UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Máster en Computación Paralela y Distribuida

Tesis de Máster

Computación Paralela Heterogénea y Homogénea para la Resolución de Problemas Estructurados de Álgebra Lineal Numérica

Miguel Óscar Bernabeu Llinares

Diciembre de 2008

Dirigida por:

Dr. Antonio Manuel Vidal Maciá

Índice general

Índice general I

Índice de figuras V

Índice de tablas VII

1. Introducción 1

1.1. Planteamiento y Objetivos 2

2. Organización de un Clúster de Computadores Heterogéneo 3

2.1. Objetivos 3

2.2. Descripción del hardware 4

2.3. Organización física 4

2.4. Descripción del software 6

2.4.1. Toolkits y distribuciones Linux orientadas a clústers 6

2.4.2. Compiladores 13

2.4.3. Bibliotecas de computación altas prestaciones 13

2.4.4. Resto de software 16

2.5. Configuración 18

2.5.1. Red 18

2.5.2. Creación de usuarios 18

2.5.3. Instalación de software para múltiples arquitecturas 19

2.5.4. Sistema de colas 20

2.6. Ejemplo de uso 21

3. Cálculo de Valores Propios de Matrices Estructuradas. Introducción 25

4. Cálculo de VPs de matrices tridiagonales simétricas: aproximación heterogénea 27

4.1. Introducción 27

4.2. Descripción del problema y solución propuesta 28

4.2.1. Matrices de prueba 31

4.3. Modelo computacional heterogéneo 31

4.4. Esquemas paralelos heterogéneos 32

4.4.1. Algoritmos implementados 34

4.5. Análisis experimental 35

4.6. Conclusiones 35

5. Implementación paralela de un algoritmo tipo Lanczos para un problema de VPs estructurado 41

5.1. Introducción 41

5.2. Descripción del problema 42

5.3. Algoritmo secuencial 43

5.3.1. Algoritmo de Lanczos con Shift-and-Invert 43

5.3.2. Factorización LDL^T 43

5.3.3. Descomposición del intervalo principal 44
5.3.4. Algoritmo secuencial 44
5.4. Implementaciones paralelas 45
5.4.1. Detalles de implementación 47
5.5. Resultados experimentales 48
5.5.1. Análisis de los resultados 49
5.6. Conclusiones 51

6. Cálculo paralelo de los valores propios de una matriz Toeplitz simétrica mediante métodos iterativos 53

6.1. Introducción 53
6.2. Descripción del método 54
6.2.1. Optimizaciones en Lanczos para el caso Toeplitz simétrico 54
6.2.2. Resolución de sistemas de ecuaciones Toeplitz simétricas 56
6.2.3. Método completo 57
6.2.4. Selección de los subintervalos 58
6.3. Implementación y paralelización del método 59
6.3.1. Paralelización 59
6.4. Resultados experimentales 60
6.5. Conclusiones 63

Bibliografía 65

A. Artículos públicados 69

Índice de figuras

2.1. Diseño del clúster 5

2.2. Bibliotecas álgebra lineal numérica. 15

2.3. Rendimiento implementaciones DGEMM. Fuente: Intel 16

2.4. Ejemplo de traza generada con MPE 17

4.1. Particionado del espectro 29

4.2. Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de espectro uniforme 37

4.3. Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de Wilkinson . 38

5.1. Matrices A y B para el problema TE de un guía tipo *ridge* 43

6.1. Speedup para $N = 5000$ 61

6.2. Speedup para $N = 10000$ 62

Índice de tablas

2.1. Características de los nodos del clúster 4

2.2. Esquema de direcciones IP del clúster 18

2.3. Colas disponibles 21

4.1. Matrices de prueba 31

4.2. Vector P_t para el cálculo de los valores propios de matrices de espectro uniforme 32

4.3. Vector P_t para el cálculo de los valores propios de matrices de Wilkinson 33

4.4. Matriz de latencias B_c (s) 33

4.5. Matriz del inverso del ancho de banda T_c (s) 34

4.6. Tiempo de ejecución, en P0, del algoritmo secuencial (A1) para ambos tipos de matrices 36

4.7. Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de espectro uniforme 36

4.8. Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de Wilkinson . 37

4.9. Speed-up del algoritmo A4 y A6 respecto del algoritmo A2 (ScaLAPACK) sobre matrices de espectro uniforme 38

4.10. Speed-up del algoritmo A4 y A6 respecto del algoritmo A2 (ScaLAPACK) sobre matrices de Wilkinson 38

5.1. Tiempo de ejecución de la versión MPI en el clúster SMP 48

5.2. Tiempo de ejecución de la versión OpenMP en el clúster SMP 48

5.3. Tiempo de ejecución de la versión MPI+OpenMP en el clúster SMP 49

5.4. Tiempo de ejecución de la versión OpenMP en el SGI Altix 49

5.5. Tiempo de ejecución de la versión MPI en el SGI Altix 49

5.6. Speed-up en el clúster SMP para las versiones OpenMP y MPI ($p = 2$). 50

5.7. Speed-up en el clúster SMP para las versiones MPI+OpenMP y MPI ($p = 4$). . . 50

5.8. Speed-up en el SGI Altix de las versiones MPI y OpenMP ($M + N = 20000$). . . 51

6.1. Tiempo de ejecución (s) de las fases de aislamiento y extracción 59

6.2. Tiempo de ejecución (s) del la versión secuencial del Algoritmo 5 y de LAPACK, calculando todos los valores propios 61

6.3. Tiempo de ejecución (s) del la versión paralela del Algoritmo 5 y de ScaLAPACK, calculando todos los valores propios 62

6.4. Tiempo de ejecución (s) del Algoritmo 5, LAPACK y ScaLAPACK, calculando el 10, 20 y 50 % de valores y vectores propios. P es el número de procesadores . . . 63

Capítulo 1

Introducción

La presentación y defensa pública de la presente Tesis de Máster es requisito para la obtención del título de Máster en Computación Paralela y Distribuida por el Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia (UPV).

Según la normativa del propio Máster: "La Tesis de Máster es un trabajo de especialización en el ámbito de la Computación Paralela y Distribuida, centrado o bien en el desarrollo de una aplicación de tipo práctico que facilite la incorporación del alumno al mundo profesional, o bien en la realización de un trabajo de investigación que le inicie en el mundo de la investigación". La presente Tesis de Máster se enmarca en el segundo tipo de trabajo: el de introducción a la carrera investigadora. Concretamente viene a resumir el trabajo realizado por el autor en el seno del Grupo de Investigación en Computación Paralela del DSIC de la UPV durante los últimos tres años.

A pesar de la orientación puramente investigadora del trabajo, no se ha dejado de lado la parte aplicada. Por una parte al presentar y utilizar soluciones hardware perfectamente implantables por cualquier empresa con necesidades reales de procesamiento de altas prestaciones. Así como por ofrecer ejemplos de transferencia a campos diversos de la ingeniería de las soluciones software desarrolladas.

Parte de la investigación presentada en este trabajo ha sido financiada por los siguientes proyectos de investigación:

- "Paralelización y Optimización de Algoritmos de Procesado Digital en Sistemas de Audio

Virtual". Vicerrectorado de Investigación, Desarrollo e Innovación. UPV

- "Desarrollo y Optimización de Código Paralelo para Sistemas de Audio 3D". Ministerio De Ciencia Y Tecnología. D.G. De Investigación. TIC 2003-083238-C02-02
- "Técnicas Computacionales Secuenciales y Paralelas Aplicadas a Problemas de Microondas y Elementos Radiantes". Vicerrectorado de Investigación, Desarrollo e Innovación. UPV

Asimismo el autor de la presente Tesis de Máster es titular de una beca para la formación de personal investigador de carácter predoctoral concedida por la Conselleria de Empresa, Universidad y Ciencia de la Generalitat Valenciana.

1.1. Planteamiento y Objetivos

En el presente trabajo se han aplicado técnicas de computación paralela en la resolución de problemas de álgebra lineal numérica.

El principal problema estudiado ha sido el de cálculo de valores propios de matrices estructuradas. Se ha estudiado el caso de las matrices tridiagonales (Capítulo 4), el de las matrices Toeplitz simétricas (Capítulo 6) y un caso de estudio particular proveniente de un problema de ingeniería (Capítulo 5). Se han aplicado técnicas de computación paralela tanto homogénea como heterogénea.

Con el fin de disponer de una plataforma donde ejecutar nuestros algoritmos heterogéneos, se ha diseñado, ensamblado y configurado un clúster de computadores heterogéneos. El Capítulo 2 documenta dicha experiencia.

Capítulo 2

Organización de un Clúster de Computadores Heterogéneo

Como apuntábamos en el capítulo primero de esta tesis, el primer objetivo a abordar es el diseño, configuración y administración de un clúster de computadores heterogéneo.

El concepto de clúster de computadores es de sobra conocido en el mundo de la computación de altas prestaciones, por lo que carece de interés definirlo aquí. El interés de este capítulo radica en las soluciones aportadas para conseguir integrar, en un mismo clúster, hardware tecnológicamente muy dispar, consiguiendo que pueda trabajar coordinadamente y ofreciendo una imagen de supercomputador lo más compacta posible.

2.1. Objetivos

- Integrar hardware tecnológicamente heterogéneo (diferentes arquitecturas, capacidades de cálculo, almacenamiento, etc)

- Adoptar soluciones que minimicen el mantenimiento de la máquina.

- Ofrecer al programador una imagen del clúster lo más compacta posible.

2.2. Descripción del hardware

La tabla 2.1 muestra una breve descripción de las características de los 7 nodos que vamos a integrar en el clúster.

					Cachés					
			Núcleos	Frec.	Nivel 1		Nivel 2			
Nodo	Tipo	CPUs	/ CPU	reloj	insts.	datos	insts.	datos	Nivel 3	RAM
1	Pentium 4	1	1	3.0GHz	150KB	8KB	1MB		–	1GB
2	Pentium 4	1	1	1.6Ghz	150KB	8KB	256KB		–	1GB
3	Pentium 4	1	1	1.7Ghz	150KB	8KB	256KB		–	1GB
4	P4 Xeon	2	1	2.2GHz	150KB	8KB	512KB		–	4GB
5	P4 Xeon	2	1	2.2GHz	150KB	8KB	512KB		–	4GB
6	Itanium II	4	2	1.4GHz	16KB	16KB	1MB	256KB	12MB	8GB
7	Itanium II	4	2	1.4GHz	16KB	16KB	1MB	256KB	12MB	8GB

Tabla 2.1: Características de los nodos del clúster

Asimismo se dispone del siguiente hardware de red para la interconexión de los nodos.

- Red Gigabit Ethernet (1GB/s)
  - 1 Switch Gigabit Ethernet Cisco Systems Catalyst 2960G
  - 5 tarjetas de red con chipset Intel Corporation 82541PI Gigabit Ethernet Controller
  - 2 tarjetas de red con chipset Intel Corporation 82545GM Gigabit Ethernet Controller
- Red Fast Ethernet (100MB/s)
  - 1 Switch Fast Ethernet 3COM
  - 8 tarjetas de red Fast Ethernet de diversos modelos

2.3. Organización física

La organización del hardware elegida es la habitual en las plataformas de tipo clúster. El primer nodo actuará como punto de conexión al sistema o "frontend" y por tanto estará conectado a la

red de área local del departamento. En dicho nodo se encontrará centralizada toda la administración del sistema (instalación de software, gestión de usuarios, monitorización, seguridad, ...). Desde el punto de vista del usuario en dicho nodo se prevé que se realicen todas las tareas de desarrollo, compilación e interacción con el sistema de colas, así como la recogida de resultados.

Tras el frontend se situarán el resto de nodos destinados a la ejecución de las tareas de cálculo, que estarán interconectados entre si y con el frontend. Se ha decidido instalar dos redes en el clúster. Una de tipo Fast Ethernet, que se encargue del trafico de red generado por las tareas de administración y monitorización del clúster, así como por el sistema de ficheros distribuido. Y otra de tipo Gigabit Ethernet dedicada exclusivamente a las comunicaciones generadas por los algoritmos paralelos que se ejecuten en el clúster. La figura 2.1 muestra un esquema del diseño.

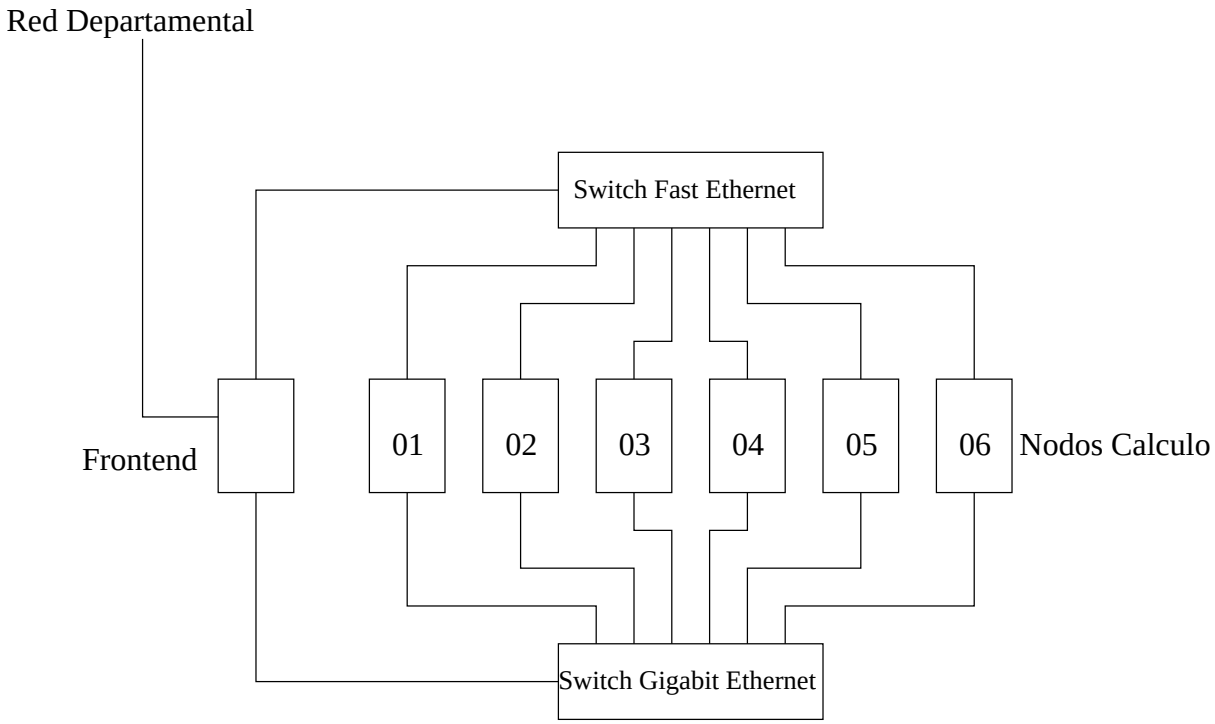

Figura 2.1: Diseño del clúster

Con esta multiplexación del tráfico de red, descargamos a la red rápida de trafico que no sea el generado por los algoritmos paralelos con el objetivo de conseguir un comportamiento lo más determinista posible. Supongamos un escenario en el que tras la sustitución de un componente defectuoso en un nodo de cálculo es necesario reinstalar el sistema operativo. El frontend puede transmitir a través de la red Fast Ethernet la imagen del sistema operativo sin sobrecargar la red Gigabit Ethernet y por tanto sin introducir perturbaciones en los programas paralelos que se estén ejecutando en el resto de nodos del clúster.

Otro escenario ventajoso para este diseño es el caso de programas paralelos que realicen E/S sobre sistemas de ficheros distribuidos (e.g. directorio /home exportado a través de NFS). Ya que si configuramos el sistema de ficheros distribuido para que trabaje con la red Fast Ethernet, podemos solapar las operaciones de E/S con otras operaciones de paso de mensajes a través de la red Gigabit Ethernet evitando problemas de sobrecarga y contención en la red.

2.4. Descripción del software

2.4.1. Toolkits y distribuciones Linux orientadas a clústers

A la hora de realizar la instalación del sistema operativo y el software necesario en el clúster podemos seguir dos aproximaciones: i) realizar la instalación y configuración del entorno nodo por nodo o ii) utilizar algún toolkit específico para la instalación centralizada de todo el software necesario y posterior configuración automática del entorno. La primera aproximación nos permite personalizar mucho más cada una de las instalaciones y por tanto puede ser ventajosa a la hora de trabajar con entornos muy heterogéneos. Además de que no requiere conocer los detalles de funcionamiento de ninguna herramienta específica, más que las propias de cualquier administrador de sistemas. Desgraciadamente una instalación de este tipo se va a encontrar rápidamente con problemas de escalabilidad y de incoherencia de configuraciones en el momento que el número de nodos del clúster aumente por encima de las pocas decenas.

Por otra parte, en los últimos años [28] han aparecido una serie de herramientas orientadas a automatizar la instalación y configuración de este tipo de arquitecturas paralelas. No se trata de soluciones específicas para clústers de cálculo, ya que se ha documentado su uso en clústers orientados a alta disponibilidad [22], granjas de servidores web, ... De cualquier forma, la idea básica es, basándose en el modelo de un nodo maestro (frontend) y un conjunto de nodos de cálculo, crear en el frontend un repositorio de todo el software que vamos a necesitar en el clúster (sistema operativo inclusive) y dejar al toolkit que se encargue de instalarlo automáticamente en los nodos de cálculo. Las principales ventajas de esta aproximación son la mejora en la escalabilidad del proceso de instalación y en la consistencia de las configuraciones, así como la facilidad para agregar nuevos nodos al clúster o recuperarse de fallos en el hardware. Por contra, su utilización requiere un esfuerzo de aprendizaje de la herramienta nada despreciable así como amplios conocimientos en administración de sistemas.

A continuación se presenta un breve estado del arte de este tipo de herramientas.

WareWulf/Perceus

El proyecto WareWulf fue originalmente desarrollado por Greg Kurtzer. Recientemente se ha producido una reestructuración del mismo, dividiéndose el original en dos subproyectos; WareWulf y Perceus. Por una parte Perceus [3], propiedad de la empresa Infiniscale, trata de recoger toda la experiencia generada por el proyecto original, para realizar un nuevo diseño, desde cero, de las herramientas de instalación y mantenimiento. Por otra parte, el proyecto original se continúa desarrollando y ofreciendo soporte para las utilidades de monitorización y las herramientas de usuario.

El proceso de instalación de un clúster con WareWulf/Perceus es, a grandes rasgos, el siguiente: tras instalar en el frontend alguna de las distribuciones de Linux soportadas, se instala el toolkit (existe documentación descriptiva del proceso). A continuación hay que preparar la instalación de los nodos de cálculo. Para ello, hay que crear, o bien descargar, uno de los llamados VNFS (Virtual Node File System). Un VNFS no es más que una imagen del sistema a instalar en los nodos. Ésta se crea a partir de un subdirectorio en el frontend (normalmente /vnfs/default) que contiene el sistema de ficheros que montarán los nodos de cálculo, con toda la estructura de directorios de configuración y de programas habitual en cualquier distribución Linux (/vnfs/default/etc, /vnfs/default/usr, ...) con la excepción del kernel y del directorio /boot. Además de los elementos propios del sistema operativo, instalaremos en dicho sistema de ficheros virtual todas las bibliotecas y aplicaciones que queramos que estén disponibles en los nodos de cálculo.

Para instalar el resto de nodos es necesario habilitarles el arranque a través de PXE (disponible en la mayoría de tarjetas de red actuales). Cuando el frontend detecte el arranque de un nodo de cálculo, se encargará de registrarlo en el sistema (asignarle una dirección IP, un nombre válido, ...), de suministrarle el kernel y los ficheros necesarios para el arranque y finalmente de

transferirle la imagen del sistema operativo (VNFS) a través del protocolo TFTP. Nótese que el sistema transferido no es instalado en la memoria secundaria del nodo, sino que se crea un ramdisk[1] para almacenarlo.

En el diseño original de WareWulf, el sistema transferido se almacenaba en memoria principal en un ramdisk, no siendo necesario que el nodo disponga de disco duro. Esto puede ser problemático en configuraciones con poca memoria principal o cuando los VNFS alcanzan un tamaño considerable. En las últimas versiones del entorno, se permite un modo de funcionamiento mixto, en el cual las partes más críticas del sistema se transfieren al ramdisk del nodo, mientras que aquellas menos críticas y de sólo lectura se exportan desde el frontend mediante el protocolo NFS. Con esta solución se reduce el consumo de memoria principal en los nodos de cálculo a cambio de introducir algo de trafico NFS en la red.

Las principales ventajas de WareWulf/Perceus son:

- Es independiente de la distribución de Linux que se quiera instalar en los nodos.
- La instalación de los nodos es muy ligera y por tanto rápida y escalable.
- Permite el uso de nodos de cálculo sin disco duro.

Algunos de sus inconvenientes son:

- Sólo soporta arquitecturas x86 y x86_64 [2].
- No da soporta a clústers heterogéneos.
- La documentación disponible es escasa y la comunidad de usuarios reducida.
- En cada reinicio de un nodo cálculo es necesario transferir la imagen del sistema.
- En la instalación básica se echan en falta gran cantidad de bibliotecas y software ampliamente usado en computación de altas prestaciones.

[1] Área en la memoria principal del nodo que emula un dispositivo de almacenamiento secundario.
[2] http://www.perceus.org/portal/node/13

NPACI Rocks

El proyecto NPACI Rocks [26] es una iniciativa del San Diego Supercomputer Center y esta financiado por la NSF de los EE.UU. Rocks puede describirse mejor como una distribución Linux para clústers que como un toolkit, ya que se basa en el sistema operativo Red Hat Linux y no ofrece la posibilidad de usar otra distribución. Tanto la documentación disponible como la comunidad de usuarios es muy amplia y es fácil recibir soporte en foros y listas de correo. Además se organizan congresos y talleres donde se presentan nuevas características y avances.

Rocks posee un diseño muy modular basado en "rolls"; cada uno de los rolls agrupa software (con sus opciones de instalación y configuración) temáticamente, por ejemplo existe un roll con bibliotecas de computación de altas prestaciones, otro que instala el gestor de colas SGE [1] o PBS [2], uno con software para integración en Grid, así como otros con aplicaciones específicas de ciertos campos de la ingeniería y la ciencia, etc. A la hora de realizar la instalación y configuración de un clúster mediante Rocks se procede de la siguiente forma: en primer lugar se arranca el frontend con el programa de instalación de Rocks, éste nos pregunta la información básica sobre nombre del clúster, configuración de red externa, etc. A continuación se nos pide que indiquemos que rolls deseamos instalar y para que arquitecturas, dichos rolls podemos suministrarlos a través de algún medio de almacenamiento extraíble o dejar que el programa se conecte a alguno de los repositorios disponibles y los descargue. Nótese que el sistema operativo se trata como un roll más, pudiendo elegir entre usar copias con licencia de Red Hat Linux o alguno de sus clones de libre distribución como CentOS.

Una vez que el sistema posee toda la información sobre las características de la instalación a realizar, procede a instalarse en el frontend. Cuando ha terminado, el administrador debe de indicar que va a instalar los nodos de cálculo. Para instalarlos se arrancan los nodos con el

mismo programa de instalación descrito anteriormente, aunque esta vez no es necesaria ninguna interacción del administrador. Automáticamente se conecta al frontend y descarga tanto el SO como el software necesario.

Una de las características destacables de esta distribución es que crea una base de datos con toda la configuración del clúster. A partir de dicha base de datos se generan todos los ficheros de configuración necesarios en los nodos de cálculo y en el frontend. De esta forma el administrador puede acceder a dicha información de una forma centralizada y propagar fácilmente los cambios a todo el clúster.

Otra de las decisiones de diseño más acertadas de la herramienta es como se gestiona el software a instalar en los nodos a base de rolls. Como ya se apuntaba anteriormente, el sistema operativo se trata como un roll más. Esto permite que puedan coexistir en el repositorio sistemas operativos (y herramientas) para diferentes arquitecturas. En el momento de instalar cada nodo de cálculo, el frontend transfiere los rolls correspondientes a su arquitectura, independientemente de que ambos la compartan. Puesto que los rolls son paquetes binarios que no ofrecen la posibilidad de configurar el software que contienen, Rocks implementa un sistema de configuración del software "post-instalación". Éste se basa en ficheros XML que definen las operaciones de creación y modificación de ficheros de configuración a realizar sobre los rolls instalados. La elección de XML como formato de descripción permite que puedan ser aplicados en distintas arquitecturas sin problemas de compatibilidad.

Las principales ventajas de NPACI Rocks son:

- Soporta arquitecturas x86, x86_64 y ia64.
- Da soporte a clústers heterogéneos.
- Existe gran cantidad de documentación disponible.
- Ofrece la posibilidad de instalar automáticamente gran cantidad de bibliotecas y software específico.
- La instalación de los nodos de cálculo requiere poca preparación e interacción del administrador.

Algunos de sus inconvenientes son:

- Sólo soporta distribuciones Red Hat Enterprise Linux y clones.
- El tamaño de la instalación en los nodos de cálculo es grande.
- En clústers de gran tamaño, la instalación de los nodos de cálculo puede tener problemas de escalabilidad debido al gran volumen de tráfico de red generado.

OSCAR

El proyecto OSCAR (Open Source Cluster Application Resources [24]) es una iniciativa de la organización OpenClusterGroup para fomentar el uso de arquitecturas clúster como alternativa de computación de altas prestaciones. El principal objetivo del proyecto es ofrecer soluciones para la instalación y administración de clústers basadas en código abierto y que sean fácilmente implantables por cualquier usuario sin amplios conocimientos de administración de sistemas.

La aproximación empleada a la hora de realizar la instalación y configuración de un clúster es similar a la de WareWulf/Perceus. En primer lugar es necesario instalar en el frontend alguna de las distribuciones Linux soportadas (todas aquellas que usan RPM como gestor de paquetes). Tras esto es necesario crear la jerarquía de directorios que albergará los paquetes RPM necesarios para crear las imágenes a instalar en los nodos de cálculo y transferirlos.

En OSCAR la creación de estas imágenes se encuentra automatizada en el proceso de instalación del toolkit. Para ello se emplea la herramienta SIS (System Installer Suite [4]), la cual ofrece un amplio abanico de opciones de configuración y administración de las imágenes instaladas, valor añadido a OSCAR.

A continuación se procede a la instalación del toolkit en sí. Dicha instalación incluye la recogida de la información necesaria sobre el clúster, la descarga de software adicional que queramos instalar (bibliotecas de comunicaciones y cálculo, aplicaciones para monitorización, etc), la creación de las imágenes a instalar y finalmente la instalación de éstas en los nodos de cálculo. Para la instalación de los nodos es necesario habilitar el arranque mediante el protocolo PXE. Cuando el frontend detecte el arranque de un nuevo nodo de cálculo se encargará de registrarlo en el

sistema y transferirle la imagen a instalar. Finalmente OSCAR dispone de una sencilla herramienta para testear el correcto funcionamiento de los principales elementos del sistema tras la instalación del clúster.

Al igual que en Rocks, OSCAR emplea una base de datos para almacenar toda la información referente a los nodos dados de alta, las IPs que se les han asignado y otros detalles de configuración. En este caso la base de datos es la propia que genera la herramienta SIS.

Las principales ventajas de NPACI Rocks son:

- Soporta arquitecturas x86, x86_64 y ia64.
- Da soporte a algunas configuraciones de clústers heterogéneos [19].
- Está basada íntegramente en soluciones de código abierto.
- La herramienta para la instalación de los nodos de cálculo es muy potente y versátil.

Algunos de sus inconvenientes son:

- Sólo soporta distribuciones basadas en el gestor de paquetes RPM.
- El tamaño de la instalación en los nodos de cálculo es grande.

Conclusiones

Tras un cuidadoso análisis de los toolkits disponibles para la instalación y configuración del clúster, nos hemos decantado por el uso de NPACI Rocks. El principal motivo es que se trata del único, entre los estudiados, que da un correcto soporte a un sistema heterogéneo como el que nos ocupa, formado por nodos con arquitecturas x86 y ia64. Como ya citábamos anteriormente, OSCAR da soporte algunas configuraciones heterogéneas [19], concretamente a aquellas en las que exista compatibilidad binaria entre todos los nodos (por ejemplo x86 y x86_64). En caso de no existir dicha compatibilidad el proceso de instalación se complica en gran manera, hecho que nos ha llevado a descartarlo. Finalmente WareWulf/Perceus está lejos de ofrecer las funcionalidades del resto de toolkits citados aquí.

2.4.2. Compiladores

Los requerimientos del sistema en cuanto a compiladores son los habituales en entornos de cálculo numérico y computación paralela: C/C++ y Fortran90. Además es necesario que ofrezcan ciertas características de autooptimización del código así como soporte para la generación de programas paralelos a partir del estándar OpenMP.

Puesto que todos los procesadores del clúster son diseños de la empresa Intel, se ha decidido instalar los compiladores de Intel para C/C++ y Fortran90 en su versión 9.1. Dichos compiladores cubren las necesidades descritas anteriormente.

Además de éstos, también se dispone de los compiladores del proyecto GNU (presentes en cualquier distribución Linux) para C/C++ y Fortran77 en su versión 3.4.6.

2.4.3. Bibliotecas de computación altas prestaciones

Junto con los compiladores, las bibliotecas numéricas y de comunicaciones son la piedra angular de la computación de altas prestaciones. Aunque es difícil prever las necesidades de los usuarios del sistema, es necesario cubrir las más habituales: álgebra lineal numérica densa y dispersa, secuencial y paralela, así como bibliotecas de comunicaciones por paso de mensajes. El soporte a la programación en memoria compartida ya lo ofrecen los compiladores instalados.

Además, no basta con ofrecer un amplio abanico de bibliotecas (evitando así que los usuarios tengan que instalar su propio software) sino que además hay que asegurarse de que se instalan versiones optimizadas de las bibliotecas para las arquitecturas en cuestión. A ésto hay que añadir la necesidad de ejecutar tests (la mayorías de veces integrados en la propia biblioteca) que aseguren un correcto comportamiento funcional y numérico.

A continuación se ofrece una pequeña descripción del software disponible:

Comunicaciones

Desde hace una década, MPI es el estándar *de facto* para la comunicación por paso de mensajes en entornos paralelos. Existen diversas implementaciones del mismo. Las más conocidas son: LAM/MPI, MPICH, MPICH 2, Intel MPI y Open MPI. Cada una de ellas presenta característi-

cas particulares en cuanto a gestión de procesos, optimización y soporte a la heterogeneidad. Aunque mantienen una interfaz unificada según los estándares MPI-1.2 o MPI-2.1.

En nuestro caso, prima que la implementación elegida dé soporte a la ejecución en plataformas heterogéneas, donde el tamaño y la representación de los datos [3] puede no coincidir entre los diferentes nodos. Hemos realizado un pequeño estudio del soporte ofrecido con los siguientes resultados:

LAM/MPI da soporte para la comunicación entre arquitecturas con diferente "endian", aunque no soporta diferencias en el tamaño de los tipos de datos [4].

MPICH1 puede manejar correctamente diferencias en la representación de los datos y en el tamaño de los tipos.

MPICH2 no ofrece ningún soporte a la heterogeneidad. Aunque según sus desarrolladores, se espera ofrecerlo a lo largo del año 2007 [5].

Intel MPI está basado en MPICH 2 por lo que los resultados son análogos.

Open MPI es la continuación de LAM/MPI y el soporte a la heterogeneidad es el mismo [6].

Por tanto, hemos elegido MPICH1 como implementación MPI para el clúster. A pesar de que se trata de un proyecto concluido (sus desarrolladores trabajan ahora en MPICH2) sigue teniendo gran número de usuarios y su última versión 1.2.7p1 es muy estable.

Álgebra Lineal Numérica

Los investigadores que van a trabajar en la máquina, lo van a hacer principalmente, en el área del álgebra lineal numérica, desarrollando y evaluando algoritmos tanto secuenciales como paralelos. Por tanto va a ser necesario proveer la máquina con las bibliotecas que ofrecen los núcleos computacionales básicos sobre los que desarrollar nuevas implementaciones o evaluar el funcionamiento y posible mejora de las existentes. Esta jerarquía básica de bibliotecas de álgebra lineal numérica es la formada por BLAS [16], LAPACK [7], BLACS [18], PBLAS [15] y ScaLAPACK [15]. La Figura 2.2 las relaciona gráficamente.

[3] little-endian o big-endian

[4] http://www.lam-mpi.org/faq/category11.php3#question2

[5] http://www-unix.mcs.anl.gov/mpi/mpich2/

[6] http://www.open-mpi.org/faq/?category=supported-systems#heterogeneous-support

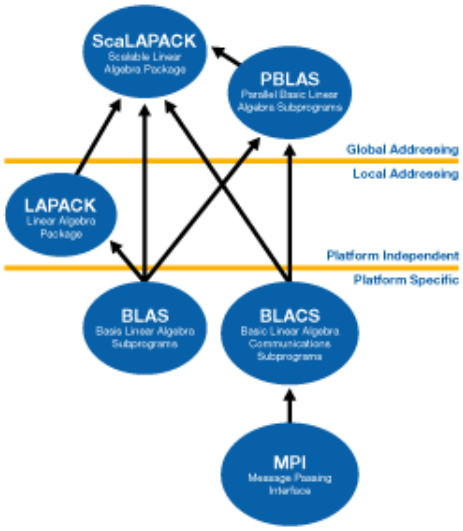

Figura 2.2: Bibliotecas álgebra lineal numérica.

Puesto que la eficiencia de estos núcleos computacionales suele ser determinante para el rendimiento de los algoritmos que se desarrollan a partir de ellos, es necesario elegir una implementación con buenas garantías de funcionalidad y rendimiento. Se ha elegido la implementación optimizada para las arquitecturas Intel que distribuye la propia empresa: Intel MKL Cluster en su versión 9.0. Los motivos para su elección han sido:

- Ofrece núcleos optimizadas para diferentes arquitecturas (x86, x86_64 y ia64).
- Ofrece versiones paralelas para memoria compartida de todas las rutinas de nivel 3 de BLAS, así como de las principales de LAPACK.
- Los miembros del grupo poseen amplia experiencia en su uso, considerándose los resultados obtenidos con ella satisfactorios. La Figura 2.4 muestra una comparativa entre el rendimiento de la implementación de Intel del producto matriz-matriz denso y la de ATLAS [33].
- Permite usar diversas implementaciones de MPI (MPICH1, MPICH2 e Intel MPI) como capa básica de comunicaciones.

Además de las bibliotecas básicas citadas anteriormente, también se incluyen las siguientes herramientas:

Sparse BLAS Subconjunto de los núcleos de BLAS para matrices dispersas (formatos CSR, CSC, Diagonal y Skyline).

