estos efectos se vean amplificados en el cluster *Tirant*. En el cluster *Kahan* no se ha apreciado este problema al ejecutar en 1 solo proceso. Probablemente la plataforma *Tirant* sea más sensible en los accesos a memoria. Además, las pruebas realizadas en este cluster han sido más grandes que las usadas en *Kahan*: matrices con el doble de tamaño, lo que supone 4 veces más memoria (sólo en las matrices).

Prestaciones paralelas de subrutinas auxiliares

Para terminar con los resultados de las pruebas realizadas, se ilustran a continuación los tiempos de ejecución de algunas de las otras rutinas desarrolladas. Estas otras rutinas aportan poco coste al proceso general, pero este hecho no ha sido razón para descuidar su paralelización.

Por ejemplo, en las figuras 5.14 y 5.15 se muestran los tiempos de las rutinas `PDTTMM` y `PMB01UD`, respectivamente. Estas rutinas sirven para multiplicar matrices triangulares, la primera, y una matriz general por una en forma de Hessenberg, la segunda. En ambos casos se observa un buen comportamiento en paralelo, ya que los tiempos van bajando de la forma característica al incrementar el número de procesos.

En la rutina `PDTTMM` incluso se aprecia, otra vez, speed-up superlineal. En esta ocasión es causado por las múltiples copias y operaciones extras que son necesarias al realizar el cálculo en paralelo. Estas operaciones también se efectúan cuando se ejecuta en 1 proceso, aunque en este caso se podría proceder de otra manera en la que no serían necesarias. Este speed-up superlineal indica que esta rutina podría optimizarse para que su ejecución en 1 proceso fuera más rápida.

Simulación de prestaciones paralelas

Para finalizar este apartado de resultados experimentales, se van a recalcular de forma "simulada" los speed-ups y las eficiencias de las rutinas `PMB03UD` y `PAB09AD`.

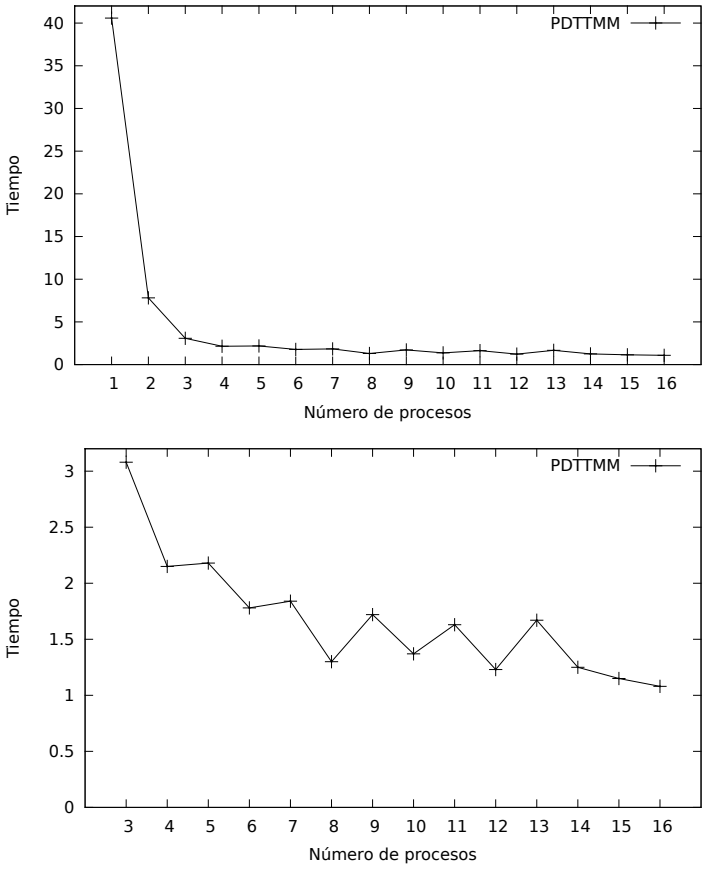

Figura 5.14: Tiempo de ejecución paralelo para la rutina `PDTTMM` con $n = 3000$ en el cluster *Tirant*

Estas rutinas presentan un notable speed-up superlineal, que se ha visto es por culpa de una mala orientación a bloques de la rutina secuencial `DBDSQR`. Si esta rutina estuviera bien orientada a bloques, su versión paralela `PDBDSQR` en ningún caso podría superar una eficiencia del 100 %.

Examinando los resultados obtenidos con esta rutina, se observa que su mayor eficiencia (superior al 100 %) ha sido para 8 procesos. Por tanto, con una buena implementación secuencial, el tiempo de la rutina secuencial debería ser como mucho 8 veces el tiempo de la rutina paralela ejecutada en 8 procesos. Se ha cambiado este tiempo de la rutina secuencial para recalcular los speed-ups y eficiencias. Esto ha supuesto que el tiempo de ejecutar en 1 proceso la rutina `PDBDSQR` que era de 1964.08 segundos, ahora se considere que es de 925.60 segundos (es un valor ficticio). Además, los tiempos en 2 y 4 procesos, donde también se superaba una eficiencia del 100 % se

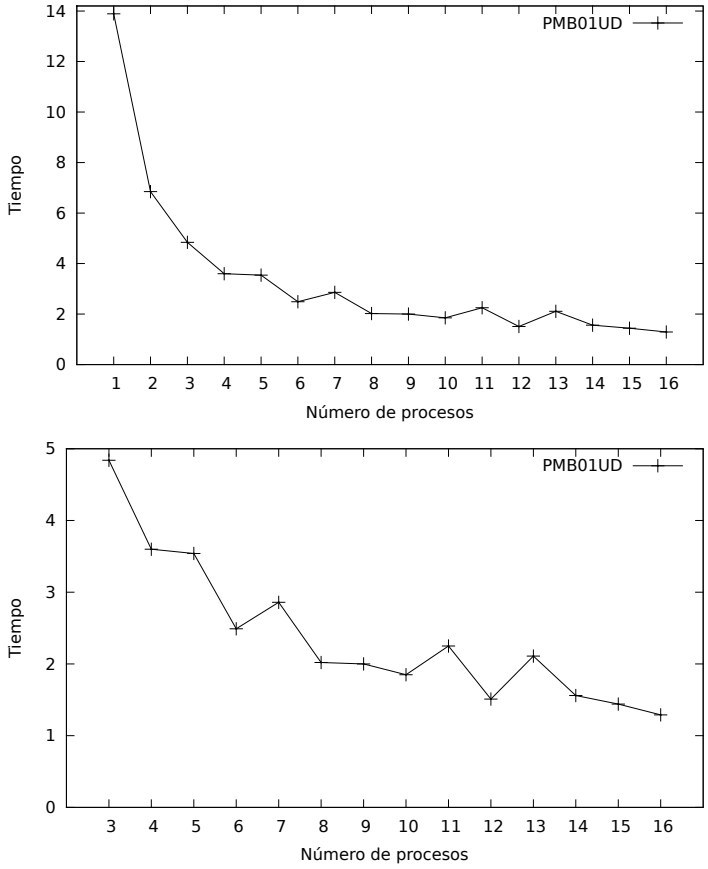

Figura 5.15: Tiempo de ejecución paralelo para la rutina `PMB01UD` con $n = 3000$ en el cluster *Tirant*

han puesto de tal forma que no se supere esa eficiencia del 100 %. Con esto, se tienen unos tiempos "ficticios" para la rutina `PDBDSQR`, pero que tratan de aproximar lo que ocurriría si se dispusiera de una versión secuencial adecuada para la rutina `DBDSQR`.

Sustituyendo los tiempos de `PDBDSQR` por su versión simulada en los tiempos de la rutina `PMB03UD` se obtienen los speed-ups y eficiencias que se muestran en la tabla 5.7.

Cambiando en los tiempos de la rutina principal de reducción de modelos `PAB09AD` la aportación de la rutina `PMB03UD` por sus nuevos tiempos simulados, se obtienen para todo el proceso los speed-ups y eficiencias que se muestran en la tabla 5.8. Estos resultados, aún siendo ficticios, se aproximan a lo que se obtendría con una buena implementación secuencial de la rutina `DBDSQR`. Puede verse que son resultados adecuados para un algoritmo paralelo que trabaja con matrices densas.

Procesos	1	2	4	8	12	16
Speed-up	1.00	1.95	3.67	7.25	10.73	13.88
Eficiencia	100.00	97.68	91.73	90.65	89.44	86.76

Tabla 5.7: Speed-ups y eficiencias "simulados" para la rutina `PMB03UD` con $n = 3000$ en el cluster *Tirant*

Procesos	1	2	4	8	12	16
Speed-up	1.00	2.07	3.84	6.90	9.83	12.23
Eficiencia	100.00	103.64	95.93	86.26	81.90	76.43

Tabla 5.8: Speed-ups y eficiencias "simulados" para la rutina `PAB09AD` con $n = 3000$ en el cluster *Tirant*

5.5. Conclusiones

Se han paralelizado las principales rutinas de reducción de modelos para sistemas estables presentes en la librería SLICOT. Se trata de los métodos de reducción de modelos de balanceado y truncamiento de la raíz cuadrada (*the Square-Root Balance & Truncate model reduction method*) con o sin balanceado y con o sin usar las fórmulas de perturbación singular.

Aparte de las rutinas principales a cargo de la reducción de modelos, también se han paralelizado todas aquellas otras operaciones que utilizan para las que no se disponía de una versión paralela.

Gracias a estos desarrollos, ahora se dispone de una implementación de altas prestaciones para estas rutinas de reducción de modelos, siguiendo los mismos buenos métodos y algoritmos que utilizan las versiones secuenciales en SLICOT.

Las pruebas prácticas realizadas muestran que las nuevas rutinas permiten obtener modelos reducidos en un tiempo muy inferior al necesario usando las equivalentes rutinas de SLICOT. Incluso la ejecución usando 1 solo proceso requiere de un menor tiempo de ejecución.

La precisión comprobada de las nuevas rutinas es equivalente a la obtenida con las rutinas secuenciales, independientemente del tamaño de bloque usado en la distribución de los datos.

Como consecuencia, se puede decir que las nuevas rutinas desarrolladas ofrecen alternativas de altas prestaciones a las versiones de SLICOT para realizar su misma función. Esto permitirá trabajar con sistemas lineales de control mucho más grandes, que es lo que se pretendía.

A pesar de que una parte importante del proceso de reducción de modelos supone calcular la descomposición en valores singulares de una matriz y se había trabajado previamente en nuevos algoritmos para esta operación, al final estos algoritmos no han sido utilizados. Las prestaciones de las rutinas existentes en LAPACK y ScaLAPACK han sido adecuadas para la reducción de modelos en paralelo.

A este respecto, resulta curioso el mal comportamiento secuencial que se ha comprobado para la rutina de LAPACK `DBDSQR`, a cargo de la obtención de la descomposición en valores singulares de una matriz bidiagonal, cuando se le requieren las matrices ortogonales usadas en la transformación. Sus prestaciones secuenciales no son adecuadas cuando se trabaja con problemas grandes. Es muy probable que no esté bien orientada a bloques o que no lo esté en absoluto.

Tampoco se ha utilizado en la descomposición en valores singulares el algoritmo que aprovecha el hecho de que sea de un producto de matrices. Supuestamente en estos casos conviene calcular la descomposición sin realizar explícitamente el producto, para así obtener una mejor precisión en el resultado. Sin embargo, la precisión de la descomposición realizada tras el producto ha resultado suficiente en las pruebas realizadas (como ocurre con la implementación secuencial de SLICOT) y además es mucho más eficiente, ya que en el producto matricial se aprovecha el hecho de que ambas matrices sean triangulares superiores. Al final, se ha dejado la forma de proceder que hay en las rutinas secuenciales de SLICOT, pero ahora implementada en paralelo.

Capítulo 6

Conclusiones

Este último capítulo concluye la memoria de la tesis. En él, primero se comentan brevemente las conclusiones más relevantes de los objetivos alcanzados con la tesis. Después, se enumeran los artículos publicados como consecuencia del trabajo desarrollado en el marco de la tesis. Por último, se mencionan algunas líneas de trabajo que se abren y constituyen posibles direcciones en las que continuar el trabajo empezado aquí.

6.1. Conclusiones del trabajo desarrollado

El trabajo realizado en esta tesis ha ido dirigido a obtener implementaciones de altas prestaciones para algoritmos de reducción de modelos. Para ello se han paralelizado todas las operaciones matemáticas necesarias de las que no se disponía aún versiones paralelas apropiadas. Como consecuencia, se han desarrollado 27 nuevas rutinas paralelas (y algunas más secuenciales) que resuelven problemas tan variados como simples multiplicaciones de matrices, desplazamientos, descomposiciones QR, problemas de valores propios, descomposición en valores singulares, ecuaciones de Lyapunov, ecuaciones de Sylvester, reducción de modelos, etc.

Todas las rutinas desarrolladas utilizan en lo posible herramientas de librerías estándar, para fomentar su portabilidad y sus buenas prestaciones. De igual modo, la interfaz que ofrecen es similar a la ofrecida por las rutinas de estas conocidas librerías, para facilitar su uso por el usuario acostumbrado a trabajar con este tipo de librerías numéricas.

De los múltiples métodos de reducción de modelos existentes se ha trabajado con aquellos que permiten una mayor aplicabilidad y que además son una base necesaria para otros métodos.

La principal aportación de la tesis en el campo de la reducción de modelos es ofrecer implementaciones de altas prestaciones completamente funcionales para cinco métodos de reducción de modelos. Se trata de diferentes variedades del método de balanceado y truncamiento de componentes de la raíz cuadrada (*the Square-Root*

Balance & Truncate model reduction method): con o sin balanceado y con o sin usar las fórmulas de perturbación singular. Todos ellos han sido implementados tanto para sistemas en tiempo continuo como para sistemas en tiempo discreto.

Las tres rutinas principales encargadas de la reducción de modelos son: `PAB09AD`, `PAB09BD` y `PAB09DDS`. Estas rutinas ofrecen la misma funcionalidad que las rutinas secuenciales equivalentes de la librería SLICOT, pero efectuando todas las operaciones en paralelo. Gracias a estas nuevas implementaciones, ahora es posible resolver los mismos problemas que antes, pero requiriendo un menor tiempo. Lo que es más importante, también es posible utilizarlas para resolver problemas mucho más grandes.

Los experimentos realizados confirman una precisión semejante a la obtenida con las rutinas secuenciales y una velocidad de cómputo superior, incluso al ejecutar las rutinas paralelas usando un único procesador.

De las muchas operaciones que han necesitado implementaciones paralelas para realizar la reducción de modelos en paralelo, merecen especial mención la resolución de ecuaciones de Lyapunov y la obtención de la forma real de Schur.

A la hora de resolver ecuaciones de Lyapunov existen múltiples métodos a utilizar. Pero en la resolución de estas ecuaciones para la reducción de modelos, la ecuación tiene unas características especiales y se desea la solución en un formato en concreto. Para ambas cosas resulta mucho más conveniente el método de Hammarling, que es el que ha sido empleado.

Se ha paralelizado el método de Hammarling para la resolución de ecuaciones de Lyapunov, tanto en tiempo continuo como en tiempo discreto y con las matrices de la ecuación normales o traspuestas. Se ha utilizado para ello una orientación a bloques especial que mantiene la buena estabilidad numérica presente en las implementaciones secuenciales.

Los resultados experimentales muestran buenas prestaciones en la paralelización del método, que ahora permite su utilización en ecuaciones de mucho mayor tamaño.

Un posible punto débil de esta paralelización del método de Hammarling es que sus mejores prestaciones se dan en mallas de procesos alargadas, lo que a veces no es conveniente. Por ejemplo, en su uso para la reducción de modelos suelen convenir mallas más cuadradas (por el resto de operaciones que intervienen), con lo que las prestaciones obtenidas para esta operación no son las mejores posibles para un determinado número de procesos. De todas formas, en el problema de la reducción de modelos el mayor coste computacional recae en otras operaciones y es suficiente con poder resolver la ecuación de Lyapunov en paralelo aunque no se haga en el menor tiempo que sería posible.

Las rutinas paralelas para resolver la ecuación de Lyapunov son independientes del resto de rutinas para la reducción de modelos. Esto hace que se puedan utilizar de forma autónoma en otras aplicaciones no necesariamente relacionadas con la reducción de modelos.

Otra operación importante en la reducción de modelos con la que se ha tenido que trabajar es la transformación de una matriz a forma real de Schur.

Aunque existen implementaciones paralelas para esta operación, las versiones disponibles no eran eficientes. Se ha trabajado con versiones más modernas, pero

que aún no habían sido convenientemente depuradas/refinadas. Tras un trabajo importante se ha conseguido tener una versión funcional, lo que ha permitido utilizarla en el problema de reducción de modelos reduciendo considerablemente el tiempo de esta operación con respecto al que se tenía con la versión anterior.

Aprovechando el trabajo invertido en estas rutinas para la transformación a forma real de Schur, se ha continuado ampliando el trabajo para permitir aplicarlo al problema completo de obtención de valores y vectores propios de una matriz. Para ello se ha desarrollado una rutina paralela que permita obtener los vectores propios a partir de los vectores de Schur, ya que para esta operación no se disponía de implementación paralela.

En este contexto, se ha colaborado con el Imperial College de Londres en un problema de simulación del flujo oceánico. En este problema hay un paso del proceso en que se necesita calcular los valores y vectores propios generalizados sobre matrices de gran tamaño.

Viendo que las características de este problema lo permitían, se ha desarrollado una nueva rutina paralela para resolver el problema generalizado transformándolo en un problema estándar. Gracias a esta nueva rutina se ha podido resolver problemas de valores propios generalizados de tamaño muy grande, habiendo llegado a resolver problemas con matrices de orden 444675. A pesar de que las matrices de entrada son dispersas, al proceder de un método de discretización, en esta aplicación se requieren todos los vectores propios (hecho poco habitual), lo que ha hecho conveniente el uso de métodos para matrices densas.

Este caso es un ejemplo del tipo de problemas que no podrían resolverse sin utilizar computación de altas prestaciones. Hacen falta máquinas de altas prestaciones como los supercomputadores que hoy en día hay en muchos centros de cálculo. Pero de nada servirían estos supercomputadores si no se implementan algoritmos paralelos que los utilicen eficientemente, algoritmos como los desarrollados en esta tesis.

Para terminar, se enumeran a continuación las principales operaciones para las que ahora se dispone de implementación paralela de altas prestaciones gracias al trabajo realizado en esta tesis. Para algunas, se ha paralelizado prácticamente todas las operaciones necesarias. Para otras, sólo se ha necesitado paralelizar algunas pequeñas operaciones que faltaban. Aunque hay muchas otras funciones que se han paralelizado, aquellas relacionadas con operaciones más generales y con un mayor ámbito de aplicación permiten resolver en paralelo los siguientes problemas:

- Reducción de sistemas lineales de control estables (tanto en tiempo continuo como en tiempo discreto).

- Resolución de la ecuación estándar de Lyapunov obteniendo el factor de Cholesky de la solución (tanto en tiempo continuo como en tiempo discreto, en versiones normal y traspuesta).

- Obtención de valores y vectores propios del problema estándar de valores propios (caso no simétrico).

- Obtención de valores y vectores propios del problema generalizado de valores propios (pero sólo en el caso de trabajar con una matriz B no singular).

6.2. Publicaciones

Como consecuencia del trabajo desarrollado en el campo de la reducción de modelos y la resolución de ecuaciones de Lyapunov, se han generado las siguientes comunicaciones a congresos, con sus respectivas publicaciones en los libros de actas correspondientes:

- *Resolución en Paralelo de la Ecuación de Lyapunov Generalizada por el Método de Hammarling en una Red de PCś*, David Guerrero, Vicente Hernández, José E. Román, Antonio M. Vidal, VIII Jornadas de Paralelismo, páginas 61-70, Cáceres, Septiembre 1997.
- *Resolución en Paralelo de la Ecuación de Lyapunov Generalizada por el Método de Hammarling*, David Guerrero, Vicente Hernández, José E. Román, Antonio M. Vidal, XV Congreso de Ecuaciones Diferenciales Y Aplicaciones / V Congreso de Matematica Aplicada (CEDYA/CMA), páginas 799-804, Vigo, Octubre 1997.
- *Parallel Algorithms for the Cholesky Factor of Generalized Lyapunov Equations*, David Guerrero, Vicente Hernández, José E. Román, Antonio M. Vidal, 5^{th} IFAC Workshop on Algorithms and Architectures for Real-Time Control (AARTC 98), páginas 237-242, Cancún (México), Abril 1998.
- *Parallel SLICOT Model Reduction Routines: The Cholesky Factor of Grammians*, David Guerrero, Vicente Hernández, José E. Román, 15^{th} World Congress of the International Federation of Automatic Control (IFAC), Barcelona, Julio 2002.
- *La reducción de modelos en problemas de control*, David Guerrero, Vicente Hernández, IV Jornadas de Investigación y Fomento de la Multidisciplinariedad (IMM 2002), páginas 3-11, Valencia, Septiembre 2002.
- *Algoritmo Paralelo para la Resolución de la Ecuación de Sylvester con Valores Propios Reales*, Juan Manuel Solís, David Guerrero, Vicente Hernández, Congreso internacional de Computación Paralela, Distribuida y Aplicaciones, Instituto Tecnologico de Linares, Linares (México), Septiembre 2003.
- *Improving accuracy of Parallel SLICOT model reduction routines for stable systems*, David Guerrero-López, José E. Román, 23^{th} Mediterranean Conference on Control and Automation (MED 2015), páginas 398-403, Torremolinos, Junio 2015.

También se ha colaborado en el trabajo de resolución del problema de la descomposición en valores singulares, fruto de lo cual se han generado diferentes comunicaciones a congresos y una publicación en revista:

- *Parallel bidiagonalization of a dense matrix*, Carlos Campos, David Guerrero, Vicente Hernández y Rui Ralha, V International Workshop on Accurate Solution of Eigenvalue Problems, Hagen (Alemania), Junio 2004.