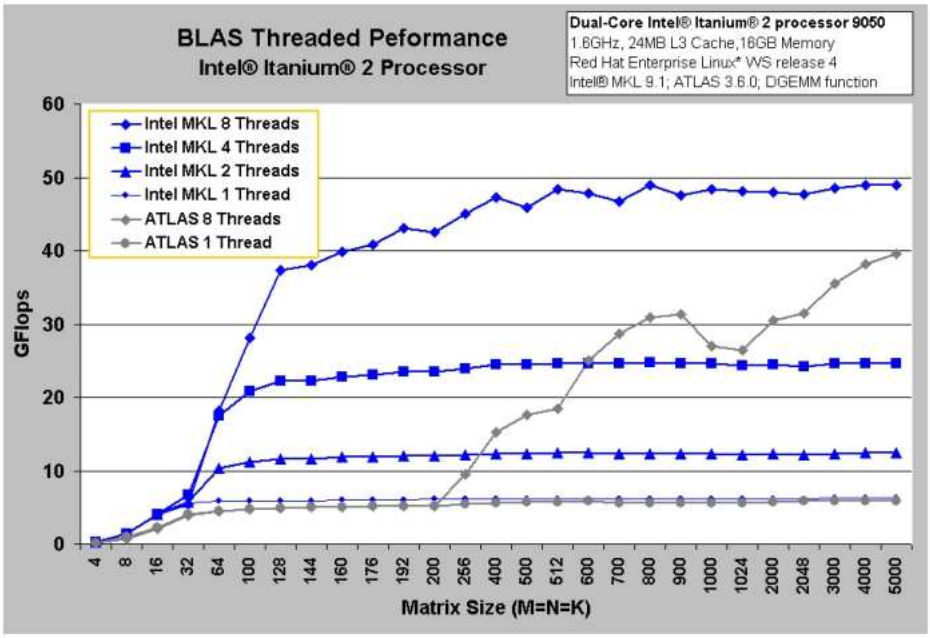

Figura 2.3: Rendimiento implementaciones DGEMM. Fuente: Intel

PARDISO Métodos directos para la resolución de sistemas de ecuaciones dispersos.

FFTs y DFTs Transformadas de Fourier.

VML Núcleos computacionales vectoriales.

VSL Generadores de números aleatorios para diferentes distribuciones de probabilidad.

Para acabar de cubrir las necesidades de software de los miembros del grupo, se han instalado las siguiente bibliotecas:

ARPACK Biblioteca para la resolución de problemas de autovalores mediante métodos iterativos de tipo Arnoldi/Lanczos (densos y dispersos).

SPARSKIT Precondicionadores y métodos iterativos para la resolución de sistemas de ecuaciones dispersos.

FFTPACK Rutinas para el cálculo de las transformadas trigonométricas y de Fourier.

2.4.4. Resto de software

Además de todo el software citado anteriormente se ha instalado las siguientes aplicaciones:

MATLAB

MATLAB es una herramienta ampliamente utilizada por los miembros del grupo en las fases iniciales del diseño de nuevos algoritmos. Se ha instalado la versión 6.5 junto a todos los "toolbox" necesarios (wavelets, análisis de señal, ...).

MPE

MPE es un conjunto de herramientas para el análisis de prestaciones de programas MPI. Ofrece librerías para la generación de trazas y logs de la ejecución de programas MPI y herramientas para su posterior visualización y análisis.

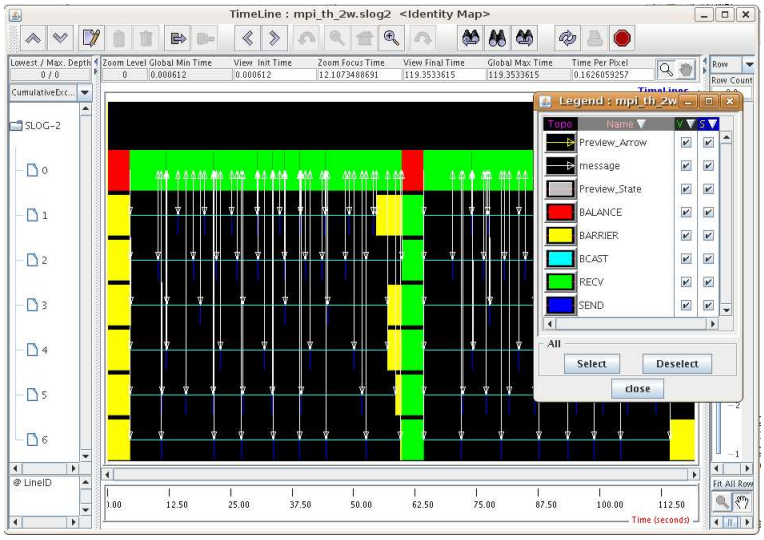

Figura 2.4: Ejemplo de traza generada con MPE

Totalview

Se trata de uno de los depuradores paralelos más potentes que existen en la actualidad. Permite depurar programas paralelos en memoria compartida y distribuida.

Sun Grid Engine

Es un sistema de gestión de colas y planificación de procesos. Viene integrado en NPACI Rocks.

2.5. Configuración

A continuación se presentan algunos de los aspectos más relevantes de la instalación de NPACI Rocks y del resto del software para facilitar su reproducción.

2.5.1. Red

Para poder implementar el esquema de red de la Figura 2.1 es necesario que asignemos un par de direcciones IP (con sus "hostnames" asociados) a cada nodo de cálculo y tres al frontend (las dos anteriores más una que lo identifique en la red departamental). La Tabla 2.2 muestra el esquema de direcciones generado

	Red Gigabit Ethernet		Red Fast Ethernet		Red Departamento	
Nodo	Dirección	Nombre	Dirección	Nombre	Dirección	Nombre
1	192.168.0.254	rosebudFE	10.1.1.1	rosebud	158.42.185.38	rosebud.dsic.upv.es
2	192.168.0.1	rosebud01	10.1.1.254	compute-0-0	–	–
3	192.168.0.2	rosebud02	10.1.1.253	compute-0-1	–	–
4	192.168.0.3	rosebud03	10.1.1.252	compute-1-0	–	–
5	192.168.0.4	rosebud04	10.1.1.251	compute-1-1	–	–
6	192.168.0.5	rosebud05	10.1.1.250	compute-2-0	–	–
7	192.168.0.6	rosebud06	10.1.1.249	compute-2-1	–	–

Tabla 2.2: Esquema de direcciones IP del clúster

2.5.2. Creación de usuarios

Existen ciertas tareas administrativas, que aunque se lleven a cabo en el frontend, es necesario que sus resultados se transfieran a todo el clúster. Rocks ofrece herramientas para esta sincronización. Por ejemplo, en el caso de la creación de usuarios se debe usar la herramienta *rocks-user-sync*. Para la creación de un nuevo usuario en el clúster se procedería de la siguiente forma:

```
useradd <nombre_usuario>
passwd <nombre_usuario>
rocks-user-sync
```

2.5.3. Instalación de software para múltiples arquitecturas

Como ya hemos citado en varias ocasiones, el clúster está formado por nodos con arquitectura x86 y ia64. Puesto que no existe compatibilidad binaria entre ambas, es necesario instalar dos versiones de cualquier aplicación o biblioteca: una de 32 y otra de 64 bits.

Dependiendo del tipo de software y su forma de instalación se deben proceder de una forma distinta. Nos hemos encontrado con los siguientes escenarios:

Software instalado automáticamente por Rocks (SGE, GANGLIA, ...): el toolkit se encarga de instalar en cada nodo la versión correspondiente a la arquitectura.

Software suministrado en RPMs (octave, vim-X11, ...): Habrá que conseguir los RPM para ambas arquitecturas y colocarlos en el repositorio que Rocks habilita para ello. En [6] se describe el proceso con detalle.

Compiladores de Intel: El programa de instalación detecta automáticamente la arquitectura de la máquina y instala la versión del compilador correspondiente. Es necesario realizar dos instalaciones; una desde el frontend sobre el directorio `/share/apps/intel/{cc,fc}/<version>/i386` y otra desde rosebud05 sobre el directorio `/share/apps/intel/{cc,fc}/<version>/ia64`. Nótese que el directorio `/share` se encuentra compartido por NFS y por tanto ambas instalaciones serán visibles desde todos los nodos.

En cada nodo será necesario hacer que la variable de sistema `$PATH` apunte al directorio correspondiente a la arquitectura. Esto también se puede automatizar mediante el comando `uname -i` que devuelve la arquitectura de la máquina. Así la inclusión de la siguiente linea en los ficheros de configuración de cualquier nodo haría que se seleccionara automáticamente el compilador correspondiente a la arquitectura.

```
export PATH=$PATH:/share/apps/intel/cc/9.1.046/‘uname -i‘/bin
export PATH=$PATH:/share/apps/intel/fc/9.1.041/‘uname -i‘/bin
```

Biblioteca Intel Cluster MKL: Al instalarla en el frontend (sobre el directorio `/share/apps/intel/cmkl`), se instalan las versiones para x86, x86_64 y ia64. El directorio de instalación está exportado por NFS, por lo que a la hora de compilar el usuario podrá seleccionar la versión necesaria. Análogamente se pueden crear reglas de compilación genéricas que se

usen desde cualquier nodo. Por ejemplo, para enlazar con los núcleos de BLAS se puede usar la siguiente linea desde cualquier arquitectura:

```
-L/share/apps/intel/cmkl/9.0/lib/‘uname -i‘/ -lmkl -lguide -lpthread
```

Software compilado desde los fuentes (MPICH, ARPACK, ...): La instalación de este tipo de software suele basarse en tres pasos: configuración, compilación e instalación. Habrá que proceder de la misma forma que para instalar los compiladores de Intel. En primer lugar y desde el frontend; configurar, compilar e instalar en el directorio `/share/apps/<nombre_aplicacion>/i386` (exportado por NFS) y en segundo lugar repetir el proceso desde una máquina ia64 instalando sobre el directorio `/share/apps/<nombre_aplicacion>/ia64`.

De forma análoga a las anteriores, usaremos el comando `uname -i` para crear ficheros de configuración genéricos que detecten automáticamente la arquitectura sobre la que se ejecutan. Por ejemplo

```
export PATH=$PATH:/share/apps/<nombre_aplicacion>/bin/‘uname -i‘
export LD_LIBRARY=$LD_LIBRARY:/share/apps/<nombre_aplicacion>/bin/‘uname
-i‘
```

2.5.4. Sistema de colas

Durante las fases de evaluación y optimización de los algoritmos paralelos a desarrollar en el clúster, es necesario obtener el uso exclusivo de los nodos sobre los que se ejecutan. Así, cuando existen varios usuarios simultáneos es necesario coordinarlos para que los experimentos de unos no interfieran en los de otros, generando resultados erróneos a veces difíciles de detectar. Para tal tarea, existe software específico que coordina la ejecución en entornos multiusuario, estos son los sistemas de gestión de colas.

La aproximación clásica de los sistemas de gestión de colas en entornos paralelos homogéneos es que el usuario pida un número concreto de nodos/procesadores donde ejecutar sus aplicaciones, sin importarle que nodos se le asignan en concreto o en que orden. De esta forma el sistema puede gestionar mucho mejor los recursos y aumentar la productividad global.

Esta aproximación no es válida en entornos heterogéneos, donde muchas veces *sí* importa que nodos se asignan o en que orden. Por tanto la idea ha sido implementar un sistema que funcione más como reserva de recursos que como planificación de tareas. Así, si un usuario quiere realizar

una ejecución en la que sitúe dos procesos MPI en los nodos rosebud03 y rosebud04 y cuatro en los nodos rosebud05 y rosebud06, deberá realizar una reserva completa de los cuatro nodos y encargarse él mismo de ubicar los procesos[7], en lugar de pedir 20 procesadores cualquiera, ya que le importa el orden concreto y un sistema de gestión de colas clásico podría no garantizárselo.

En la Sección 2.6 se muestra un ejemplo de ejecución. La Tabla 2.3 lista las colas que se han creado y que nodos reservan.

Id. cola	rosebud01	rosebud02	rosebud03	rosebud04	rosebud05	rosebud06
all.q	√	√	√	√	√	√
i386.q	√	√	√	√		
ia64.q					√	√
p4s.q	√	√				
xeons.q			√	√		
itaniums.q					√	√

Tabla 2.3: Colas disponibles

2.6. Ejemplo de uso

A continuación se muestra un ejemplo de como compilar y ejecutar un sencillo programa MPI sobre todo el clúster (con y sin sistema de colas)

En primer lugar, sea `cpi.c` un programa MPI correcto sin ninguna dependencia de librerías externas. Como ya se ha citado anteriormente, vamos a necesitar dos compilaciones del fuente (una para cada arquitectura). Desde el frontend ejecutamos:

```
mpicc -o cpi.i386 cpi.c       (compilar en 32 bits)
ssh rosebud05                 (Para conectarse al nodo ia64)
mpicc -o cpi.ia64 cpi.c       (compilar en 64 bits)
exit                          (para volver al frontend)
```

Para ejecutar el programa anterior en todo el clúster (en los 22 cores) se puede invocar una orden como la siguiente:

[7] MPICH ofrece las herramientas necesarias

```
mpirun -nolocal -arch i386 -np 6 -machinefile machine.i386 -arch ia64 -np
16 -machinefile machine.ia64 ./cpi.%a
```

Es importante incluir el parámetro `-nolocal` para que `mpirun` no sitúe el primer proceso en el frontend.

El fichero `machines.i386` tiene el siguiente contenido:

```
rosebud01
rosebud02
rosebud03:2
rosebud04:2
```

El fichero `machines.ia64` tiene el siguiente contenido:

```
rosebud05:8
rosebud06:8
```

Para ejecutar a través del sistema de colas se debe crear el fichero `cpi.sh` con el siguiente contenido:

```
#!/bin/bash
#
# use bash as default shell
#$ -S /bin/bash
#
# output to the current directory
#$ -cwd
#
# Set the Parallel Environment and number of procs.
#$ -pe mpich 22
#
# Select the queue:
# p4s.q(2), xeons.q(4), itaniums.q(16)
# i386.q(6), ia64.q(16)
```

```
# all.q(22)
#$ -q all.q

mpirun -arch i386 -np 6 -machinefile machine.i386 -arch ia64 -np 16 -machinefile
machine.ia64 ./cpi.%a
```

Obsérvese que se han seleccionado los 22 procesadores de la cola `all.q` y que después mediante nuestros ficheros de máquinas (`machine.i386` y `machine.ia64`) nos encargamos de situar cada proceso donde corresponda. Existen otras colas más específicas por si sólo se quiere trabajar con un subconjunto de la máquina (e.g. `i386.q` agrupa todas las máquinas de 32 bits o `xeons.q` los biprocesadores Xeon). La Tabla 2.3 proporciona la descripción de las distintas colas.

Para enviar el trabajo a la cola, se usa el comando `qsub` seguido del nombre del script:

```
qsub cpi.sh
```

El sistema de colas devolverá un identificador para el trabajo que nos permitirá identificarlo cuando consultemos el estado de las colas (mediante el comando `qstat`) y que identificará la salida del mismo (`cpi.sh.oXX` para la salida estándar y `cpi.sh.eXX` para la salida de error, siendo `XX` dicho identificador).

Véase [5] para más información sobre el sistema gestor de colas y la sintaxis de los distintos comandos.

Capítulo 3

Cálculo de Valores Propios de Matrices Estructuradas. Introducción

Informalmente, diremos que una matriz $A \in \mathbb{R}^{n\times n}$ es estructurada si sus valores siguen una distribución "espacial" que nos permite realizar una representación de la misma sin necesidad de almacenar los n^2 elementos y sin perder información. Dentro de esta definición podemos incluir tanto las clásicas matrices simétricas, banda, Toeplitz, ... como otras que aparezcan en la resolución de problemas numéricos como las presentadas en la Sección 5.

La principal ventaja al trabajar con matrices estructuradas es que existen formatos de representación que nos permiten reducir la cantidad de memoria necesaria para su almacenamiento en la memoria de un ordenador. Gracias a esto podremos representar y abordar problemas mucho mayores que en el caso de las matrices "densas". A esto hay que añadir la posibilidad de diseñar nuevos algoritmos para la resolución de la mayoría de problemas del álgebra lineal numérica que aprovechen la estructura de la matrices para reducir el coste temporal.

Un problema estándar de valores propios consiste en dada una matriz $A \in \mathbb{C}^{n\times n}$ resolver la ecuación:

$$Ax = \lambda x \tag{3.1}$$

con $x \neq 0 \in \mathbb{C}^n, \lambda \in \mathbb{C}$. La selección del algoritmo a aplicar depende en gran medida de las características del problema: si la matriz A es real y simétrica, todos los valores propios

son reales, mientras que si es no simétrica los valores propios pueden ser complejos. También es posible que solamente se necesite calcular unos pocos valores propios: el de mayor o menor magnitud, los contenidos en cierto intervalo, ... En este caso los métodos iterativos tipo Lanczos, Arnoldi o Jacobi-Davidson son los más apropiados.

Si es necesario calcularlos todos o la gran mayoría, la solución pasa por tridiagonalizar A mediante transformaciones de Householder y aplicar alguno de los algoritmos para matrices tridiagonales: iterativo QR, bisección, divide-y-vencerás, RRR, ... Ésta es la aproximación implementada en LAPACK y ScaLAPACK. Sin embargo, para problemas muy grandes, los requerimientos de memoria pueden hacerla inabordable. Además, resulta muy difícil aprovechar la estructura de la matriz, ya que la tridiagonalización elimina su estructura.

Un problema generalizado de valores propios se define como, dadas $A, B \in \mathbb{C}^{n\times n}$, hallar la solución de la ecuación:

$$Ax = \lambda Bx \tag{3.2}$$

con $x \neq 0 \in \mathbb{C}^n, \lambda \in \mathbb{C}$. El problema de selección de un algoritmo es análogo al anterior. Para el cálculo del espectro completo, el algoritmo más habitual es el QZ. Una vez más, este algoritmo requiere la reducción de A a forma Hessenberg superior y B a forma triangular superior, por lo que no es apropiado para problemas estructurados ya que se pierde la estructura de las matrices.

Otro algoritmo útil cuando se quieren calcular todos los valores propios (siempre que B sea simétrica y definida positiva) se basa en calcular la factorización de Cholesky de la matriz B ($B = ZZ^H$) y transformar el problema generalizado en uno estándar:

$$Ax = \lambda Bx \Longrightarrow Z^{-1}AZ^{-H}Z^Hx = \lambda Z^Hx \Longrightarrow Cy = \lambda y \tag{3.3}$$

con $C = Z^{-1}AZ^{-H}$ y $y = Z^Hx$. Será necesario estudiar para cada caso concreto si el problema resultante mantiene la estructura y que método de los anteriores aplicar.

Finalmente, también existen versiones de los algoritmos iterativos citados anteriormente para el problema generalizado, útiles cuando se quieren calcular unos pocos valores propios. La principal ventaja de estos métodos es que la única operación que requieren hacer sobre las matrices es la solución de sistemas y el producto matriz-vector, con lo que se pueden desarrollar núcleos que aprovechen la estructura de la matriz para reducir el tiempo de ejecución.

En los siguientes tres capítulos se muestran los resultados de varios trabajos realizados en este campo, citando las publicaciones generadas.

Capítulo 4

Cálculo de VPs de matrices tridiagonales simétricas: aproximación heterogénea

4.1. Introducción

El cálculo de valores propios de matrices tridiagonales simétricas es un problema de gran relevancia en el álgebra lineal numérica. Este tipo de matrices aparecen en la discretización de ciertos problemas de ingeniería y además se trata del núcleo computacional central en la resolución de cualquier problema de valores propios simétrico.

Las principales bibliotecas de álgebra lineal numérica paralela, como ScaLAPACK, implementan los algoritmos descritos anteriormente, aunque orientados a ejecutarse en multiprocesadores de memoria distribuida homogéneos. Este tipo de implementaciones, presentan resultados pobres en entornos paralelos heterogéneos, ya que muchas veces el rendimiento global del sistema se ve acotado por el del procesador menos potente. Por tanto es necesario desarrollar núcleos computacionales optimizados para este tipo de arquitecturas con el objetivo de llegar a una versión heterogénea de ScaLAPACK.

4.2. Descripción del problema y solución propuesta

Sea $T \in \mathbb{R}^{n\times n}$ una matriz tridiagonal simétrica

$$T = \begin{bmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_1 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{bmatrix} \tag{4.1}$$

y $\lambda(T)$ el conjunto de todos los valores propios de T, también conocido como espectro de T.

Dicho espectro puede representarse como un conjunto puntos sobre la recta $\mathbb{R}$ (por ser T simétrica). Es posible aislar estos puntos (valores propios de T) mediante la factorización LDL^T y el teorema de la inercia de Sylvester. Si

$$T - \mu I = LDL^T, \quad T = T^T \in \mathbb{R}^{n\times n} \tag{4.2}$$

es la factorización LDL^T de $T-\mu I$ con $D = diag(d_1, \ldots, d_n)$, entonces la cantidad de d_i negativos es equivalente al número de valores propios menores que μ [21].

La secuencia $(d_1, \ldots, d_n)$ se puede calcular mediante la siguiente recurrencia, siendo $d_i = q_i(c), i = 1, \ldots, n$ para cualquier $c \in \mathbb{R}$:

$$\begin{cases} q_0(c) = 1, \quad q_1(c) = a_1 - c \\ q_i(c) = (a_i - c) - \frac{b_{i-1}^2}{q_{i-1}(c)} \quad i : 2, 3, \ldots, n \end{cases} \tag{4.3}$$

Gracias a este resultado es posible definir la función

$$neg_n(T, c) : \mathbb{R}^{n\times n} \times \mathbb{R} \to \mathbb{N} \tag{4.4}$$

que calcula el número de valores propios de T menores que c. A partir de esta función se ha diseñado el Algoritmo 1 capaz de dividir el espectro de T en m subintervalos $]a_i, b_i]$ que contengan un número de valores propios $v \leq max_{val}$ (Figura 4.1).

Dicho algoritmo necesita como entrada un intervalo $[a, b]$ que contenga todos los valores propios de la matriz T para iniciar la bisección, éste puede calcularse mediante el teorema de los círculos de Gershgorin.

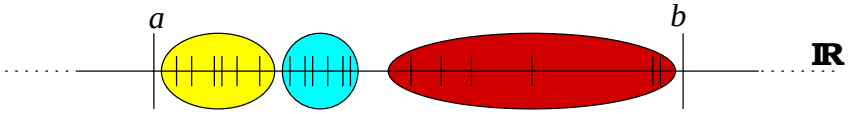

Figura 4.1: Particionado del espectro

Algoritmo 1 Algoritmo de tipo bisección para el particionado del espectro de una matriz.

ENTRADA: matriz T, intervalo $[a,b]$ que contiene todos los valores propios de T y número máximo de valores propios por subintervalo max_{val}

SALIDA: conjunto de subintervalos $[a_i,b_i]$ y número de valores propios que contienen, $[a,b] = \bigcup_{i=1}^{m}[a_i,b_i]$ y $[a_i,b_i] \cap [a_j,b_j] = \emptyset, \forall i,j : i \neq j$

Sea n el tamaño del problema

1. *left = $neg_n(T,a)$*
2. *in = $neg_n(T,b)$ - left*
3. *right = n - left - in*
4. *active = {<[a,b], left, in, right>}*
5. *processed = $\emptyset$*

6. *while active $\neq \emptyset$ do*
7. *active $\leftarrow$ active - <[x,y], l, i, r>*
8. *shift = (y-x)/2*
9. *u = $neg_n(T, shift)$*
10. *e1 = u - l*
11. *e2 = r - u*

12. *if (e1 > 0) then*
13. *if (e1 > max_{val}) then*
14. *active $\leftarrow$ active $\cup$ {<[x,shift], l, e1, r+e2>}*
15. *else*
16. *processed $\leftarrow$ processed $\cup$ {<[x,shift], e1>}*

17. *end if*

18. *end if*

19. *if (e2 > 0) then*

20. *if (e2 > max_{val}) then*

21. *active* $\leftarrow$ *active* $\cup$ {*<[shift,y], l+e1, e2, r>*}

22. *else*

23. *processed* $\leftarrow$ *processed* $\cup$ {*<[shift,y], e2>*}

24. *end if*

25. *end if*

26. end do

27. return processed

Obsérvese que el parámetro max_{val} determina el número máximo de valores propios que contendrá cada $]a_i, b_i]$, si ajustamos su valor a 1 conseguimos definir un definir intervalos que contengan un único valor propio que podrá ser fácilmente extraído mediante métodos de cálculo de raíces. Si lo hacemos a un número mayor conseguiremos dividir el problema original en m problemas independientes (descartando además partes de la recta real que no contengan valores propios). Deberá diseñarse alguna técnica específica para procesar dichos subproblemas.

La segunda fase de la solución recibe como entrada los m subintervalos $]a_i, b_i]$ y extrae los valores propios contenidos en ellos. Existen varias alternativas para su extracción:

1. Volver a aplicar el Algoritmo 1 para extraerlos por bisección.

2. Usar métodos de convergencia rápida como Newton o Laguerre.

3. Usar los núcleos que proporciona LAPACK para la resolución secuencial del problema.

Obsérvese que esta segunda fase es potencialmente paralelizable dado que no hay dependencias de datos entre los diferentes subproblemas.

4.2.1. Matrices de prueba

El rendimiento de los algoritmos de bisección depende en gran medida de como se distribuyan los valores propios a lo largo del espectro. Además, la presencia de clústers de valores propios o valores propios ocultos incrementa considerablemente el tiempo de extracción [9].

Para realizar un correcto análisis experimental de los algoritmos implementados se debe elegir un conjunto de matrices de prueba apropiado. En nuestro caso hemos elegido dos tipos de matrices (Tabla 4.1) que presentan distintas características de distribución de los valores propios.

Tipo	Elementos	Valores propios
Matrices de espectro uniforme	$a_i = 0$ $b_i = \sqrt{i(n-i)}$	$\{-n+2k-1\}_{k=1}^{n}$ Distribuidos uniformemente
Matrices de Wilkinson	$a_i = \begin{cases} \frac{m}{2} - i + 1 & i : 1, \ldots, \frac{m}{2} \\ i - \frac{m}{2} & i : \frac{m}{2}+1, \ldots, m \end{cases}$ $m = \begin{cases} n & con\ n\ par \\ n+1 & con\ n\ impar \end{cases}$	La mayoría de valores propios agrupados en clústers de dos.

Tabla 4.1: Matrices de prueba

4.3. Modelo computacional heterogéneo

Un modelo computacional de una máquina paralela es una abstracción matemática de su funcionamiento que oculta los detalles arquitecturales al programador. Debe ser suficientemente detallado como para reflejar los aspectos que influyen en el rendimiento del software, aunque lo suficientemente abstracto como para ser independiente de la máquina. Vamos a usar un modelo que evalúa el rendimiento de cada procesador y de la red que los une.

Sea un conjunto de p procesadores interconectados por una red. En primer lugar, se define un vector P_t que representa la potencia relativa de un procesador con respecto del global de la máquina. Esta potencia relativa depende del problema a resolver y de su talla. En segundo lugar, el modelo de comunicaciones usado define el tiempo necesario para enviar n bytes del procesador i al j como

$$T_{ij}(n) = \beta + n\tau \tag{4.5}$$

, donde β representa la latencia de la red y τ la inversa del ancho de banda. Para resumir el modelo se definen dos matrices B y T, el valor de la posición (i, j) de cada una de ellas representa el β y el τ aplicable a las comunicaciones entre i y j.

Para evaluar el modelo y ejecutar los algoritmos presentados se han usado los nodos 1, 3, 4 y 5 del clúster descrito en el Capítulo 2. Las tablas 4.2, 4.3, 4.4 y 4.5 muestran el resultado de evaluar esta máquina con el modelo descrito anteriormente.

	nodo 1	nodo 4		nodo 5		nodo 3
n	P0	P1	P2	P3	P4	P5
5000	0.2245	0.1740	0.1692	0.1605	0.1598	0.1120
6000	0.2265	0.1735	0.1706	0.1591	0.1591	0.1112
7000	0.2264	0.1748	0.1688	0.1586	0.1600	0.1113
8000	0.2245	0.1749	0.1701	0.1590	0.1600	0.1116
9000	0.2269	0.1735	0.1700	0.1592	0.1590	0.1115
10000	0.2264	0.1739	0.1703	0.1588	0.1596	0.1110
11000	0.2265	0.1749	0.1690	0.1577	0.1604	0.1115
12000	0.2268	0.1739	0.1711	0.1578	0.1595	0.1108
13000	0.2265	0.1743	0.1700	0.1586	0.1597	0.1108
14000	0.2270	0.1747	0.1700	0.1588	0.1593	0.1101
15000	0.2266	0.1737	0.1708	0.1598	0.1599	0.1093

Tabla 4.2: Vector P_t para el cálculo de los valores propios de matrices de espectro uniforme

4.4. Esquemas paralelos heterogéneos

El problema del equilibrado de la carga ha sido ampliamente estudiado en computación paralela homogénea. Sin embargo, en entornos heterogéneos éste se complica aún más, ya que no basta repartir datos y cálculo a partes iguales entre los procesadores disponibles, sino que habrá que tener en cuenta factores como la potencia de cálculo, la memoria disponible o la red. La propia evaluación de estos parámetros puede ser una tarea complicada, añadiendo complejidad al proceso.

Existen diversas aproximaciones a la hora de realizar esta distribución, dando lugar a múltiples

n	P0	P1	P2	P3	P4	P5
5000	0.2219	0.1747	0.1681	0.1635	0.1607	0.1113
6000	0.2177	0.1762	0.1687	0.1638	0.1623	0.1131
7000	0.2193	0.1752	0.1687	0.1648	0.1612	0.1110
8000	0.2145	0.1756	0.1692	0.1647	0.1642	0.1117
9000	0.2213	0.1745	0.1681	0.1632	0.1622	0.1107
10000	0.2232	0.1748	0.1678	0.1632	0.1611	0.1099
11000	0.2231	0.1751	0.1661	0.1640	0.1614	0.1103
12000	0.2203	0.1759	0.1680	0.1636	0.1620	0.1101
13000	0.2239	0.1748	0.1675	0.1640	0.1608	0.1091
14000	0.2263	0.1744	0.1664	0.1639	0.1601	0.1089
15000	0.2237	0.1735	0.1662	0.1654	0.1611	0.1101

Tabla 4.3: Vector P_t para el cálculo de los valores propios de matrices de Wilkinson

	P0	P1	P2	P3	P4	P5
P0	0	2.5074E-05	4.6149E-05	4.5745E-05	4.0809E-05	5.1629E-05
P1	2.4774E-05	0	4.6042E-05	4.5852E-05	4.0348E-05	5.317E-05
P2	4.6268E-05	4.6073E-05	0	2.4727E-05	4.2951E-05	5.2753E-05
P3	4.564E-05	4.5583E-05	2.4898E-05	0	4.1091E-05	5.2756E-05
P4	4.0829E-05	4.0706E-05	4.1424E-05	4.0211E-05	0	7.8713E-05
P5	5.0371E-05	4.9166E-05	5.1836E-05	4.6695E-05	1.0234E-4	0

Tabla 4.4: Matriz de latencias B_c (s)

algoritmos paralelos. Nosotros hemos contemplado las siguientes.

- Ignorar la heterogeneidad del sistema y realizar una distribución de la carga equitativa.
- Basándose en un modelo computacional como el anterior realizar una distribución de la carga proporcional. En este caso bastará dividir el espectro en p (número de procesadores) subconjuntos con un número de valores propios proporcional a la potencia del procesador correspondiente.
- Implementar un algoritmo de distribución dinámica de la carga de tipo "maestro-esclavo". En este caso nos valdremos del Algoritmo 1 para particionar el problema original en $m \gg p$

	P0	P1	P2	P3	P4	P5
P0	0	6.4244E-09	1.4045E-08	1.1898E-08	2.1316E-08	5.5421E-08
P1	6.3307E-09	0	1.2879E-08	1.3066E-08	2.1954E-08	5.5108E-08
P2	1.2966E-08	1.3761E-08	0	6.3347E-09	2.0886E-08	5.2581E-08
P3	1.2548E-08	1.3327E-08	6.3294E-09	0	2.0014E-08	5.4863E-08
P4	2.1369E-08	2.1076E-08	2.269E-08	2.0776E-08	0	1.0859E-07
P5	2.1369E-08	5.4601E-08	5.6684E-08	5.1962E-08	1.0991E-07	0

Tabla 4.5: Matriz del inverso del ancho de banda T_c (s)

subproblemas independientes, que se asignaran a los procesadores disponibles bajo demanda.

4.4.1. Algoritmos implementados

Basándonos en el método presentado en la Sección 4.2 y las soluciones para el equilibrado de la carga descritas anteriormente se han implementado un algoritmo secuencial y cinco paralelos:

Algoritmo secuencial. A1: En esta versión se implementa secuencialmente la metodología descrita en la Sección 4.2.

Algoritmo ScaLAPACK. A2: En esta versión se calculan los valores propios de la matriz T mediante una llamada a la subrutina *pdstebz*, que asigna un número idéntico de valores propios a cada procesador y los extrae por bisección.

Algoritmo estático. A3: En esta versión asignamos estáticamente al procesador $i = 0, \ldots, p-1$ el cálculo de los valores propios del $i\frac{n}{p}+1$ al $(i+1)\frac{n}{p}$ según una ordenación ascendente de los mismos. Este algoritmo puede implementarse con p llamadas concurrentes a la rutina de LAPACK *dstevr*, que recibe como entrada dos enteros que definen el conjunto de valores propios a calcular.

Algoritmo estático proporcional. A4: La estrategia implementada es análoga a la anterior pero el número de valores propios asignados a cada procesador depende de su potencia relativa.

Algoritmo dinámico. A5: La primera etapa de este algoritmo consiste en dividir el intervalo $[a, b]$ que contiene todos los valores propios en p subintervalos de longitud $\frac{b-a}{p}$. Cada uno de los subintervalos es asignado a un procesador que aplica el Algoritmo 1. Se ha ajustado el parámetro max_{val} a 1, por tanto el número de subproblemas generados m es igual a la talla del problema n. Los resultados de la fase de aislamiento se reúnen en el proceso maestro. Finalmente, la etapa de extracción se ha implementado mediante el algoritmo "maestro-esclavo" descrito anteriormente y llamadas a la función *dstevr* de LAPACK.

Algoritmo dinámico modificado. A6: Esta versión es similar a la anterior pero la condición $m \gg p$ ha sido relajada. En lugar de $max_{val} = 1$ se han elegido valores entre 1 y 100. Se ha observado que el valor óptimo se encuentra en 20. Con esta modificación pretendemos evitar que el algoritmo de bisección tenga que procesar clústers de valores propios, situación que tiende a degradar sus prestaciones.

4.5. Análisis experimental

Las Tablas 4.6, 4.7 y 4.8 y las Figuras 4.2 y 4.3 muestran los tiempos de ejecución de los 6 algoritmos presentados para distintas tallas. Para ambos tipos de matrices, los algoritmos estático proporcional (A4) y dinámico modificado (A6) presentan los tiempos de ejecución menores, seguidos de ScaLAPACK (A2) y dinámico modificado (A5) que presentan resultados similares. Finalmente, el algoritmo estático (A3) ofrece las prestaciones más bajas de los estudiados.