- *Algoritmos de altas prestaciones para la bidiagonalización de matrices densas*, Carlos Campos, David Guerrero, Vicente Hernández y Rui Ralha, XV Jornadas de Paralelismo, páginas 78-83, Almería, Septiembre 2004.

- *A new bidiagonalization method*, Carlos Campos, David Guerrero, Vicente Hernández y Rui Ralha, Jornada INGRID 06, Braga, Noviembre 2006.

- *Parallel bidiagonalization of a dense matrix*, Carlos Campos, David Guerrero, Vicente Hernández y Rui Ralha, SIAM Journal on Matrix Analysis and Applications, volumen 29, número 3, páginas 826-837, Julio 2007.

- *Towards a parallel code without communication for the eigenvalues of symmetric tridiagonals*, Carlos Campos, Rui Ralha, Vicente Hernández y David Guerrero, Parallel Matrix Algorithms and Applications, Neuchatel (Suiza), Junio 2008.

- *Nuevos resultados en la bidiagonalización de matrices densas*, Carlos Campos, Rui Ralha, Vicente Hernández y David Guerrero, XIX Jornadas de Paralelismo, Castellón, Septiembre 2008.

A pesar de ello, al final su aplicación en los algoritmos de este trabajo no ha resultado más eficiente que la de otros métodos existentes, que han sido los que se han utilizado.

Ha de decirse que el autor principal de todos estos articulos sobre la descomposición en valores singulares es el Dr. Carlos Campos. Fueron el fruto de su trabajo para la realización de su tesis doctoral.

Por último, queda decir que hay una publicación más, que ha sido realizada en cooperación con el grupo de investigación del Imperial College. Se trata de un artículo sobre la aplicación práctica en la simulación del flujo oceánico. Ha sido enviado a una revista del sector y está en proceso de revisión:

- *On low-frequency variability of the midlatitude ocean gyres*, I. V. Shevchenko, P. S. Berloff, D. Guerrero-López y J. E. Roman, *Journal of Fluid Mechanics*, *en revisión.*

Parte de los resultados de esta tesis y también del trabajo descrito en estas publicaciones se ha desarrollado en el marco de estos proyectos de investigación:

- *Network for development and evaluation of numerically reliable software in control engineering and its implementation in production technologies (NICONET)* (ERRT-CT97-5040).

- *Extensión de la librería SLEPc para polinomios matriciales, funciones matriciales y ecuaciones matriciales en plataformas de computación emergentes* (TIN2013-41049-P).

6.3. Posibles líneas de trabajo futuro

El trabajo realizado ofrece rutinas paralelas que permiten obtener modelos reducidos para sistemas lineales de control mediante algunos métodos de reducción de modelos. A partir de aquí se puede continuar prestando atención a dos aspectos.

Primero, se puede ir en la direccción de mejorar lo hecho. Aunque las prestaciones obtenidas con las nuevas rutinas desarrolladas son adecuadas, es cierto que algunas operaciones todavía permitirían mejoras en su implementación.

Segundo, se puede ir en la dirección de ampliar lo realizado. Por ejemplo, se pueden desarrollar nuevos métodos de reducción de modelos que utilicen como base los ya desarrollados.

A continuación se muestran algunas propuestas en ambas direcciones separadas en las tres temáticas principales sobre las que se ha trabajado.

En la resolución del problema de valores propios se puede continuar trabajando en:

- Aunque se ha desarrollado una rutina paralela para resolver el problema generalizado de valores propios, se ha hecho transformando el problema a su forma estándar. Esto sólo es posible cuando la matriz B del problema es no singular y está bien condicionada. Sería conveniente tratar de adaptar la rutina para el caso más general (utilizando los nuevos desarrollos que se están realizando para la transformación a forma real de Schur generalizada [AKK07, AKK14]).

Algunos posibles nuevos desarrollos en la resolución de la ecuación de Lyapunov son:

- La rutina `PSB03OT`, a cargo de la resolución de ecuaciones de Lyapunov reducidas en paralelo, tiene buenas prestaciones, pero mejores en mallas de procesos horizontales (o verticales en el caso de la ecuación traspuesta). Sería interesante tratar de mejorar sus prestaciones para cualquier tipo de malla de procesos, especialmente para las mallas cuadradas que son más adecuadas para otras operaciones.

- En la implementación actual, la resolución de la ecuación reducida no está optimizada para aquellos casos en los que la matriz que aparece premultiplicada por su traspuesta en la parte derecha de la ecuación es rectangular (de igual forma como tampoco está aprovechado en la rutina secuencial equivalente de SLICOT). Convendría analizar en qué partes puede aprovecharse esta característica y hacerlo. Probablemente sea suficiente con tenerlo en cuenta en las múltiples descomposiciones QR de matrices más grandes en las que aparece esa matriz.

- Una vez que se ha estudiado la forma más adecuada de paralelizar la resolución de esta ecuación, puede utilizarse el conocimiento adquirido para optimizar la resolución secuencial. Podría desarrollarse una versión orientada a bloques fuertemente basada en la forma de trabajar en paralelo. Ya se ha comprobado en las pruebas que las prestaciones del algoritmo paralelo ejecutado con un solo procesador son comparables a las de otras versiones orientadas a bloques.

- También podría trabajarse en el desarrollo de una versión paralela de memoria compartida. Se trataría de implementar una versión multi-hilo.

Con las rutinas paralelizadas para la reducción de modelos se puede:

- Una vez que estén disponibles algoritmos de altas prestaciones para la factorización de Schur generalizada, se podrían adaptar los métodos desarrollados para trabajar con sistemas lineales de control también generalizados.
- Tras la experiencia de paralelizar algunos métodos de reducción de modelos de la librería SLICOT, se pueden analizar el resto de métodos implementados en esta librería y paralelizar los que tengan una forma de trabajar similar a los ya hechos.
- Ahora que ya se tienen versiones paralelas de los principales métodos de reducción de modelos para sistemas estables de la librería SLICOT, es posible enfrentarse a la paralelización de aquellos métodos para sistemas inestables que se basan principalmente en descomponer el sistema en varias partes y aplicar la reducción para sistemas estables sobre alguna o varias de estas partes.
- Se ha comprobado que la rutina de LAPACK `DBDSQR`, a cargo de obtener los valores singulares de una matriz en forma bidiagonal, tiene un coste secuencial muy elevado. Esto ocurría al acumular las transformaciones ortogonales utilizadas. Viendo que la versión paralela tenía super speed-up, puede ser una buena idea el orientar a bloques la rutina secuencial empleando un sistema lo más parecido posible a lo realizado en la rutina paralela. Así se conseguiría mejorar las prestaciones de la rutina secuencial para problemas grandes.

Apéndice A

Algunos detalles de implementación

En este apéndice se comentan algunos detalles de la implementación utilizada para resolver los grandes problemas de valores propios que aparecen en la simulación de flujos oceánicos. Se trata de algunos factores que no suelen aparecer en problemas de menor tamaño, pero que se vuelven importantes al trabajar con problemas grandes.

Primero se explica una nueva forma de gestionar el espacio de trabajo (el `WORK`) que necesitan muchas rutinas de LAPACK y ScaLAPACK. En la implementación de solución del problema de valores propios que se ha utilizado hay algunas rutinas recursivas que dificultan la forma habitual de trabajar con el `WORK`. Ha sido necesario utilizar una forma alternativa de gestión de este espacio de trabajo.

Después se cuenta cómo se han solucionado algunos problemas que aparecen en el uso de matrices muy grandes. Se ha implementado una nueva rutina que permite realizar el reparto/recogida de matrices de gran tamaño. También se han desarrollado diferentes funciones para leer y escribir en disco matrices grandes, permitiendo incluso una lectura/escritura parcial, que resulta conveniente para matrices muy grandes.

Por último, al resolver problemas cuyo tiempo de ejecución supera el máximo que se permite reservar en los cluster utilizados resulta necesario poder guardar y luego recuperar el estado de una ejecución del programa. Aprovechando el conocimiento del funcionamiento del programa, se ha desarrollado un sencillo sistema que permite crear y luego continuar *puntos de restauración* en el problema de resolución de valores propios. Este sistema se explica al final de este apéndice.

A.1. Rutinas de gestión del `WORK`

Muchas rutinas de LAPACK y ScaLAPACK necesitan un espacio extra, más allá del espacio de los datos de entrada y/o salida, para realizar cálculos intermedios. Este espacio es lo que se conoce como *espacio de trabajo* o *workspace*.

Cuando este espacio es ridículamente pequeño, suelen utilizarse variables locales a cada rutina para él. Pero es muy habitual que este espacio sea grande, aunque normalmente no tanto como las matrices de entrada y/o salida a la rutina. En estos casos, a estas funciones se les pasa un parámetro extra (`WORK`) como una matriz más, pero para usar como espacio de trabajo.

También es común el pasarles un parámetro extra indicando el tamaño de este espacio, de forma que la rutina pueda comprobar que es suficiente. Se trata del parámetro `LWORK`.

En múltiples ocasiones el tamaño del espacio de trabajo necesario no es conocido en tiempo de compilación por depender del tamaño del problema a resolver y/o de otros parámetros o peculiaridades del problema. En estos casos, las rutinas correspondientes suelen permitir hacer lo que se conoce como una *consulta del espacio requerido*. Esto suele hacerse indicando como tamaño `LWORK=-1`, lo que provoca que la rutina no haga nada excepto calcular el espacio necesario y almacenarlo en la primera posición del `WORK` proporcionado (que deberá tener al menos una posición).

Aunque hoy en día la mayoría de lenguajes de programación ofrecen herramientas para la gestión de memoria dinámica, esto no siempre ha sido así. Con memoria dinámica, se podría pedir directamente la memoria necesaria dentro de cada rutina y luego liberarla. Pero cuando no se tenían herramientas de memoria dinámica, resultaba necesaria otra forma de realizar esta labor, como ha sido el uso del `WORK`.

Además, por cuestiones de eficiencia sigue siendo interesante el uso de un espacio de trabajo externo a la rutina en lugar de pedir y liberar lo necesario cada vez que esta es llamada. Piénsese por ejemplo en una rutina que vaya a ser llamada cientos de veces para problemas similares. Si se gestiona el espacio desde dentro de la rutina, se va a pedir/liberar memoria cientos de veces, lo que será ineficiente.

Por todo esto, LAPACK y ScaLAPACK siguen trabajando con un espacio de trabajo externo a las rutinas y, en la mayoría de casos, permitiendo realizar una consulta del espacio requerido.

A pesar de que esto *parece* una buena idea, lo cierto es que el uso del `WORK` es una de las cosas más engorrosas, feas y molestas cuando se trabaja con LAPACK y/o ScaLAPACK. Sobre todo en el caso de ScaLAPACK, porque los cálculos del tamaño del espacio de trabajo se complican mucho al depender de la distribución empleada, del tamaño de bloque, del tipo de malla de procesos...

Y si ya es algo lioso manejar el `WORK` desde el punto de vista de un usuario de las rutinas en que se necesita, lo es todavía más desde el punto de vista de un desarrollador que pretenda diseñar, implementar y ofrecer una nueva rutina con un uso similar al ofrecido por las rutinas presentes en LAPACK y ScaLAPACK.

En el caso de las rutinas a cargo de la solución del problema de valores propios desarrolladas por Granat y Kågström [GKK10, GKKS15], el asunto se complica aún más debido a que parte de ellas se han implementado de forma recursiva. Para echar más leña al fuego, ni tan siquiera son rutinas recursivas tradicionales, sino que se trata de una recursividad indirecta. Una rutina llama a otra, que posteriormente puede llamar a la primera. Esto puede verse en la figura 3.1, que muestra el grafo de llamadas entre estas rutinas.

Por todo esto, el cálculo del tamaño necesario para el espacio de trabajo se

complica muchísimo. Depende del tamaño de problema y de sus características, que van a afectar al número de llamadas recursivas y a otros factores. Hasta el punto de que tal como están implementadas estas rutinas, no se calcula bien el espacio necesario. Y tratar de calcularlo de forma correcta resulta inviable.

Para poder utilizar estas rutinas en el problema de cálculo de valores propios, se han desarrollado nuevas rutinas auxiliares para lograr una *gestión dinámica del espacio de trabajo*.

Se ha procurado hacer algo similar a lo que ocurriría si se explotasen las características de gestión dinámica de memoria presentes en los lenguajes de programación actuales, pero evitando la sobrecarga de hacer múltiples reservas/liberaciones de memoria.

En cada rutina, de las problemáticas con el espacio de trabajo, se hace una petición de memoria en el momento en que se conoce el tamaño real necesario para el espacio de trabajo. Y posteriormente no se libera ese espacio hasta finalizar por completo la ejecución del programa. Se ha implementado de tal forma que, tras volver de una rutina, otras rutinas a las que se llame reutilizarán el espacio de trabajo que se concedió a la rutina anterior, tan sólo ampliándolo con una nueva petición en caso de que sea necesario.

De esta manera se consigue una gestión del espacio de trabajo más cómoda para el usuario, sin la contrapartida de efectuar muchas peticiones/liberaciones de memoria.

Se han modificado las rutinas donde el cálculo del espacio de trabajo era erróneo y además era complicado a causa de la recursividad. Ahora hacen uso de estas nuevas funciones de gestión del espacio de trabajo de forma dinámica. Se trata de las rutinas `PDLAQR0` (recursiva), `PDLAQR1` y `PDLAQR3` (recursiva indirecta).

En ellas se usan estas nuevas rutinas:

- `GET_WORK`, para solicitar espacio de trabajo cuando se necesita. Internamente esta rutina sólo pedirá nuevo espacio al sistema operativo si no se tiene bastante con lo concedido anteriormente.

- `ADD_WORK_POINTER`, para cada variable que apunte a zonas del espacio de trabajo. Teniendo en cuenta que otras rutinas o esta misma pueden solicitar más espacio, puede ocurrir que se mueva en memoria el espacio reservado hasta ahora y que está siendo utilizado por variables en esta u otras rutinas. Por esto, resulta necesario mantener una lista de los punteros a zonas del espacio de trabajo, para actualizarlos en caso de cambios en memoria.

- `UPDATE_WORK_POINTER`, que será llamada después de usar alguna rutina que pueda haber movido el espacio de trabajo en memoria. Esta rutina sería para actualizar los punteros a zonas que se hayan movido. En realidad esto se hace usando una lista interna de tales punteros, de forma que actualmente esta rutina no hace *nada*. Pero sí aparecen llamadas a ella para cada puntero en las zonas adecuadas, básicamente para que el compilador vea que puede haber cambiado este puntero. En la mayoría de compiladores no hará falta, pero así funcionará también en los que sí haga falta este *aviso* al compilador.

- `DEL_WORK_POINTERS`, que es llamada al final de cada rutina para eliminar de la lista interna los punteros a zonas del espacio de trabajo de esta rutina, que ya no necesitarán ser actualizados cuando haya algún cambio en memoria.

Además, en el programa principal se debe usar la nueva rutina `FREE_WORK` que libera *todo* el espacio de trabajado gestionado por estas nuevas rutinas. Esto deberá hacerse al final de programa o también cuando ya no vaya a usarse este espacio de trabajo definitivamente o por un tiempo.

Aparte de estas rutinas también se ha incorporado una rutina `CURRENT_LWORK` que informa del uso actual de espacio de trabajo gestionado de esta manera y que resulta muy útil para depuración o mostrar información extra al usuario.

Todas estas rutinas han sido necesarias para poder usar las nuevas rutinas de cálculo de valores propios, en las que el uso habitual del `WORK` no es posible por errores en los cálculos globales del espacio de trabajo necesario.

Sin embargo, también pueden ser útiles en futuros nuevos desarrollos ya que facilitan enormemente la gestión del `WORK`, al hacer innecesario conocer a priori el tamaño necesario para él. Serán especialmente útiles para gestión de `WORK` en rutinas recursivas o donde su tamaño sea difícil o imposible de conocer a priori.

A.2. `my_pdgemr2d`

En ScaLAPACK hay una rutina que permite copiar los datos de una (sub)matriz distribuida a otra (sub)matriz que puede estar distribuida de forma diferente. Se trata de la rutina `PDGEMR2D`. Esta rutina es muy cómoda de usar a la hora de mover datos entre matrices con diferentes distribuciones.

Si las (sub)matrices entre las que se quiere copiar tienen exactamente la misma distribución, es más eficiente usar la rutina `PDLACPY` que efectúa la copia sin hacer ninguna comunicación.

En el problema de valores propios, esta rutina se usará habitualmente tanto en el reparto inicial de los datos y la recogida final de los resultados como durante el cálculo de los valores propios y la forma real de Schur, donde se realizan redistribuciones de matrices en sub-mallas de procesos (dentro de las rutinas de ScaLAPACK `PDLAQR0` y `PDLAQR3`).

La rutina `PDGEMR2D` de copia de (sub)matrices distribuidas realiza múltiples peticiones de memoria extra para llevar a cabo la operación. De hecho, en principio solicita en cada proceso tanta memoria como está ocupando la (sub)matriz origen de la copia y también tanta como está ocupando la (sub)matriz destino de la copia. Esto parece algo excesivo, pero sin esta memoria extra la operación de comunicación para cualquier posible distribución de los datos podría complicarse mucho.

Sin embargo, hay un pequeño problema en las rutinas internas donde se gestiona la memoria.

La rutina interna `setmemory` es utilizada para pedir memoria para un bloque local. Recibe un parámetro de tipo entero de 32 bits con signo donde se especifica la cantidad de números reales en doble precisión para los que se quiere memoria. El

hecho de utilizar un entero de 32 bits con signo para este dato limita la cantidad máxima de números para los que reservar memoria a $2^{31} - 1$.

Si, por ejemplo, se utiliza esta rutina para hacer la copia de una matriz grande totalmente almacenada en un proceso a una distribución entre muchos procesos, se va a necesitar pedir espacio para su copia local en el proceso que la tiene toda. Con la limitación de esa cantidad máxima de números reales, el tamaño máximo de la matriz (suponiéndola cuadrada) sería 46340×46340. El problema es que en ocasiones se necesita operar con matrices mucho más grandes que eso.

Además, en realidad el problema se agrava, porque internamente la función `setmemory` llama a otra para hacer la petición de memoria: la rutina `mr2d_malloc`, que recibe en un entero de 32 bits con signo la cantidad de memoria a reservar **en bytes**.

Teniendo en cuenta que un número real en doble precisión ocupa 8 bytes, el tamaño máximo del ejemplo anterior se ve reducido en esa cantidad. Ahora, el tamaño máximo posible de memoria es de $2^{31} - 1$ bytes y no elementos. Esto limita el tamaño de una matriz cuadrada a 16383×16383. Es un tamaño máximo adecuado para muchos problemas, pero no para *problemas de gran dimensión*.

En una primera aproximación se ha trabajado desarrollando una nueva rutina de distribución que no tenga estas limitaciones de tamaño.

Debe notarse que el problema crítico es la distribución inicial y final de los datos. Porque en estos casos la matriz origen o destino de la copia está completamente almacenada en un proceso y es cuando se requiere mucha más memoria. En las redistribuciones realizadas internamente en las rutinas de cálculo de valores propios se hacen copias de matrices grandes ya distribuidas a matrices con otra distribución. Por tanto, aquí el problema es menor.

Se ha implementado una nueva rutina `my_pdgemr2d` que permita realizar la distribución inicial de los datos y la recogida final de los resultados, pero necesitando mucha menos memoria que la rutina de distribución de ScaLAPACK `PDGEMR2D`. En esta nueva rutina no se permite cualquier distribución de las matrices origen y destino de la copia. Esta rutina es para la distribución inicial y la recogida final, así que tiene la restricción de que o bien la matriz origen de la copia o bien la matriz destino de la copia deben estar completamente almacenadas en un único proceso. Exigir esto simplifica la implementación (pero no la hace trivial), haciendo posible realizar la copia sin necesitar mucha más memoria.

Ya que se pretende ahorrar memoria, en la nueva rutina no se utiliza nada de memoria extra para realizar la copia. Además se envía/recibe un único mensaje por cada proceso involucrado en la copia.

Para no pedir memoria y realizar un único mensaje por proceso, la nueva rutina realiza un reordenamiento previo (o posterior) de los datos en la misma matriz donde están o donde se quedarán.

El algoritmo utilizado puede esquematizarse en estos pasos, donde A representa la matriz cuyos datos se quieren copiar en la matriz B:

Algoritmo A.1 `my_pdgemr2d`

1. *Si la matriz origen A está completamente almacenada en un proceso, entonces:*

   a) *En el proceso que tiene toda la matriz A:*

      1) *Reordenar la matriz A sobre ella misma de forma que lo que va a quedar repartido en cada proceso quede contiguo en memoria.*
      2) *Hacer un bucle que recorra todos los procesos de la malla de procesos. Para cada uno de ellos que tenga algún elemento de B: si es el proceso que tiene toda A, copiar el fragmento que le corresponde de A a B; si no, enviarle el fragmento que le corresponde de A (un único mensaje para cada proceso).*
      3) *Deshacer el reordenamiento de la matriz A para que quede en el orden original.*

   b) *En el resto de procesos: si tiene algún elemento de B, recibir todo el fragmento de B que tiene del proceso emisor (un único mensaje).*

2. *Si no, es la matriz destino B la que está completamente almacenada en un proceso y se procede así:*

   a) *En el proceso que tiene toda la matriz B:*

      1) *Hacer un bucle que recorra todos los procesos de la malla de procesos. Para cada uno de ellos que tenga algún elemento de A: si es el proceso que tiene toda B, copiar el fragmento que le corresponde de A a B; si no, recibir el fragmento que le corresponde en B (un único mensaje para cada proceso).*
      2) *Reordenar la matriz B, donde se tiene de forma contigua lo recibido de cada proceso, para que quede en el orden adecuado.*

   b) *En el resto de procesos: si tiene algún elemento de A, enviar todo el fragmento de A que tiene al proceso que tiene toda B (un único mensaje).*

Los reordenamientos que aparecen en el algoritmo no son procesos triviales, ya que hay que utilizar lo que se sabe de la distribución de la correspondiente matriz. Además se realizan sobre la misma matriz, sin espacio extra, lo que los complica un poco.