Las Tablas 4.9 y 4.10 muestran la ganancia de tiempo, en términos de speed-up, de las dos mejores versiones paralelas: A4 y A6, respecto de la resolución mediante ScaLAPACK (algoritmo A2). Se observa que ambos algoritmos presentan ganancias similares. Aunque se observan mayores ganancias en el caso de las matrices Wilkinson.

4.6. Conclusiones

En el presente trabajo un algoritmo secuencial y 5 paralelos han sido presentados para la extracción de los valores propios de una matriz tridiagonal. Tres de ellos han sido específicamente diseñados para su ejecución en multicomputadores paralelos heterogéneos.

En la parametrización del clúster se ha observado que el vector de potencias P_t sufre poca va-

n	Uniforme	Wilkinson
5000	20.14	8.63
6000	28.76	12.20
7000	38.80	16.50
8000	50.54	21.57
9000	63.35	27.07
10000	78.03	33.15
11000	94.13	39.94
12000	113.96	47.58
13000	130.69	55.38
14000	151.30	64.67
15000	172.80	74.31

Tabla 4.6: Tiempo de ejecución, en P0, del algoritmo secuencial (A1) para ambos tipos de matrices

n	A2	A3	A4	A5	A6
5000	5.15	6.39	4.44	5.21	4.43
6000	7.35	9.12	6.36	7.26	6.31
7000	9.98	12.41	8.61	9.78	8.54
8000	13.02	16.09	11.06	12.64	11.13
9000	16.29	20.22	13.94	15.88	13.95
10000	20.26	24.97	17.10	19.56	17.14
11000	24.45	30.15	20.66	23.68	20.78
12000	28.85	35.88	24.53	28.07	24.58
13000	33.85	42.12	28.76	33.00	28.68
14000	39.74	48.94	33.09	38.57	33.24
15000	45.67	56.50	37.96	43.72	38.04

Tabla 4.7: Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de espectro uniforme

riación para todos los tamaños de problema resueltos. Esta escasa variabilidad contradice la hipótesis de que existe un vector P_t para cada tamaño de problema. Creemos que se trata de una circunstancia propia de este problema en concreto y no generalizable. Debida al hecho de

n	A2	A3	A4	A5	A6
5000	2.48	2.98	2.03	3.48	2.08
6000	3.49	4.25	2.85	4.95	3.06
7000	4.70	5.68	3.86	6.72	3.92
8000	6.20	7.38	5.02	8.77	5.09
9000	7.68	9.36	6.33	11.15	6.35
10000	9.48	11.54	7.77	14.16	7.87
11000	11.45	13.96	9.38	17.42	9.45
12000	13.64	16.61	11.14	20.88	11.20
13000	15.95	19.63	13.04	24.58	13.15
14000	18.53	22.49	15.11	28.58	15.12
15000	21.21	25.99	17.39	32.85	17.35

Tabla 4.8: Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de Wilkinson

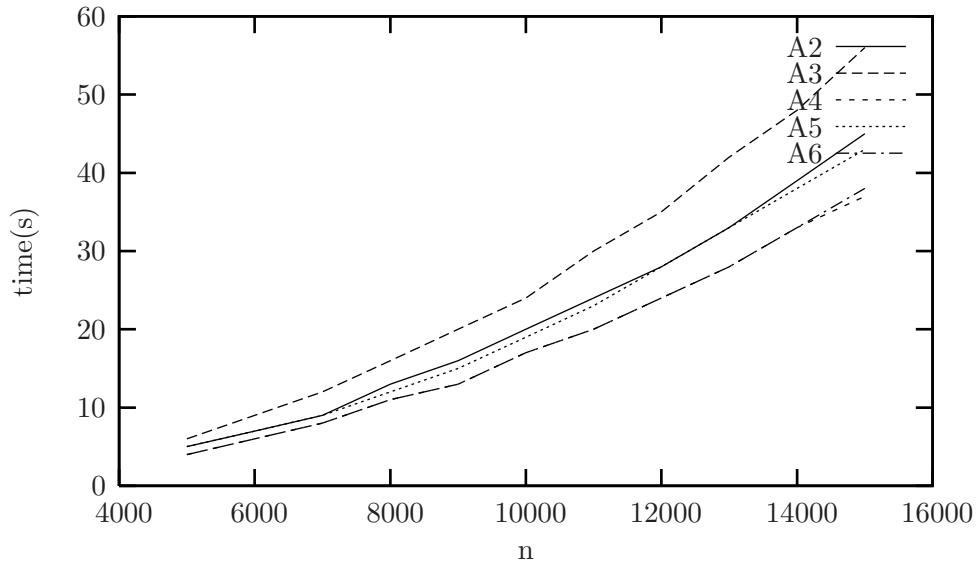

Figura 4.2: Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de espectro uniforme

almacenar las matrices en formato comprimido, circunstancia que repercute en que la complejidad espacial del algoritmo sea lineal y por tanto se atenúe el impacto de las diferencias en la jerarquía de memoria de los distintos procesadores.

Se ha observado que los algoritmos que tienen en cuenta la heterogeneidad del entorno a la

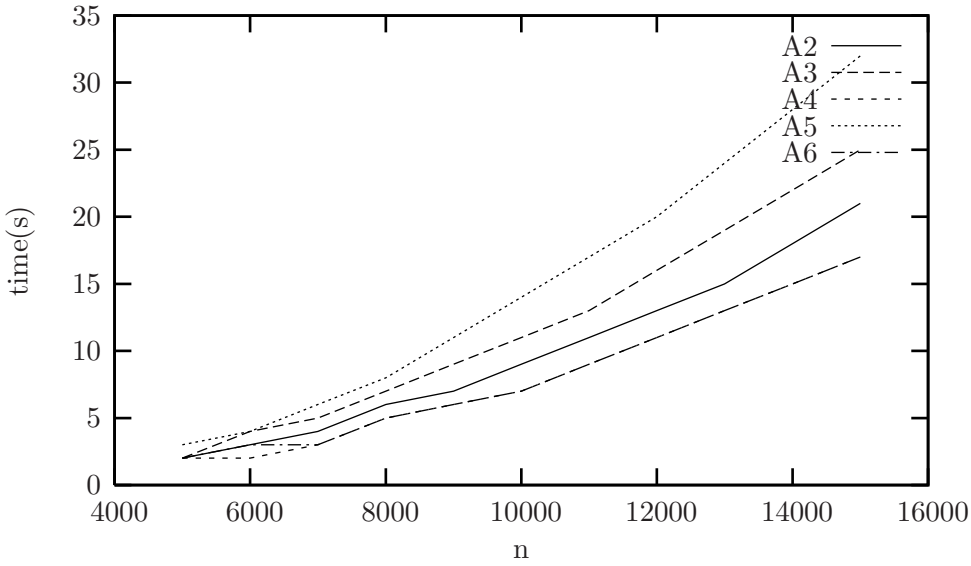

Figura 4.3: Tiempo de ejecución de los 5 algoritmos paralelos sobre matrices de Wilkinson

n	5000	7000	9000	11000	13000	15000
A4	1.16	1.16	1.17	1.18	1.18	1.2
A6	1.16	1.17	1.17	1.18	1.18	1.2

Tabla 4.9: Speed-up del algoritmo A4 y A6 respecto del algoritmo A2 (ScaLAPACK) sobre matrices de espectro uniforme

n	5000	7000	9000	11000	13000	15000
A4	1.22	1.22	1.21	1.22	1.22	1.22
A6	1.2	1.2	1.21	1.21	1.21	1.22

Tabla 4.10: Speed-up del algoritmo A4 y A6 respecto del algoritmo A2 (ScaLAPACK) sobre matrices de Wilkinson

hora de distribuir la carga computacional (A4, A5 y A6) siempre obtienen tiempos de ejecución menores que no la tienen en cuenta(A2 y A3). Este hecho justifica la necesidad de implementar técnicas de equilibrado de carga específicas para arquitecturas heterogéneas.

Se ha observado que el tiempo de ejecución del algoritmo Dinámico (A5) es siempre superior al del algoritmo Dinámico Modificado (A6). Esto es debido al esfuerzo extra que se realiza en la etapa de aislamiento. Además, la existencia de clústers de valores propios (matrices de

Wilkinson) repercute negativamente en el número de pasos de bisección necesarios para aislar individualmente los valores propios en dicha etapa.

El hecho de que los algoritmos Estático Proporcional (A4) y Dinámico Modificado (A6) presenten tiempos de ejecución casi idénticos, valida ambas metodologías de equilibrado de la carga. A pesar de esto, los autores consideran que el esfuerzo necesario para alcanzar un buen equilibrado de la carga en A4 (cálculo del vector de potencias para cada tipo de matriz y tamaño) es muy superior. Por tanto, queda justificada la elección del algoritmo Dinámico Modificado (A6) como el más conveniente en entornos heterogéneos.

Finalmente, el hecho de que los algoritmos A4 y A6 presenten mejores prestaciones que las subrutinas que ofrece ScaLAPACK para la resolución del problema abordado justifica la necesidad del diseño y la implementación de bibliotecas de álgebra lineal para arquitecturas paralelas heterogéneas.

Este trabajo a dado lugar a dos publicaciones: [13, 12]

Capítulo 5

Implementación paralela de un algoritmo tipo Lanczos para un problema de VPs estructurado

5.1. Introducción

En este trabajo nos hemos centrado en la paralelización de un algoritmo de tipo Lanczos para la resolución de un problema generalizado de valores propios, con la peculiaridad que tan solo necesitamos calcular los valores propios comprendidos en un intervalo conocido $[\alpha, \beta]$ (y sus vectores propios asociados).

Dicho problema proviene de una aplicación real de diseño de componentes pasivos de microondas. La descripción y el algoritmo secuencial propuesto se puede encontrar en [20]. No obstante, cuando se simulan componentes muy complejos el tiempo de ejecución es demasiado grande. Un primer algoritmo paralelo fue introducido en [30], usando MPI sobre un clúster homogéneo. Los resultados presentados en dicho artículo muestran que el algoritmo paraleliza extremadamente bien.

Se tiene previsto integrar el código desarrollado en una herramienta CAD para el diseño de componentes electromagnéticos. Dicha herramienta será ejecutada en estaciones de trabajo o, a lo sumo, en pequeños clústers de PCs. Para este tipo de sistemas se debe trabajar con otras

técnicas de paralelización.

El principal objetivo de este trabajo es explorar las distintas alternativas de programación paralela disponibles para la paralelización del algoritmo secuencial descrito en [20], en pequeños clústers y estaciones de trabajo.

Se han estudiado tres aproximaciones: en primer lugar se ha diseñado una versión en OpenMP para su ejecución en pequeñas máquinas de memoria compartida (multi-procesadores, multi-cores, ...). A continuación, hemos implementado una versión para clústers de PCs usando MPI. Finalmente, se ha desarrollado una solución híbrida (MPI+OpenMP) para clústers de multi-procesadores o multi-cores.

Se han introducido diversas mejoras sobre el algoritmo publicado anteriormente con el objetivo de aumentar la robustez del código. Éstas han sido: i) la utilización de la implementación del algoritmo de Lanczos que ofrece ARPACK, ii) la corrección del algoritmo de equilibrado de la carga y iii) la mejora del algoritmo para la resolución de sistemas, originalmente basado en la factorización LU y el complemento de Schur y ahora basado en la factorización LDL^T.

5.2. Descripción del problema

En este trabajo, el cálculo de los modos de distintas guías electromagnéticas se basa en la técnica conocida como BI-RME (Boundary Integral - Resonant Mode Expansion, se puede consultar su formulación en [20]). Esta técnica proporciona las frecuencias de corte modales de una guía electromagnética a partir de la resolución de dos problemas generalizados de valores propios. El primero es el problema que modela la familia de modos TE (Tranversal Electric) de la guía. Las matrices A y B (Figura 5.1) presentan una estructura muy dispersa que es convenientemente aprovechada en este trabajo. Ambas matrices tienen un diagonal principal y un pequeño bloque $n \times n$ en la esquina derecha inferior. Además la matriz B, tiene dos bandas R (con dimensiones $n \times m$) y R_t $(m \times n)$, en las últimas n filas y n columnas respectivamente. Así pues, la matriz es de tamaños $(m+n) \times (m+n)$ con $m \gg n$ y por tanto muy dispersa. El cálculo de la familia de modos TM (Transversal Magnetic) también puede formularse como un problema generalizado de valores propios, con unas matrices muy similares a las del caso TE.

En este trabajo solo vamos a considerar el caso TE.

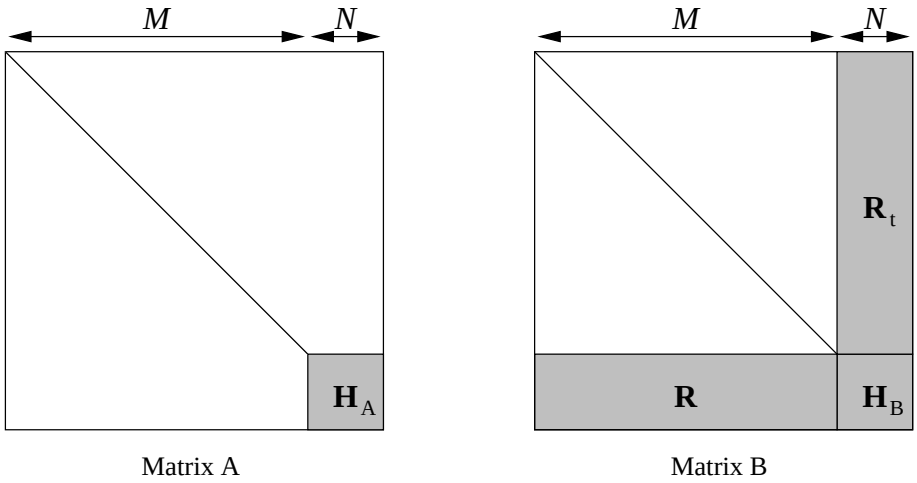

Figura 5.1: Matrices A y B para el problema TE de un guía tipo *ridge*

5.3. Algoritmo secuencial

5.3.1. Algoritmo de Lanczos con Shift-and-Invert

La técnica propuesta en [20] por los autores se basa en el algoritmo de Lanczos [27]. Este algoritmo, en su formulación más básica, permite el cálculo de un número reducido de valores y vectores propios situados a los extremos del espectro (los mayores y menores en magnitud). Sin embargo, dado un número real (conocido como *shift* o desplazamiento) σ, se puede aplicar el algoritmo sobre la matriz $W = (A - \sigma B)^{-1}B$, para obtener los valores propios del problema original más próximos a σ. El uso del método de Lanczos en este problema requiere la resolución de diversos sistemas de ecuaciones lineales, con $A-\sigma B$ como matriz de coeficientes. La estructura de las matrices permite optimizar dicha resolución mediante la técnica del complemento de Schur. Este método, descrito en [20], fue implementado originalmente basándose en la factorización LU, aunque en esta última versión se ha usado la LDL^T, descrita a continuación.

5.3.2. Factorización LDL^T

La factorización LDL^T de la matriz $A - \sigma B$ puede definirse de la siguiente forma:

$$\mathbf{A} - \sigma\mathbf{B} \;=\; \begin{pmatrix} \mathbf{U}_\sigma & \mathbf{R}_{\sigma t} \\ \mathbf{R}_\sigma & \mathbf{H}_\sigma \end{pmatrix} = \begin{pmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{F} & \mathbf{T} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{D}_l & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_s \end{pmatrix} \cdot \begin{pmatrix} \mathbf{D_t} & \mathbf{F_t} \\ \mathbf{0} & \mathbf{T_t} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{D} \cdot \mathbf{D}_l \cdot \mathbf{D_t} & \mathbf{D} \cdot \mathbf{D}_l \cdot \mathbf{F_t} \\ \mathbf{F} \cdot \mathbf{D}_l \cdot \mathbf{D}_t & \mathbf{F} \cdot \mathbf{D}_l \cdot \mathbf{F_t} + \mathbf{T} \cdot \mathbf{D}_s \cdot \mathbf{T_t} \end{pmatrix} \tag{5.1}$$

siendo la estructura de $A - \sigma B$ idéntica a la de la matriz B (Figura 5.1).

Si se toma D como la matriz identidad (ya que U_σ es diagonal) se puede llegar al siguiente procedimiento para el cálculo de la factorización LDL^T:

1. Tomar D_l como U_σ
2. $F = R_\sigma D_l^{-1}$ (trivial, ya que D_l es diagonal)
3. T y D_s se obtienen calculando la factorización LDL^T de $H_\sigma - FD_lF_t$ mediante la rutina *dsytrf* de LAPACK.

5.3.3. Descomposición del intervalo principal

Como hemos comentado anteriormente, la versión *shift-and-invert* del algoritmo de Lanczos calcula un subconjunto del espectro centrado en el *shift*. El número de valores propios a extraer va a determinar tanto el número de iteraciones de Lanczos como su coste espacial [23]. Obviamente, no podemos aplicar Lanczos al intervalo principal $[\alpha, \beta]$. El problema original debe ser dividido en un conjunto de problemas de coste menor para asegurar unas prestaciones globales óptimas.

Para dividir el intervalo principal se va a usar la técnica presentada en la Sección 4.2 basada en el Teorema de la Inercia. Para poder aplicar el Algoritmo 1 debemos definir una nueva función

$$neg_n(A, B, c) : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times n} \times \mathbb{R} \to \mathbb{N} \tag{5.2}$$

Al igual que en el caso anterior se basa en la factorización LDL^T de $A - cB$ y el número de valores propios menores que c es equivalente al número de valores negativos en D. Para este tipo de matrices no se dispone de una recurrencia que simplifique dicho cálculo, por lo que será necesario calcular la factorización explícitamente mediante la técnica descrita anteriormente.

5.3.4. Algoritmo secuencial

El algoritmo secuencial consiste en: dividir el intervalo $[\alpha, \beta]$ en un conjunto de subintervalos de longitud menor, para cada subintervalo, seleccionar un *shift* (posiblemente el punto central) y

aplicar la variante *shift-and-invert* del algoritmo de Lanczos. De esta forma se pueden calcular los valores propios de cada subintervalo de forma independiente del resto. La cantidad de valores propios por subintervalo debe de mantenerse moderada para evitar problemas de memoria.

Algoritmo 2 Algoritmo secuencial completo.

ENTRADA: matrices $\mathbf{A}$ *y* $\mathbf{B}$*, intervalo de búsqueda* $[\alpha, \beta]$ *y número máximo de valores propios por subintervalo* max_{val}

SALIDA: valores propios de $(\mathbf{A}, \mathbf{B})$ *contenidos en* $[\alpha, \beta]$ *y sus vectores propios correspondientes*

1. *Aplicar el Algoritmo 1 para dividir el intervalo principal* $[\alpha, \beta]$ *en subintervalos*

2. *Para cada subintervalo* $[\alpha_i, \beta_i]$
4. $\sigma = (\beta_i - \alpha_i)/2$
5. *Aplicar Lanczos con shift-and-invert para extraer los valores propios alrededor de* σ *y sus vectores propios*
6. *Fin para*

7. *Retornar valores y vectores propios*

En las últimas versiones del código, se ha pasado a utilizar la subrutina *dsaupd* de ARPACK, con el objetivo de mejorar la robustez del código. Esta rutina es más rápida y segura que nuestras versiones previas de Lanczos.

5.4. Implementaciones paralelas

La idea básica para la paralelización del método es distribuir los subintervalos entre los procesadores disponibles; para cada subintervalo, la extracción de los valores propios se realiza de forma secuencial.

Una vez se han calculado las límites de cada subintervalo $[\alpha_i, \beta_i]$, se asignan $\frac{m}{p}$ subintervalos a cada procesador. Esta asignación se realiza al principio del algoritmo y no hay más comunicaciones entre los procesadores hasta que terminen su trabajo y se tengan que reunir los resultados.

Como ya se comentaba anteriormente, el tiempo necesario para resolver cada subintervalo se espera que sea casi constante. Por tanto, dado que vamos a trabajar en un entorno homogéneo, distribuyéndolos de forma equitativa entre los procesadores disponibles se espera que el equilibrio de la carga sea bueno.

Algoritmo 3 Algoritmo paralelo completo.

ENTRADA: matrices $\mathbf{A}$ *y* $\mathbf{B}$*, intervalo de búsqueda* $[\alpha, \beta]$ *y número máximo de valores propios por subintervalo* max_{val}

SALIDA: valores propios de $(\mathbf{A}, \mathbf{B})$ *contenidos en* $[\alpha, \beta]$ *y sus vectores propios correspondientes*

Sea m múltiplo de p

1. *En el procesador maestro*
2. *Aplicar el Algoritmo 1 para dividir el intervalo principal* $[\alpha, \beta]$ *en subintervalos*
3. *Asignar* m/p *subintervalos a cada procesador*
4. *Fin en el procesador maestro*

5. *En cada procesador*
6. *Para cada subintervalo asignado* $[\alpha_i, \beta_i]$
7. $\sigma = (\beta_i - \alpha_i)/2$
8. *Aplicar Lanczos con shift-and-invert para extraer los valores propios alrededor de* σ *y sus vectores propios*
9. *Fin para*

10. *Enviar valores y vectores propios al procesador maestro*
11. *Fin en cada procesador*

12. En el procesador maestro

13. Reunir los resultados de todos los procesadores

14. Fin en el procesador maestro

5.4.1. Detalles de implementación

Se han implementado tres versiones del Algoritmo 3:

1. Versión MPI

2. Versión OpenMP

3. Versión MPI+OpenMP

En la versión MPI, todos los procesadores leen los datos de entrada de disco (matrices y intervalo principal). Cada procesador aplica el algoritmo para la división del intervalo principal y en función de su índice selecciona los subintervalos que le corresponden y los resuelve. Finalmente, los resultados se reúnen en el proceso maestro que deberá ordenarlos en caso de necesidad. Esta versión está orientada a ejecutarse en pequeños clústers de computadores homogéneos.

En la versión OpenMP, tan sólo el hilo maestro lee los datos de disco y realiza el particionado del intervalo $[\alpha, \beta]$. A continuación, los subintervalos son asignados y distribuido entre los hilos. Esta versión está orientada a ejecutarse en máquinas de memoria compartida.

Finalmente, la versión MPI+OpenMP combina las dos aproximaciones descritas anteriormente. En el primer nivel de paralelismo, se crea un conjunto de p procesos MPI que ejecutan la versión MPI del algoritmo. Entonces, en la etapa en la que cada proceso MPI resuelve sus $\frac{m}{p}$ subintervalos, se introduce un nuevo nivel de paralelismo. Se crea un grupo de p' hilos y los $\frac{m}{p}$ subintervalos se distribuyen entre ellos. Finalmente habrá $p \times p'$ procesadores trabajando en la solución del problema. Esta versión es una combinación de las dos previas y está orientada a ejecutarse en clústers de multi-procesadores o multi-cores.

5.5. Resultados experimentales

Para evaluar las implementaciones realizadas se han elegido dos entornos diferentes: un pequeño clúster de multi-procesadores formado por los nodos 4 y 5 del clúster descrito en la Capítulo 2 y un SGI Altix 3700. De éste sólo se han utilizado 4 procesadores Itanium II de los 44 disponibles.

El algoritmo ha sido diseñado para ejecutarse en máquinas de un coste moderado, por lo que este clúster es un entorno muy apropiado para su evaluación. A pesar de esto, también hemos querido evaluar sus prestaciones en una máquina mucho más potente

Las Tablas 5.1 a 5.5 muestran los tiempos de ejecución de las 3 implementaciones en ambas plataformas de prueba. Se han elegido como banco de pruebas guías de tipo *ridge* de diferentes tamaños, véase [20] para más información sobre las características de la guía.

$M+N$	$p=1$	$p=2$	$p=4$
5000	71.68	40.92	20.45
8000	199.26	121.22	67.98
11000	426.32	257.13	140.06
14000	772.10	413.06	221.21
17000	1247.71	655.40	367.26
20000	1685.27	1003.56	540.88

Tabla 5.1: Tiempo de ejecución de la versión MPI en el clúster SMP

$M+N$	$p=1$	$p=2$
5000	71.68	38.11
8000	199.26	109.78
11000	426.32	246.32
14000	772.10	419.12
17000	1247.71	646.51
20000	1685.27	963.91

Tabla 5.2: Tiempo de ejecución de la versión OpenMP en el clúster SMP

$M+N$	$p=1$	$p=4$
5000	71.68	20.53
8000	199.26	61.59
11000	426.32	134.88
14000	772.10	216.84
17000	1247.71	333.86
20000	1685.27	534.69

Tabla 5.3: Tiempo de ejecución de la versión MPI+OpenMP en el clúster SMP

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	25.44	18.66	14.95
8000	161.99	86.46	69.25	55.67
11000	321.68	185.16	148.37	133.35
14000	598.13	337.35	249.26	247.38
17000	893.64	494.42	405.15	351.61
20000	1259.16	665.58	556.76	532.72

Tabla 5.4: Tiempo de ejecución de la versión OpenMP en el SGI Altix

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	23.69	16.17	13.09
8000	161.99	86.34	60.85	49.24
11000	321.68	172.88	117.61	91.53
14000	598.13	310.42	217.38	170.07
17000	893.64	498.16	304.24	241.64
20000	1259.16	658.08	446.46	349.44

Tabla 5.5: Tiempo de ejecución de la versión MPI en el SGI Altix

5.5.1. Análisis de los resultados

Los resultados anteriores certifican que el método descrito para el cálculo de valores y vectores propios de matrices estructurados paraleliza extremadamente bien en diferentes arquitecturas. Las claves para dicho comportamiento son el uso del Teorema de la Inercia para asegurar el

correcto equilibrio de la carga, así como la ausencia de comunicaciones durante la ejecución del algoritmo.

La Tabla 5.6 muestra el speed-up de las versiones MPI y OpenMP en uno de los biprocesadores. Los resultados de la versión OpenMP son ligeramente superiores a los de la versión MPI. Este comportamiento es el que cabe esperar ya que OpenMP puede realizar un uso más eficiente de las características de memoria compartida de la máquina.

$M+N$	OpenMP	MPI
5000	1.88	1.75
8000	1.82	1.64
11000	1.73	1.66
14000	1.84	1.87
17000	1.93	1.90
20000	1.75	1.68

Tabla 5.6: Speed-up en el clúster SMP para las versiones OpenMP y MPI ($p = 2$).

La Tabla 5.7 muestra el speed-up de las versiones MPI y MPI+OpenMP ejecutándose sobre los 4 procesadores del clúster. En este tipo de entornos con dos niveles de paralelismo (memoria compartida en los nodos y distribuida entre ellos) la combinación de MPI y OpenMP ofrece mejores resultados que el uso de MPI exclusivamente.

$M+N$	MPI+OpenMP	MPI
5000	3.49	3.51
8000	3.24	2.93
11000	3.16	3.04
14000	3.56	3.49
17000	3.74	3.40
20000	3.15	3.12

Tabla 5.7: Speed-up en el clúster SMP para las versiones MPI+OpenMP y MPI ($p = 4$).

La Tabla 5.8 muestra el speed-up de las versiones MPI y OpenMP en el SGI Altix. En esta máquina, la versión MPI escala mucho mejor que la versión OpenMP. Se trata de un resultado sorprendente, ya que se trata de una máquina ccNUMA que en teoría debería ofrecer buenas

prestaciones cuando se programa con el paradigma de memoria compartida. Tras un análisis de la arquitectura y del funcionamiento de los algoritmo de planificación de la máquina, hemos llegado a la conclusión de que hay problemas de contención en el acceso a memoria de los distintos hilos y que estos se agravan cuando crece su número. Esto es debido a que todas las matrices y estructuras de datos son reservadas por el hilo principal en la memoria correspondiente al procesador en el que se está ejecutando. La latencia en el acceso a memoria no es constante en este tipo de arquitecturas y cuando aumente el número de hilos se producen dos efectos: i) empiezan a haber hilos ejecutándose lejos[1] del procesador principal, lo cual aumenta la latencia ii) el ancho de banda de la memoria es limitado y puede llegar a saturarse.

version	$p = 2$	$p = 3$	$p = 4$
MPI version	1,91	2,82	3,60
OpenMP version	1,89	2,26	2,36

Tabla 5.8: Speed-up en el SGI Altix de las versiones MPI y OpenMP ($M + N = 20000$).

5.6. Conclusiones

Se han presentado 3 implementaciones paralelas de un algoritmo de tipo Lanczos para la resolución de un problema generalizado de valores propios. Dicho problema posee algunas características especiales: las matrices son dispersas y estructuradas y los valores y vectores propios a extraer vienen definidos por un intervalo que se nos proporciona.

La técnica propuesta paraleliza muy bien y cualquiera de las implementaciones propuestas alcanza buen speed-up, incluso para un número pequeño de procesadores.

OpenMP es la mejor elección para programar máquinas multi-procesador o multi-core. En cambio para arquitecturas ccNUMA, se concluye que OpenMP puede presentar problemas de escalabilidad que habrá que tener en cuenta e intentar aliviar.

La programación multinivel (MPI+OpenMP) es la mejor opción para máquinas híbridas (con dos niveles de paralelismo), ya que este paradigma puede explotar las características de la memoria compartida y distribuida de la máquina.

[1] En términos de número de enlaces a atravesar en el hipercubo que forman los 44 procesadores disponibles.

Finalmente, podemos concluir que el tiempo de ejecución en ambas máquinas no difiere mucho, mientras que el speed-up es claramente superior en el caso del clúster de bi-procesadores. Por lo que la relación precio-prestaciones es mucho mejor en el segundo.

Este trabajo a dado lugar a dos publicaciones: [14] y [11]

Capítulo 6

Cálculo paralelo de los valores propios de una matriz Toeplitz simétrica mediante métodos iterativos

6.1. Introducción

En este trabajo nos centramos en el cálculo de valores y vectores propios de matrices Toeplitz simétricas. En la literatura específica no abundan los métodos prácticos para la resolución de este tipo de problemas. Los únicos algoritmos que hemos podido encontrar están basados en técnicas de bisección; puesto que existen diversas recurrencias para la evaluación del polinomio característico de este tipo de matrices, es posible aislar intervalos que contengan un único valor propio y posteriormente extraerlo mediante alguna técnica de cálculo de raíces como Newton, Pegasus, secante, ... [29, 25, 8]

En nuestro caso, vamos a proponer una aproximación basada en dividir el intervalo principal de búsqueda en un subintervalos que contengan más de un valor propio (unas decenas) y posteriormente procesarlos mediante la variante del algoritmo de Lanczos conocida como "shift-and-invert". Además, usaremos los algoritmos para la resolución de sistemas de ecuaciones Toeplitz que hemos desarrollado en otros trabajos para la resolución de los sistemas que aparecen en la

iteración de Lanczos.

6.2. Descripción del método

Al igual que en el trabajo anterior vamos a basar la extracción de los valores propios en la versión "shift-and-invert" del algoritmo de Lanczos. Puede consultarse [17] para una descripción y discusión detallada del algoritmo.

La única diferencia con la formulación anterior es que se trata de un problema estándar de valores propios y por tanto trabajaremos con la matriz $T - \sigma I \in \mathbb{R}^{n\times n}$, siendo I la matriz identidad. Dicha matriz mantiene la estructura Toeplitz por lo que se pueden aplicar algoritmos específicos para el caso.

6.2.1. Optimizaciones en Lanczos para el caso Toeplitz simétrico

En [32], Voss introduce optimizaciones en el algoritmo de Lanczos para el cálculo del valor propio de menor magnitud de una matriz Toeplitz simétrica. Concretamente, se aprovecha el conocimiento que se tiene de la estructura del espectro de una matriz Toeplitz para reducir el tiempo de cálculo. En nuestro trabajo hemos descubierto que esa misma técnica puede ser aplicada al problema del cálculo del espectro completo reduciéndose también el tiempo de cálculo respecto a las versiones clásicas de Lanczos. Los siguiente párrafos resumen la aportación de Voss:

Sea $J_n = (\delta_{i,n+1-i})_{i,j=1,\ldots,n}$ la matriz (n,n) que aplicada a un vector lo invierte. Un vector v es simétrico si $x = J_n x$ y anti-simétrico si $x = -J_n x$. Se sabe que los vectores propios de una matriz Toeplitz son o bien simétricos o anti-simétricos. Extendiendo la notación diremos que un valor propio es simétrico (o anti-simétrico) si su vector propio asociado es simétrico (o anti-simétrico).

Considérese ahora el algoritmo de Lanczos. Si se inicializa éste con un vector simétrico, todo el espacio de Krylov generado estará en la misma clase de simetría y por tanto los vectores propios que engendre también lo serán. El razonamiento es análogo si se inicializa con un vector anti-simétrico.

Se sabe que en la mayoría de ocasiones los valores propios simétricos y antisimétricos de una matriz Toeplitz simétrica se encuentran entrelazados. Por tanto si restringimos la búsqueda a una de las dos clases conseguiremos aumentar la separación entre los valores propios y, por tanto,

aumentar la velocidad de convergencia.

En su trabajo, Voss calcula el menor de los valores propios de la matriz. Desgraciadamente no es posible saber a cual de las dos clases pertenecerá.

Para resolver este problema, Voss ideó un método de Lanczos de dos vías, la clave del mismo es la siguiente: si $T \in \mathbb{R}^{n\times n}$ es una matriz Toeplitz simétrica, y $v \in \mathbb{R}^n$ es la solución del sistema $Tv = w$, entonces la parte simétrica de $v_s = 0{,}5(v + J_n v)$ resuelve el sistema $Tv_s = 0{,}5(w + J_n w)$ y la parte anti-simétrica $v_a = v - v_s$ resuelve el sistema $Tv_a = w - w_s$.

A partir de esta propiedad se puede diseñar un algoritmo de tipo Lanczos con dos "vías" de trabajo, que extraiga simultáneamente los valores propios simétricos y antisimétricos. Cada "vía" calcula el valor propio de menor magnitud de cada una de las clases de simetría, aplicando el método de Lanczos invertido para resolver un problema de valores propios de talla m (la mitad de la talla original). Ambos procesos de Lanczos trabajan en paralelo hasta que se llega a la resolución del sistema de ecuaciones. Entonces, a partir de la propiedad citada anteriormente, se resuelve un único sistema lineal y se extrae la parte simétrica y anti-simétrica, que será entrada para la siguiente etapa de Lanczos. De esta forma, se construyen dos matrices tridiagonales, una para cada clase de simetría:

$$\widetilde{SYM}_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix}; \widetilde{SKS}_j = \begin{bmatrix} \gamma_1 & \delta_1 & & \\ \delta_1 & \gamma_2 & \ddots & \\ & \ddots & \ddots & \delta_{k-1} \\ & & \delta_{k-1} & \gamma_k \end{bmatrix} \tag{6.1}$$

junto con dos matrices ortogonales $P_k = [p_1, p_2, \ldots, p_k]$ y $Q_k = [q_1, q_2, \ldots, q_k]$ que contienen los vectores de Lanczos para cada clase de simetría.

El siguiente algoritmo describe el proceso. Básicamente es el mismo algoritmo presentado por Voss en [32], pero implementando la técnica del desplazamiento y con reortogonalizaciones (necesarias si se quieren extraer varios valores propios).

Algoritmo 4 *:* **Algoritmo de Lanczos con "Shift–and–Invert" y dos vías.**
Sea $T \in \mathbb{R}^{n\times n}$ una matriz Toeplitz simétrica, este algoritmo retorna los valores propios próximos a σ y sus vectores propios asociados.

1. *Let* $p_1 = J_n p_1 \neq 0$ *and* $q_1 = -J_n q_1 \neq 0$ *initial vectors*
2. *Let* $p_0 = q_0 = 0; \beta_0 = \delta_0 = 0$;
3. $p_1 = p_1 / \|p_1\|$; $q_1 = q_1 / \|q_1\|$;
4. *for* $j = 1, 2, ...$ *until convergence:*
5. $w = p_k + q_k$;
6. *solve* $(T - \sigma I)\, v = w$;
7. $v_s = 0{,}5 \cdot (v + J_n v)$; $v_a = 0{,}5 \cdot (v - J_n v)$;
8. $\alpha_k = v_s^t \cdot p_k$; $\gamma_k = v_a^t \cdot q_k$;
9. $v_s = v_s - \alpha_k \cdot p_k - \beta_{k-1} \cdot p_{k-1}$; $v_a = v_a - \gamma_k \cdot q_k - \delta_{k-1} \cdot q_{k-1}$;
10. *full re-orthogonalization* ;
11. $\beta_k = \|v_s\|_2$; $\delta_k = \|v_a\|_2$;
12. $p_{k+1} = v_s / \beta_k$; $q_{k+1} = v_a / \delta_k$;
13. *obtain eigenvalues of* $\widetilde{SYM}_k$; *obtain eigenvalues of* $\widetilde{SKS}_k$;
14. *test bounds for convergence* ;
15. *end for*
16. *compute associated eigenvectors.*

Esta reordenación del método de Lanczos, pretende doblar la velocidad de la versión estándar (en términos de número de valores propios extraídos por iteración). Los experimentos así lo confirman, pero además, se ha detectado que con el mismo número de iteraciones la versión con dos vías de Lanczos extrae 3 o 4 veces más valores propios que la "shift-and-invert" clásica (usando el mismo shift, claro). Esto se debe al aumento en la separación de los valores propios.

6.2.2. Resolución de sistemas de ecuaciones Toeplitz simétricas

La eficiencia del método descrito anteriormente depende de la resolución del sistema $(T - \sigma I)v = w$ en la linea 6 del Algoritmo 4. En [10] se presenta el método optimizado para la resolución de sistemas de ecuaciones Toeplitz que hemos usado en éste y en otros trabajos, así como implementaciones secuenciales y paralelas del mismo.

6.2.3. Método completo

El algoritmo de Lanczos en su versión "shift-and-invert" de dos vías, junto con el algoritmo para la resolución sistemas de ecuaciones Toeplitz simétricos, permite el cálculo eficiente de los valores propios situados cerca del desplazamiento σ. Si se quiere calcular todo el espectro (o un subconjunto amplio) se deberán usar varios desplazamientos de tal forma que se consiga cubrir todo el intervalo sobre el que se quiere trabajar.

Algoritmo 5 *:* **Algoritmo completo.**

1. *Elegir el intervalo* $[a, b]$ *que contenga todos los valores propios*
2. *Dividir* $[a, b]$ *en subintervalos*
3. *Para cada subintervalo: (* en paralelo *)*
4. *Calcular el "shift"*σ*, posiblemente* $\sigma = (a + b)/2$
5. *Descomponer la matriz* $(T - \sigma I)$ *en las matrices de tipo Cauchy* C_0 *and* C_1
6. *Calcular la factorización* LDL^T *de las matrices de tipo Cauchy*
7. *Aplicar el método de Lanczos con "Shift–and–Invert" y dos vías (Algoritmo 4) para extraer todos los valores propios del subintervalo y sus vectores propios asociados*
8. *Fin Para*

El coste espacial del algoritmo es $O((\frac{n}{2})^2)$ y viene determinado por el almacenamiento de los factores triangulares de la descomposición LDL^T de las matrices de Cauchy. Si dicho coste no fuera asumible se podría cambiar el solver de Cauchy por uno de tipo Levinson con unas necesidades de memoria menores, aunque con un coste temporal mayor.

El procesamiento de cada subintervalo es independiente del resto, por tanto la paralelización es trivial, simplemente repartiendo los subintervalos entre los procesadores disponibles. No obstante, la eficiencia del método (tanto en secuencial como en paralelo) va a depender de la correcta selección de estos subintervalos. Si alguno de los subintervalos contiene demasiados valores propios entonces se necesitarán muchas iteraciones de Lanczos para su cálculo (y posiblemente reinicializaciones del proceso). Por tanto la selección de los intervalos es otra parte clave del

algoritmo.

6.2.4. Selección de los subintervalos

Existen diversos factores a tener en cuenta a la hora de particionar el intervalo principal. En primer lugar, se necesitan varias iteraciones de Lanczos antes de que empiecen a converger los valores propios (normalmente unas 5-7 para que converja el primero). Tras esto, los valores propios empiezan a converger de forma bastante rápida, especialmente con la versión de dos vías. Un resultado bastante realista sería obtener 25-30 valores propios tras 40 iteraciones.

Por otra parte, el Algoritmo 4 ha sido implementado con reortogonalización completa; esto significa que el coste computacional aumenta con cada iteración. Además, para poder tratar casos especialmente adversos o valores propios con multiplicidad mayor que uno, se ha fijado un número máximo de iteraciones de Lanczos (tamaño máximo del espacio de Krylov). Cuando dicho límite es alcanzado, el algoritmo realiza una reinicialización consistente en volver inicializar el algoritmo con un vector ortogonal a los vectores propios ya convergidos. Este mecanismo aporta robustez al algoritmo, pero degrada mucho las prestaciones, por lo que habrá que evitarlo siempre que se pueda.

Estos dos factores nos obligan a que el número de valores propios por subintervalo no sea excesivamente grande, y que su longitud tampoco lo sea, ya que subintervalos muy grandes con valores propios en ambos extremos ralentizan mucho la convergencia. Una cota razonable para el número máximo de valores propios podría ser entre 30 y 50 por subintervalo, aunque este factor es dependiente del problema y de la implementación. Nosotros hemos elegido 40 en este caso.

Para dividir el intervalo principal se va a usar la técnica presentada en la Sección 4.2 basada en el Teorema de la Inercia. Para poder aplicar el Algoritmo 1 debemos definir una nueva función

$$neg_n(T, c) : \mathbb{R}^{n\times n} \times \mathbb{R} \to \mathbb{N} \tag{6.2}$$

Al igual que en el casos anteriores se basa en la factorización LDL^T de $T - cI$ y el número de valores propios menores que c es equivalente al número de valores negativos en D. Para este tipo de matrices no se dispone de una recurrencia que simplifique dicho cálculo, por lo que será necesario calcular la factorización explícitamente.

n	tiempo de aislamiento	% del tiempo total	tiempo de extracción	% del tiempo total
1000	0,38	7,4	4,73	92,6
5000	19,14	7,1	251,88	92,9
10000	136,25	6,6	1933,76	93,4
15000	440,53	5,7	7267,41	94,3

Tabla 6.1: Tiempo de ejecución (s) de las fases de aislamiento y extracción

6.3. Implementación y paralelización del método

La versión secuencial del Algoritmo 5 ha sido implementada en Fortran95 (Intel Fortran Compiler 8.1) usando BLAS y LAPACK siempre que haya sido posible (Intel MKL 8.1). Para añadir robustez al algoritmo, ha sido implementado usando reinicializaciones y reortogonalización completa. Los valores y vectores propios de las matrices $\widetilde{SYM}_k$ y $\widetilde{SKS}_k$ ha sido calculados mediante la función *dstevr* de LAPACK.

Los tests de convergencia han podido ser simplificados en gran medida al conocerse el número exacto de valores propios en cada subintervalo. Como ya se había dicho, éste se ha fijado a un máximo de 40. Si se aumenta dicho valor, el tiempo de aislamiento (etapas 1 y 2) disminuirá mientras que el tiempo de extracción (etapas 3-10) aumentará. La Tabla 6.1 muestran los tiempos de aislamiento y extracción para diferentes tallas.

6.3.1. Paralelización

Se ha desarrollado una versión paralela para memoria distribuida del Algoritmo 5 usando MPI. La fase de extracción del algoritmo empieza en la linea 3 y su paralelización es trivial. La Tabla 6.1 muestra que la fase de aislamiento tiene un coste proporcionalmente mucho menor, aunque no despreciable. Por tanto, también se ha paralelizado mediante el siguiente esquema.

Algoritmo 6 *:* **Algoritmo aislamiento paralelo**

1. *Seleccionar el intervalo* $[a, b]$ *que contenga todos los valores propios requeridos*
2. *Dividir el intervalo* $[a, b]$ *en* p *subintervalos* $[a_i, b_i]$ *de igual longitud*

3. *Para cada procesador* $i = 0, ..., p-1$
4. *Aplicar el Algoritmo 1 al subintervalo* $[a_i, b_i]$
5. *Fin Para*
6. *Reunir todos los subintervalos en el proceso maestro*

En la versión paralela del Algoritmo 5 la fase de extracción se ha diseñado mediante el paradigma maestro-esclavo, el maestro gestiona una cola con los resultados de la fase anterior y los va asignando a los esclavos. Cuando un esclavo termina de procesar un subintervalo pide uno nuevo al maestro, hasta que se vacíe la cola. Los esclavos procesan los subintervalos de una forma puramente secuencial.

Finalmente, se puede observar que el algoritmo ofrece niveles extra de paralelismo, como resolver ambas "vías" del Algoritmo 4 en paralelo, o resolver en paralelo ambos sistemas de Cauchy. Estas posibles mejoras se están investigando en estos momentos. Sin embargo, no se espera obtener mejoras substanciales sobre la paralelización basada en distribución de los intervalos.

6.4. Resultados experimentales

Se han usado matrices Toeplitz simétricas aleatorias para verificar el código y evaluar sus prestaciones, con tallas $N = 1000, 5000, 10000$ y 15000. Los programas implementados se han ejecutado en un clúster formado por 55 biprocesadores Intel Xeon, interconectados por una red SCI. Con fines comparativos, se han resuelto los mismos problemas con las rutinas que ofrece LAPACK, en el caso secuencial, y ScaLAPACK en el caso paralelo.

Se ha observado diferencia de prestaciones significativa entre las distintas rutinas disponibles en LAPACK y ScaLAPACK para la resolución del problema estándar de valores propios. Hemos constatado que la rutina *dsyevr* de LAPACK y la rutina *pdsyevd* de ScaLAPACK son las que mejores prestaciones ofrecen y por tanto las que hemos usado para las comparaciones. Ambas librerías necesitan que trabajar con la matriz en formato denso (distribuida en el caso de ScaLAPACK), por lo que de entrada el consumo de memoria es un orden de magnitud superior. El caso $N = 15000$ no ha podido ser ejecutado en un solo procesador por no disponer de memoria suficiente.

	N=1000	N=5000	N=10000	N=15000
FSTW Lanczos	5,12	271,02	2070,01	7707,94
LAPACK (DSYEVR)	2,27	225,57	1754,64	memoria insuficiente

Tabla 6.2: Tiempo de ejecución (s) del la versión secuencial del Algoritmo 5 y de LAPACK, calculando todos los valores propios

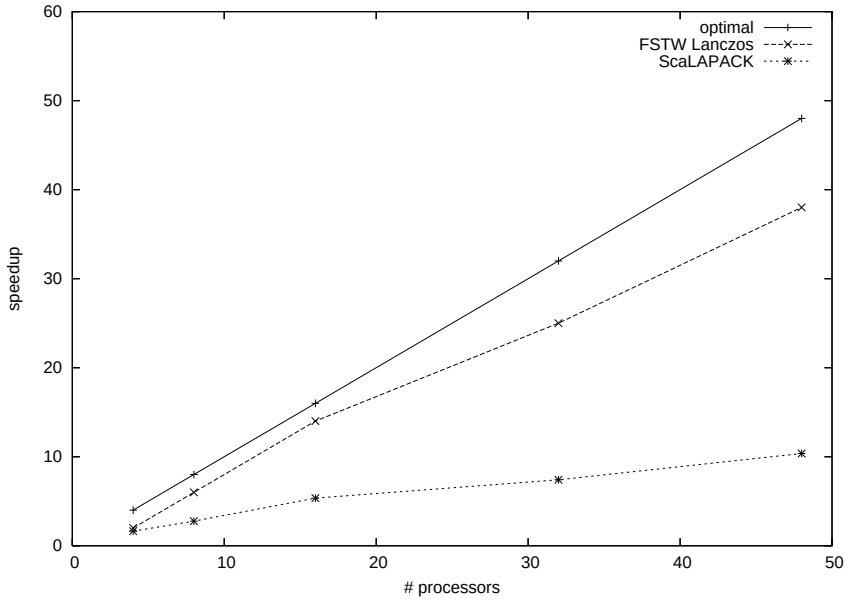

Figura 6.1: Speedup para $N = 5000$

Las Tablas 6.2 y 6.3 recogen los resultados del cálculo del espectro completo de una matriz Toeplitz simétrica. En la Tabla 6.3 la columna de la izquierda indica el número de procesadores; se ha elegido el formato (1+x) para remarcar que el algoritmo paralelo sigue una estructura maestro-esclavo y que por tanto hay un procesador que no participa en el cálculo.

Los tiempos obtenidos revelan que, aunque LAPACK es algo más rápido que la versión secuencial del Algoritmo 5, la versión paralela es claramente superior a ScaLAPACK en la mayoría de casos y las diferencias crecen a medida que se va aumentando el número de procesadores. Las Figuras 6.1 y 6.2 muestran el speed-up para los casos $n = 5000$, $n = 10000$. Se observa que la versión FSTW de Lanczos obtiene speed-ups excelentes y que la ventaja respecto a *pdsyevd* aumenta con la talla del problema.

Si sólo se desea calcular una fracción del espectro, es comparativamente aún más eficiente. El motivo es la forma en que las rutinas de LAPACK y ScaLAPACK abordan el problema. En estas en primer lugar se tridiagonaliza la matriz y luego se resuelve el problema de valores propios

	N=1000		N=5000	
Procesadores	FSTW Lanczos	PDSYEVD	FSTW Lanczos	PDSYEVD
4(1+3)	1,62	3,59	94,40	137,10
8 (1+7)	0,72	2,38	39,49	81,69
16 (1+15)	0,39	1,32	19,22	42,17
32 (1+31)	0,34	4,17	10,77	30,45
48 (1+47)	0,33	4,60	6,98	21,77
	N=10000		N=15000	
Procesadores	FSTW Lanczos	PDSYEVD	FSTW Lanczos	PDSYEVD
4(1+3)	699,49	925,92	2540,23	2184,28
8 (1+7)	304,66	488,83	1099,98	1146,04
16 (1+15)	146,63	251,37	568,98	691,48
32 (1+31)	75,23	152,31	277,01	454,38
48 (1+47)	53,80	112,01	190,74	298,57

Tabla 6.3: Tiempo de ejecución (s) del la versión paralela del Algoritmo 5 y de ScaLAPACK, calculando todos los valores propios

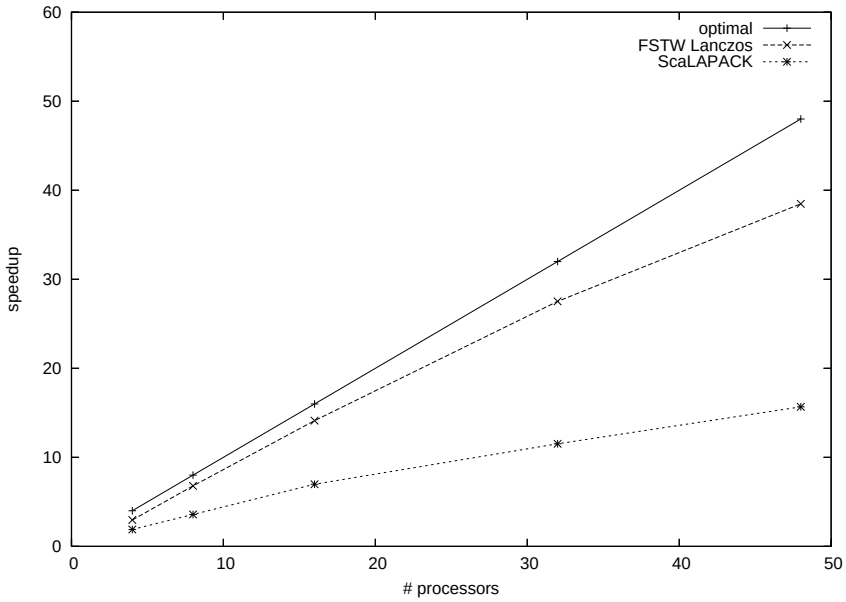

Figura 6.2: Speedup para $N = 10000$

	P=1, N=1000		P=1, N=5000	
	FSTW Lanczos	DSTEBZ	FSTW Lanczos	DSTEBZ
10 %	0,52	1,32	32,07	129,96
20 %	1,16	2,09	82,79	199,19
50 %	2,49	4,32	194,76	441,83
	P=16(4x4), N=5000		P=16(4x4), N=10000	
	FSTW Lanczos	PDSTEBZ	FSTW Lanczos	PDSTEBZ
10 %	3,03	23,12	26,00	123,99
20 %	6,69	23,74	52,37	131,60
50 %	14,52	24,78	98,35	152,37

Tabla 6.4: Tiempo de ejecución (s) del Algoritmo 5, LAPACK y ScaLAPACK, calculando el 10, 20 y 50 % de valores y vectores propios. P es el número de procesadores

tridiagonal. Esta primera etapa tiene un peso importante que no puede ser reducido aunque solo se quiera calcular parte del espectro.

Hemos realizado experimentos calculando una fracción del espectro (10, 25 y 50 % del mismo), en el caso de 1 y 16 procesadores (con grid 4×4 para las rutinas de ScaLAPACK). En este caso se ha comparado con las rutinas de bisección de LAPACK (*dstebz*) y ScaLAPACK (*pdstebz*), ya que son las que permiten calcular un subconjunto del espectro y por tanto nos parecen más apropiadas para la comparación. La Tabla 6.4 recoge los resultados. (Obsérvese que la subrutina *dstebz* sólo es competitiva respecto a *dsyevr* cuando se quiere calcular una pequeña fracción del espectro; por ejemplo, se observa que para $n = 5000$ calcular el espectro completo con *dsyevr* es más rápido que calcular el 50 % del mismo con *dstebz*.)

Se observa claramente que cuando la fracción del espectro a calcular disminuye, el coste fijo de la tridiagonalización se transforma en el coste principal de las rutinas de LAPACK y ScaLAPACK y nuestro algoritmo es comparativamente mucho más rápido.

6.5. Conclusiones

Se ha presentado un nuevo algoritmo, basado en otros ya existentes, para el cálculo de los valores y vectores propios de una matriz Toeplitz simétrica. Dicho algoritmo puede ejecutarse

secuencialmente, aunque es intrínsecamente paralelo. Los resultados muestran que es más rápido que las rutinas de ScaLAPACK para el mismo problema, dicha ventaja aumenta aún más cuando sólo se necesita calcular un subconjunto del espectro.

Otra ventaja del algoritmo es su reducido coste espacial. En secuencial, puede abordar problemas imposibles de resolver con las rutinas de LAPACK. A esto hay que añadir que se puede sustituir el algoritmo para la resolución de sistemas lineales Toeplitz simétricos basado en descomposición de Cauchy por uno de tipo Levinson, para reducir la cantidad de memoria necesaria y poder abordar tallas aún mayores.

Finalmente, la escalabilidad del algoritmo está cerca de valores óptimos, siendo mucho mayor que la de las rutinas de ScaLAPACK.

Este trabajo ha dado lugar a la siguiente publicación: [31]

Bibliografía

[1] Página web del proyecto Grid Engine: *http://gridengine.sunsource.net*.

[2] Página web del proyecto OpenPBS: *http://www.openpbs.org*.

[3] Página web del proyecto Perceus: *http://www.perceus.org*.

[4] Página web del proyecto SIS: *http://www.systemimager.org*.

[5] *N1 Grid Engine 6 User's Guide*, May 2005.

[6] *Rocks Cluster Distribution: Users Guide: User's Guide for Rocks version 4.2.1 Edition*, December 2006.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[8] J. M. Badía and A. M. Vidal. Parallel algorithms to compute the eigenvalues and eigenvectors of symmetric toeplitz matrices. *Parallel Algorithms and Applications*, 13:75–93, 1989.

[9] J. M. Badía and A. M. Vidal. *High Performance Algorithms for Structured Matrix Problems*, chapter Parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem. Nova Science Publishers, 1998.

[10] Miguel O. Bernabeu, Pedro Alonso, and Antonio M. Vidal. A multilevel parallel algorithm to solve symmetric toeplitz linear systems. *J. Supercomput.*, 44(3):237–256, 2008.

[11] Miguel O. Bernabeu, Máriam Taroncher, Victor M. Garcia, and Ana Vidal. Robust parallel implementation of a lanczos-based algorithm for an structured electromagnetic eigenvalue problem. *Scalable Computing: Practice and Experience*, 8(3), 2007.

[12] Miguel O. Bernabeu and Antonio M. Vidal. Static versus dynamic heterogeneous parallel schemes to solve the symmetric tridiagonal eigenvalue problem. In *ACS06*. WSEAS, 2006.

[13] Miguel O. Bernabeu and Antonio M. Vidal. The symmetric tridiagonal eigenvalue problem: a heterogeneous parallel approach. *WSEAS Transactions on Mathematics*, 6(4):587–594, 2007.

[14] Miguel Oscar Bernabeu, Mariam Taroncher, Victor M. Garcia, and Ana Vidal. Parallel implementation in pc clusters of a lanczos-based algorithm for an electromagnetic eigenvalue problem. In *ISPDC*, pages 296–300. IEEE Computer Society, 2006.

[15] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[16] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[17] James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[18] J. J. Dongarra, R. A. van de Geijn, and R. C. Whaley. A users' guide to the BLACS. Manuscript. Department of Computer Science, University of Tennessee, Knoxville, TN 37996., 1993.

[19] Erich Focht. Heterogeneous clusterswith oscar: Infrastructure and administration. In *HPCS*, page 37. IEEE Computer Society, 2006.

[20] V. M. García, A. Vidal, V. E. Boria, and A. M. Vidal. Efficient and accurate waveguide mode computation using bi-rme and lanczos methods. *International Journal for Numerical Methods in Engineering*, 65(11):1773–1788, 2006.

[21] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[22] Chokchai Leangsuksun, Lixin Shen, Tong Liu, and Stephen L. Scott. Achieving high availability and performance computing with an ha-oscar cluster. *Future Generation Comp. Syst.*, 21(4):597–606, 2005.

[23] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users' Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*, October 1997.

[24] Timothy G. Mattson. High performance computing at intel: The oscar software solution stack for cluster computing. In *CCGRID*, pages 22–25. IEEE Computer Society, 2001.

[25] Michael K. Ng and William F. Trench. Numerical solution of the eigenvalue problem for Hermitian Toeplitz-like matrices. Technical Report TR-CS-97-14, Canberra 0200 ACT, Australia, 1997.

[26] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. Npaci rocks: tools and techniques for easily deploying manageable linux clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):707–725, 2003.

[27] A. Ruhe. *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*, chapter Generalized Hermitian Eigenvalue Problem; Lanczos Method. SIAM, 2000.

[28] Stephen L. Scott. Oscar and the beowulf arms race for the "cluster standard". In *CLUSTER*, page 137. IEEE Computer Society, 2001.

[29] F. William Trench. Numerical solution of the eigenvalue problem for hermitian toeplitz matrices. *SIAM Journal on Matrix Analysis and Applications*, 10(2):135–146, 1989.

[30] A. M. Vidal, A. Vidal, V. E. Boria, and V. M. García. Parallel computation of arbitrarily shaped waveguide modes using bi-rme and lanczos methods. *Communications in Numerical Methods in Engineering*, 2006.

[31] Antonio M. Vidal, Victor M. Garcia, Pedro Alonso, and Miguel O. Bernabeu. Parallel computation of the eigenvalues of symmetric toeplitz matrices through iterative methods. *J. Parallel Distrib. Comput.*, 68(8):1113–1121, 2008.

[32] Heinrich Voss. A symmetry exploiting Lanczos method for symmetric Toeplitz matrices. *Numerical Algorithms*, 25(1–4):377–385, 2000.

[33] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

Apéndice A

Artículos públicados

A continuación se adjuntan las públicaciones generadas a partir de trabajo presentado en esta Tesis.

Parallel Implementation in PC Clusters of a Lanczos-based Algorithm for an Electromagnetic Eigenvalue Problem

Miguel Óscar Bernabeu
Dpt. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
mbernabeu@dsic.upv.es

Máriam Taroncher
Dpt. de Comunicaciones,
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
matacal@iteam.upv.es

Víctor M. García
Dpt. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
vmgarcia@dsic.upv.es

Ana Vidal
Dpt. de Comunicaciones,
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
avidal@dcom.upv.es

Abstract

This paper describes a parallel implementation of a Lanczos-based method to solve generalised eigenvalue problems related to the modal computation of arbitrarily shaped waveguides. This efficient implementation is intended for execution in moderate-low cost workstations (2 to 4 processors). The problem under study has several features: the involved matrices are sparse with a certain structure, and all the eigenvalues needed are contained in a given interval. The novel parallel algorithms proposed show excellent speed-up for small number of processors.

1. Introduction

This paper is focused on the parallelisation of a Lanczos-based method for the solution of the following generalised eigenvalue problem: Given a symmetric pencil $\mathbf{A}x = \lambda\mathbf{B}x$, find all the generalised eigenvalues (and the corresponding eigenvectors) comprised in a given interval. This interval contains a large number of eigenvalues.

An efficient sequential method was already proposed in [1]. However, when the number of desired eigenvalues is very large, the execution time is still too long. A first parallel algorithm was recently introduced in [2], using MPI and a distributed-memory approach. The results presented in that paper show that the method parallelises extremely well for a large number of processors.

A code based in the proposed technique will be included in a CAD tool for design of passive waveguide components. However, this CAD tool will usually run in low cost workstations or, at most, small PC clusters. For these small systems, a different approach should be chosen.

Therefore, the goal of this paper is to explore different parallel programming approaches for the implementation of the sequential technique described in [1], in low cost workstations and small clusters.

Three different approaches have been examined: First, we have designed an OpenMP version of the Lanczos algorithm to take advantage of bi-processor machines. Next, we implemented a version for distributed memory machines using MPI (Message Passing Interface), to execute it on clusters of PCs. Finally, a mixed approach was proposed in order to achieve optimum performance on clusters of bi-processors.

The paper is organised as follows: first, we will briefly outline the sequential problem (fully described in [1]). Then, the new parallelisation schemes will be completely described, taking into account the different proposed options: i.e. MPI, OpenMP, MPI+OpenMP and so on. Finally, some numerical results are shown, and then the conclusions of this work are given.

2. Problem Description and Sequential Algorithm

2.1. The electromagnetic problem

In this study, the efficient and accurate modal computation of arbitrary waveguides is based on the Boundary Integral - Resonant Mode Expansion (BI-RME) method (see the

Proceedings of The Fifth International Symposium
on Parallel and Distributed Computing (ISPDC'06)
0-7695-2638-1/06 $20.00 © 2006 IEEE

detailed formulation in [1, 3]). This technique provides the modal cut-off frequencies of an arbitrary waveguide from the solution of two eigenvalue problems. The first one is a generalised eigenvalue problem that models the transversal electric (TE) family of modes of the arbitrary waveguide. The structure of the corresponding matrices $\mathbf{A}$ and $\mathbf{B}$, shown in Fig. 1, presents a very sparse nature that is conveniently exploited in this work. Both matrices have a non-zero main diagonal, and a small $N \times N$ block in the right, bottom corner. Furthermore, the B matrix has two thin nonzero stripes R (with dimensions $N \times M$) and R_t ($M \times N$), in the last N rows and the last N columns. The size of the matrices is $(M+N) \times (M+N)$, but since M is far larger than N the matrices are very sparse (see [1]). This situation is given when a large number of cut-off frequencies is demanded. The transversal magnetic (TM) family of modes can be also formulated as a generalised eigenvalue problem (see [1]) with matrices $\mathbf{A}$ and $\mathbf{B}$ very similar to those explained before for the TE modes.

Here we will consider only the TE case.

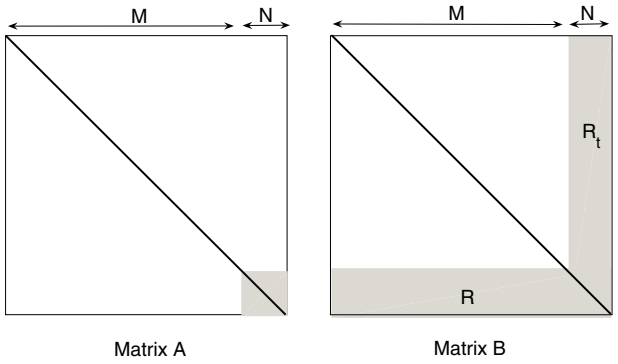

Figure 1. Structured matrices $\mathbf{A}$ and $\mathbf{B}$ for the TE problem in a ridge waveguide.

2.2. The sequential algorithm

The standard techniques for generalised eigenvalue problems is the QZ algorithm. However, as was described in [1], in this case is not efficient since it does not use the structure of the matrices.

The technique proposed in [1] by the authors is based on Lanczos algorithm [6]. This algorithm, in its most basic form, allows the computation of a reduced number of extremal eigenvalues (the largest or smallest in magnitude). However, given a real number (usually called *shift*) σ, Lanczos' algorithm can be applied to the matrix $\mathbf{W} = (\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}$. Lanczos' algorithm applied to this matrix will deliver as result the eigenvalues of the original problem closer to the shift σ. (This is called the "Shift-and-Invert" version of the Lanczos' algorithm.) The application of the Lanczos' method to this problem requires the solution of several linear systems, with $\mathbf{A} - \sigma\mathbf{B}$ as coefficient matrix. However, the structure of the matrices $\mathbf{A}$ and $\mathbf{B}$ allows a very efficient solution of these systems, through the Schur complement technique.

Using this technique, the full sequential algorithm is as follows: The interval where lie the eigenvalues desired, $[\alpha, \beta]$, is divided in many small subintervals. Then, in each subinterval, a shift (possibly the middle point) is selected, and then the "Shift-and Invert" Lanczos algorithm is applied independently to each subinterval. This will compute all the eigenvalues in each subinterval, independently of the other subintervals. The number of subintervals and its width are chosen so that the number of eigenvalues in each subinterval is not too large.

This allows to obtain all the eigenvalues in the full interval in a reasonable time and without memory problems (see [1] for all the details).

3. Parallel implementations

3.1. Algorithmic approach

Clearly, the basic idea for the parallel implementation is to distribute the subintervals among the available processors; in each subinterval, the extraction of the eigenvalues will still be carried out in a sequential way.

The first problem to face is the work load balance. If the length of each subinterval is arbitrarily chosen (e.g. $(\alpha - \beta)/p$), it is almost sure that computation time will not be equal for each single subinterval, since the eigenvalues may not be uniformly distributed along the $[\alpha, \beta]$ interval.

As shown in [1], it is possible to use the Inertia Theorem to know in advance how many eigenvalues contain a given interval $[\alpha, \beta]$. For such interval, the $\mathbf{LDL_t}$ decompositions of $\mathbf{A} - \alpha\mathbf{B}$ (equal to $\mathbf{L}^{\alpha}\mathbf{D}^{\alpha}{\mathbf{L_t}}^{\alpha}$) and $\mathbf{A} - \beta\mathbf{B}$ (equal to $\mathbf{L}^{\beta}\mathbf{D}^{\beta}{\mathbf{L_t}}^{\beta}$) can be computed with a moderated cost (again, taking profit from the structures of the matrices). Then, the number of eigenvalues in the interval is simply the number $\nu(\mathbf{D}^{\beta}) - \nu(\mathbf{D}^{\alpha})$, where $\nu(\mathbf{D})$ denotes the number of negative elements in the diagonal $\mathbf{D}$.

Thus, we can divide the original $[\alpha, \beta]$ interval into m subintervals $[\alpha_i, \beta_i]$ of different length, but containing all of them the same number of eigenvalues. Therefore, the CPU time needed to compute the eigenvalues of every subinterval is expected to be nearly constant.

Once we have computed the bounds of every $[\alpha_i, \beta_i]$ subinterval, m/p subintervals are assigned to each processor. This assignation is performed at the beginning of the algorithm, and there is no more communication among the

Proceedings of The Fifth International Symposium
on Parallel and Distributed Computing (ISPDC'06)
0-7695-2638-1/06 $20.00 © 2006 IEEE

processors until all of them have ended its work and results have been gathered.

Algorithm 1 Lanczos parallel algorithm with static organisation.
Let us suppose that m is multiple of p

1. *Apply the Inertia Theorem to the full interval $[\alpha, \beta]$ to divide it into m smaller subintervals $[\alpha_i, \beta_i]$.*
2. *Assign m/p subintervals to each processor.*
3. *For each processor*
4. *In each subinterval, compute a suitable "shift" σ*
5. *Apply Lanczos' method in each subinterval to the "shifted and inverted pencil" $\mathbf{C} = (\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}$, to obtain all the eigenvalues in the subinterval*
6. *Repeat the steps 4,5 in all subintervals*
7. *End For each processor*
8. *Gather the results from all the processors.*

3.2. Implementation details

The new proposed algorithms have been implemented in Fortran 90, making use of the Intel Fortran Compiler for Linux. OpenMP and MPI standards have been used for the shared-memory version and distributed-memory version, respectively.

We have implemented three versions of Algorithm 1:

1. MPI version of algorithm 1
2. OpenMP version of algorithm 1
3. MPI+OpenMP version of algorithm 1

In the MPI version, all the processes read the input data from disk (matrices and main interval). Then, the main interval is divided with the technique described in the previous section. Next, a distributed algorithm is executed to assign the subintervals that should be solved by each process. Once it is done, every process solves its corresponding subintervals sequentially. Then, the results are gathered by the master process. This version is oriented to distributed memory machines, although it should work as well in shared memory machine.

In the OpenMP version, only the main thread reads the input data from disk. Then, the $[\alpha, \beta]$ interval is divided by the main thread, again. Next, the subintervals are assigned and distributed among the threads. This version is designed to run on shared memory machines.