En estos reordenamientos se han realizado dos pequeñas mejoras sobre la versión inicial:

- Cada reordenamiento implica en realidad dos permutaciones: una de las filas de la matriz y otra de las columnas de la matriz. Dado que se trabaja con matrices muy grandes y para minimizar los fallos de caché, ambas permutaciones se hacen procediendo en orden por las diversas *columnas* de la matriz. Como se trabaja con el formato en memoria de FORTRAN para matrices, los datos se almacenan por columnas y es conveniente accederlos de esta manera para minimizar los fallos de caché.

- Precisamente por hacer las permutaciones de filas agrupadas por columnas, habría que recalcular para cada columna las permutaciones a realizar. Pero esto supondría muchos cálculos. Para evitarlo, se calcula el reordenamiento de filas necesario una única vez y se almacena en un vector auxiliar que se reutiliza para realizar la permutación en cada columna. Este vector auxiliar es un vector de memoria dinámica. Se reserva memoria para él y luego se libera. Ya no es cierto que en esta rutina no se use memoria extra. Pero la cantidad de memoria pedida y luego liberada es de tan sólo m números enteros para matrices de tamaño $m \times n$, lo que es muy poco.

Gracias a esta nueva rutina desaparecen las limitaciones en cuanto a tamaño de matriz para realizar las distribuciones iniciales y finales de los datos del problema. Pero la cantidad de memoria física disponible en un único nodo para cada proceso sigue imponiendo limitaciones para los problemas más grandes. Esto se puede solucionar cambiando la forma de repartir/recoger la matriz, pasando por no cargarla por completo en ningún proceso. Al final, la nueva rutina `my_pdgemr2d` ya no ha hecho falta en el problema de valores propios, tras realizar este cambio que reduce aún más la memoria necesaria en el proceso que gestiona los datos iniciales y finales del problema. Esto se explica en el siguiente apartado.

Afortunadamente, esta nueva rutina sí ha hecho falta y así se ha utilizado en los otros problemas grandes que han aparecido (en la resolución de ecuaciones de Lyapunov con matrices de tamaño superior a 16383×16383).

A.3. Rutinas de lectura/escritura de grandes matrices en disco

En el cálculo de valores propios de problemas de gran tamaño se ha utilizado un formato de fichero propio del lenguaje FORTRAN. Las *enormes* matrices del problema a resolver provienen de un programa escrito en lenguaje FORTRAN que las escribe, en principio, de la forma más característica del lenguaje. En este formato, las matrices se encuentran almacenadas en disco en formato binario en bloques de un tamaño razonablemente grande, yendo cada bloque precedido y seguido de su tamaño, también en binario.

Se ha desarrollado una nueva librería en lenguaje C, pero con interfaz tanto para C como para FORTRAN, con rutinas para la gestión de estos ficheros, de forma que se puedan leer y escribir con facilidad.

También se han desarrollado nuevos programas muy básicos para realizar operaciones triviales con estos ficheros desde la línea de comandos de Linux: copiar, comparar, obtener información, obtener estadísticas, generar matrices de ceros...

Por necesidades de la aplicación práctica, también se ha permitido desde la aplicación la generación del fichero resultado en forma de múltiples ficheros, uno para cada columna del resultado. Esto permite realizar un tratamiento posterior trabajando sólo con las columnas deseadas y no necesitando enfrentarse cada vez con el *enorme* fichero completo.

Se han resuelto problemas de valores propios con matrices de hasta 444675×444675 elementos, lo que en formato *double* (8 bytes por cada elemento) implica un espacio tanto en memoria como en disco de 1581886845000 bytes, esto es 1508604.86 megabytes, 1473.25 gigabytes, ¡¡ 1.44 terabytes !! [1] En el caso de tener la información en disco, el tamaño es un poco mayor porque falta sumar lo usado en cabeceras e información extra, aunque esto es despreciable frente al tamaño de los datos.

Además, al tratarse en este caso práctico de la resolución de un problema de valores propios *generalizado*, son dos las matrices de entrada al problema, lo que duplica los problemas de espacio.

Por todo esto, se ha necesitado optimizar el uso de espacio tanto en memoria como en disco para hacer factible la resolución de estos grandes problemas.

A continuación se comentan las ideas llevadas a cabo dentro de esta nueva librería de gestión en disco de matrices para reducir los requisitos espaciales necesarios. Es la parte más interesante de esta nueva librería.

A.3.1. Reducción de espacio en memoria

El hecho de trabajar con problemas tan sumamente grandes no sólo afecta al espacio necesario en disco sino también al uso de memoria. Almacenar en memoria **dos** matrices, ocupando cada una casi un terabyte y medio de información es inviable en prácticamente todos los ordenadores actuales. Esto justifica la necesidad de trabajar con multiprocesadores de memoria distribuida que permitan mantener ambas matrices en memoria repartidas entre muchos procesadores.

En la mayoría de problemas que se paralelizan para su ejecución en plataformas de multiprocesadores, la razón fundamental de esta paralelización es la mejora de prestaciones. Es decir, ya se puede resolver el problema en un monoprocesador, pero se quiere resolverlo necesitando un menor tiempo de ejecución. En estos casos, lo típico y que casi siempre se hace es cargar los datos procedentes de disco en uno de los procesos y desde allí distribuirlos al resto.

Otra forma de proceder es que todos los procesos lean de disco, leyendo cada uno la parte que va a necesitar. Pero esto a veces no puede hacerse de forma sencilla por dos razones.

Primero, porque la gestión de la carga por cada proceso de la parte que necesite del problema suele ser bastante complicada. Piénsese que habitualmente esta parte del problema no es un bloque contiguo de datos en disco.

Pero además en ocasiones ocurre en los sistemas paralelos que sólo uno de los procesos tiene acceso al disco físico. O también que aunque todos tengan acceso, resulte más eficiente evitar múltiples accesos simultáneos al disco. Aunque esto depende del hardware disponible y ya existen sistemas de ficheros optimizados para accesos simultáneos.

Por una u otra razón, normalmente es un proceso el que trabaja con disco y es el encargado de leer los datos y repartirlos entre todos los procesos. Al final recogerá los

[1] Se han usado potencias de 2, como suele ser habitual en las medidas informáticas. Actualmente las unidades que usan potencias de 2 tienden a ser nombradas mebibytes, gibibytes y tebibytes.

resultados y los escribirá a disco. Este esquema es con diferencia el más utilizado y es el que se ha utilizado en este trabajo.

Pero no es posible proceder de esta manera para problemas tan grandes. En ninguna de las múltiples plataformas físicas con las que se ha trabajado se disponía de suficiente memoria RAM para almacenar por completo el problema. Por lo tanto, no se puede trabajar de la forma habitual. No es factible cargar en un proceso todo el problema y luego repartirlo desde ahí al resto de procesos.

Así que se ha optado por modificar ligeramente esta forma de proceder. En función de la memoria disponible, se van cargando fragmentos del problema y se van distribuyendo a los procesos que les corresponda. Es similar al método tradicional, pero ahora no se necesita almacenar por completo en memoria el problema.

Gracias a esta nueva forma de proceder se ha hecho posible el trabajar con problemas tan grandes como se desee, siempre que quepan **distribuidos** entre todos los procesos. Ya no existe la limitación de que el problema quepa por completo en la memoria de un proceso. El límite vendrá dado por la memoria agregada de todos los procesos utilizados para la resolución del problema.

Para implementar esta nueva forma de repartir/recoger los datos, se ha modificado la parte de distribución y recogida del programa de cálculo. Pero también ha resultado imprescindible modificar las rutinas de almacenamiento/recuperación en disco para que permitan ir leyendo/escribiendo bloques de tamaño arbitrario en disco.

Debe notarse que esto no es simplemente ir leyendo/escribiendo los datos que se deseen cada vez, puesto que el formato de los ficheros utilizados incorpora datos sobre tamaños de bloques y otras cosas.

Se han desarrollado rutinas nuevas en esta librería de gestión de matrices en disco para:

- Abrir un fichero en lectura o escritura sabiendo que las operaciones van a ser parciales. Esta operación es diferente a la apertura de un fichero que vaya a ser leído/escrito por completo, puesto que se necesita mantener información varia para poder ir trabajando de forma parcial con el fichero.

- Leer un fragmento de disco. Aparte de leer la cantidad especificada de datos, hay que actualizar y usar la información extra que se mantiene en memoria sobre el fichero y que permite realizar estas lecturas parciales.

- Escribir un bloque de datos a disco. Al igual que para la lectura, aparte de escribir los datos se deben actualizar y usar los datos que se mantienen en memoria y que permiten trabajar con el formato del fichero realizando operaciones parciales.

- Cerrar un fichero abierto para lectura/escritura parcial. Esta operación no hace nada especial con respecto al trabajo con un fichero sin realizar operaciones parciales. Pero resulta necesario diferenciarla del caso general, porque sí que se hacen cosas especiales cuando se utiliza el formato comprimido que se explica a continuación.

A.3.2. Reducción de espacio en disco

La lectura/escritura parcial de datos explicada en el punto anterior es totalmente necesaria e imprescindible para permitir trabajar con problemas más grandes de la cantidad de memoria disponible para un proceso. Lo que se va a explicar a continuación no es imprescindible ni completamente necesario para trabajar con problemas grandes, aunque resulta muy conveniente.

Dadas las características del problema con el que se ha trabajado, resulta que las matrices de entrada al mismo son dispersas. Las matrices A y B que almacenan los datos de las dos matrices de las que se necesitan calcular los valores y vectores propios generalizados son dispersas. Nótese que la matriz de vectores propios no lo es.

El hecho de que las matrices sean muy dispersas puede hacer pensar en la conveniencia de utilizar otros métodos de cálculo que no necesiten tener las matrices en forma densa en memoria. Pero estos métodos no resultan apropiados si, como ocurre en este caso, se desean obtener *todos* los valores y vectores propios. Por tanto, aunque los datos de entrada al problema son dispersos, las operaciones con ellos se hacen de forma densa y el resultado (los vectores propios) es una matriz densa.

La matriz de vectores propios va a ser una matriz enorme y densa, que habrá que almacenar en disco como tal, ocupando mucho espacio. Pero los datos de entrada son dispersos y esto puede explotarse para reducir los requerimientos de disco al menos para esta parte.

Esto es lo que se ha desarrollado, añadiendo nuevas funcionalidades a las rutinas de gestión de matrices en disco. Se ha diseñado un nuevo formato de matrices en disco, que permita aprovechar la dispersidad de las matrices de entrada y así ahorrar mucho espacio en disco, al menos para los datos de entrada.

Se han desarrollado nuevas rutinas en la librería de gestión de matrices en disco para todas las operaciones existentes, pero ahora en versión comprimida: obtener información de fichero, abrir para leer, abrir para escribir, abrir para lecturas parciales, abrir para escrituras parciales, leer, escribir, efectuar una lectura parcial, efectuar una escritura parcial, cerrar fichero abierto para lecturas parciales, cerrar fichero abierto para escrituras parciales y alguna otra funcionalidad más.