Finally, the MPI+OpenMP version combines both techniques. In the first level of parallelism, a set of p MPI processes are spawned and they execute the MPI algorithm described before. Then, in the step where each MPI process solves its m/p subintervals, a second level of parallelism is introduced. Instead of sequentially solving those intervals, a group of p' OpenMP threads are created and the m/p intervals are divided among them in the same way described in the OpenMP version. There are $p * p'$ processors working on the solution of the problem. Note that this version is a combination of the two previous ones , and has been designed to run on a cluster of SMP machines.

4. Experimental results

4.1. Description of the test environment

In order to test the performance of the three implemented versions of algorithm 1, we have chosen two different environments: an SMP Cluster and an SGI Computation Server.

The SMP Cluster consists of two Intel Xeon biprocessors running at 2.2 GHz with 4GB of RAM each one, interconnected through a Gigabit-Ethernet network.

The SGI Computation Server is an SGI Altix 3700. This machine is a cluster of 44 Itanium II tetraprocessors, although it has been designed as a ccNUMA machine [5] and therefore can be programmed as a SMP machine.

As mentioned previously, the algorithms have been designed to be included into a CAD tool of complex passive waveguide components. This kind of tools are expected to run in moderate-low cost workstations, so the SMP Cluster is the perfect testing environment. Despite of this, we have also chosen a more complex and powerful machine, the SGI Server, in order to test the algorithm performance using more expensive machines. Obviously, we will only use 4 of the 44 processors available for fair comparison purposes with the cheaper machine.

4.2. Experimental results

The following tables show the execution times of the implementations listed in section 3 for both test environments.

For the testbed, we have considered a single ridge waveguide described in [1].

$M+N$	$p=1$	$p=2$	$p=4$
5000	71.68	40.92	20.45
8000	199.26	121.22	67.98
11000	426.32	257.13	140.06
14000	772.10	413.06	221.21
17000	1247.71	655.40	367.26
20000	1685.27	1003.56	540.88

Table 1. Execution time (s) for MPI implementation at the SMP Cluster.

Proceedings of The Fifth International Symposium
on Parallel and Distributed Computing (ISPDC'06)
0-7695-2638-1/06 $20.00 © 2006 IEEE

$M+N$	$p=1$	$p=2$
5000	71.68	38.11
8000	199.26	109.78
11000	426.32	246.32
14000	772.10	419.12
17000	1247.71	646.51
20000	1685.27	963.91

Table 2. Execution time (s) for OpenMP implementation at the SMP Cluster.

$M+N$	$p=1$	$p=4$
5000	71.68	20.53
8000	199.26	61.59
11000	426.32	134.88
14000	772.10	216.84
17000	1247.71	333.86
20000	1685.27	534.69

Table 3. Execution time (s) for MPI+OpenMP implementation at the SMP Cluster.

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	25.44	18.66	14.95
8000	161.99	86.46	69.25	55.67
11000	321.68	185.16	148.37	133.35
14000	598.13	337.35	249.26	247.38
17000	893.64	494.42	405.15	351.61
20000	1259.16	665.58	556.76	532.72

Table 4. Execution time (s) for OpenMP implementation at the SGI Server.

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	23.69	16.17	13.09
8000	161.99	86.34	60.85	49.24
11000	321.68	172.88	117.61	91.53
14000	598.13	310.42	217.38	170.07
17000	893.64	498.16	304.24	241.64
20000	1259.16	658.08	446.46	349.44

Table 5. Execution time (s) for MPI implementation at the SGI Server.

4.3. Analysis of the experimental results

The previous results show that the method described in section 2 parallelises extremely well in affordable machines. The key points for this good behaviour are the application of Inertia Theorem to ensure a good work-load balance, as well as the absence of communications during the execution of the algorithm.

The different versions of the developed algorithms differ in the parallel programming standard used: MPI, OpenMP, or both of them. Both standards offer good performance and the final choice depends more on the machine architecture rather than on the sequential algorithm characteristics.

$M+N$	OpenMP	MPI
5000	1.88	1.75
8000	1.82	1.64
11000	1.73	1.66
14000	1.84	1.87
17000	1.93	1.90
20000	1.75	1.68

Table 6. Speed-up @ the SMP Cluster. Comparative study between OpenMP and MPI versions ($p=2$).

Table 6 shows the speed-up of MPI and OpenMP versions in a biprocessor board (one of the nodes of the SMP Cluster). OpenMP results are slightly better than MPI ones. This result was expected because OpenMP can take more advantage of the shared memory architecture of the machine.

$M+N$	MPI+OpenMP	MPI
5000	3.49	3.51
8000	3.24	2.93
11000	3.16	3.04
14000	3.56	3.49
17000	3.74	3.40
20000	3.15	3.12

Table 7. Speed-up @ the SMP Cluster. Comparative study between MPI+OpenMP and MPI versions ($p=4$).

Table 7 shows the speed-up of MPI and MPI+OpenMP versions in a cluster of 2 biprocessor boards. In this kind of environments with two levels of parallelism (shared memory at each node and distributed memory for the global view of the machine) the combination of MPI and OpenMP standards show better results than the use of MPI only. Again, OpenMP is taking advantage of shared memory features of the machine while MPI is not doing so.

Table 8 shows the speed-up of MPI and OpenMP versions at the SGI Server. In this machine, the MPI ver-

Proceedings of The Fifth International Symposium
on Parallel and Distributed Computing (ISPDC'06)
0-7695-2638-1/06 $20.00 © 2006 IEEE

version	$p = 2$	$p = 3$	$p = 4$
MPI version	1,91	2,82	3,60
OpenMP version	1,89	2,26	2,36

Table 8. Comparative analysis between OpenMP and MPI versions @ SGI Server ($M + N = 20000$).

sion scales better than OpenMP version. This rather surprising result is due to the scheduling policy. When the batch system runs the parallel algorithm, it can schedule the p threads/processes to different boards. With the MPI algorithm this does not create problems, since each process owns all the necessary data to perform its part of the algorithm. However, for the OpenMP implementation it is different, because all the threads need to access master thread's memory. This would create accesses to memory placed in a different board, which shall slow down the algorithm. Obviously, the problem worsens as the number of threads increases.

5. Conclusions

Three parallel implementations of a Lanczos-based method for solving a generalised eigenvalue problem have been successfully developed. The problem has got several distinct characteristics: matrices are sparse and structured, and the search of eigenvalues is reduced to a fixed interval.

The proposed technique parallelises very well and any of the implementations present very good speed-up even for a small number of processors.

OpenMP is the best choice for parallel programming of biprocessors boards (and any SMP environment). For NUMA systems, it is concluded that OpenMP may present some problems and its use should be studied carefully.

Multi level programming (MPI + OpenMP) is the best choice for hybrid machines (those with two levels of parallelism), since this paradigm can take advantage of both shared and distributed memory features of the machine.

Finally, we can conclude that execution times in both machines are not too different, while speed-up is clearly better at the SMP Cluster. So, in this case, the performance-cost ratio is clearly better for the SMP Cluster.

Acknowledgement

Contract/grant sponsor: partially supported by Ministerio de Educación y Ciencia, Spanish Government, and FEDER funds, European Commission; contract/grant number: TIC2003-08238- C02-02, and by Programa de Incentivo a la Investigacion UPV-Valencia 2005 Project 005522

References

[1] García V.M., Vidal A., Boria V.E., Vidal A.M.: Efficient and accurate waveguide mode computation using BI-RME and Lanczos method; Int. Journal for Numerical Methods in Engineering. DOI:10.1002/nme.1520 (2005)

[2] Vidal A.M., Vidal A., Boria V.E., García V.M.: Parallel computation of arbitrarily shaped waveguide modes using BI-RME and Lanczos method; Submitted to Int. Journal for Numerical Methods in Engineering. (2006)

[3] Conciauro G., Bressan M., Zuffada, C.: Waveguide modes via an integral equation leading to a linear matrix eigenvalue problem; IEEE Transactions on Microwave Theory and Techniques. (1984)

[4] Snir M., Otto S., Huss-Lederman S., Walker D. and Dongarra J.: MPI: The Complete Reference; MIT Press, (1996)

[5] Grbic A: Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors; Phd Thesis, University of Toronto (2003)

[6] A. Ruhe. *Generalized Hermitian Eigenvalue Problem; Lanczos Method, In Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, Philadelphia, first edition, 2000.

Scalable Computing: Practice and Experience
Volume 8, Number 3, pp. 263–270. http://www.scpe.org
ISSN 1895-1767

ROBUST PARALLEL IMPLEMENTATION OF A LANCZOS-BASED ALGORITHM FOR AN STRUCTURED ELECTROMAGNETIC EIGENVALUE PROBLEM

MIGUEL O. BERNABEU*, MÁRIAM TARONCHER†, VÍCTOR M. GARCÍA‡, AND ANA VIDAL§

Abstract. This paper describes a parallel implementation of a Lanczos-based method to solve generalised eigenvalue problems related to the modal computation of arbitrarily shaped waveguides. This efficient implementation is intended for execution mainly in moderate-low cost workstations (2 to 4 processors). The problem under study has several features: the involved matrices are sparse with a certain structure, and all the eigenvalues needed are contained in a given interval. The novel parallel algorithms proposed show excellent speed-up for small number of processors.

Key words. large eigenvalue problem, structured matrices, microwaves

1. Introduction and examples. This paper is focused on the parallelisation of a Lanczos-based method for the solution of the following generalised eigenvalue problem: Given a symmetric pencil $\mathbf{A}x = \lambda\mathbf{B}x$, find all the generalised eigenvalues (and the corresponding eigenvectors) comprised in a given interval. This interval contains a large number of eigenvalues.

An efficient sequential method was already proposed in [1]. However, when the number of desired eigenvalues is very large, the execution time is still too long. A first parallel algorithm was recently introduced in [2], using MPI and a distributed-memory approach. The results presented in that paper show that the method parallelises extremely well.

A code based in the proposed technique will be included in a CAD tool for design of passive waveguide components. However, this CAD tool will usually run in low cost workstations or, at most, small PC clusters. For these small systems, a different approach should be chosen.

Therefore, the main goal of this paper is to explore different parallel programming approaches for the implementation of the sequential technique described in [1], in low cost workstations and small clusters.

Three different approaches have been examined: First, we have designed an OpenMP version of the Lanczos algorithm to take advantage of two-processor machines. Next, we implemented a version for distributed memory machines using MPI (Message Passing Interface), to execute it on clusters of PCs. Finally, a mixed approach was proposed in order to achieve optimum performance on clusters of two-processors.

A number of modifications have been carried out lately in the algorithm, to improve the reliability of the code, these shall be described as well. The main corrections have been i) the inclusion of ARPACK [7] routines for the extraction of all the generalised eigenvalues in a small subinterval, ii) the correction of the algorithm for balancing workload, and iii) the improvement of the linear solver, formerly based in the LU-Schur complement and now based on the LDL_t decomposition.

The paper is organised as follows: first, we will briefly outline the sequential problem (described in [1]), including the algorithmic modifications. Then, the new parallelisation schemes will be completely described, taking into account the different proposed options: i. e. MPI, OpenMP, MPI+OpenMP and so on. Finally, some numerical results are shown, and then the conclusions of this work are given.

2. Problem Description and Sequential Algorithm.

2.1. The electromagnetic problem. In this study, the efficient and accurate modal computation of arbitrary waveguides is based on the Boundary Integral - Resonant Mode Expansion (BI-RME) method (see the detailed formulation in [1, 3]). This technique provides the modal cut-off frequencies of an arbitrary waveguide from the solution of two eigenvalue problems. The first one is a generalised eigenvalue problem that models the transversal electric (TE) family of modes of the arbitrary waveguide. The structure of the corresponding matrices $\mathbf{A}$ and $\mathbf{B}$, shown in Fig. 2.1, presents a very sparse nature that is conveniently exploited in this work.

*Dpt. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain, mbernabeu@dsic.upv.es

† Dpt. de Comunicaciones, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain, matacal@iteam.upv.es

‡Dpt. de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain, vmgarcia@dsic.upv.es

§Dpt. de Comunicaciones, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, Spain, avidal@dcom.upv.es

Both matrices have a non-zero main diagonal, and a small $N \times N$ block in the right, bottom corner. Furthermore, the B matrix has two thin nonzero stripes $\mathbf{R}$ (with dimensions $N \times M$) and $\mathbf{R_t}$ ($M \times N$), in the last N rows and the last N columns. The size of the matrices is $(M+N) \times (M+N)$, but since M is far larger than N the matrices are very sparse (see [1]). This situation is given when a large number of cut-off frequencies is demanded. The transversal magnetic (TM) family of modes can be also formulated as a generalised eigenvalue problem (see [1]) with matrices $\mathbf{A}$ and $\mathbf{B}$ very similar to those explained before for the TE modes.

Here we will consider only the TE case.

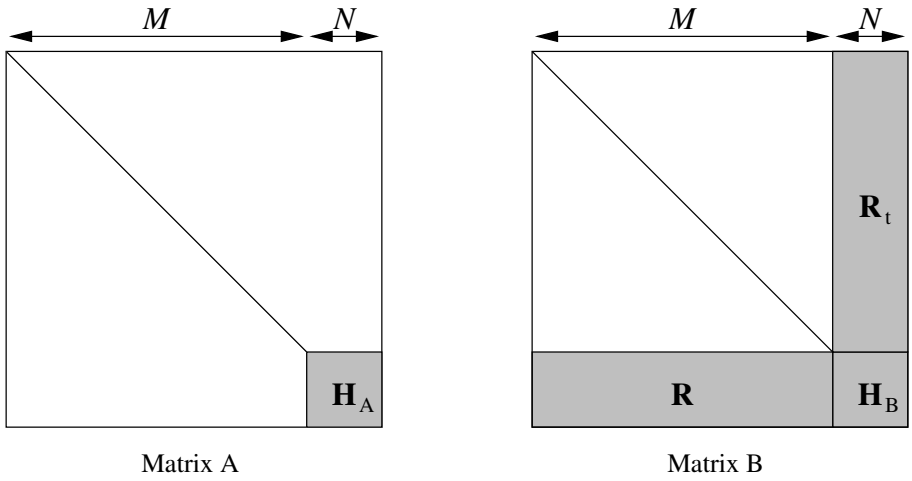

FIG. 2.1. *Structured matrices* $\mathbf{A}$ *and* $\mathbf{B}$ *for the TE problem in a ridge waveguide.*

2.2. The sequential algorithm.

2.2.1. Shift-and-Invert Lanczos' algorithm. The standard techniques for generalised eigenvalue problems is the QZ algorithm. However, as was described in [1], in this case is not efficient since it does not use the structure of the matrices.

The technique proposed in [1] by the authors is based on Lanczos algorithm [6]. This algorithm, in its most basic form, allows the computation of a reduced number of extremal eigenvalues (the largest or smallest in magnitude). However, given a real number (usually called *shift*) σ, Lanczos' algorithm can be applied to the matrix $\mathbf{W} = (\mathbf{A} - \sigma\mathbf{B})^{-1}\mathbf{B}$. Lanczos' algorithm applied to this matrix will deliver as result the eigenvalues of the original problem closer to the shift σ. (This is called the "Shift-and-Invert" version of the Lanczos' algorithm.) The application of the Lanczos' method to this problem requires the solution of several linear systems, with $\mathbf{A} - \sigma\mathbf{B}$ as coefficient matrix. However, the structure of the matrices $\mathbf{A}$ and $\mathbf{B}$ allows a very efficient solution of these systems, using the Schur complement method. This method, described in [1] for this problem, was based in the LU decomposition; one of the algorithmic improvements mentioned above has been to change the LU-based technique to a LDL_t based algorithm, described next.

2.2.2. LDL_t **decomposition.** Let us now find out how is the $\mathbf{LDL_t}$ decomposition of the matrices in our case. For a matrix $(\mathbf{A} - \sigma\mathbf{B})$ with $\mathbf{A}$ and $\mathbf{B}$ as above, we can write

$$\begin{aligned}\mathbf{A} - \sigma\mathbf{B} &= \begin{pmatrix} \mathbf{U}_\sigma & \mathbf{R}_{\sigma t} \\ \mathbf{R}_\sigma & \mathbf{H}_\sigma \end{pmatrix} = \begin{pmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{F} & \mathbf{T} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{D}_l & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_s \end{pmatrix} \cdot \begin{pmatrix} \mathbf{D_t} & \mathbf{F_t} \\ \mathbf{0} & \mathbf{T_t} \end{pmatrix} \\ &= \begin{pmatrix} \mathbf{D} \cdot \mathbf{D}_l \cdot \mathbf{D_t} & \mathbf{D} \cdot \mathbf{D}_l \cdot \mathbf{F_t} \\ \mathbf{F} \cdot \mathbf{D}_l \cdot \mathbf{D}_t & \mathbf{F} \cdot \mathbf{D}_l \cdot \mathbf{F_t} + \mathbf{T} \cdot \mathbf{D}_s \cdot \mathbf{T_t} \end{pmatrix} \end{aligned} \tag{2.1}$$

where the structure of matrix $\mathbf{A} - \sigma\mathbf{B}$ is identical to that of matrix $\mathbf{B}$ (Figure 2.1).

It is easy to check that we can take $\mathbf{D}$ as the identity matrix (since $\mathbf{U}_\sigma$ is diagonal), so that equating parts of this equation we arrive to the following procedure to compute the $\mathbf{LDL_t}$ decomposition:

1. Take $\mathbf{D}_l$ equal to $\mathbf{U}_\sigma$.
2. $\mathbf{F} = \mathbf{R}_\sigma \cdot \mathbf{D}_l^{-1}$ (trivial, since $\mathbf{D}_l$ is diagonal).
3. $\mathbf{T}$ and $\mathbf{D}_s$ are obtained computing the $\mathbf{LDL_t}$ decomposition of $\mathbf{H}_\sigma - \mathbf{F} \cdot \mathbf{D}_l \cdot \mathbf{F_t}$, through the LAPACK routine *dsytrf* .

2.2.3. Main interval decomposition. As we have mentioned before, the shift-and-invert version of the Lanczos' algorithm computes a subset of the spectrum centred in the shift point. The number of eigenvalues required will determine the number of iterations of the Lanczos' algorithm and its spatial cost [7]. Obviously, we cannot apply the Lanczos' algorithm to the main interval $[\alpha, \beta]$ where all the desired eigenvalues lie. The original problem should be split into many smaller ones to ensure the optimal performance of the Lanczos' algorithm.

As shown in [1], it is possible to use the Inertia Theorem to know in advance how many eigenvalues contain a given interval $[\alpha, \beta]$. For such interval, the $\mathbf{LDL_t}$ decompositions of $\mathbf{A} - \alpha\mathbf{B}$ (equal to $\mathbf{L}^\alpha\mathbf{D}^\alpha{\mathbf{L_t}}^\alpha$) and $\mathbf{A} - \beta\mathbf{B}$ (equal to $\mathbf{L}^\beta\mathbf{D}^\beta{\mathbf{L_t}}^\beta$) can be computed with a moderated cost (as described above). Then, the number of eigenvalues in the interval is simply the number $\nu(\mathbf{D}^\beta) - \nu(\mathbf{D}^\alpha)$, where $\nu(\mathbf{D})$ denotes the number of negative elements in the diagonal $\mathbf{D}$. It must be taken into account that the diagonal returned by *dsytrf* may not be completely "diagonal"; instead, it can be diagonal with $1 * 1$ and $2 * 2$ blocks, as a consequence of the special pivoting strategy. In this case, the eigenvalues of this special diagonal matrix can be easily found (solving the characteristic equation for the 2*2 blocks), which allows to compute the inertia quite efficiently anyway.

Thus, we can divide the original $[\alpha, \beta]$ interval into m subintervals $[\alpha_i, \beta_i]$ of different length, but containing nearly the same number of eigenvalues, and where the number of eigenvalues in each subinterval is known exactly. Therefore, the CPU time needed to compute the eigenvalues of every subinterval is expected to be nearly constant.

2.2.4. Sequential algorithm. The full sequential algorithm is as follows: The interval where lie the desired eigenvalues, $[\alpha, \beta]$, is divided in many small subintervals. Then, in each subinterval, a shift (possibly the middle point) is selected, and then the "Shift-and Invert" Lanczos algorithm is applied independently to each subinterval. This will compute all the eigenvalues in each subinterval, independently of the other subintervals. The number of subintervals and its width are chosen so that the number of eigenvalues on each subinterval is not too large.

This allows to obtain all the eigenvalues in the full interval in a reasonable time and without memory problems (see [1] for all the details).

ALGORITHM 1. **Sequential overall algorithm**.

INPUT: matrices $\mathbf{A}$ *and* $\mathbf{B}$*, the main subinterval* $[\alpha, \beta]$ *and the maximum number of eigenvalues per subinterval*
OUTPUT: eigenvalues of the pair $(\mathbf{A}, \mathbf{B})$ *contained in* $[\alpha, \beta]$ *ant its corresponding eigenvectors*

1. *Apply the Inertia Theorem to the full interval* $[\alpha, \beta]$ *to divide it into m smaller subintervals* $[\alpha_i, \beta_i]$

2. *for every subinterval* $[\alpha_i, \beta_i]$
4. $\sigma = (\beta_i - \alpha_i)/2$
5. *Apply Lanczos' shift-and-invert method to extract the eigenvalues closer to* σ *and its eigenvectors*
6. *end for*

7. *return eigenvalues and eigenvectors*

In the latest versions of the code, the robustness has been improved by using the ARPACK [7] routines for the symmetric generalised eigenvalue problem *dsaupd*. This routine is faster and safer than our previous versions of the Lanczos algorithm.

3. Parallel implementations.

3.1. Algorithmic approach. Clearly, the basic idea for the parallel implementation is to distribute the subintervals among the available processors; in each subinterval, the extraction of the eigenvalues will still be carried out in a sequential way.

Once we have computed the bounds of every $[\alpha_i, \beta_i]$ subinterval, m/p subintervals are assigned to each processor. This assignation is performed at the beginning of the algorithm, and there is no more communication among the processors until all of them have ended its work and results have been gathered.

As we have mentioned in Section 2.2.3, the CPU time needed to extract the eigenvalues of every subinterval is expected to be nearly constant. Thus, just distributing them among the available processors the work load balance is expected to be close to the optimal.

ALGORITHM 2. **Parallel overall algorithm**.

INPUT: matrices $\mathbf{A}$ *and* $\mathbf{B}$*, the main interval* $[\alpha, \beta]$ *and the maximum number of eigenvalues per subinterval*
OUTPUT: eigenvalues of the pair $(\mathbf{A}, \mathbf{B})$ *contained in* $[\alpha, \beta]$ *ant its corresponding eigenvectors*

Let us suppose that m *is multiple of* p

1. At the master processor
2. Apply the Inertia Theorem to the full interval $[\alpha, \beta]$ *to divide it into* m *smaller subintervals* $[\alpha_i, \beta_i]$
3. Assign m/p *subintervals to each processor*
4. End at the master processor

5. For each processor
6. for every assigned subinterval $[\alpha_i, \beta_i]$
7. $\sigma = (\beta_i - \alpha_i)/2$
8. Apply Lanczos' shift-and-invert method to extract the eig eigenvalues closer to σ *and its eigenvectors*
9. end for

10. Send eigenvalues and eigenvectors to the master processor
11. End for each processor

12. At the master processor
13. Gather the results from all the processors
14. End at the master processor

3.2. Implementation details. The new proposed algorithms have been implemented in Fortran 90, making use of the Intel Fortran Compiler for Linux. OpenMP and MPI standards have been used for the shared-memory version and distributed-memory version, respectively. In addition, BLAS and LAPACK [8] have been used whenever it was possible.

We have implemented three versions of Algorithm 2:

1. MPI version of algorithm 2
2. OpenMP version of algorithm 2
3. MPI+OpenMP version of algorithm 2

In the MPI version, all the processes read the input data from disk (matrices and main interval). Then, the main interval is divided with the technique described in the previous section. Next, a distributed algorithm is executed to assign the subintervals that should be solved by each process. Once it is done, every process solves its corresponding subintervals sequentially. Then, the results are gathered by the master process. This version is oriented to distributed memory machines, although it should work as well in shared memory machine.

In the OpenMP version, only the main thread reads the input data from disk. Then, the $[\alpha, \beta]$ interval is divided by the main thread, again. Next, the subintervals are assigned and distributed among the threads. This version is designed to run on shared memory machines.

Finally, the MPI+OpenMP version combines both techniques. In the first level of parallelism, a set of p MPI processes are spawned and they execute the MPI algorithm described before. Then, in the step where each MPI process solves its m/p subintervals, a second level of parallelism is introduced. Instead of sequentially solving those intervals, a group of p' OpenMP threads are created and the m/p intervals are divided among them in the same way described in the OpenMP version. There are $p * p'$ processors working on the solution of the problem. Note that this version is a combination of the two previous ones , and has been designed to run on a cluster of SMP machines.

4. Experimental results.

4.1. Description of the test environment. In order to test the performance of the three implemented versions of algorithm 2, we have chosen two different environments: an SMP Cluster and an SGI Computation Server.

The SMP Cluster consists of two Intel Xeon bi-processors running at 2.2 GHz with 4GB of RAM each one, interconnected through a Gigabit-Ethernet network.

The SGI Computation Server is an SGI Altix 3700. This machine is a cluster of 44 Itanium II tetraprocessors, although it has been designed as a ccNUMA machine [5] and therefore can be programmed as a SMP machine.

As mentioned previously, the algorithms have been designed to be included into a CAD tool of complex passive waveguide components. This kind of tools are expected to run in moderate-low cost workstations, so the SMP Cluster is the perfect testing environment. Despite of this, we have also chosen a more complex and powerful machine, the SGI Server, in order to test the algorithm performance using more expensive machines. Obviously, we will only use 4 of the 44 processors available for fair comparison purposes with the cheaper machine.

4.2. Experimental results. The following tables show the execution times of the implementations listed in section 3 for both test environments.

For the testbed, we have considered a single ridge waveguide described in [1].

TABLE 4.1
Execution time (s) for MPI implementation at the SMP Cluster.

$M+N$	$p=1$	$p=2$	$p=4$
5000	71.68	40.92	20.45
8000	199.26	121.22	67.98
11000	426.32	257.13	140.06
14000	772.10	413.06	221.21
17000	1247.71	655.40	367.26
20000	1685.27	1003.56	540.88

TABLE 4.2
Execution time (s) for OpenMP implementation at the SMP Cluster.

$M+N$	$p=1$	$p=2$
5000	71.68	38.11
8000	199.26	109.78
11000	426.32	246.32
14000	772.10	419.12
17000	1247.71	646.51
20000	1685.27	963.91

4.3. Analysis of the experimental results. The previous results show that the method described in section 2 parallelises extremely well in affordable machines. The key points for this good behaviour are the

TABLE 4.3
Execution time (s) for MPI+OpenMP implementation at the SMP Cluster.

$M+N$	$p=1$	$p=4$
5000	71.68	20.53
8000	199.26	61.59
11000	426.32	134.88
14000	772.10	216.84
17000	1247.71	333.86
20000	1685.27	534.69

TABLE 4.4
Execution time (s) for OpenMP implementation at the SGI Server.

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	25.44	18.66	14.95
8000	161.99	86.46	69.25	55.67
11000	321.68	185.16	148.37	133.35
14000	598.13	337.35	249.26	247.38
17000	893.64	494.42	405.15	351.61
20000	1259.16	665.58	556.76	532.72

TABLE 4.5
Execution time (s) for MPI implementation at the SGI Server.

$M+N$	$p=1$	$p=2$	$p=3$	$p=4$
5000	44.14	23.69	16.17	13.09
8000	161.99	86.34	60.85	49.24
11000	321.68	172.88	117.61	91.53
14000	598.13	310.42	217.38	170.07
17000	893.64	498.16	304.24	241.64
20000	1259.16	658.08	446.46	349.44

application of Inertia Theorem to ensure a good work-load balance, as well as the absence of communications during the execution of the algorithm.

The different versions of the developed algorithms differ in the parallel programming standard used: MPI, OpenMP, or both of them. Both standards offer good performance and the final choice depends more on the machine architecture rather than on the sequential algorithm characteristics.

TABLE 4.6
Speed-up @ the SMP Cluster. Comparative study between OpenMP and MPI versions ($p=2$).

$M+N$	OpenMP	MPI
5000	1.88	1.75
8000	1.82	1.64
11000	1.73	1.66
14000	1.84	1.87
17000	1.93	1.90
20000	1.75	1.68

Table 4.6 shows the speed-up of MPI and OpenMP versions in a two-processor board (one of the nodes of the SMP Cluster). OpenMP results are slightly better than MPI ones. This result was expected because OpenMP can take more advantage of the shared memory architecture of the machine.

Table 4.7 shows the speed-up of MPI and MPI+OpenMP versions in a cluster of 2 two-processor boards. In this kind of environments with two levels of parallelism (shared memory at each node and distributed memory

TABLE 4.7
Speed-up @ the SMP Cluster. Comparative study between MPI+OpenMP and MPI versions ($p = 4$).

$M + N$	MPI+OpenMP	MPI
5000	3.49	3.51
8000	3.24	2.93
11000	3.16	3.04
14000	3.56	3.49
17000	3.74	3.40
20000	3.15	3.12

for the global view of the machine) the combination of MPI and OpenMP standards show better results than the use of MPI only. Again, OpenMP is taking advantage of shared memory features of the machine while MPI is not doing so.

TABLE 4.8
Comparative analysis between OpenMP and MPI versions @ SGI Server ($M + N = 20000$).

version	$p = 2$	$p = 3$	$p = 4$
MPI version	1,91	2,82	3,60
OpenMP version	1,89	2,26	2,36

Table 4.8 shows the speed-up of MPI and OpenMP versions at the SGI Server. In this machine, the MPI version scales better than OpenMP version. This rather surprising result is due to the scheduling policy. When the batch system runs the parallel algorithm, it can schedule the p threads/processes to different boards. With the MPI algorithm this does not create problems, since each process owns all the necessary data to perform its part of the algorithm. However, for the OpenMP implementation it is different, because all the threads need to access master thread's memory. This would create accesses to memory placed in a different board, which shall slow down the algorithm. Obviously, the problem worsens as the number of threads increases.

5. Conclusions. Three parallel implementations of a Lanczos-based method for solving a generalised eigenvalue problem have been successfully developed. The problem has got several distinct characteristics: matrices are sparse and structured, and the search of eigenvalues is reduced to a fixed interval.

The proposed technique parallelises very well and any of the implementations present very good speed-up even for a small number of processors.

OpenMP is the best choice for parallel programming of two-processors boards (and any SMP environment). For NUMA systems, it is concluded that OpenMP may present some problems and its use should be studied carefully.

Multi level programming (MPI + OpenMP) is the best choice for hybrid machines (those with two levels of parallelism), since this paradigm can take advantage of both shared and distributed memory features of the machine.

Finally, we can conclude that execution times in both machines are not too different, while speed-up is clearly better at the SMP Cluster. So, in this case, the performance-cost ratio is clearly better for the SMP Cluster.

Acknowledgement. Contract/grant sponsor: partially supported by Ministerio de Educación y Ciencia, Spanish Government, and FEDER funds, European Commission; contract/grant number: TIC2003-08238- C02-02, and by Programa de Incentivo a la Investigacion UPV-Valencia 2005 Project 005522

REFERENCES

[1] GARCÍA V. M., VIDAL A., BORIA V. E., VIDAL A. M.: *Efficient and accurate waveguide mode computation using BI-RME and Lanczos method,* Int. Journal for Numerical Methods in Engineering. DOI:10.1002/nme.1520 (2005).

[2] VIDAL A. M., VIDAL A., BORIA V. E., GARCÍA V. M.: *Parallel computation of arbitrarily shaped waveguide modes using BI-RME and Lanczos method,* Submitted to Int. Journal for Numerical Methods in Engineering. (2006).

[3] CONCIAURO G., BRESSAN M., ZUFFADA, C.: *Waveguide modes via an integral equation leading to a linear matrix eigenvalue problem,* IEEE Transactions on Microwave Theory and Techniques. (1984).

[4] Snir M., Otto S., Huss-Lederman S., Walker D. and Dongarra J.: MPI: The Complete Reference; MIT Press, (1996).
[5] Grbic A.: Assessment of Cache Coherence Protocols in Shared-memory Multiprocessors; Phd Thesis, University of Toronto (2003).
[6] A. Ruhe.: *Generalized Hermitian Eigenvalue Problem; Lanczos Method,* In Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide. SIAM, Philadelphia, first edition, (2000).
[7] R. B. Lehoucq, D. C. Sorensen, and C. Yang: *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods* SIAM, Philadelphia, (1998).
[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen: *LAPACK Users' Guide*. SIAM, Philadelphia, (1999).

Edited by: ...
Received: ...
Accepted: ...

Static versus dynamic heterogeneous parallel schemes to solve the symmetric tridiagonal eigenvalue problem

Miguel O. Bernabeu
Dept. de Sistemas Informáticos
y Computación
Universidad Politécnica de Valencia
Camino de Vera S/N. 46022 Valencia
Spain

Antonio M. Vidal
Dept. de Sistemas Informáticos
y Computación
Universidad Politécnica de Valencia
Camino de Vera S/N. 46022 Valencia
Spain

Abstract: Computation of the eigenvalues of a symmetric tridiagonal matrix is a problem of great relevance. Many linear algebra libraries provide subroutines for solving it. But none of them is oriented to be executed in heterogeneous distributed memory multicomputers. In this work we focus on this kind of platforms. Two different load balancing schemes are presented and implemented. The experimental results show that only the algorithms that take into account the heterogeneity of the system when balancing the workload obtain optimum performance. This fact justifies the need of implementing specific load balancing techniques for heterogeneous parallel computers.

Key–Words: Symmetric tridiagonal eigenvalue problem, heterogeneous parallel computing, load balancing

1 Introduction

Computation of the eigenvalues of a symmetric tridiagonal matrix is a problem of great relevance in numerical linear algebra and in many engineering fields, mainly due to two reasons: first, this kind of matrices arises in the discretisation of certain engineering problems and secondly, and more important, this operation is the main computational kernel in the computation of the eigenvalues of any symmetric matrix when tridiagonalisation techniques are used as a previous step.

Nowadays, there are a large amount of eigenvalue computation algorithms that exploit the tridiagonality properties of the matrix. Four main techniques can be found in the specialised literature to solve this problem: QR iteration, homotopy method, bisection and multisection methods and divide and conquer techniques. None of them is clearly superior to the rest since every one presents exclusive advantages, for example: computing all matrix eigenvalues or just a defined subset of them, precision of the results or simultaneous eigenvector computation. See [1] for an exhaustive comparison.

In [2] Badía and Vidal proposed two parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem on distributed memory multicomputers, including a deep study of the two step bisection algorithm. In that work, special emphasis was put in load balancing since this is the main difficulty when parallelising the bisection algorithm for the computation of the eigenvalues of a symmetric tridiagonal matrix.