Además, se han modificado todas las rutinas ya existentes para el trabajo con matrices en formato no comprimido. Ahora en las funciones de apertura de fichero (para cualquier tipo de lectura o escritura), se detecta la extensión del fichero con el que se va a trabajar. Si la extensión es la del formato comprimido (`.cd1`), se marca una variable de estado que hará que para este fichero se usen todas las nuevas rutinas que trabajan en formato comprimido, aún llamando a las rutinas antiguas que trabajan con el formato sin comprimir. De esta forma, el uso de las nuevas rutinas de compresión es bastante transparente tanto al programador como al usuario. De hecho, al ejecutar el programa de cálculo se pueden especificar ficheros de entrada y/o salida con o sin la extensión usada para formato comprimido, lo que hará que la librería trabaje de forma automática con ficheros comprimidos o no, según corresponda.

Esto mismo se ha hecho con los programas auxiliares de trabajo con ficheros desde

línea de comandos que se han comentado anteriormente. Gracias a esto, por ejemplo, el programa para copiar ficheros de matrices con estos formatos puede usarse con gran comodidad para comprimir/descomprimir una matriz. Basta simplemente con especificar como origen de la copia un fichero sin extensión de comprimida y como destino uno con extensión de comprimida. Esto comprimiría la matriz dejándola en un nuevo archivo en disco. Y de forma análoga se podría descomprimir si se deseara.

Las nuevas rutinas son algo más complicadas que las básicas de lectura/escritura, ya que ha habido que añadirles la parte de compresión/descompresión, que complica bastante la implementación. Más todavía si cabe en las rutinas de lecturas/escrituras parciales, donde a la información que permite trabajar con el fichero de forma parcial hay que añadir información extra que permita ir comprimiendo/descomprimiendo de forma parcial.

Pero gracias a esto las matrices de entrada al problema ocupan mucho menos espacio en disco, lo que las hace más manejables y transportables. Además, la carga del sistema de discos será menor al tener que transmitir menos datos a la CPU, lo que se traducirá en una mayor velocidad de lectura y/o escritura de la información en disco.

Por ejemplo, en uno de los problemas, que no de los más grandes, las matrices A y B son de 49926×49926 elementos cada una. Estas matrices almacenadas en el formato habitual ocupan en disco 19940843888 bytes (18.57 gigabytes) cada una. Las mismas matrices almacenadas en el nuevo formato comprimido ocupan 11383456 bytes (10.86 megabytes) para la matriz A y 5055704 bytes (4.82 megabytes) para la matriz B. Esto supone un ahorro considerable de espacio en disco, además de permitir transportarlas, ya sea físicamente o por internet, de manera mucho más cómoda.

A.4. Puntos de restauración

Para poder resolver problemas de valores propios de gran dimensión teniendo en cuenta que en la mayoría de plataformas paralelas suele haber una limitación máxima de tiempo de ejecución al lanzar trabajos, es conveniente disponer de un sistema de *puntos de restauración*.

Se ha desarrollado un sencillo sistema que permite trabajar con *puntos de restauración*. Esto es que se permite suspender el problema que se está resolviendo para continuar con su resolución en una ejecución posterior.

Tras observar las diferentes etapas del problema de cálculo de valores propios, se ha constatado que es en la parte propiamente dicha de cálculo de valores propios (en el paso de transformación a forma real de Schur) donde normalmente se finalizaba la ejecución del trabajo por superar el tiempo máximo de ejecución permitido. Los pasos previos (lectura de los datos, transformación de problema generalizado a problema estándar y transformación de matriz general a forma de Hessenberg) también son costosos, pero normalmente al estar al principio les da tiempo a realizarse. Por esto, se ha implementado un sistema de puntos de restauración en las rutinas de cálculo de valores propios y obtención de la forma real de Schur.

Estas rutinas de ScaLAPACK están implementadas utilizando una recursividad indirecta, lo que complica el realizar un punto de restauración en cualquier sitio. Además, dependiendo de dónde se pretenda suspender la ejecución, puede necesitarse guardar en disco demasiadas variables para luego poder continuar con la ejecución.

Un punto adecuado para reducir las variables a guardar en disco y no tener que enfrentarse con la pila de llamadas recursivas es realizar los puntos de restauración en la rutina `PDLAQR0`, pero solamente en su primera llamada y no en llamadas recursivas. Se ha optado por hacer una llamada a las rutinas encargadas de crear puntos de restauración en cada iteración del algoritmo iterativo QR implementado en la rutina `PDLAQR0`. Como esta rutina de ScaLAPACK ha sido modificada (como luego se explica) para corregir algunos errores, se ha aprovechado para incluir esta llamada en su bucle principal. Esto no implica que se vaya a realizar un punto de restauración en cada iteración del método, pero sí que se pueda hacer si se desea.

La rutina encargada de los puntos de restauración, que es llamada para cada iteración, se comporta de forma diferente en función de las opciones especificadas al programa:

- Si no se ha indicado nada, no se hace nada.
- Si se ha especificado que se desea realizar puntos de restauración, se habrá indicado el intervalo de tiempo deseado para crear cada punto de restauración. En este caso, si desde la última creación de un punto de restauración (o desde el inicio del programa si aún no se ha creado ninguno) ha transcurrido un tiempo superior al indicado, se crea un nuevo punto de restauración. Se puede especificar que el programa finalice inmediatamente tras crear un punto de restauración.
- Si se ha indicado que se desea continuar la ejecución a partir de un punto de restauración creado en una ejecución previa, se cargan los datos necesarios de disco y se continúa la ejecución desde el punto guardado en disco.

La rutina `PDGGEV` también ha sido modificada para que no se realicen todas las operaciones previas a la factorización de Schur si se pretende continuar la ejecución a partir de un punto de restauración.

Por el lugar del programa donde se efectúan los puntos de restauración, la cantidad de datos necesaria a almacenar es reducida y está formada por los siguientes valores:

- La matriz que se está pasando a forma real de Schur y que estará a mitad camino entre Hessenberg y forma real de Schur. Dado que esta matriz tiene un elevado número de ceros, se almacena en el nuevo formato comprimido `.cd1`, lo que hará que ocupe mucho menos espacio en disco.
- Los vectores con la parte real y compleja de los valores propios que ya se han calculado.
- La matriz que acumula las transformaciones ortogonales utilizadas para la obtención de la forma real de Schur.

- El número actual de iteración.
- Los índices de principio y fin de la submatriz que aún queda por pasar a forma real de Schur.
- El tiempo acumulado transcurrido desde el principio de la resolución del problema. Esto permitirá ir acumulando los tiempos de todas las ejecuciones para conocer cuánto es el tiempo total de solución del problema.

Gracias a esta implementación de puntos de restauración, el tamaño del problema a resolver ya no queda limitado por el máximo tiempo de ejecución permitido en el sistema de colas de la plataforma paralela con la que se trabaje.

Este tiempo sí que limitará el tamaño del problema cuando:

- No baste para realizar todos los pasos previos al comienzo del cálculo de valores propios y transformación a forma real de Schur de la matriz en forma de Hessenberg. Si el tiempo es insuficiente para los pasos previos, habría que pensar en realizar puntos de restauración en los pasos previos o al menos realizarlos de alguna forma entre varias ejecuciones.
- Alguna iteración del algoritmo iterativo QR requiera más de ese tiempo para efectuarse. Pero ya tendría que ser un problema tremendamente grande para que *una única iteración* requiera más de ese tiempo.

Bibliografía

[ABB+92] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorenson. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[ADK02] Björn Adlerborn, Krister Dackland, and Bo Kågström. Parallel and blocked algorithms for reduction of a regular matrix pair to Hessenberg-triangular and generalized Schur forms. In Juha Fagerholm, Juha Haataja, Jari Järvinen, Mikko Lyly, Peter Råback, and Ville Savolainen, editors, *Applied Parallel Computing*, volume 2367 of *Lecture Notes in Computer Science*, pages 319–328. Springer Berlin Heidelberg, 2002.

[ADO92] E. Anderson, J. Dongarra, and S. Ostrouchov. LAPACK working note 41: Installation guide for LAPACK. Technical Report UT-CS-92-151, Department of Computer Science, University of Tennessee, Febrero 1992.

[AKK07] Björn Adlerborn, Bo Kågström, and Daniel Kressner. Parallel variants of the multishift QZ algorithm with advanced deflation techniques. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 117–126. Springer Berlin Heidelberg, 2007.

[AKK14] B. Adlerborn, B. Kågström, and D. Kressner. A parallel QZ algorithm for distributed memory HPC systems. *SIAM Journal on Scientific Computing*, 36(5):C480–C503, 2014.

[ASG01] A. C. Antoulas, D. C. Sorensen, and S. Gugercin. A survey of model reduction methods for large-scale systems. *Contemporary Mathematics*, 280:193–219, 2001.

[AT97] Mario Ahues and Françoise Tisseur. A new deflation criterion for the QR algorithm. Technical Report 122, LAPACK Working Note, March 1997.

[Bai02] Zhaojun Bai. Krylov subspace techniques for reduced-order modeling of large-scale dynamical systems. *Applied Numerical Mathematics*, 43(1-2):9–44, 2002.

[BBM02a] Karen Braman, Ralph Byers, and Roy Mathias. The Multishift QR Algorithm. Part I: Maintaining Well-Focused Shifts and Level 3 Performance. *SIAM Journal on Matrix Analysis and Applications*, 23, 2002.

[BBM02b] Karen Braman, Ralph Byers, and Roy Mathias. The Multishift QR Algorithm. Part II: Aggressive Early Deflation. *SIAM Journal on Matrix Analysis and Applications*, 23, 2002.

[BC85] Stephen Barnett and R. G. Cameron. *Introduction to Mathematical Control Theory*. Oxford applied mathematics and computing science series. Clarendon Press, Oxford, UK and New York, NY, USA, 1985.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[BCHV97] Ignacio Blanquer, Héctor Claramunt, Vicente Hernández, and Antonio M. Vidal. Resolución de la Ecuación de Lyapunov Generalizada por el Método de Bartels-Stewart mediante Librerías Estándar Secuenciales y Paralelas de Álgebra Lineal. In *VIII Jornadas de Paralelismo*, pages 11–20, Cáceres, 1997.

[BCQ98] Peter Benner, José M. Claver, and Enrique S. Quintana-Ortí. Efficient solution of coupled Lyapunov equations via matrix sign function iteration. In *Proc. 3rd portuguese conf. on automatic control CONTROLO'98, Coimbra*, pages 205–210, 1998.

[BD89] Zhaojun Bai and James Demmel. On a block implementation of Hessenberg multishift QR iteration. *International Journal of High Speed Computing*, 1(1):97–112, 1989.

[BEK+11] Peter Benner, Pablo Ezzatti, Daniel Kressner, Enrique S. Quintana-Ortí, and Alfredo Remón. A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. *Parallel Computing*, 37(8):439–450, 2011.

[Ben04] Peter Benner. Solving large-scale control problems. *IEEE Control Systems*, 24(1):44–59, 2004.

[Ben06] Peter Benner. Numerical linear algebra for model reduction in control and simulation. *GAMM-Mitteilungen*, 29(2):275–296, 2006.