Both, ScaLAPACK subroutines and those presented in [2] achieve good performance in homogeneous distributed memory multicomputers. We can define an homogeneous distributed memory multicomputer as a distributed memory multicomputer where all the processors are equal in computing and communication capabilities. In this work we focus on heterogeneous distributed memory multicomputers, those formed by processors with different computing and communication capabilities. These kind of platforms are expected to be the best solution in order to achieve great performance/cost ratio, to reuse obsolete computational resources or simply to obtain the maximum performance from several powerful computers with different architectures but able to work coordinately.

Parallel numerical linear algebra libraries, like ScaLAPACK, do not take into account the possible heterogeneity of the hardware and, for that reason, their performance considerably decrease when working in this kind of systems. There is a big gap in this context and, nowadays, the design of parallel linear algebra libraries for heterogeneous architectures is a must. The study of computational kernels and basic algorithms is the previous step to achieve this objec-

tive. In this work we present algorithms for the computation of the eigenvalues of a symmetric tridiagonal matrix which attain good performance in heterogeneous multicomputer, analysing different load balancing strategies and different problem instances.

The rest of the paper is organised as follows: in the next section we present the mathematical description of the problem and the sequential algorithm implemented in the solution. Section 3 describes the heterogeneous computational model used. In Section 4, the different parallel schemes used are described. In Section 5, experimental results are presented. Finally, the main conclusions of the work are given in Section 6.

2 Problem description and proposed solution

2.1 Problem definition

Let T be a symmetric tridiagonal matrix $T \in \mathbb{R}^{nxn}$, defined as follows

$$T = \begin{bmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_1 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{bmatrix} \quad (1)$$

The eigenvalues of T are the n roots of its characteristic polynomial $p(z) = det(zI - T)$. The set of these roots is called the spectrum and is denoted by $\lambda(T)$.

It is possible to compute specific eigenvalues of a symmetric matrix by using the LDL^T factorization and exploiting the Sylvester inertia theorem. If

$$A - \mu I = LDL^T \quad A = A^T \in R^{nxn}$$

is the LDL^T factorization of $A - \mu I$ with $D = diag(d_1, \ldots, d_n)$, then the number of negative d_i equals the number of $\lambda(A)$ that are smaller than μ [6].

Sequence $(d_1, \ldots, d_n)$ can be computed by using the following recurrence, where $d_i = q_i(c), i = 1, \ldots, n$ for a given c:

$$\begin{cases} q_0(c) = 1, \quad q_1(c) = a_1 - c \\ q_i(c) = (a_i - c) - \frac{b_{i-1}^2}{q_{i-1}(c)} \quad i : 2, 3, \ldots, n \end{cases}$$

Thanks to this result it is possible to define a function $neg_n(c)$ that for any value of c computes the number of eigenvalues smaller than c. With this function it is easy to implement a bisection algorithm that isolates eigenvalues of T.

The bisection algorithm needs, for initialisation purpose, an initial interval $[a, b]$ which contains all the eigenvalues of matrix T. The Gershgorin circle theorem can be used to calculate it.

So, based on the initial interval $[a, b]$ and through the bisection algorithm is possible to isolate the m subintervals $]lb_i, ub_i]$ which contain a number of eigenvalues $v \leq max_v al$ and will be used as the input for the next step of the algorithm.

The importance of this step lies in the fact that it will help us to discard parts of the real line where no eigenvalue is located and therefore to reduce the number of iterations of the extraction methods used in the following step. In addition, the isolation step will be used to balance the workload of the parallel algorithms presented in Section 4, since the initial problem is divided into m subproblems with similar workload, susceptible to be solved in parallel.

The second step of the algorithm gets as input the m subintervals $]lb_i, ub_i]$ obtained in the previous step to compute m eigenvalues of matrix T.

There are several alternatives for the eigenvalues extraction:

1. To apply again the bisection method described in previous subsection.
2. To use a fast convergence method like Newton or Laguerre [1].
3. To use standard computational kernels like LAPACK and let them choose the best method for eigenvalue extraction. These subroutines are expected to efficiently implement the sequential solution of the problem.

In this work, we have chosen to use LAPACK subroutines, specifically the driver subroutine $dstevr$ which can compute the eigenvalues of a tridiagonal symmetric matrix contained into an interval $]vl, vu]$.

2.2 Test matrices

The bisection algorithm above described is problem-dependent, because the number of iterations to reach each eigenvalue with a specified precision could be different. Thus, the behaviour of the algorithm depends on the distribution of the eigenvalues along the spectrum. In addition, the presence of clusters of eigenvalues or hidden eigenvalues considerably increases the extraction time, see [2].

Therefore, in order to perform a correct experimental analysis of the algorithms implemented, a suitable set of test matrices should be chosen. In our case,

we have chosen two kinds of matrices that present different eigenvalue distribution characteristics, so it can affect the performance of the algorithm.

Table 1 shows matrices used.

3 Heterogeneous computational model

3.1 PC model description

The following theoretical machine model is called PC (Power-Communications). Let be a set of p processors interconnected via a communication network. This model is expected to evaluate the power of each processor as well as the communication capabilities of the network.

First of all, a power vector P_t that summarizes the relative power of each processor (related to the global machine power) is defined. This relative power depends on the operation and on the problem size, so there is a vector P_t for each pair of operation and problem size. However, we consider here that the power vector does not depend on time.

Secondly, the communication model used defines the time needed to send n bytes from processor i to processor j as $T_{ij}(n) = \beta + n\tau$, where β stands for network latency and τ is the bandwidth inverse. In order to summarize the model two matrices B_c and T_c are defined. The (i, j) entry of each matrix represents the β or τ applicable to the communication from processor i to j. We consider also here that both matrices do not depend on time.

3.2 Model implementation

The cluster used to evaluate the model and to run the parallel algorithms consists of four machines with six processors. They are:

- Intel Pentium IV at 3.0 GHz with 1 MB of cache and 1 GB of main memory.
- Intel Xeon two-processor at 2.2 GHz with 512 KB of cache and 4 GB of main memory.
- Intel Xeon two-processor at 2.2 GHz with 512 KB of cache and 4 GB of main memory.
- Intel Pentium IV at 1,6 GHz with 256 KB of cache and 1GB of main memory.

A Gigabit Ethernet network is used to interconnect the six machines with 1 Gbit/s of theoretical bandwidth. Note that communications between each CPU of the two-processors boards have been evaluated with the same model.

Despite the algorithms have been implemented and evaluated in this machine, they only depend on the theoretical model. So, they can be executed in any other distributed memory multicomputer with similar predictable performance, under the condition of being evaluated with the previous model.

3.3 Evaluation

Tables 2, 3, 4 and 5 show the result of the evaluation of the cluster described before, following the PC model. Tables 2 and 3 show the power vector P_t obtained in the computation of eigenvalues of uniform spectrum matrices and Wilkinson matrices with different sizes. Tables 4 and 5 show the matrices B_c and T_c obtained in the same experiments.

As it can be observed, the variations of the vector power P_t with the size of the problem are very small. This is a characteristic of this problem, because only two vectors (main diagonal and subdiagonal) have to be stored in memory. This may be different with problems which require more memory space.

4 Heterogeneous parallel schemes

4.1 Available alternatives

Among different techniques proposed in the literature (see [2]) to parallelise the bisection method, probably the most effective is the computation of groups of eigenvalues simultaneously in different processors. However, this division could not be arbitrarly done, since the performance of the parallel algorithm will be determined by the correctness of the load balancing.

The problem of the load balancing is already known in homogeneous parallel computing but it affects more to the performance of the parallel algorithms when the power and the communication capabilities of the processors are not equal.

Different approaches can be taken to solve the problem in our case:

1. To ignore the difference of power and communication capabilities and perform an equitable workload distribution. With this approach, the spectrum is divided into subintervals containing the same number of eigenvalues.
2. Based on a heterogeneous machine model, like the one presented on Section 3, perform a distribution of the workload proportional to the power and communication features of each processor. Now, the spectrum is divided into subintervals

Kind	Elements	Eigenvalues
Uniform spectrum matrices	$a_i = 0$ $b_i = \sqrt{i(n-i)}$	$\{-n+2k-1\}_{k=1}^{n}$ Uniformly distributed
Wilkinson matrices	$a_i = \begin{cases} \frac{m}{2} - i + 1 & i : 1, \ldots, \frac{m}{2} \\ i - \frac{m}{2} & i : \frac{m}{2} + 1, \ldots, m \end{cases}$ $m = \begin{cases} n & with\ even\ n \\ n+1 & with\ odd\ n \end{cases}$	Most eigenvalues grouped in clusters of two.

Table 1: Test Matrices

P0	P1	P2	P3	P4	P5
0.2245	0.1740	0.1692	0.1605	0.1598	0.1120

Table 2: Relative power vector P_t for uniform spectrum matrices eigenvalue computation

P0	P1	P2	P3	P4	P5
0.2219	0.1747	0.1681	0.1635	0.1607	0.1113

Table 3: Relative power vector P_t for Wilkinson matrices eigenvalue computation

	P0	P1	P2	P3	P4	P5
P0	0	2.5074E-05	4.6149E-05	4.5745E-05	4.0809E-05	5.1629E-05
P1	2.4774E-05	0	4.6042E-05	4.5852E-05	4.0348E-05	5.317E-05
P2	4.6268E-05	4.6073E-05	0	2.4727E-05	4.2951E-05	5.2753E-05
P3	4.564E-05	4.5583E-05	2.4898E-05	0	4.1091E-05	5.2756E-05
P4	4.0829E-05	4.0706E-05	4.1424E-05	4.0211E-05	0	7.8713E-05
P5	5.0371E-05	4.9166E-05	5.1836E-05	4.6695E-05	1.0234E-4	0

Table 4: Latency matrix B_c (s)

	P0	P1	P2	P3	P4	P5
P0	0	6.4244E-09	1.4045E-08	1.1898E-08	2.1316E-08	5.5421E-08
P1	6.3307E-09	0	1.2879E-08	1.3066E-08	2.1954E-08	5.5108E-08
P2	1.2966E-08	1.3761E-08	0	6.3347E-09	2.0886E-08	5.2581E-08
P3	1.2548E-08	1.3327E-08	6.3294E-09	0	2.0014E-08	5.4863E-08
P4	2.1369E-08	2.1076E-08	2.269E-08	2.0776E-08	0	1.0859E-07
P5	2.1369E-08	5.4601E-08	5.6684E-08	5.1962E-08	1.0991E-07	0

Table 5: Inverse bandwidth matrix T_c (s)

with a number of eigenvalues proportional to the relative power of each processor.

3. To implement a dynamical workload distribution algorithm based on the master-slave programming paradigm. With this approach, the spectrum is divided into a number of subintervals $m \gg p$ that are assigned to the processors on demand.

4.2 Implemented algorithms

Based on the algorithm presented in Section 2 and on the approaches for the solution of the load balancing problem described before, we have implemented a sequential and five parallel algorithms.

Sequential algorithm. A1: This version implements sequentially the bisection algorithm described in

Section 2.

ScaLAPACK algorithm. A2: This version computes the eigenvalues of the matrix T by means of calling ScaLAPACK subroutine $pdstebz$. This subroutine uses the bisection method for isolating and extracting the eigenvalues.

Static algorithm. A3: In this version we statically assign to the processor $i = 0, \ldots, p-1$ the calculation of eigenvalues $[i\frac{n}{p}+1, (i+1)\frac{n}{p}]$ according to an ascending classification of them.

This algorithm have been implemented by means of p concurrent calls to the LAPACK subroutine $dstevr$ that takes as input parameters two integers that define the subset of desired eigenvalues.

We have used this algorithm to model the cluster and to obtain the data shown in Section 3.3. Also it could be a better comparative reference for the rest of the algorithms as it has been implemented with a similar style and without the optimisation of ScaLAPACK library.

Proportional Static algorithm. A4: This version uses a similar strategy to the one described in the previous algorithm but the number of eigenvalues assigned to each processor depends on its relative power.

Dynamic algorithm. A5: This version implements, in parallel, both steps of bisection algorithm described in Section 2. The first step of the algorithm consists of dividing the interval $[a, b]$ which contains all the eigenvalues into p subintervals of length equal to $\frac{b-a}{p}$. Each of this subintervals is assigned to a processor that applies the isolation step described in 2. Finally, the results are gathered by the master process.

In order to fulfil the $m \gg p$ constraint, parameter max_val of the isolation algorithm has been adjusted to 1. Thus m is equal to problema size n.

Finally, the extraction step has been implemented with the master-slave technique described before. Note that the most powerful processor has been chosen to allocate the master process. The master process also assigns intervals to this most powerful processor. In this way it acts as a slave process too, in order to take advantage of its greater power.

Modified Dynamic algorithm. A6: This version is similar to the previous one, but the $m \gg p$ constraint has been relaxed. Instead of $max\_val = 1$, we have assigned values between 1 and 100. We have done it for two reasons; first, to diminish the drawbacks produced by clusters of eigenvalues in the isolation step and, second, to study the impact in the execution time of the number of eigenvalues computed in the extraction step.

5 Experimental Analysis

Tables 6, 7 and 8 show the execution time of the six algorithms presented. For both kind of matrices, the Proportional Static algorithm (A4) and the Modified Dynamic algorithm (A6) present the smaller execution times, followed by ScaLAPACK (A2) and by Dynamic algorithm (A5) that present similar results. Finally the Static algorithm (A3) has the poorest performance of all tested algorithms.

n	Uniform	Wilkinson
5000	20.14	8.63
7000	38.80	16.50
9000	63.35	27.07
11000	94.13	39.94
13000	130.69	55.38
15000	172.80	74.31

Table 6: Execution time (s), on P0, for the sequential algorithm (A1) on both kinds of matrices

n	A2	A3	A4	A5	A6
5000	5.15	6.39	4.44	5.21	4.43
7000	9.98	12.41	8.61	9.78	8.54
9000	16.29	20.22	13.94	15.88	13.95
11000	24.45	30.15	20.66	23.68	20.78
13000	33.85	42.12	28.76	33.00	28.68
15000	45.67	56.50	37.96	43.72	38.04

Table 7: Execution time (s) for the 5 parallel algorithms on uniform spectrum matrices

Tables 9 and 10 show the speedup of the two best parallel versions, A4 and A6, with regard to the ScaLAPACK version (algorithm A2). Both algorithms present similar performance, with a slightly better speedup when they are applied on Wilkinson matrices.

6 Conclusions

In the present work one sequential and five parallel algorithms have been presented for the extraction of the

n	A2	A3	A4	A5	A6
5000	2.48	2.98	2.03	3.48	2.08
7000	4.70	5.68	3.86	6.72	3.92
9000	7.68	9.36	6.33	11.15	6.35
11000	11.45	13.96	9.38	17.42	9.45
13000	15.95	19.63	13.04	24.58	13.15
15000	21.21	25.99	17.39	32.85	17.35

Table 8: Execution time (s) for the 5 parallel algorithms on Wilkinson matrices

n	5000	7000	9000	11000	13000	15000
A4	1.16	1.16	1.17	1.18	1.18	1.2
A6	1.16	1.17	1.17	1.18	1.18	1.2

Table 9: Speedup of algorithm A4 and A6 with regard to algorithm A2 (ScaLAPACK) on uniform spectrum matrices

n	5000	7000	9000	11000	13000	15000
A4	1.22	1.22	1.21	1.22	1.22	1.22
A6	1.2	1.2	1.21	1.21	1.21	1.22

Table 10: Speedup of algorithm A4 and A6 with regard to algorithm A2 (ScaLAPACK) on Wilkinson matrices

eigenvalues of a symmetric tridiagonal matrix. Three of them have been specifically designed to be executed in heterogeneous distributed memory multicomputers.

The parallel algorithms implemented are based on the bisection method. Basically, two strategies have been used: a static strategy, trying to get a good load balancing through a distribution of processes proportional to the power of processors, and a dynamic strategy, based on the master-slave paradigm. For the implementation of the dynamic algorithms we have used a bisection algorithm with two steps: isolation and extraction. The bisection technique has been chosen to implement the isolation step. For the extraction step LAPACK subroutines have been used.

Finally, these are the main conclusions of the work:

- The algorithms that take into account the heterogeneity of the system when balancing the workload (A4, A5 and A6) always obtain better execution time than those that do not that (A2 and A3). This fact justifies the need of implementing specific load balancing techniques for heterogeneous architectures.
- The execution time of the Dynamic algorithm (A5) is always larger than that of the Modified Dynamic algorithm (A6). This is due to the extra effort made in the isolation step. In addition, the presence of clusters of eigenvalues (Wilkinson matrices) increases the number of iterations needed in this step.
- The fact that Proportional Static algorithm (A4) and Modified Dynamic algorithm (A6) present almost the same execution time validates both strategies to get a good load balancing. However, the effort necessary to reach a good workload balance in A4 (compute the power vector for each kind of matrix and problem size) could be a huge amount of extra work. Therefore, the authors consider the Modified Dynamic algorithm (A6) is the most suitable solution for heterogeneous environments.
- The fact that algorithms A4 and A6 present better performance than ScaLAPACK subroutines justifies the need of designing and implementing numerical linear algebra libraries for heterogeneous parallel architectures.

References:

[1] J.M.Badía, A.M.Vidal. "Cálculo de los valores propios de matrices tridiagonales simétricas mediante la iteración de Laguerre". Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería. Vol. 16, num 2, pp 227–149 (2000)

[2] J.M.Badía, A.M.Vidal. "Parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem". In the book High Performance Algorithms for Structured Matrix Problems from the series Advances in the Theory of Computation and Computational Mathematics. Nova Science Publishers (1998)

[3] J. L. Bosque, L. P. Perez. "HLogGP: a new parallel computational model for heterogeneous clusters". CCGRID 2004: 403-410 (2004)

[4] Anderson, E., Bai, Z., Bishof, C., Demmel, J., Dongarra, J.; LAPACK User Guide; Second edition. SIAM (1995)

[5] Blackford, L.S., Choi, J., Clearly, A.; ScaLAPACK User's Guide. SIAM (1997)

[6] G. H. Golub, C. F. van Loan; Matrix Computations; John Hopkins University Press, 3rd edition (1996)

The symmetric tridiagonal eigenvalue problem: a heterogeneous parallel approach

Miguel O. Bernabeu
Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera S/N. 46022 Valencia
Spain
mbernabeu@dsic.upv.es

Antonio M. Vidal
Dept. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera S/N. 46022 Valencia
Spain
avidal@dsic.upv.es

Abstract: Computation of the eigenvalues of a symmetric tridiagonal matrix is a problem of great relevance in numerical linear algebra. There exist a wide range of algorithms for its solution and many implementations, both sequential and parallel, can be found. Despite this fact, none of them is oriented to be executed in heterogeneous distributed memory multicomputers. In these architectures, the workload balance is the key factor for the overall performance of the algorithms. In this work, we present and compare two different load balancing schemes and their corresponding implementations. Besides, the experimental results show that only the algorithms that take into account the heterogeneity of the system when balancing the workload obtain optimum performance.

Key–Words: Symmetric tridiagonal eigenvalue problem, heterogeneous parallel computing, load balancing

1 Introduction

Computation of the eigenvalues of a symmetric tridiagonal matrix is a problem of great relevance in numerical linear algebra and in many engineering fields, mainly due to two reasons: first, this kind of matrices arises in the discretisation of certain engineering problems and secondly, and more important, this operation is the main computational kernel in the computation of the eigenvalues of any symmetric matrix when tridiagonalisation techniques are used as a previous step.

Nowadays, there are a large amount of eigenvalue computation algorithms that exploit the tridiagonality properties of the matrix. Four main techniques can be found in the specialised literature to solve this problem: QR iteration, homotopy method, bisection and multisection methods and divide and conquer techniques. None of them is clearly superior to the rest since every one presents exclusive advantages, for example: computing all matrix eigenvalues or just a defined subset of them, precision of the results or simultaneous eigenvector computation. See [1] for an exhaustive comparison.

The importance of the problem and the different available algorithms is reflected by the large number of subroutines provided by linear algebra libraries for the computation of the eigenvalues of a symmetric tridiagonal matrix. For example, LAPACK offers implementations based on QR iteration, bisection method and divide and conquer technique. In addition, ScaLAPACK also offer parallel implementations of both bisection method and QR iteration.

In [2] Badía and Vidal proposed two parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem on distributed memory multicomputers, including a deep study of the two step bisection algorithm. In that work, special emphasis was put in load balancing since this is the main difficulty when parallelising the bisection algorithm for the computation of the eigenvalues of a symmetric tridiagonal matrix.

Both, ScaLAPACK subroutines and those presented in [2] achieve good performance in homogeneous distributed memory multicomputers. We can define an homogeneous distributed memory multicomputer as a distributed memory multicomputer where all the processors are equal in computing and communication capabilities. In this work we focus on heterogeneous distributed memory multicomputers, those formed by processors with different computing and communication capabilities. These kind of platforms are expected to be the best solution in order to achieve great performance/cost ratio, to reuse obsolete computational resources or simply to obtain the maximum performance from several powerful computers with different architectures but able to work coordinately.

Parallel numerical linear algebra libraries, like ScaLAPACK, do not take into account the possible heterogeneity of the hardware and, for that reason, their performance considerably decrease when working in this kind of systems. There is a big gap in this context and, nowadays, the design of parallel linear algebra libraries for heterogeneous architectures is a must. The study of computational kernels and basic algorithms is the previous step to achieve this objective. In this work we present algorithms for the computation of the eigenvalues of a symmetric tridiagonal matrix which attain good performance in heterogeneous multicomputer, analysing different load balancing strategies and different problem instances.

The rest of the paper is organised as follows: in the next section we present the mathematical description of the problem and the sequential algorithm implemented in the solution. Section 3 describes the heterogeneous computational model used. In Section 4, the different parallel schemes used are described. In Section 5, experimental results are presented. Finally, the main conclusions of the work are given in Section 6.

2 Problem description and proposed solution

2.1 Problem definition

Let T be a symmetric tridiagonal matrix $T \in \mathbb{R}^{nxn}$, defined as follows

$$T = \begin{bmatrix} a_1 & b_1 & & & 0 \\ b_1 & a_2 & b_1 & & \\ & b_2 & a_3 & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ 0 & & & b_{n-1} & a_n \end{bmatrix} \quad (1)$$

The eigenvalues of T are the n roots of its characteristic polynomial $p(z) = det(zI - T)$. The set of these roots is called the spectrum and is denoted by $\lambda(T)$.

It is possible to compute specific eigenvalues of a symmetric matrix by using the LDL^T factorization and exploiting the Sylvester inertia theorem. If

$$A - \mu I = LDL^T \quad A = A^T \in R^{nxn}$$

is the LDL^T factorization of $A - \mu I$ with $D = diag(d_1, \ldots, d_n)$, then the number of negative d_i equals the number of $\lambda(A)$ that are smaller than μ [6].

Sequence $(d_1, \ldots, d_n)$ can be computed by using the following recurrence, where $d_i = q_i(c), i = 1, \ldots, n$ for a given c:

$$\begin{cases} q_0(c) = 1, \quad q_1(c) = a_1 - c \\ q_i(c) = (a_i - c) - \frac{b_{i-1}^2}{q_{i-1}(c)} \quad i : 2, 3, \ldots, n \end{cases}$$

Thanks to this result it is possible to define a function $neg_n(c)$ that for any value of c computes the number of eigenvalues smaller than c. With this function it is easy to implement a bisection algorithm that isolates eigenvalues of T.

The bisection algorithm needs, for initialisation purpose, an initial interval $[a, b]$ which contains all the eigenvalues of matrix T. The Gershgorin circle theorem can be used to calculate it.

So, based on the initial interval $[a, b]$ and through the bisection algorithm is possible to isolate the m subintervals $]lb_i, ub_i]$ which contain a number of eigenvalues $v \leq max_v al$ and will be used as the input for the next step of the algorithm.

The importance of this step lies in the fact that it will help us to discard parts of the real line where no eigenvalue is located and therefore to reduce the number of iterations of the extraction methods used in the following step. In addition, the isolation step will be used to balance the workload of the parallel algorithms presented in Section 4, since the initial problem is divided into m subproblems with similar workload, susceptible to be solved in parallel.

The second step of the algorithm gets as input the m subintervals $]lb_i, ub_i]$ obtained in the previous step to compute m eigenvalues of matrix T.

There are several alternatives for the eigenvalues extraction:

1. To apply again the bisection method described in previous subsection.

2. To use a fast convergence method like Newton or Laguerre [1].

3. To use standard computational kernels like LAPACK and let them choose the best method for eigenvalue extraction. These subroutines are expected to efficiently implement the sequential solution of the problem.

In this work, we have chosen to use LAPACK subroutines, specifically the driver subroutine $dstevr$ which can compute the eigenvalues of a tridiagonal symmetric matrix contained into an interval $]vl, vu]$.

2.2 Test matrices

The bisection algorithm above described is problem-dependent, because the number of iterations to reach each eigenvalue with a specified precision could be different. Thus, the behaviour of the algorithm depends on the distribution of the eigenvalues along the spectrum. In addition, the presence of clusters of eigenvalues or hidden eigenvalues considerably increases the extraction time, see [2].

Therefore, in order to perform a correct experimental analysis of the algorithms implemented, a suitable set of test matrices should be chosen. In our case, we have chosen two kinds of matrices that present different eigenvalue distribution characteristics, so it can affect the performance of the algorithm.

Table 1 shows matrices used.

3 Heterogeneous computational model

A parallel computational model is a mathematical abstraction of the parallel machine that hides the architectural details to the software designers. The models should be detailed enough to reflect those aspects with significant impact in the program performance, abstract enough to be machine independent and simple in order to allow an efficient analysis of the algorithms [3].

3.1 PC model description

The following theoretical machine model is called PC (Power-Communications). Let be a set of p processors interconnected via a communication network. This model is expected to evaluate the power of each processor as well as the communication capabilities of the network.

First of all, a power vector P_t that summarizes the relative power of each processor (related to the global machine power) is defined. This relative power depends on the operation and on the problem size, so there is a vector P_t for each pair of operation and problem size. However, we consider here that the power vector does not depend on time.

Secondly, the communication model used defines the time needed to send n bytes from processor i to processor j as $T_{ij}(n) = \beta + n\tau$, where β stands for network latency and τ is the bandwidth inverse. In order to summarize the model two matrices B_c and T_c are defined. The (i, j) entry of each matrix represents the β or τ applicable to the communication from processor i to j. We consider also here that both matrices do not depend on time.

3.2 Model implementation

The cluster used to evaluate the model and to run the parallel algorithms consists of four machines with six processors. They are:

- Intel Pentium IV at 3.0 GHz with 1 MB of cache and 1 GB of main memory.
- Intel Xeon two-processor at 2.2 GHz with 512 KB of cache and 4 GB of main memory.
- Intel Xeon two-processor at 2.2 GHz with 512 KB of cache and 4 GB of main memory.
- Intel Pentium IV at 1,6 GHz with 256 KB of cache and 1GB of main memory.

A Gigabit Ethernet network is used to interconnect the six machines with 1 Gbit/s of theoretical bandwidth. Note that communications between each CPU of the two-processors boards have been evaluated with the same model.

Despite the algorithms have been implemented and evaluated in this machine, they only depend on the theoretical model. So, they can be executed in any other distributed memory multicomputer with similar predictable performance, under the condition of being evaluated with the previous model.

3.3 Evaluation

Tables 2, 3, 4 and 5 show the result of the evaluation of the cluster described before, following the PC model. Tables 2 and 3 show the power vector P_t obtained in the computation of eigenvalues of uniform spectrum matrices and Wilkinson matrices with different sizes. Tables 4 and 5 show the matrices B_c and T_c obtained in the same experiments.

As it can be observed, the variations of the vector power P_t with the size of the problem are very small. This is a characteristic of this problem, because only two vectors (main diagonal and subdiagonal) have to be stored in memory. This may be different with problems which require more memory space.

4 Heterogeneous parallel schemes

4.1 Available alternatives

Among different techniques proposed in the literature (see [2]) to parallelise the bisection method, probably the most effective is the computation of groups of eigenvalues simultaneously in different processors. However, this division could not be arbitrarly done,

Kind	Elements	Eigenvalues
Uniform spectrum matrices	$a_i = 0$ $b_i = \sqrt{i(n-i)}$	$\{-n+2k-1\}_{k=1}^{n}$ Uniformly distributed
Wilkinson matrices	$a_i = \begin{cases} \frac{m}{2} - i + 1 & i : 1, \ldots, \frac{m}{2} \\ i - \frac{m}{2} & i : \frac{m}{2} + 1, \ldots, m \end{cases}$ $m = \begin{cases} n & with\ even\ n \\ n+1 & with\ odd\ n \end{cases}$	Most eigenvalues grouped in clusters of two.

Table 1: Test Matrices

n	P0	P1	P2	P3	P4	P5
5000	0.2245	0.1740	0.1692	0.1605	0.1598	0.1120
6000	0.2265	0.1735	0.1706	0.1591	0.1591	0.1112
7000	0.2264	0.1748	0.1688	0.1586	0.1600	0.1113
8000	0.2245	0.1749	0.1701	0.1590	0.1600	0.1116
9000	0.2269	0.1735	0.1700	0.1592	0.1590	0.1115
10000	0.2264	0.1739	0.1703	0.1588	0.1596	0.1110
11000	0.2265	0.1749	0.1690	0.1577	0.1604	0.1115
12000	0.2268	0.1739	0.1711	0.1578	0.1595	0.1108
13000	0.2265	0.1743	0.1700	0.1586	0.1597	0.1108
14000	0.2270	0.1747	0.1700	0.1588	0.1593	0.1101
15000	0.2266	0.1737	0.1708	0.1598	0.1599	0.1093

Table 2: Relative power vector P_t for uniform spectrum matrices eigenvalue computation

n	P0	P1	P2	P3	P4	P5
5000	0.2219	0.1747	0.1681	0.1635	0.1607	0.1113
6000	0.2177	0.1762	0.1687	0.1638	0.1623	0.1131
7000	0.2193	0.1752	0.1687	0.1648	0.1612	0.1110
8000	0.2145	0.1756	0.1692	0.1647	0.1642	0.1117
9000	0.2213	0.1745	0.1681	0.1632	0.1622	0.1107
10000	0.2232	0.1748	0.1678	0.1632	0.1611	0.1099
11000	0.2231	0.1751	0.1661	0.1640	0.1614	0.1103
12000	0.2203	0.1759	0.1680	0.1636	0.1620	0.1101
13000	0.2239	0.1748	0.1675	0.1640	0.1608	0.1091
14000	0.2263	0.1744	0.1664	0.1639	0.1601	0.1089
15000	0.2237	0.1735	0.1662	0.1654	0.1611	0.1101

Table 3: Relative power vector P_t for Wilkinson matrices eigenvalue computation

since the performance of the parallel algorithm will be determined by the correctness of the load balancing.

The problem of the load balancing is already known in homogeneous parallel computing but it affects more to the performance of the parallel algorithms when the power and the communication capabilities of the processors are not equal.

Different approaches can be taken to solve the problem in our case:

1. To ignore the difference of power and communication capabilities and perform an equitable workload distribution. With this approach, the spectrum is divided into subintervals containing the same number of eigenvalues.

2. Based on a heterogeneous machine model, like the one presented on Section 3, perform a distribution of the workload proportional to the power and communication features of each processor.

	P0	P1	P2	P3	P4	P5
P0	0	2.5074E-05	4.6149E-05	4.5745E-05	4.0809E-05	5.1629E-05
P1	2.4774E-05	0	4.6042E-05	4.5852E-05	4.0348E-05	5.317E-05
P2	4.6268E-05	4.6073E-05	0	2.4727E-05	4.2951E-05	5.2753E-05
P3	4.564E-05	4.5583E-05	2.4898E-05	0	4.1091E-05	5.2756E-05
P4	4.0829E-05	4.0706E-05	4.1424E-05	4.0211E-05	0	7.8713E-05
P5	5.0371E-05	4.9166E-05	5.1836E-05	4.6695E-05	1.0234E-4	0

Table 4: Latency matrix B_c (s)

	P0	P1	P2	P3	P4	P5
P0	0	6.4244E-09	1.4045E-08	1.1898E-08	2.1316E-08	5.5421E-08
P1	6.3307E-09	0	1.2879E-08	1.3066E-08	2.1954E-08	5.5108E-08
P2	1.2966E-08	1.3761E-08	0	6.3347E-09	2.0886E-08	5.2581E-08
P3	1.2548E-08	1.3327E-08	6.3294E-09	0	2.0014E-08	5.4863E-08
P4	2.1369E-08	2.1076E-08	2.269E-08	2.0776E-08	0	1.0859E-07
P5	2.1369E-08	5.4601E-08	5.6684E-08	5.1962E-08	1.0991E-07	0

Table 5: Inverse bandwidth matrix T_c (s)

Now, the spectrum is divided into subintervals with a number of eigenvalues proportional to the relative power of each processor.

3. To implement a dynamical workload distribution algorithm based on the master-slave programming paradigm. With this approach, the spectrum is divided into a number of subintervals $m \gg p$ that are assigned to the processors on demand.

4.2 Implemented algorithms

Based on the algorithm presented in Section 2 and on the approaches for the solution of the load balancing problem described before, we have implemented a sequential and five parallel algorithms.

4.2.1 Sequential algorithm. A1

This version implements sequentially the bisection algorithm described in Section 2.

4.2.2 ScaLAPACK algorithm. A2

This version computes the eigenvalues of the matrix T by means of calling ScaLAPACK subroutine $pdstebz$. This subroutine uses the bisection method for isolating and extracting the eigenvalues.

4.2.3 Static algorithm. A3

In this version we statically assign to the processor $i = 0, \ldots, p-1$ the calculation of eigenvalues $[i\frac{n}{p} + 1, (i+1)\frac{n}{p}]$ according to an ascending classification of them.

This algorithm have been implemented by means of p concurrent calls to the LAPACK subroutine $dstevr$ that takes as input parameters two integers that define the subset of desired eigenvalues.

We have used this algorithm to model the cluster and to obtain the data shown in Section 3.3. Also it could be a better comparative reference for the rest of the algorithms as it has been implemented with a similar style and without the optimisation of ScaLAPACK library.

4.2.4 Proportional Static algorithm. A4

This version uses a similar strategy to the one described in the previous algorithm but the number of eigenvalues assigned to each processor depends on its relative power.

4.2.5 Dynamic algorithm. A5

This version implements, in parallel, both steps of bisection algorithm described in Section 2. The first step of the algorithm consists of dividing the interval $[a, b]$ which contains all the eigenvalues into p subintervals of length equal to $\frac{b-a}{p}$. Each of this subintervals is assigned to a processor that applies the isolation step

described in 2. Finally, the results are gathered by the master process.

In order to fulfil the $m \gg p$ constraint, parameter max_val of the isolation algorithm has been adjusted to 1. Thus m is equal to problema size n.

Finally, the extraction step has been implemented with the master-slave technique described before. Note that the most powerful processor has been chosen to allocate the master process. The master process also assigns intervals to this most powerful processor. In this way it acts as a slave process too, in order to take advantage of its greater power.

4.2.6 Modified Dynamic algorithm. A6

This version is similar to the previous one, but the $m \gg p$ constraint has been relaxed. Instead of $max\_val = 1$, we have assigned values between 1 and 100. We have done it for two reasons; first, to diminish the drawbacks produced by clusters of eigenvalues in the isolation step and, second, to study the impact in the execution time of the number of eigenvalues computed in the extraction step.

5 Experimental Analysis

Tables 6, 7 and 8 and Figures 1 and 2 show the execution time of the six algorithms presented. For both kind of matrices, the Proportional Static algorithm (A4) and the Modified Dynamic algorithm (A6) present the smaller execution times, followed by ScaLAPACK (A2) and by Dynamic algorithm (A5) that present similar results. Finally the Static algorithm (A3) has the poorest performance of all tested algorithms.

n	Uniform	Wilkinson
5000	20.14	8.63
6000	28.76	12.20
7000	38.80	16.50
8000	50.54	21.57
9000	63.35	27.07
10000	78.03	33.15
11000	94.13	39.94
12000	113.96	47.58
13000	130.69	55.38
14000	151.30	64.67
15000	172.80	74.31

Table 6: Execution time (s), on P0, for the sequential algorithm (A1) on both kinds of matrices

n	A2	A3	A4	A5	A6
5000	5.15	6.39	4.44	5.21	4.43
6000	7.35	9.12	6.36	7.26	6.31
7000	9.98	12.41	8.61	9.78	8.54
8000	13.02	16.09	11.06	12.64	11.13
9000	16.29	20.22	13.94	15.88	13.95
10000	20.26	24.97	17.10	19.56	17.14
11000	24.45	30.15	20.66	23.68	20.78
12000	28.85	35.88	24.53	28.07	24.58
13000	33.85	42.12	28.76	33.00	28.68
14000	39.74	48.94	33.09	38.57	33.24
15000	45.67	56.50	37.96	43.72	38.04

Table 7: Execution time (s) for the 5 parallel algorithms on uniform spectrum matrices

n	A2	A3	A4	A5	A6
5000	2.48	2.98	2.03	3.48	2.08
6000	3.49	4.25	2.85	4.95	3.06
7000	4.70	5.68	3.86	6.72	3.92
8000	6.20	7.38	5.02	8.77	5.09
9000	7.68	9.36	6.33	11.15	6.35
10000	9.48	11.54	7.77	14.16	7.87
11000	11.45	13.96	9.38	17.42	9.45
12000	13.64	16.61	11.14	20.88	11.20
13000	15.95	19.63	13.04	24.58	13.15
14000	18.53	22.49	15.11	28.58	15.12
15000	21.21	25.99	17.39	32.85	17.35

Table 8: Execution time (s) for the 5 parallel algorithms on Wilkinson matrices

Tables 9 and 10 show the speedup of the two best parallel versions, A4 and A6, with regard to the ScaLAPACK version (algorithm A2). Both algorithms present similar performance, with a slightly better speedup when they are applied on Wilkinson matrices.

n	5000	7000	9000	11000	13000	15000
A4	1.16	1.16	1.17	1.18	1.18	1.2
A6	1.16	1.17	1.17	1.18	1.18	1.2

Table 9: Speedup of algorithm A4 and A6 with regard to algorithm A2 (ScaLAPACK) on uniform spectrum matrices

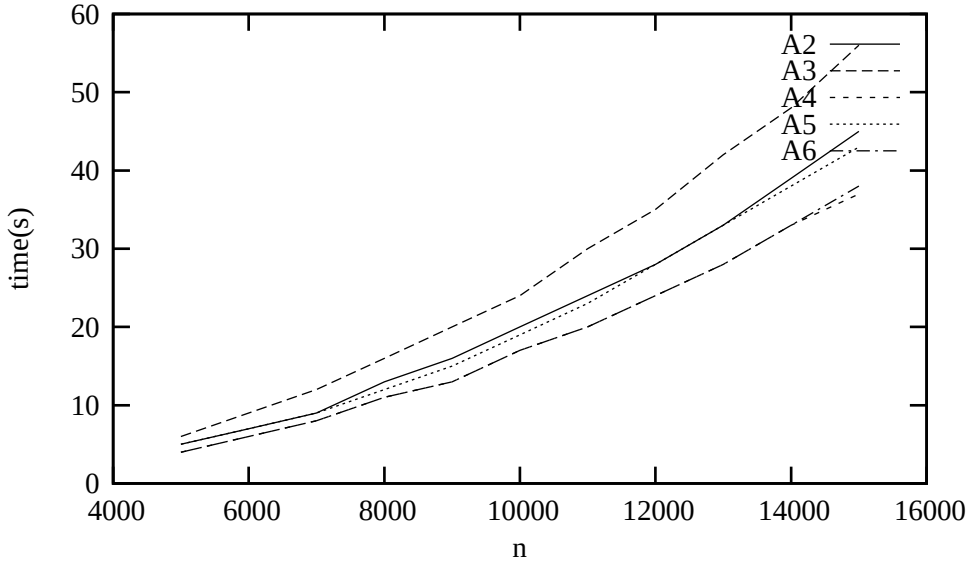

Figure 1: Execution time for the 5 parallel algorithms on uniform spectrum matrices

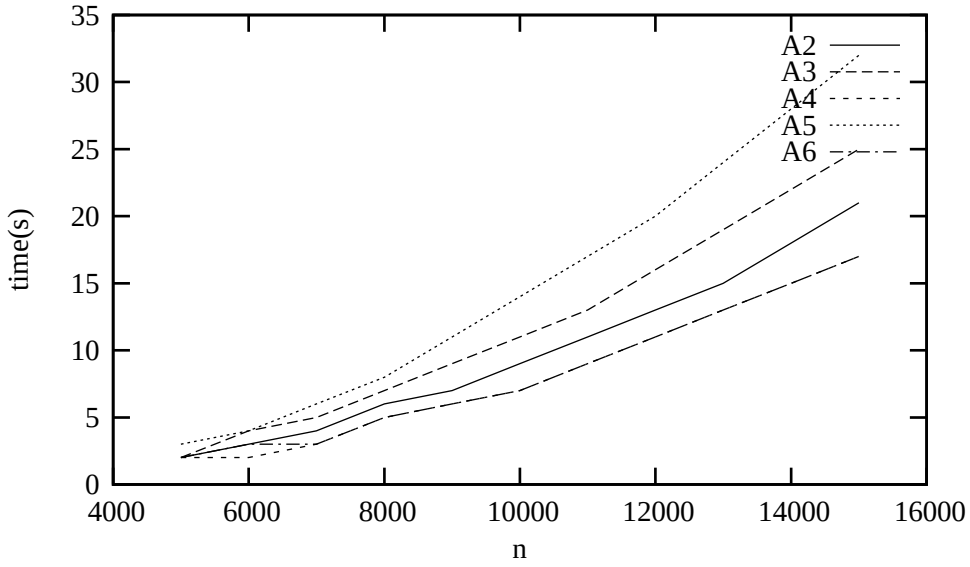

Figure 2: Execution time for the 5 parallel algorithms on Wilkinson matrices

n	5000	7000	9000	11000	13000	15000
A4	1.22	1.22	1.21	1.22	1.22	1.22
A6	1.2	1.2	1.21	1.21	1.21	1.22

Table 10: Speedup of algorithm A4 and A6 with regard to algorithm A2 (ScaLAPACK) on Wilkinson matrices

6 Conclusions

In the present work one sequential and five parallel algorithms have been presented for the extraction of the eigenvalues of a symmetric tridiagonal matrix. Three of them have been specifically designed to be executed in heterogeneous distributed memory multicomputers.

The parallel algorithms implemented are based on the bisection method. Basically, two strategies have been used: a static strategy, trying to get a good load balancing through a distribution of processes proportional to the power of processors, and a dynamic strategy, based on the master-slave paradigm. For the implementation of the dynamic algorithms we have used a bisection algorithm with two steps: isolation and ex-

traction. The bisection technique has been chosen to implement the isolation step. For the extraction step LAPACK subroutines have been used.

Finally, these are the main conclusions of the work:

- The algorithms that take into account the heterogeneity of the system when balancing the workload (A4, A5 and A6) always obtain better execution time than those that do not that (A2 and A3). This fact justifies the need of implementing specific load balancing techniques for heterogeneous architectures.
- The execution time of the Dynamic algorithm (A5) is always larger than that of the Modified Dynamic algorithm (A6). This is due to the extra effort made in the isolation step. In addition, the presence of clusters of eigenvalues (Wilkinson matrices) increases the number of iterations needed in this step.
- The fact that Proportional Static algorithm (A4) and Modified Dynamic algorithm (A6) present almost the same execution time validates both strategies to get a good load balancing. However, the effort necessary to reach a good workload balance in A4 (compute the power vector for each kind of matrix and problem size) could be a huge amount of extra work. Therefore, the authors consider the Modified Dynamic algorithm (A6) is the most suitable solution for heterogeneous environments.
- The fact that algorithms A4 and A6 present better performance than ScaLAPACK subroutines justifies the need of designing and implementing numerical linear algebra libraries for heterogeneous parallel architectures.

References:

[1] J.M.Badía, A.M.Vidal. "Cálculo de los valores propios de matrices tridiagonales simétricas mediante la iteración de Laguerre". Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería. Vol. 16, num 2, pp 227–149 (2000)

[2] J.M.Badía, A.M.Vidal. "Parallel bisection algorithms for solving the symmetric tridiagonal eigenproblem". In the book High Performance Algorithms for Structured Matrix Problems from the series Advances in the Theory of Computation and Computational Mathematics. Nova Science Publishers (1998)

[3] J. L. Bosque, L. P. Perez. "HLogGP: a new parallel computational model for heterogeneous clusters". CCGRID 2004: 403-410 (2004)

[4] Anderson, E., Bai, Z., Bishof, C., Demmel, J., Dongarra, J.; LAPACK User Guide; Second edition. SIAM (1995)

[5] Blackford, L.S., Choi, J., Clearly, A.; ScaLAPACK User's Guide. SIAM (1997)

[6] G. H. Golub, C. F. van Loan; Matrix Computations; John Hopkins University Press, 3rd edition (1996)

J. Parallel Distrib. Comput. 68 (2008) 1113–1121

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Parallel computation of the eigenvalues of symmetric Toeplitz matrices through iterative methods

Antonio M. Vidal, Victor M. Garcia *, Pedro Alonso, Miguel O. Bernabeu

Departamento de Sistemas Informáticos y Computación, Univ. Politécnica de Valencia, Camino de Vera s/n 46022 Valencia, Spain

ARTICLE INFO

Article history:
Received 7 February 2007
Received in revised form
22 February 2008
Accepted 9 March 2008
Available online 15 March 2008

Keywords:
Toeplitz matrices
Eigenvalue problem
Parallel computing
Shift and Invert Lanczos

ABSTRACT

This paper presents a new procedure to compute many or all of the eigenvalues and eigenvectors of symmetric Toeplitz matrices. The key to this algorithm is the use of the "Shift–and–Invert" technique applied with iterative methods, which allows the computation of the eigenvalues close to a given real number (the "shift"). Given an interval containing all the desired eigenvalues, this large interval can be divided in small intervals. Then, the "Shift–and–Invert" version of an iterative method (Lanczos method, in this paper) can be applied to each subinterval. Since the extraction of the eigenvalues of each subinterval is independent from the other subintervals, this method is highly suitable for implementation in parallel computers. This technique has been adapted to symmetric Toeplitz problems, using the symmetry exploiting Lanczos process proposed by Voss [H. Voss, A symmetry exploiting Lanczos method for symmetric Toeplitz matrices, Numerical Algorithms 25 (2000) 377–385] and using fast solvers for the Toeplitz linear systems that must be solved in each Lanczos iteration. The method compares favourably with ScaLAPACK routines, specially when not all the spectrum must be computed.

1. Introduction

The computation of the eigenvalues of a symmetric Toeplitz matrix is a task that appears quite often in digital signal processing and control applications [5,2,12].

The eigenvalue problem for symmetric real matrices can be stated as:

$$Ax = \lambda x, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$, $x \neq 0 \in \mathbb{C}^n$, $\lambda \in \mathbb{R}$. The algorithms to be applied to solve this problem depend mainly on the concrete characteristics of the problem. It is possible that only a few eigenvalues are needed, maybe the largest or the smallest; in this case, the so called "iterative" methods can be used, for example Arnoldi, Lanczos, or Jacobi–Davidson [3]. If all or most eigenvalues are needed, then the best solution is the tridiagonalization of the matrix through Householder reflections. we can then use one of the available algorithms for tridiagonal matrices such as iterative QR, bisection, divide–and–conquer or Relatively Robust Representation (RRR). This is the philosophy implemented in LAPACK [1] and ScaLAPACK [4].

However, for very large problems, the memory needed may render this approach infeasible. Furthermore, it is usually difficult or impossible to take advantage of any structure that the matrix may have, since the tridiagonalization usually wipes out the structure of the matrix.

The iterative methods mentioned above can be advantageous in such situations, because they usually access the matrix only through matrix–vector products. These methods cannot be used directly to obtain all the eigenvalues, since the process would converge very slowly. Nevertheless, these methods can be modified (through the "Shift-and-Invert" technique) so that they can compute the eigenvalues closer to a given real number σ (the *shift*).

The main drawback of the "Shift-and-Invert" modification is that instead of computing matrix–vector products, linear systems must be solved, using as coefficient matrix: $A - \sigma I$. If the solution of these systems is too costly, the method will not be efficient. Therefore, this method is best suited for matrices where systems of the form $(A - \sigma I)x = b$ can be solved efficiently.

Our target problem is the computation of many (or all) the eigenvalues and eigenvectors of symmetric Toeplitz matrices. There are relatively few practical procedures proposed in the literature for the symmetric Toeplitz eigenvalue problem. The only specific methods are similar to the bisection method; since there are efficient recurrences to evaluate the characteristic polynomial, it becomes possible to find intervals containing a single eigenvalue, and then, use some suitable root–finding procedure (Newton, Pegasus, secant, …) to extract all the eigenvalues [2,12,16].

* Corresponding author.
E-mail addresses: avidal@dsic.upv.es (A.M. Vidal), vmgarcia@dsic.upv.es (V.M. Garcia), palonso@dsic.upv.es (P. Alonso), mbernabeu@dsic.upv.es (M.O. Bernabeu).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.03.003

A technique for solving the generalized eigenvalue problem for symmetric sparse matrices was proposed in [9], and parallelized in [19]. There, the interval that contains all the desired eigenvalues is split into subintervals of appropriate width, and all the eigenvalues of each subinterval are computed independently (of the eigenvalues in other subintervals) with the "Shift–and–Invert" technique. In this paper, we propose a similar method for the computation of all (or many) the eigenvalues of a symmetric Toeplitz matrix, where high efficiency is obtained using a symmetry exploiting version of Lanczos's method [18] and through fast Toeplitz solvers.

As shall be shown experimentally, sequential implementations of this algorithm are slower than LAPACK routines. However, the main idea behind this algorithm is the fact that the eigenvalues of each subinterval can be computed independently of the other subintervals. Therefore, the eigenvalues belonging to different intervals can be computed by different processors achieving easily a very high degree of parallelism. The parallel version of this algorithm becomes faster than the ScaLAPACK routines, even for a small number of processors.

The rest of the paper is structured as follows: First, we will describe the "Shift-and-Invert" Lanczos algorithm, and the symmetry exploiting version for Toeplitz matrices. Then, the fast solver for symmetric Toeplitz matrices will be described, followed by the overall sequential and parallel methods. Finally, we shall present some numerical results and the conclusions.

2. Method description

2.1. Iterative methods for eigenvalue problems

Iterative methods, such as Jacobi–Davidson, Arnoldi, Lanczos and variants, are mainly used to compute one or a few eigenvalue–eigenvector pairs [3]. In this work, we have chosen the Lanczos method, but any of the others could have also been selected.

Since the early 50's, the Lanczos method is used for computing eigenvalues of a symmetric matrix A. Given an initial vector r, it builds an orthonormal basis $v_1, v_2, \ldots, v_m$ of the Krylov subspace $K^m(A, r)$. In the new orthonormal basis, the matrix A is represented as the following tridiagonal matrix

$$\widetilde{A}_j = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{j-1} \\ & & \beta_{j-1} & \alpha_j \end{bmatrix}, \tag{2}$$

where the computation of the α_j, β_j is carried out as shown in Algorithm 1 (the formulation included below has been taken from [14]).

Algorithm 1. Lanczos algorithm for computing a few eigenvalues of A.

1. $\beta_0 = \|r\|_2$
2. for $j = 1, 2, \ldots$ until convergence
3. $\quad v_j = r/\beta_{j-1}$
4. $\quad r = Av_j$
5. $\quad r = r - v_{j-1} \cdot \beta_{j-1}$
6. $\quad \alpha_j = v_j^t r$
7. $\quad r = r - v_j \cdot \alpha_j$
8. $\quad$ re–orthogonalize if necessary
9. $\quad \beta_j = \|r\|_2$
10. $\quad$ compute approximate eigenvalues of $\widetilde{A}_j$
11. $\quad$ test bounds for convergence
12. end for
13. compute approximate eigenvectors.

Some of the eigenvalues of the $\widetilde{A}$ matrix (called Ritz values) are good approximations to the eigenvalues of A, and the eigenvectors can be easily obtained, as well, from the eigenvectors of the $\widetilde{A}$ matrix and the v_i vectors.

This method, as such, is not suitable for computing many eigenvalues; among other problems the convergence speed would be very poor and the matrix $V = (v_i)$ might become huge. Fortunately, the method can be adapted in a very convenient way, described in the following sections.

2.2. "Shift-and-Invert" technique

The Lanczos method converges first to the eigenvalues that are largest in magnitude. However, given a scalar σ, this method can be used to obtain the eigenvalues closest to σ, through the "Shift-and-Invert" technique [14].

This technique consists of finding the eigenvalues of $W = (A - \sigma I)^{-1}$. Any eigenvalue of this matrix gives an eigenvalue of A: if λ is such that $Ax = \lambda x$, then $\lambda_\sigma = \frac{1}{(\lambda - \sigma)}$ is an eigenvalue of $(A - \sigma I)^{-1}$. Clearly, since iterative methods converge first to eigenvalues of largest magnitude, these methods applied to the matrix W will deliver first the eigenvalues of A closest to σ. The algorithm for the "Shift-and-Invert" version of Lanczos would be like Algorithm 1 but after changing line 4 to: $r = (A - \sigma I)^{-1} v_j$. This algorithm, along with the implementation details, can be found in [14].

This algorithm can be applied to any symmetric matrix; however, it should work much better for matrices that allow fast resolution of the linear systems with coefficient matrix $A - \sigma I$.

For a symmetric Toeplitz matrix $T \in \mathbb{R}^{n \times n}$ the systems $T - \sigma I$ are symmetric Toeplitz as well. For these kind of matrices, there are several fast solvers that can be integrated in this algorithm; the chosen one and its implementation are discussed in Section 2.4.

However, this is not the only optimization that can be applied to this algorithm, when the target matrix is symmetric Toeplitz. In the next section, we describe an adaptation of the Lanczos method to the symmetric Toeplitz case, which allows a much faster extraction of the eigenvalues in an interval.

2.3. Symmetry exploiting Lanczos method

The symmetry exploiting Lanczos method was proposed by Voss in [18], for the computation of the smallest eigenvalue of a symmetric Toeplitz matrix. It takes advantage of the special structure of the eigenvalues of a Toeplitz matrix, to reduce the computation time needed. We have found out that this technique, applied to the problem of extracting all the eigenvalues of an interval, is even more profitable than for the problem of the smallest eigenvalue, since it allows to extract all the eigenvalues of the interval much faster than the standard Lanczos method. The following description summarizes the paper from Voss:

Let $J_n = (\delta_{i,n+1-i})_{i,j=1,\ldots,n}$ be the (n, n) matrix which, applied to a vector, reverses it. A vector v is symmetric if $x = J_n x$ and skew-symmetric if $x = -J_n x$. It is well known that the eigenvectors of a Toeplitz matrix are either symmetric or skew-symmetric. With a small abuse of terminology, we will say as well that a eigenvalue is symmetric (or skew-symmetric) if its associated eigenvector is symmetric (or skew-symmetric).

Consider now the Lanczos method. If the initial vector for the Lanczos recurrence is symmetric, then the whole Krylov Space is in the same symmetry class, and the eigenvectors generated shall be symmetric; the same would happen if the initial vector is skew-symmetric, only skew-symmetric eigenvectors can be generated.

Very often (though not always) the symmetric and skew-symmetric eigenvalues of a symmetric Toeplitz matrix are interlaced; therefore, if we can restrict the Lanczos method to only one of these classes, the relative separation between eigenvalues

may increase, and the speed of convergence could improve. In Voss' paper, the computation of the smallest eigenvalue is the goal; however, the symmetry class of the smallest eigenvalue is usually not known in advance.

This problem was overcomed by Voss devising a nice two-way Lanczos process, whose key is the following fact: if $T \in \mathbb{R}^{n\times n}$ is a symmetric Toeplitz matrix, and $v \in \mathbb{R}^n$ is the solution of the linear system $Tv = w$, then the symmetric part $v_s = 0.5\,(v + J_n v)$ solves the linear system $Tv_s = 0.5\,(w + J_n w)$ and the skew-symmetric part $v_a = v - v_s$ solves the linear system $Tv_a = w - w_s$.

Using this property, it is possible to setup a two-way Lanczos process which works extracting simultaneously symmetric and skew-symmetric eigenvalues. Each "way" extracts the smallest eigenvalues of one of the two symmetry classes, applying the inverted Lanczos method to solve an eigenvalue problem with size m (half the size of the original problem). Both Lanczos processes work in parallel, until the solution of the linear system is needed. Then, a single linear system is solved, using the above property, and the symmetric (and skew-symmetric) parts of that vector are extracted as the next vectors of the Lanczos recurrence (see [18] for the details). Two tridiagonal matrices are built, one for each symmetry class:

$$\widetilde{SYM}_k = \begin{bmatrix} \alpha_1 & \beta_1 & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & \ddots & \beta_{k-1} \\ & & \beta_{k-1} & \alpha_k \end{bmatrix};$$

$$\widetilde{SKS}_j = \begin{bmatrix} \gamma_1 & \delta_1 & & \\ \delta_1 & \gamma_2 & \ddots & \\ & \ddots & \ddots & \delta_{k-1} \\ & & \delta_{k-1} & \gamma_k \end{bmatrix}, \tag{3}$$

plus two orthogonal matrices $P_k = [p_1, p_2, \ldots, p_k]$ and $Q_k = [q_1, q_2, \ldots, q_k]$ containing the Lanczos vectors of each symmetry class.

The following algorithm describes the process. It is basically the same algorithm described in [18], but for the shifted case and, since many eigenvalues are desired, reorthogonalizations are needed.

Algorithm 2. "Shift-and-Invert" Two-way Lanczos method.

Given $T \in \mathbb{R}^{n\times n}$ a symmetric Toeplitz matrix, this algorithm returns the eigenvalues closest to the shift σ and the associated eigenvectors.

1. Let $p_1 = J_n p_1 \neq 0$ and $q_1 = -J_n q_1 \neq 0$ initial vectors
2. Let $p_0 = q_0 = 0$; $\beta_0 = \delta_0 = 0$
3. $p_1 = p_1 / \|p_1\|$; $q_1 = q_1 / \|q_1\|$
4. for $j = 1, 2, \ldots$ until convergence:
5. $w = p_k + q_k$
6. solve $(T - \sigma I)\ v = w$
7. $v_s = 0.5 \cdot (v + J_n v)$; $v_a = 0.5 \cdot (v - J_n v)$
8. $\alpha_k = v_s^t \cdot p_k$; $\gamma_k = v_a^t \cdot q_k$
9. $v_s = v_s - \alpha_k \cdot p_k$ $v_a = v_a - \gamma_k \cdot q_k$
   $-\beta_{k-1} \cdot p_{k-1}$ $-\delta_{k-1} \cdot q_{k-1}$
10. full re-orthogonalization
11. $\beta_k = \|v_s\|_2$; $\delta_k = \|v_a\|_2$
12. $p_{k+1} = v_s / \beta_k$; $q_{k+1} = v_a / \delta_k$
13. obtain eigenvalues of $\widetilde{SYM}_k$; obtain eigenvalues of $\widetilde{SKS}_k$;
14. test bounds for convergence ;
15. end for
16. compute associated eigenvectors.

In Voss' paper, the smallest eigenvalue was found obtaining the smallest eigenvalues in both symmetry classes, and comparing the extracted eigenvalues. For our problem (to extract all the eigenvalues of an interval), the symmetry exploiting Lanczos process is even more appealing, since we need to extract as many eigenvalues as possible, with the fewest Lanczos iterations. In the smallest eigenvalue problem, the computational effort of one of the symmetry classes (the one that the smallest eigenvalue does not belong to) is, in some sense, lost. In our case problem, all the eigenvalues of the interval must be extracted, so that we will collect the eigenvalues from both classes, taking full profit of the algorithm.

This reorganization of the Lanczos method should double the speed of the standard Lanczos method (measured as number of eigenvalues extracted per Lanczos iteration). This is usually confirmed by experiments, but, furthermore, in many cases we have detected that, with the same number of iterations, the two-way shift and invert Lanczos algorithm extracts three or four times the number of eigenvalues that extracts the standard "Shift-and-Invert" Lanczos iteration, (of course, using the same shift). As pointed out by Voss, this is due to the improvement of relative separation between eigenvalues.

However, the efficiency of the method will depend on the fast resolution of the linear systems $(T - \sigma I)\ v = w$ in line 6 of Algorithm 2; next, we will describe a fast solver for these problems.

2.4. Solution of symmetric Toeplitz linear systems

There are several solvers available for symmetric Toeplitz linear systems. Maybe the best known solvers for this problem are those based on Levinson's algorithm. A different approach is the decomposition of the symmetric Toeplitz linear system into two Cauchy-like linear systems, described below. In our case, this method has given better results than the based on Levinson's algorithm, and we have taken it as the basic algorithm for experimentation.

2.4.1. Solution of symmetric Toeplitz linear systems through decomposition in Cauchy-like systems

For the solution of a symmetric Toeplitz linear system

$$T\,x = b\,, \tag{4}$$

we split the linear system in two linear systems of an order of about the half the original one as follows.

Given the normalized Discrete Sine Transformation (DST) $\mathcal{S}$ as defined in [17], the Toeplitz linear system (4) is transformed into

$$C\,\bar{x} = \bar{b}\,, \tag{5}$$

where $C = \mathcal{S}T\mathcal{S}$, $\bar{x} = \mathcal{S}x$ and $\bar{b} = \mathcal{S}b$, since $\mathcal{S}$ is symmetric and orthogonal.

Matrix $C = [c_{ij}]_{i,j=0}^{n-1}$ is known as a *Cauchy-like* matrix. It has the following property: $c_{i,j} = 0$ if $i + j$ is odd, that is, about the half of the entries are zero. To exploit this feature we define the odd–even permutation matrix P_{oe} as the matrix that, after applied to a vector, groups the odd entries in the first positions and the even entries in the last ones, $P_{oe}(\ x_1\ \ x_2\ \ x_3\ \ x_4\ \ x_5\ \ x_6\ \ \ldots\ )^{\mathrm{T}} = (\ x_1\ \ x_3\ \ x_5\ \ \ldots\ \ x_2\ \ x_4\ \ x_6\ \ \ldots\ )^{\mathrm{T}}$. Applying transformation $P_{oe}(\cdot)P_{oe}^{\mathrm{T}}$ to the symmetric Cauchy-like matrix C (5)

$$P_{oe} C P_{oe}^{\mathrm{T}} = \begin{pmatrix} C_0 & \\ & C_1 \end{pmatrix}, \tag{6}$$

gives the symmetric Cauchy-like matrices C_0 and C_1 of order $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively.

Thus, the linear system (4) can be solved by solving the following two Cauchy-like systems

$$C_j\hat{x}_j = \hat{b}_j, \quad j = 0, 1, \tag{7}$$

being $\hat{x} = \left(\hat{x}_0^{\mathrm{T}} \quad \hat{x}_1^{\mathrm{T}}\right)^{\mathrm{T}} = P_{oe}\mathcal{S}x$ and $\hat{b} = \left(\hat{b}_0^{\mathrm{T}} \quad \hat{b}_1^{\mathrm{T}}\right)^{\mathrm{T}} = P_{oe}\mathcal{S}b$.

Both Cauchy-like systems can be solved efficiently (in $O(n^2)$ steps) using the LDL^{T} decomposition for Cauchy-like matrices, described below, in Appendix A. Therefore, we could solve a single linear system (4) using the following Algorithm:

Algorithm 3. Solution of a symmetric Toeplitz system with Cauchy-like transformation.

Given $T \in \mathbb{R}^{n\times n}$ a symmetric Toeplitz matrix and $b \in \mathbb{R}^n$ an independent term vector, this algorithm returns the solution vector $x \in \mathbb{R}^n$ of the linear system $Tx = b$.

1. Obtain $C_0, C_1, \hat{b}_0, \hat{b}_1$ through DST $\mathcal{S}$ and permutation P_{oe}
2. Compute LDL^{T} decompositions: $C_0 = L_0D_0L_0^{\mathrm{T}}$ and $C_1 = L_1D_1L_1^{\mathrm{T}}$
3. Solve $L_0D_0L_0^{\mathrm{T}}\hat{x}_0 = \hat{b}_0$ and $L_1D_1L_1^{\mathrm{T}}\hat{x}_1 = \hat{b}_1$
4. Compute $x = \mathcal{S}P_{oe}^{\mathrm{T}}\begin{pmatrix}\hat{x}_0\\ \hat{x}_1\end{pmatrix}$.

In our problem, for each subinterval, we need to solve several linear systems (one per Lanczos iteration) with the same coefficient matrix. Therefore, steps 1 and 2 in Algorithm 3 will be carried out only once for each subinterval; for each new linear system, only steps 3 (back substitution) and 4 apply DST and reorder solution) are needed.

The algorithms needed to compute the LDL^t decomposition were published in [7]. For the sake of completeness, and because they are needed for an efficient implementation of the interval selection, they have been included in Appendix A.

2.5. Overall method

The "Shift-and-Invert" two-way Lanczos algorithm, along with the technique described above to solve the Toeplitz linear systems of equations, allows the efficient computation of all the eigenvalues of the matrix located in the neighborhood of the shift σ. If we desire the computation of all the eigenvalues of the matrix (or all the eigenvalues in a, possibly large, interval) it can be done by finding first a large interval containing all the desired eigenvalues, and slicing this large interval in small subintervals. Then, a shift can be selected in the middle of each subinterval, and the "Shift-and-Invert" iterative method can be applied to extract all the eigenvalues of the subinterval. We will denote this algorithm as Full Spectrum Two-Way Lanczos based algorithm (FSTW Lanczos), although it can be used to compute the spectrum contained in any interval.

Algorithm 4. Overall Method: FSTW Lanczos.

1. Choose the interval $[a, b]$ containing the desired eigen values
2. Divide the interval $[a, b]$ in small subintervals
3. for each subinterval: (* in parallel *)
4. Compute a "shift" σ, possibly $\sigma = (a + b)/2$
5. Decompose the matrix $(T - \sigma I)$ in Cauchy-like matrices C_0 and C_1, as shown in Section 2.4.1
6. Obtain the LDL^{T} decompositions of the Cauchy-like matrices (Algorithm A.1)
7. Apply "Shift-and-Invert" two-way Lanczos method (Algorithm 2) to extract all the eigenvalues in subinterval and the associated eigenvectors
9. end for
10. end algorithm.

The basic idea behind this algorithm, the "spectrum slicing", was already proposed in [9,19] for a different problem, the symmetric generalized eigenvalue problem for sparse matrices. Apart from the obvious differences (the Two-Way Lanczos process and the fast Toeplitz linear system solver), there are more differences in the technique for interval selection, which will be discussed in Section 2.5.1.

The main memory requirement of this algorithm is the storage of the triangular factors of the Cauchy-like matrices. If the memory needed is too large, the symmetric Toeplitz solver might be changed to a suitable version of the Levinson's algorithm. The memory needed should be then much less, although the computing time should increase.

It is clear that the procedure in each subinterval is independent of the other subintervals, so that this algorithm parallelizes trivially, just assigning different subintervals to different processors. However, the efficiency of the method (both in parallel and sequential computing) will depend on a good choice of the subintervals. If some of the subintervals have too many eigenvalues or are too wide, then many Lanczos iterations (and, possibly many restarts) will be needed. Therefore, the selection of the subintervals is another important part of the algorithm.

2.5.1. Selection of the intervals

There are many factors that must be taken into account, when determining the subintervals. First, it takes a few Lanczos iterations until the eigenvalues start to be extracted (usually a minimum of 5–7 iterations until the first eigenvalue converges). After that, new eigenvalues converge quite fast, specially with the symmetry–exploiting Lanczos version. This would demand subintervals with many eigenvalues. A typical result would be to obtain 25–30 eigenvalues after 40 iterations.

On the other hand, Algorithm 2 has been implemented with full reorthogonalization; this means that the computational cost grows after each iteration. Furthermore, to deal with difficult cases and with multiple eigenvalues, a maximum number of Lanczos iterations (maximum dimension of the Krylov space) is set. When this number of iterations is reached, the algorithm performs a explicit restart (the method starts again, but reorthogonalizing the starting vector with respect to the already converged eigenvectors). This mechanism gives robustness to the algorithm, but when a restart takes place there is an important loss of efficiency. These two factors indicate that the number of eigenvalues on each subinterval must not be too large, and that the width of the subinterval must be controlled as well, since very large subintervals with eigenvalues in the extremes would slow down the convergence.

A reasonable maximum number of eigenvalues would be between 30 and 50 in each interval, although of course this is problem-and-implementation dependent. We have chosen this number as 40.

The tool for an efficient subinterval selection is the Inertia Theorem [14]. Given an interval $[\alpha, \beta]$, this theorem can be used to find out how many eigenvalues are in the interval. This could be done computing the LDL^{T} decompositions of $T-\alpha I$ (equal to $L_\alpha D_\alpha L_\alpha^{\mathrm{T}}$) and $T - \beta I$ (equal to $L_\beta D_\beta L_\beta^{\mathrm{T}}$). Then, the number of eigenvalues in the interval $[\alpha, \beta]$ is simply the number $\nu(D^\beta) - \nu(D^\alpha)$, where $\nu(D)$ denotes the number of negative elements in the diagonal D. In our case, this number can be computed efficiently using Algorithm A.1 (see Appendix A). Furthermore, since only the count of signs in D is needed, it is possible to obtain a modified version of Algorithm A.1 where neither L nor D are stored.