[BGH+98] Ignacio Blanquer, David Guerrero, Vicente Hernández, Enrique S. Quintana-Ortí, and Pedro A. Ruiz. Parallel-SLICOT implementation and documentation standards. Technical report, SLICOT Working Note 1998-1, 1998.

[BHD07] P. Berloff, A. Hogg, and W. Dewar. The turbulent oscillator: A mechanism of low-frequency variability of the wind-driven ocean gyres. *Journal of Physical Oceanography*, 9:2363–2386, 2007.

[BKS13] Peter Benner, Patrick Kürschner, and Jens Saak. Efficient handling of complex shift parameters in the low-rank Cholesky factor ADI method. *Numerical Algorithms*, 62(2):225–251, 2013.

[BLP08] Peter Benner, Jing-Rebecca Li, and Thilo Penzl. Numerical solution of large-scale Lyapunov equations, Riccati equations, and linear-quadratic optimal control problems. *Numerical Linear Algebra with Applications*, 15(9):755–777, 2008.

[BM99] P. S. Berloff and J. C. McWilliams. Large-scale, low-frequency variability in wind-driven ocean gyres. *Journal of Physical Oceanography*, 29:1925–1949, 1999.

[BMS+99] Peter Benner, Volker Mehrmann, Vasile Sima, Sabine Van Huffel, and Andras Varga. SLICOT–a subroutine library in systems and control theory. In *Applied and Computational Control, Signals, and Circuits*, pages 499–539. Birkhäuser Boston, 1999.

[BMS05] P. Benner, V. Mehrmann, and D. C. Sorensen. *Dimension Reduction of Large-Scale Systems: Proceedings of a Workshop Held in Oberwolfach, Germany, October 19-25, 2003*. Lecture Notes in Computational Science and Engineering. Springer, 2005.

[BQ99] Peter Benner and Enrique S. Quintana-Ortí. Solving stable generalized Lyapunov equations with the matrix sign function. *Numerical Algorithms*, 20(1):75–100, 1999.

[BQO05] Peter Benner and Enrique S. Quintana-Ortí. Model reduction based on spectral projection methods. In Peter Benner, Danny C. Sorensen, and Volker Mehrmann, editors, *Dimension Reduction of Large-Scale Systems*, volume 45 of *Lecture Notes in Computational Science and Engineering*, pages 5–48. Springer Berlin Heidelberg, 2005.

[BQQ99] Peter Benner, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. A portable subroutine library for solving linear control problems on distributed memory computers. Technical Report NICONET 1999-1, Working Group on Software WGS, ESAT - Katholieke Universiteit Leuven (Belgium), January 1999.

[BQQ00a] Peter Benner, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Balanced truncation model reduction of large-scale dense systems on parallel computers. *Mathematical and Computer Modelling of Dynamical Systems*, 6(4):383–405, 2000.

[BQQ00b] Peter Benner, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. PSLICOT routines for model reduction of stable large-scale systems. Technical report, January 22 2000.

[BRD+15] Peter Benner, Alfredo Remón, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S. Quintana-Ortí. Extending LYAPACK for the solution of band Lyapunov equations on hybrid CPU–GPU platforms. *The Journal of Supercomputing*, 71(2):740–750, 2015.

[BS72] R. H. Bartels and G. W. Stewart. Solution of the equation $AX+XB = C$. *Communications of the ACM*, 15:820–826, 1972.

[Bye07] Ralph Byers. LAPACK 3.1 xHSEQR: Tuning and implementation notes on the small bulge multi-shift QR algorithm with aggressive early deflation. Technical Report 187, LAPACK Working Note, May 2007.

[Cam14] Carlos A. Campos. *Algoritmos de Altas Prestaciones para el Cálculo de la Descomposición en Valores Singulares y su Aplicación a la Reducción de Modelos de Sistemas Lineales de Control.* Tesis doctoral. Dpto. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Octubre 2014.

[CD02] Younès Chahlaoui and Paul Van Dooren. A collection of benchmark examples for model reduction of linear time invariant dynamical systems. Technical report, SLICOT Working Note 2002-2, February 2002.

[CDOP96] J. Choi, J. Dongarra, S. Ostrouchov, and A. Petitet. A proposal for a set of Parallel Basic Linear Algebra Subprograms. *Lecture Notes in Computer Science*, 1041:107–, 1996.

[CGHR04] C. Campos, D. Guerrero, V. Hernández, and R. Ralha. Algoritmos de altas prestaciones para la bidiagonalización de matrices densas. In *XV Jornadas de Paralelismo*, pages 78–83, Almería, Septiembre 2004.

[CGHR07] C. Campos, D. Guerrero, V. Hernández, and R. Ralha. Parallel bidiagonalization of a dense matrix. *SIAM Journal on Matrix Analysis and Applications*, 29(3):826–837, 2007.

[CH99] Jose M. Claver and Vicente Hernandez. Parallel adaptive wavefront algorithms solving Lyapunov equations for the Cholesky factor on message passing multiprocessors. *Concurrency: Practice and Experience*, 11(14):849–862, 1999.

[Chu87] King-wah Eric Chu. The solution of the matrix equations AXB-CXD=E and (YA-DZ,YC-BZ)=(E,F). *Linear Algebra and its Applications*, 93:93–105, 1987.

[CIGL01] K. I. Chang, K. Ide, M. Ghil, and C.-C. A. Lai. Transition to aperiodic variability in a wind-driven double-gyre circulation model. *Journal of Physical Oceanography*, 31:1260–1286, 2001.

[Cla99] Jose M. Claver. Parallel wavefront algorithms solving Lyapunov equations for the Cholesky factor on message passing multiprocessors. *The Journal of Supercomputing*, 13(2):171–189, 1999.

[CRHG08] C. Campos, R. Ralha, V. Hernández, and D. Guerrero. Nuevos resultados en la bidiagonalización de matrices densas. In *XIX Jornadas de Paralelismo*, Castellón, Septiembre 2008.

[Dat95] Biswa Nath Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole, Pacific Grove, CA, USA, 1995.

[Dat04] B.N. Datta. *Numerical Methods for Linear Control Systems: Design and Analysis*. Elsevier Academic Press, 2004.

[DCDH89] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms. In *Proceedings of the 3rd Conference on Parallel Processing for Scientific Computing*, pages 40–44, Philadelphia, PA, USA, Diciembre 1989. SIAM Publishers.

[DCHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[Dem97] J. W. Demmel. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[DSH89] Jack J. Dongarra, Danny C. Sorensen, and Sven J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215–227, 1989.

[DW95] Jack J. Dongarra and R. Clint Whaley. LAPACK working note 94: A user's guide to the BLACS v1.0. Technical Report UT-CS-95-281, Department of Computer Science, University of Tennessee, Marzo 1995.

[FGG92] L. Fortuna, G. Nunnary, and A. Gallo. *Model Order Reduction Techniques with Applications in Electrical Engineering*. Springer-Verlag, 1992.

[Fra61] J. G. F. Francis. The QR transformation: A unitary analogue to the LR transformation. *The Computer Journal*, 4(3):265–271, 1961.

[Fre03] R. W. Freund. Model reduction methods based on Krylov subspaces. *Acta Numerica*, 12:267–319, 2003.

[GHR02] D. Guerrero, V. Hernández, and J. E. Román. Parallel SLICOT Model Reduction Routines: The Cholesky Factor of Grammians. In *15th World Congress of the International Federation of Automatic Control (IFAC)*, Barcelona, Julio 2002.

[GHRV97a] David Guerrero, Vicente Hernández, José E. Román, and Antonio M. Vidal. Resolución en Paralelo de la Ecuación de Lyapunov Generalizada por el Método de Hammarling. In *XV Congreso de Ecuaciones Diferenciales y Aplicaciones/V Congreso de Matemática Aplicada*, pages 799–804, Vigo, 1997.

[GHRV97b] David Guerrero, Vicente Hernández, José E. Román, and Antonio M. Vidal. Resolución en Paralelo de la Ecuación de Lyapunov Generalizada por el Método de Hammarling en una Red de PC's. In *VIII Jornadas de Paralelismo*, pages 61–70, Cáceres, 1997.

[GJK09] Robert Granat, Isak Jonsson, and Bo Kågström. RECSY and SCASY library software: Recursive blocked and parallel algorithms for Sylvester-type matrix equations with some applications. In *Parallel Scientific Computing and Optimization*, volume 27 of *Springer Optimization and Its Applications*, pages 3–24. Springer New York, 2009.

[GK10a] R. Granat and B. Kågström. Algorithm 904: The SCASY library - parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part II. *ACM Transactions on Mathematical Software*, 37(3):Article 33, 2010.

[GK10b] R. Granat and B. Kågström. Parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part I: Theory and algorithms. *ACM Transactions on Mathematical Software*, 37(3), 2010.

[GKK09] R. Granat, B. Kågström, and D. Kressner. Parallel eigenvalue reordering in real Schur forms. *Concurrency and Computation: Practice and Experience*, 21:1225–1250, 2009.

[GKK10] Robert Granat, Bo Kågström, and Daniel Kressner. A novel parallel QR algorithm for hybrid distributed memory HPC systems. *SIAM Journal on Scientific Computing*, 32(4):2345–2378, 2010.

[GKKS15] Robert Granat, Bo Kågström, Daniel Kressner, and Meiyue Shao. Algorithm 953: Parallel library software for the multishift QR algorithm with aggressive early deflation. *ACM Transactions on Mathematical Software*, 41(4), 2015.

[GL91] J. Gardiner and A. Laub. Parallel algorithms for algebraic Riccati equations. *International Journal of Control*, 54(6):1317–1333, 1991.

[GL95] Michael Green and David J. N. Limebeer. *Linear Robust Control*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[GL05] Serkan Gugercin and Jing-Rebecca Li. Smith-type methods for balanced truncation of large sparse systems. *Dimension Reduction of Large-Scale Systems*, pages 49–82, 2005.

[GLR15] David Guerrero-López and José E. Román. Improving accuracy of Parallel SLICOT model reduction routines for stable systems. In *23th Mediterranean Conference on Control and Automation (MED 2015)*, pages 398–403, Torremolinos, Junio 2015.

[GSD01] Gene Golub, Knut Solna, and Paul Van Dooren. Computing the SVD of a general matrix product/quotient. *SIAM Journal on Matrix Analysis and Applications*, 22(1):1–19, 2001.

[GV13] G. H. Golub and C. F. Van Loan. *Matrix Computations, 4th ed.* Johns Hopkins University Press, Baltimore, 2013.

[Ham82] S. J. Hammarling. Numerical solution of the stable, non-negative definite Lyapunov equation. *IMA Journal of Numerical Analysis*, 2:303–323, 1982.

[HLPW86] M. T. Heath, A. J. Laub, C. C. Paige, and R. C. Ward. Computing the singular value decomposition of a product of two matrices. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1147–1159, 1986.

[HTP96] A. Scottedward Hodel, Bruce Tenison, and Kameshwar R. Poolla. Numerical solution of the Lyapunov equation by approximate power iteration. *Linear Algebra and its Applications*, 236:205–230, 1996.

[HVDG96] G. Henry and R. Van De Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: Myths and reality. *SIAM Journal on Scientific Computing*, 17(4):870–883, 1996.

[HWD02] Greg Henry, David Watkins, and Jack Dongarra. A parallel implementation of the nonsymmetric QR algorithm for distributed memory architectures. *SIAM Journal on Scientific Computing*, 24(1):284–311, 2002.

[ICQ+12] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. van de Geijn, and Field G. Van Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 72(9):1134–1143, 2012.

[JK94] I. M. Jaimoukha and E. M. Kasenally. Krylov subspace methods for solving large Lyapunov equations. *SIAM Journal on Numerical Analysis*, 31:227–251, 1994.

[Kal63] R. E. Kalman. Mathematical description of linear dynamical systems. *J. SIAM Control Ser. A*, 1:152–192, 1963.

[KB15] D. Kondrashov and P. Berloff. Stochastic modeling of decadal variability in ocean gyres. *Geophysical Research Letters*, 42(5):1543–1553, 2015.

[KBD+06] S. Kravtsov, P. Berloff, W. K. Dewar, M. Ghil, and J. C. McWilliams. Dynamical Origin of Low-Frequency Variability in a Highly Nonlinear Midlatitude Coupled Model. *Journal of Climate*, 19(24):6391–6408, 2006.

[KFA69] R. E. Kalman, P. L. Falb, and M. A. Arbib. *Topics in mathematical system theory*. International series in pure and applied mathematics. McGraw-Hill, 1969.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[KK11] Lars Karlsson and Bo Kågström. Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. *Parallel Computing*, 37(12):771–782, 2011.

[KKQQ08] B. Kagstrom, D. Kressner, E. S. Quintana-Orti, and G. Quintana-Orti. Blocked algorithms for the reduction to Hessenberg-triangular form revisited. *BIT Numerical Mathematics*, 48(3):1–22, 2008.

[KPT13] Daniel Kressner, Martin Plešinger, and Christine Tobler. A preconditioned low-rank CG method for parameter-dependent Lyapunov matrix equations. *Numerical Linear Algebra with Applications*, 21:666–684, 2013.

[Kre08] Daniel Kressner. Block variants of Hammarling's method for solving Lyapunov equations. *ACM Transactions on Mathematical Software*, 34(1):1:1–1:15, 2008.

[KS11] L. Knizhnerman and V. Simoncini. Convergence analysis of the extended Krylov subspace method for the Lyapunov equation. *Numerische Mathematik*, 118(3):567–586, 2011.

[LA89] Y. Liu and B. D. O. Anderson. Singular perturbation approximation of balanced systems. In *Proceedings of the 28th IEEE Conference on Decision and Control*, pages 1355–1360, 1989.

[Lau80] A. J. Laub. On computing balancing transformations. In *Joint Automatic Control Conference*, volume 1, pages FA8–E, San Francisco, 1980.

[Law78] C. Lawson. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1978.

[Lev96] W. S. Levine. *The Control Handbook*. Electrical Engineering Handbook series. Taylor & Francis, 1996.

[LHPW87] A. J. Laub, M. T. Heath, C. C. Paige, and R. C. Ward. Computation of system balancing transformations and other applications of simultaneous diagonalization algorithms. *IEEE Transactions on Automatic Control*, AC-32:115–122, 1987.

[LS85] G. J. Lastman and N. K. Sinha. An error analysis of the balanced matrix method of model reduction, with application to the selection of reduced-order models. *Large Scale Systems*, 9:63–71, 1985.

[LT85] P. Lancaster and M. Tismenetsky. *The Theory of Matrices*. Academic Press, New York, 1985.

[LW01] Jing-Rebecca Li and Jacob White. Reduction of large circuit models via low rank approximate gramians. *International Journal of Applied Mathematics and Computer Science*, 11(5):1151–1171, 2001.

[Mea00] S. P. Meacham. Low-Frequency Variability in the Wind-Driven Circulation. *Journal of Physical Oceanography*, 30(2):269–293, 2000.

[Moo81] B. C. Moore. Principal component analysis in lineary systems: Controllability, observability, and model reduction. *IEEE Transactions on Automatic Control*, AC-26:17–32, 1981.

[MPI94] MPI Forum. MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, 1994.

[MPW70] R. Martin, G. Peters, and J. Wilkinson. The QR algorithm for real Hessenberg matrices. *Numerische Mathematik*, 14:219–231, 1970.

[MS73] C. B. Moler and G. W. Stewart. An algorithm for generalized matrix eigenvalue problems. *SIAM Journal on Numerical Analysis*, 10(2):241–256, 1973.

[Mut99] A. G. O. Mutambara. *Design and Analysis of Control Systems*. Taylor & Francis, 1999.

[MW68] R. S. Martin and J. H. Wilkinson. Similarity reduction of a general matrix to Hessenberg form. *Numerische Mathematik*, 12(5):349–368, 1968.

[Par98] Beresford N. Parlett. *The symmetric eigenvalue problem*. Classics in Applied Mathematics. SIAM, Philadelphia, 1998.

[PCK91] P. Hr. Petkov, N. D. Christov, and M. M. Konstantinov. *Computational Methods for Linear Control Systems*. Prentice Hall Int. Series in Systems and Control Engineering. Prentice Hall, 1991.

[PDM14] Stefano Pierini, Henk A. Dijkstra, and Mu Mu. Intrinsic low-frequency variability and predictability of the Kuroshio current and of its extension. *Advances in Oceanography and Limnology*, 5(2):79–122, 2014.

[Pen98] Thilo Penzl. Numerical solution of generalized Lyapunov equations. *Advances in Computational Mathematics*, 8(1-2):33–48, 1998.

[Pen00] T. Penzl. *LYAPACK Users Guide: A MATLAB Toolbox for Large Lyapunov and Riccati Equations, Models Reduction Problems, and Linear Quadratic Optimal Control Problems*. Preprint-Reihe des Chemnitzer SFB 393. SFB 393, 2000.

[PMvdG+13] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):13:1–13:24, 2013.

[PR55] D. Peaceman and H. Rachford. The numerical solution of elliptic and parabolic differential equations. *Journal of SIAM*, 3(1):28–41, 1955.

[Rob80] J. D. Roberts. Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. *International Journal of Control*, 32(4):677–687, 1980.

[RS12] D. Raczynski and W. Stanislawski. Controllability and observability gramians parallel computation using GPU. *Journal of Theoretical and Applied Computer Science*, 6:47–66, 2012.

[Saa11] Youcef Saad. *Numerical methods for large eigenvalue problems*. SIAM, 2011.

[SBD+76] Brian T. Smith, James M. Boyle, Jack Dongarra, Burton S. Garbow, Yasuhiko Ikebe, Virginia C. Klema, and Cleve B. Moler. *Matrix Eigensystem Routines - EISPACK Guide, Second Edition*, volume 6 of *Lecture Notes in Computer Science*. Springer, 1976.

[SD02] Eric Simonnet and Henk A. Dijkstra. Spontaneous Generation of Low-Frequency Modes of Variability in the Wind-Driven Ocean Circulation. *Journal of Physical Oceanography*, 32(6):1747–1762, 2002.

[SG90] Youcef Saad and X. V. Gv. Numerical solution of large Lyapunov equations. In *Signal Processing, Scattering and Operator Theory, and Numerical Methods, Proc. MTNS-89*, pages 503–511. Birkhauser, 1990.

[Sha75] Y. Shamash. Model reduction using the Routh stability criterion and the Padé aproximation technique. *International Journal of Control*, 21(3):475–484, 1975.

[Ske80] R. E. Skelton. Cost decomposition of linear systems with application to model reduction. *International Journal of Control*, 32(6):1031–1055, 1980.

[Smi68] R. A. Smith. Matrix equation $XA + BX = C$. *SIAM Journal on Applied Mathematics*, 16(1):198–201, 1968.

[SOH94] Thomas Schreiber, Peter Otto, and Fridolin Hofmann. A new efficient parallelization strategy for the QR algorithm. *Parallel Computing*, 20(1):63–75, 1994.

[SOHL+96] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[Son98] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems (2nd Ed.)*. Springer-Verlag New York, Inc., 1998.

[SS12] T. Stykel and V. Simoncini. Krylov subspace methods for projected Lyapunov equations. *Applied Numerical Mathematics*, 62(1):35–50, 2012.

[SZ70] J. Snyders and M. Zakai. On nonnegative solutions of the equation $AD+DA' = -C$. *SIAM Journal on Applied Mathematics*, 18:704–714, 1970.

[SZ03] D. C. Sorensen and Y. Zhou. Direct methods for matrix Sylvester and Lyapunov equations. *Journal of Applied Mathematics*, 2003(6):277–303, 2003.

[TND10] Stanimire Tomov, Rajib Nath, and Jack Dongarra. Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. *Parallel Computing*, 36(12):645–654, 2010.

[TP87] M. S. Tombs and I. Postlethwaite. Truncated balanced realization of a stable non-minimal state-space system. *International Journal of Control*, 46(4):1319–1330, 1987.

[Var91a] A. Varga. Balancing-free square-root algorithm for computing singular perturbation approximations. *Proc. 30-th IEEE CDC*, 2:1062–1065, 1991.

[Var91b] A. Varga. Efficient minimal realization procedure based on balancing. *Prepr. of IMACS Symposium on Modelling and Control of Technological Systems*, 2:42–49, 1991.

[Var00] A. Varga. Model reduction software in the SLICOT library. In *Applied and Computational Control, Signals, and Circuits, volume 629 of The Kluwer International Series in Engineering and Computer Science*, pages 239–282. Kluwer Academic Publishers, 2000.

[VdG88] R. A. Van de Geijn. *Storage Schemes for Parallel Eigenvalue Algorithms*. Technical report. University of Texas at Austin, Department of Computer Sciences, 1988.

[vdG93] Robert A. van de Geijn. Deferred shifting schemes for parallel QR methods. *SIAM Journal on Matrix Analysis and Application*, 14(1):180–194, 1993.

[vdGH89] R. A. van de Geijn and D. G. Hudson. An efficient parallel implementation of the nonsymmetric QR algorithm. In *Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 697–700, 1989.

[vdGW95] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. Technical Report TR-95-13, Department of Computer Sciences, The University of Texas, Abril 1995.

[Wac88] E. L. Wachspress. Iterative solution of the Lyapunov matrix equation. *Applied Mathematics Letters*, 1:87–90, 1988.

[Wal95] David W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[Wat94] D. S. Watkins. Shifting strategies for the parallel QR algorithm. *SIAM Journal on Scientific Computing*, 15(4):953–958, 1994.

[Wat07] David S. Watkins. *The matrix eigenvalue problem: GR and Krylov subspace methods*. SIAM, 2007.

[Wat08] David S. Watkins. The QR algorithm revisited. *SIAM review*, 50(1):133–145, 2008.

[WP05] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[WZS+14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi(TM)*, pages 167–188. Springer International Publishing, 2014.

[WZZY13] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 25:1–25:12, New York, NY, USA, 2013. ACM.

[ZDG96] Kemin Zhou, John C. Doyle, and Keith Glover. *Robust and Optimal Control*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.