Once an appropriate number of eigenvalues has been decided, the main interval containing all the desired eigenvalues is divided in a reasonable number of subintervals, choosing division points σ_i where the Inertia Theorem is used to determine the number of

Table 1
CPU time (s) for isolation and extraction phases

n	Isolation time	% of total time	Extraction time	% of total
1 000	0.43	8.3	4.73	91.6
5 000	19.39	7.1	251.88	92.8
10 000	137.01	6.6	1933.76	93.4
15 000	442.02	5.7	7267.41	94.3

eigenvalues in the left and in the right of the chosen point. Then, applying the Inertia Theorem in each new point, a bisection–like search is performed, until we obtain a division of the main interval where none of the subintervals has more eigenvalues than the chosen number, and none is wider than a pre–set tolerance. Empty subintervals (without eigenvalues) are automatically discarded.

As a final result, a set of subintervals will be obtained which include the full spectrum; and, as a by–product, the exact number of eigenvalues in each subinterval is obtained. This is very useful to avoid unnecesary iterations in the FSTW Lanczos based algorithm.

The strategies followed in [9,19] are quite different. There, the matrices considered are symmetric sparse matrices, and the factorizations needed for the "Shift-and-Invert" method and for the computation of inertias are relatively more expensive. In both references, the computation of LDL^t factorizations was minimized by reusing the factorizations with two goals, computation of inertias and computation of eigenvalues. In [9] the algorithm is oriented to sequential processing; the algorithm starts by computing all the eigenvalues in an interval, and then extends that interval progressively; of course, this is not appropriate for parallel processing.

In [19] (where the proposed algorithm is oriented to parallel processing) the main interval is initially split in subintervals, assigned to processors, and a first shift is computed at the midpoint of each subinterval. The eigenvalues close to this shift are computed, using the associated factorization. Then, in each interval new shifts are selected (and the LDL^t factorizations computed), at the left and right of the main shift so that the number of eigenvalues in the subinterval contained between these new shifts is determined. This information is spread among the processors, so that the limits of the subintervals are adaptively adjusted, and therefore the workload is balanced. This process is repeated as many times as needed, switching between computation of eigenvalues close to the new shifts and computation of inertias, to progressively balance the workload.

The algorithm proposed in this paper takes advantage of the efficient computation of the LDL^t decomposition of symmetric Toeplitz matrices to avoid these problems; a balanced set of intervals is computed prior to the computation of any eigenvalue, which ensures the load balance. Although these factorizations are not reused, the scalability analysis detailed below seems to confirm that this relatively simple choice gives good overall results.

2.5.2. Enforcing orthogonality

The orthogonality among the eigenvectors computed with the same shift is guaranteed, thanks to the "full reorthogonalization" strategy. Also, the eigenvectors associated with well separated eigenvalues do not suffer of orthogonality problems [13]. However, it is possible that loss of orthogonality may appear when very close eigenvalues are computed (along with their eigenvectors) using different shifts. This may happen when a cluster of eigenvalues lie just in the limit between two subintervals.

A simple cure for this problem consists in extending the limits of both subintervals, so that there is a small overlap (we have chosen to extend both subintervals just a 1%, relative to the length of the smaller subinterval; this is enough to guarantee enough relative separation between eigenvalues outside the overlap zone).

Each eigenvalue in the overlap zone will be computed twice, using both shifts (so that orthogonality shall be achieved), but, once the computing has finished, one of the processors (the one that originally did not had the repeated eigenvalue in its interval) will discard the repeated eigenvalue.

This strategy enforces orthogonality of the computed eigensolutions, at the cost of duplicating the computation of some eigenpairs. The extra cost of this technique is two new LDL^t decompositions for each interval (which are computed very fast) and some extra Lanczos iterations when there is a cluster exactly in the limit between two subintervals, but, even in this case, the extra CPU time is small, thanks to the fast convergence of the FSTW algorithm.

3. Implementation and parallelization of the method

The sequential version of Algorithm 4 has been implemented in Fortran 95 (Intel Fortran Compiler 8.1), using BLAS and LAPACK kernels (Intel MKL 8.1). To give robustness to the algorithm, it has been implemented using explicit restarts to account for the non-converged eigenvalues, and with full reorthogonalization. The eigenvalues and eigenvectors of the matrices $\widetilde{SYM}_k$ and $\widetilde{SKS}_k$ are computed using the routine DSTEVR from LAPACK.

The convergence tests were greatly simplified since it is known in advance the number of eigenvalues in each subinterval.

As mentioned above, the maximum number of eigenvalues per subinterval was chosen as 40. If this value is larger, isolation time (steps 1 and 2) will decrease while extraction time (steps 3–10) will increase. For the chosen value (40), the times of extraction and isolation phases are displayed in Table 1 for different problem sizes.

3.1. Parallelization

We have developed a distributed memory parallel version of Algorithm 4 using MPI [10].

The extraction phase of Algorithm 4 starts in step 3 of the algorithm and parallelizes trivially. As the Table 1 shows, the isolation phase has a much smaller cost, but still it represents a non-negligible portion of the execution time. So, it has been parallelized following this template:

Algorithm 5. Parallel Isolation of Intervals.

1. Choose the interval $[a, b]$ containing the desired eigenvalues
2. Divide interval $[a, b]$ into p subintervals $[a_i, b_i]$ of equal length
3. for each processor $i = 0, \ldots, p-1$
4. Apply the bisection–like technique described in section 2.5.1 to the i–th subinterval
5. end for
6. Gather all the subintervals in the master node.

In the parallel Algorithm 4, the extraction step has been structured using a master–slave paradigm, where the master holds a

Table 2
CPU time (s) for sequential Algorithm 4 and LAPACK, obtaining all the eigenvalues

	$N = 1000$	$N = 5000$	$N = 10\,000$	$N = 15\,000$
FSTW Lanczos	5.25	271.48	2071.33	7710.34
LAPACK (DSYEVR)	2.27	225.57	1754.64	No mem

Table 3
CPU time (s) for parallel Algorithm 4 and ScaLAPACK, obtaining all the eigenvalues

Processors	$N = 1000$		$N = 5000$	
	FSTW Lanczos	PDSYEVD	FSTW Lanczos	PDSYEVD
$4(1+3)$	1.64	3.59	94.49	137.10
$8(1+7)$	0.73	2.38	39.80	81.69
$16(1+15)$	0.39	1.32	19.90	42.17
$32(1+31)$	0.34	4.17	10.91	30.45
$48(1+47)$	0.33	4.60	7.11	21.77
Processors	$N = 10\,000$		$N = 15\,000$	
	FSTW Lanczos	PDSYEVD	FSTW Lanczos	PDSYEVD
$4(1+3)$	702.79	925.92	2545.08	2184.28
$8(1+7)$	305.66	488.83	1103.00	1146.04
$16(1+15)$	143.22	251.37	570.66	691.48
$32(1+31)$	75.92	152.31	278.39	454.38
$48(1+47)$	54.61	112.01	192.70	298.57

queue with the results of the previous step, and assigns them to the slaves. When a slave finishes the processing of a subinterval, asks for another to the master, until the queue is empty. Each slave processes the subintervals in a purely sequential form.

Finally, it can be observed that the algorithm offers some extra levels of parallelism, such as applying both "ways" of Algorithm 2 in parallel, or solving in parallel both Cauchy-like systems (Eq. (7)). These possibilities are currently under research. However, the gains obtained are not so relevant as the obtained with this simple interval-based parallelization.

4. Experimental results

The codes developed have been tested using randomly generated full symmetric Toeplitz matrices, with sizes $N = 1000$, 5000, 10 000, 15 000. The problems (computation of eigenvalues and eigenvectors) were solved with different number of processors, in a cluster with 55 Intel Xeon biprocessors, interconnected with a SCI network. The same problems were solved using LAPACK with 1 processor, and ScaLAPACK in the cases with several processors.

The spectrum of these matrices is reasonably well spread, although always appears a large eigenvalue, quite far away from the others. We have tested as well our algorithms with Prolate Toeplitz matrices, as can be generated with Matlab's "gallery" command [15]; these are full matrices, extremely bad conditioned, and have most eigenvalues clustered around 0 or 1. Despite the difficulties, we could not find significative differences between the results (neither in execution time, nor accuracy) with these matrices and with the randomly generated.

We have observed significative speed differences among the available LAPACK and ScaLAPACK routines; after testing several of them, we found DSYEVR (in LAPACK) and PDSYEVD (in ScaLAPACK) to be the fastest in these problems, and have used them for the comparison. To perform this comparison, the full symmetric Toeplitz matrix had to be allocated (distributed, in the case with several processors); the case with $N = 15\,000$ could not be executed with a single processor, because the processor did not have enough memory.

The main test was to compute the full spectrum of the matrices. The time results are summarized in Tables 2 and 3.

The column on the left shows the number of processors; the format $(1+x)$ has been chosen to remark that the parallel program has a master + slave structure, and therefore there is a processor relatively "idle".

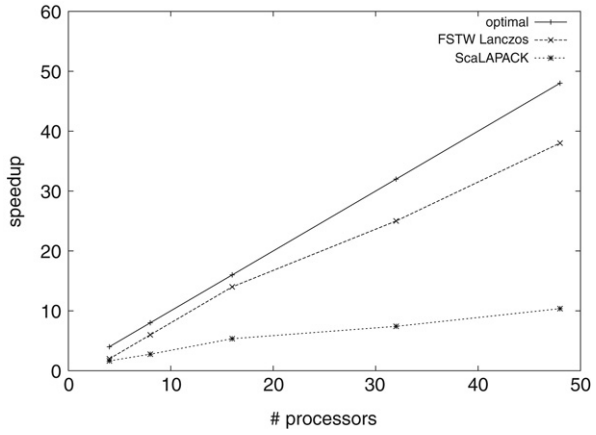

Fig. 1. Speedup for the case $N = 5000$.

The times obtained reveal that, while LAPACK is somewhat faster than Algorithm 4 running sequentially, the parallel version of our algorithm is faster than ScaLAPACK in most cases, and the differences become greater when the number of processors grows.

Figs. 1 and 2 display the speedups for the cases $N = 5000$, $N = 10\,000$. It is quite clear that the FSTW Lanczos obtains excellent speedups, and the advantage with respect to PDSYEVD grows with the size of the problem.

If a fraction of the spectrum is desired, then the Lanczos-based algorithm is comparatively more efficient. The reason for this lies in the way in which LAPACK and ScaLAPACK routines work. These routines carry out first a tridiagonalization of the matrix, and then extract the eigenvalues of the tridiagonal matrix. The main computational task is the tridiagonalization of the matrix; this has a fixed cost that cannot be reduced, even if not all the spectrum is computed.

We have carried out experiments computing a fraction of the spectrum, with 10%, 25% and 50% of the spectrum computed, in the cases with 1 and 16 processors (a 4×4 processor grid for ScaLAPACK routines). In this case, we have compared against the Bisection routines of LAPACK (DSTEBZ) and ScaLAPACK (PDSTEBZ) since these subroutines allow to compute only the portion desired

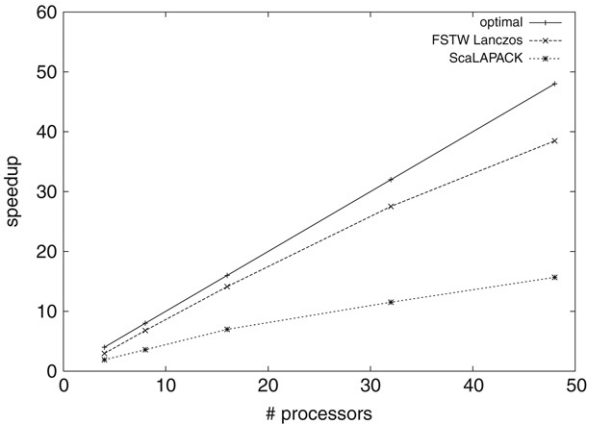

Fig. 2. Speedup for the case $N = 10\,000$.

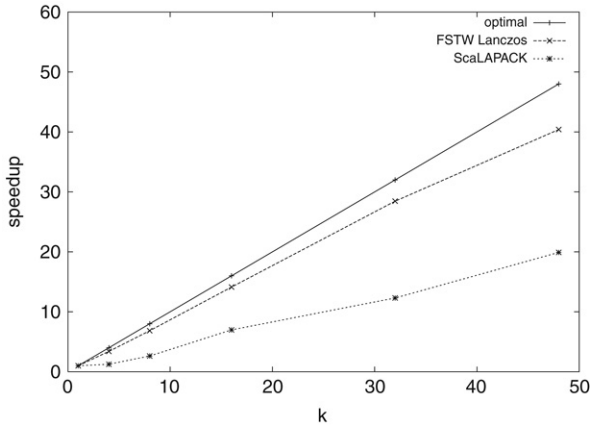

Fig. 3. Scaled speedup.

Table 4
CPU time (s) for Algorithm 4, LAPACK and ScaLAPACK, obtaining 10%, 20% or 50% of the eigenvalues and eigenvectors

	$P = 1, N = 1000$		$P = 1, N = 5000$	
	FSTW Lanczos	DSTEBZ	FSTW Lanczos	DSTEBZ
10%	0.52	1.32	32.07	129.96
20%	1.16	2.09	82.81	199.19
50%	2.50	4.32	194.98	441.83
	$P = 16(4 \times 4), N = 5000$		$P = 16(4 \times 4), N = 10\,000$	
	FSTW Lanczos	PDSTEBZ	FSTW Lanczos	PDSTEBZ
10%	3.03	23.12	26.00	123.99
20%	6.69	23.74	52.41	131.60
50%	14.55	24.78	99.06	152.37

P is the number of processors.

of the spectrum, and therefore they seem more appropriate for this experiment. The results are summarized in Table 4. (It must be noted that the LAPACK routine DSTEBZ only compares well with DSYEVR when a small fraction of the spectrum must be computed; for example, it can be seen in the $N = 5000$ case that computing the full spectrum with DSYEVR is faster than computing 50% of the spectrum with DSTEBZ.)

Clearly, when the fraction of spectrum to be computed diminishes, the fixed cost of tridiagonalization becomes more important for the LAPACK and ScaLAPACK routines, and our algorithm becomes comparatively faster.

4.1. Scalability analysis

We study now the scalability of the parallel algorithm. The scalability of a parallel system is a measure of its capacity to increase speedup in proportion to the number of processing elements. It reflects a parallel systems ability to utilize increasing processing resources effectively [8]. In our case we use the scaled speedup as defined in Reference [8] due to its ease of use when experimental data are available. The scaled speedup SS_p is defined as the speedup obtained when the problem size is increased linearly with the number of processing elements:

$$SS_p = \frac{T_1(kW)}{T_p(kp, kW)} \tag{8}$$

being W the problem size, p an initial number of processors and k the number of times W and p are increased.

Thus, the scaled speedup can be plotted as a function of k in order to analyse the behaviour of the scalability of a parallel algorithm in a determined parallel computer. A system (an algorithm–computer pair) is considered scalable if the scaled speedup is close to linear with respect to the number of processing elements. To analyse the scalability, the problem size must be scaled with the number of processors, where the problem size is defined as the number of computations carried out in the best sequential algorithm.

In this case we consider that the theoretical computational cost of the sequential algorithm is $W = O(n^3)$ (using the results of the Tables 2 and 3 it is easy to check that this is a reasonable assumption). Thus, scaling the number of processors p and the problem size W in the same ratio implies that if we scale p as $p = kp_0$ then W must be scaled as

$$W = k \cdot W_0 = k \cdot O(n_0^3) = O(k^{1/3} \cdot n_0)^3 \tag{9}$$

that means that n must be scaled by $k^{1/3}$.

In Fig. 3 we have plotted the scaled speedup, for both the ScaLAPACK parallel subroutine PDSYEVD and our FSTW Lanczos parallel algorithm. We have considered as initial points $p_0 = 1$, $n_0 = 1000$ and we have increased p_0 and n_0 by k and $k^{1/3}$ respectively, for $k = 1, 4, 8, 16, 32, 48$. For the case $k = 48$, the size of the target matrix corresponds to $n = 15\,000$. The memory required for LAPACK subroutine for this size is very large and it has been not possible to run it in a single processor of our system. Hence, in order to compute the speedup for the LAPACK routine in this case, we have taken an estimation of T_1 by assuming $T_1(n) = O(n^3)$, thus allowing to estimate SS_{48}:

$$SS_{48} = \frac{T_1(n = 15\,000)}{T_{48}(n = 15\,000)}. \tag{10}$$

As it can be observed in the Fig. 3, the scaled speedup is close to linear behaviour in the case of the FSTW Lanczos parallel algorithm thus indicating a good scalability of the developed algorithm. Moreover, the behaviour is clearly superior to that of the ScaLAPACK subroutine. This is due to the fact that our algorithm does not require communications, practically during all its life cycle. This is not similar for the ScaLAPACK subroutine.

5. Conclusions

A new algorithm, combination of other known algorithms, has been proposed for computation of many eigenvalues and eigenvectors of symmetric Toeplitz matrices. The algorithm can be executed sequentially, but it is intrinsically parallel. It has shown to be faster that ScaLAPACK routines, and the comparison is still better if only part of the spectrum needs to be computed.

A further advantage of this algorithm is its low memory need. In its sequential form, it can handle problems which cannot be solved by LAPACK. Additionally, changing the fast solver described

in Section 2.4 to a Levinson-type solver, the memory needed would be lower still, allowing the computation of eigenvalues of very large matrices.

As expected, the scalability of the algorithm is very good (far better than the ScaLAPACK routines).

Acknowledgment

This work has been supported by Spanish MCYT and FEDER under Grant TIC2003-08238-C02-02, and by Programa de Incentivo a la Investigacion UPV-Valencia (SPAIN), 2005 Project 005522.

Appendix. Efficient LDL^{T} decomposition of Cauchy-like matrices

The LDL^{T} decomposition of the Cauchy-like matrices appearing in Eq. (7) can be obtained in $O(n^2)$ steps by exploiting the *displacement property* of the structured matrices. As Kailath, Kung and Morf introduced in [11], a matrix of order n is *structured* if its *displacement representation* has a lower rank regarding n. The *displacement representation* of a symmetric Toeplitz matrix T can be defined in several ways depending on the form of the displacement matrices. A useful form for our purposes is

$$\nabla_F T = F\,T - T\,F = \mathcal{G}\,\mathcal{H}\,\mathcal{G}^{\mathrm{T}}; \tag{A.1}$$

where F, also called *displacement matrix*, is the symmetric Toeplitz matrix defined as $F = [f_{ij}]_{i,j=0,\ldots,n-1}$, being all entries 0 except those where $i = j+1$ or $j = i+1$ which are 1; $\mathcal{G} \in \mathbb{R}^{n\times 4}$ is the *generator* matrix and $\mathcal{H} \in \mathbb{R}^{4\times 4}$ is a skew-symmetric *signature* matrix. The rank of $\nabla_F T$ is 4, that is, quite low and independent of n.

Matrices C_0 and C_1 (6) are also structured. The displacement representation for them is easily obtained by applying the following two transformations to (A.1). First, with $\mathcal{S}(\cdot)\mathcal{S}$, we obtain the displacement representation of the Cauchy-like matrix C in (5):

$$\mathcal{S}(FT - TF)\mathcal{S} = \mathcal{S}(\mathcal{G}\mathcal{H}\mathcal{G}^{\mathrm{T}})\mathcal{S} \;\rightarrow\; \Lambda C - C\Lambda = \hat{\mathcal{G}}\mathcal{H}\hat{\mathcal{G}}^{\mathrm{T}}, \tag{A.2}$$

where $\hat{\mathcal{G}} = \mathcal{S}\mathcal{G}$ and being $\Lambda = \mathcal{S}F\mathcal{S}$ a diagonal matrix. Next, transformation $P_{oe}(\cdot)P_{oe}^{\mathrm{T}}$ is applied to (A.2) to obtain

$$\begin{pmatrix}\Lambda_0 & \\ & \Lambda_1\end{pmatrix}\begin{pmatrix}C_0 & \\ & C_1\end{pmatrix} - \begin{pmatrix}C_0 & \\ & C_1\end{pmatrix}\begin{pmatrix}\Lambda_0 & \\ & \Lambda_1\end{pmatrix} = P_{oe}\hat{\mathcal{G}}\mathcal{H}\hat{\mathcal{G}}^{\mathrm{T}}P_{oe}^{\mathrm{T}}, \tag{A.3}$$

where, again, Λ_0 and Λ_1 are diagonal matrices. Equating each of the two Cauchy-like matrices we have

$$\Lambda_j C_j - C_j \Lambda_j = G_j H_j G_j^{\mathrm{T}}, \quad j = 0, 1, \tag{A.4}$$

where $H_0 = H_1 = \begin{pmatrix}0 & 1\\ -1 & 0\end{pmatrix}$. Furthermore, the rank of the displacement representation of each of the two Cauchy-like matrices is 2 instead of 4 as it is in the case of T and C. The significance of the displacement representation of a structured matrix falls in the fact that the triangularization algorithm works on the generators G_0 and G_1. If the first column of the symmetric Toeplitz matrix T is $(t_0, \quad t_1, \quad \cdots, \quad t_{n-2}, \quad t_{n-1})^{\mathrm{T}}$, and using (A.1)–(A.4), an explicit form for the generators G_0 and G_1 can be derived:

$$G = (G_{:,0}, G_{:,1}) = \begin{pmatrix}G_0\\ G_1\end{pmatrix} = \sqrt{2}P_{oe}\mathcal{S}\,(u \quad e_0), \tag{A.5}$$

being $u^{\mathrm{T}} = (0, \quad t_2, \quad t_3, \quad \cdots, \quad t_{n-2}, \quad t_{n-1}, \quad 0)^{\mathrm{T}}$ and being e_0 the first column of the identity matrix.

Algorithm A.1 summarizes the steps to perform the LDL^{T} decomposition of a symmetric Cauchy-like matrix.

Algorithm A.1 (LDL^{T} *Decomposition of Symmetric Cauchy-Like Matrices*). Let $G \in \mathbb{R}^{n\times 2}$ be the generator, $H \in \mathbb{R}^{2\times 2}$ be the signature matrix, λ be an array with the diagonal entries of $\Lambda \in \mathbb{R}^{n\times n}$ of the displacement of a symmetric Cauchy-like matrix C of the form (A.4) and the diagonal entries of C ($c_{k,k}$, for $k = 0, \ldots, n-1$) obtained from Algorithm A.2, this algorithm returns a unit lower triangular matrix L and the diagonal entries of a diagonal factor D, stored in the diagonal of L, of the LDL^{T} decomposition of C.

```
for k = 0, ..., n − 1
    d = c_{k,k} .
    l_{k,k} = d .
    for i = k + 1, ..., n − 1
        c_{i,k} = (G_{i,:} H G_{k,:}^T)/(λ_i − λ_k) .
        l_{i,k} = c_{i,k}/d .
        c_{i,i} ← c_{i,i} − d l_{i,k}2 .
        g_{i,0} ← g_{i,0} − g_{k,0} l_{i,k} .
        g_{i,1} ← g_{i,1} − g_{k,1} l_{i,k} .
    end for
end for
```

Algorithm A.1 receives a generator (G_0 or G_1 (A.5)), the signature matrix H and the displacement matrix (Λ_0 or Λ_1). The algorithm is based on the property that the Schur complement of a structured matrix is also structured and on the fact that the entries off the main diagonal of a symmetric Cauchy-like matrix are implicitly known through the generator and the signature matrix. Each (i, j)th element of C can be computed as

$$C_{i,j} = \frac{G_{i,:}HG_{j,:}^{\mathrm{T}}}{\lambda_i - \lambda_j}, \quad \text{iff } \lambda_i \neq \lambda_j, \tag{A.6}$$

that is, for all elements off the main diagonal. The main diagonal elements of C must be computed prior to the start of the factorization since they cannot be computed by means of (A.6), so these entries are an entry to Algorithm A.1. The computation of the diagonal entries of a symmetric Cauchy-like matrix can be carried out in $O(n \log n)$ operations using the DST, as it is shown in Algorithm A.2 [6].

Algorithm A.2. Computation of the diagonal of a symmetric Cauchy-like matrix.

Let $(t_0, t_1, \ldots, t_{n-1})^{\mathrm{T}}$ be the first column of a symmetric Toeplitz matrix T and $\mathcal{S}$ the DST transformation matrix, this algorithm returns the diagonal of the Cauchy-like matrix $C = \mathcal{S}T\mathcal{S}$ in array c.

```
r_0 = nt_0
r_1 = 2(n − 1)t_1
w_0 = −t_0
v_0 = −2t_1
for k = 2, ..., ⌊n/2⌋
    r_{2k−2} = 2(n − 2k + 2)t_{2k−2} + 2w_{k−2}
    w_{k−1} = w_{k−2} − 2t_{2k−2}
    r_{2k−1} = 2(n − 2k + 1)t_{2k−1} + 2v_{k−2}
    v_{k−1} = v_{k−2} − 2t_{2k−1}
end for
if n is odd
    r_{n−1} = 2t_{n−1} + 2w_{((n−1)/2)−1}
end if
c = Sr .
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 1999.

[2] J.M. Badia, A.M. Vidal, Parallel algorithms to compute the eigenvalues and eigenvectors of symmetric Toeplitz matrices, Parallel Algorithms and Applications 13 (1998) 75–93.

[3] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. Van der Vorst (Eds.), Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, 1st edition, SIAM, Philadelphia, 2000.

[4] L. Blackford, et al., ScaLAPACK Users' Guide, SIAM, Philadelphia, 1997.

[5] J.R. Bunch, Stability of methods for solving Toeplitz systems of equations, SIAM Journal on Scientific and Statistical Computing 6 (1985) 349–364.

[6] R.H. Chan, M.K. Ng, C.K. Wong, Sine transform based preconditioners for symmetric Toeplitz systems, Linear Algebra and its Applications 232 (1–3) (1996) 237–259.

[7] I. Gohberg, T. Kailath, V. Olshevsky, Fast Gaussian elimination with partial pivoting for matrices with displacement structure, Mathematics of Computation 64 (212) (1995) 1557–1576.

[8] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, 2nd edition, Pearson Education Limited, 2003.

[9] R.G. Grimes, J.G. Lewis, H.D. Simon, A shifted block Lanczos algorithm for solving Sparse symmetric generalized eigenproblems, SIAM Journal on Matrix Analysis and Applications 15 (1994) 228–272.

[10] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Portable Parallel Programming with Message Passing Interface, MIT Press, 1994.

[11] T. Kailath, S.Y. Kung, M. Morf, Displacement ranks of a matrix, Bulletin of the American Mathematical Society 1 (1979) 769–773.

[12] M.K. Ng, W.F. Trench, Numerical solution of the eigenvalue problem for Hermitian Toeplitz-like matrices, The Australian National University, TR-CS-97-14, 1997, http://cs.anu.edu.au/techreports/1997/TR-CS-97-14.pdf.

[13] B. Parlett, The Symmetric Eigenvalue Problem, Prentice Hall, Englewood Cliffs, NJ, 1980.

[14] A. Ruhe, Hermitian eigenvalue problem; Lanczos method, in: Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, 1st edition, SIAM, Philadelphia, 2000.

[15] The Mathworks Inc., MATLAB R14 Natick MA, 2004.

[16] W.F. Trench, Numerical solution of the eigenvalue problem for Hermitian Toeplitz matrices, SIAM J. Matrix Th. Appl. 9 (1988) 291–303.

[17] C. Van Loan, Computational Frameworks for the Fast Fourier Transform, SIAM Press, Philadelphia, 1992.

[18] H. Voss, A symmetry exploiting Lanczos method for symmetric Toeplitz matrices, Numerical Algorithms 25 (2000) 377–385.

[19] H. Zhang, B. Smith, M. Sternberg, P. Zapol, SIPs: Shift-and-Invert parallel spectral transformations, ACM Transactions on Mathematical Software. 33 (2) (2007).

Antonio M. Vidal was born in Alicante, Spain, in 1949. He received the MS in physics from the Universidad de Valencia, Spain, in 1972, and the PhD in computer science from the Universidad Politécnica de Valencia, Spain, in 1990. Since 1992 he has been at the Universidad Politécnica de Valencia, Spain, where he is currently a full professor and director of the Parallel and Distributed Master Studies in the Department of Computer Science. His main areas of interest include parallel computing with applications in numerical linear algebra and signal processing.

Victor M. Garcia obtained a degree in mathematics and computer science (Universidad Complutense, Madrid) in 1991, later an MSc in industrial mathematics (University of Strathclyde, Glasgow) in 1992, and a Ph D in mathematics (Universidad Politécnica de Valencia) in 1998. He is a senior lecturer at the Universidad Politécnica de Valencia, and his areas of interest are numerical computing and parallel numerical methods and applications.

Pedro Alonso was born in Valencia, Spain, in 1968. He received the engineer degree in computer science from the Universidad Politecnica de Valencia, Spain, in 1994 and the PhD degree from the same University in 2003. His dissertation was on the design of parallel algorithms for structured matrices with application in several fields of digital signal analysis. Since 1996 he has been a senior lecturer in the Department of Computer Science of the Universidad Politecnica de Valencia. He is a member of the High Performance Networking and Computing Research Group of the Universidad Politecnica de Valencia. His main areas of interest include parallel computing for the solution of structured matrices with applications in digital signal processing.

Miguel O. Bernabeu received his engineer degree in computer science from the Universidad Politécnica de Valencia, Spain, in 2005. He was a research fellow with the Universidad Politécnica de Valencia from 2004 through 2007. He is currently a research assistant with the Computing Laboratory of the University of Oxford, UK. His research interests include parallel computing and numerical linear algebra and its applications to signal processing and cardiac modeling and simulation.

J Supercomput
DOI 10.1007/s11227-007-0157-x

A multilevel parallel algorithm to solve symmetric Toeplitz linear systems

Miguel O. Bernabeu · Pedro Alonso · Antonio M. Vidal

Abstract This paper presents a parallel algorithm to solve a structured linear system with a symmetric-Toeplitz matrix. Our main result concerns the use of a combination of shared and distributed memory programming tools to obtain a multilevel algorithm that exploits the actual different hierarchical levels of memory and computational units present in parallel architectures. This gives, as a result, a so-called parallel *hybrid* algorithm that is able to exploit each of these different configurations. Our approach has been done not only by means of combining standard implementation tools like OpenMP and MPI, but performing the appropriate mathematical derivation that allows this derivation. The experimental results over different representations of available parallel hardware configurations show the usefulness of our proposal.

Keywords Toeplitz matrix · Cauchy-like matrix · Rank displacement · Multilevel parallel programming · MPI · OpenMP

1 Introduction

In this paper, we present a parallel algorithm for the solution of the linear system

$$Tx = b, \tag{1}$$

where $T \in \mathbb{R}^{n\times n}$ is a symmetric Toeplitz matrix $T = (t_{ij})_{i,j=0}^{n-1} = (t_{|i-j|})_{i,j=0}^{n-1}$ and $b, x \in \mathbb{R}^n$ are the independent and the solution vectors, respectively.

M.O. Bernabeu (✉) · P. Alonso · A.M. Vidal
Departament de Sistemes Informàtics i Computació, Universitat Politècnica de València, 46022 Valencia, Spain
e-mail: mbernabeu@dsic.upv.es

P. Alonso
e-mail: palonso@dsic.upv.es

A.M. Vidal
e-mail: avidal@dsic.upv.es

The solution of a linear system of equations is extremely important in computer science, applied mathematics and engineering. New techniques are always needed to improve the efficiency of algorithms to solve this problem. When the linear system is *structured*, computations can be sped up by applying the *displacement structured* property of such matrices that describe the linear system. In particular, algorithms to solve Toeplitz linear systems can be classified into two categories, namely, the *Levinson*-type and the *Schur*-type. Levinson-type algorithms produce factorizations of the inverse of the Toeplitz while the Schur-type algorithms produce factorizations of the Toeplitz matrix itself.

The seminal work performed by Schur [1] to derive a fast recursive algorithm to check if a power series is analytic and bounded in the unit disk, interestingly, provided a fast factorization of matrices with *displacement rank* of 2. Toeplitz matrices have a displacement rank of 2 [2]. The first fast Toeplitz solver of this type was proposed by Bareiss [3]. Later on, closely related algorithms were presented [4, 5].

Levinson-type algorithms are based on the classical Levinson algorithm [6] and a very well-known set of variants due to Durbin [7], Trench [8], and Zohar [9, 10].

There are a number of differences between both kind of algorithms. Levinson-type algorithms require $O(n)$ size of memory due to the inverse of the Toeplitz matrix is not stored while Schur-type algorithms require $O(n^2)$. Schur-type algorithms are about 30% more expensive than Levinson-type algorithms [11]. Levinson-type algorithms require the computation of saxpy's and of inner products while Schur-type algorithms involve only saxpy's. On parallel machines, i.e., on a linear array of $O(n)$ processors, Schur-type methods require only $O(n)$ computing time, as compared to $O(n \log n)$ computing time for Levinson-type parallel methods. There exist some experiences with parallel algorithms over linear systolic arrays in [12–14]. Other theoretical algorithms (BSP model) which perform the parallelization of the Bareiss algorithm can be found in [15, 16]. Parallel examples of the Levinson-type algorithm can be found in [17, 18]; in this case with shared memory architecture.

Efficient parallel algorithms for the solution of this problem are not easy to develop due to the high dependency between operations and the low cost of the fast algorithms regarding the weight of communications. Some parallel algorithms are based on a *recursive factorization* of matrices using the divide and conquer technique used in theoretically efficient sequential algorithms for matrix inversion. Such techniques can be found originally in the LU factorization of symmetric positive definite matrices by recursive factorization of the Schur complement submatrices induced by the block Gaussian elimination [19]. Trench also used this technique for the inversion of Toeplitz matrices [8]. More recently, other parallel algorithms for structured matrices based on recursive partitioning have been presented in [20] with the particular case of Toeplitz-like matrices filled with integers, or for symmetric positive definite Toeplitz-like matrices [21].

Unfortunately, these last references are only theoretical proposals. No experimental results or any implement code is available to compare with new experimental results. We miss empirical information about the communication cost and the particular behavior of the algorithms over real parallel computer architectures. More practical works on parallel algorithms for structured matrices tested on a cluster of workstations have been developed. Parallel Schur-type algorithms can be found, i.e.,

in [22] where the least squares problem is solved with a refinement step to improve the accuracy of the solution. Also, the block–Toeplitz case is a practical study in [23, 24].

Schur-type as well as Levinson-type algorithms have some limitations to be the basis of efficient parallel algorithms [25]. In past years, new efficient parallel algorithms have risen based on the idea of the translation of the Toeplitz-like system to a Cauchy-like one. This idea was first proposed to introduce partial pivoting without the destruction of the structured property of Toeplitz matrices [26]. Furthermore, it was proposed in [12] to use this approach to solve Toeplitz linear systems in a linear systolic array. Recently, Thirumalai [27] proposed a parallel algorithm to solve symmetric Toeplitz linear systems but only for two processors. This last work was successful followed in depth to develop efficient parallel algorithms for different kinds of Toeplitz matrices [28–30] even with refinement techniques to improve the accuracy of the solution [31].

The present work to solve symmetric Toeplitz linear systems translates the Toeplitz matrix into a Cauchy-like one by means of trigonometric transformations. This step gives rise to an interesting sparsity that is exploited in order to split the problem into two independent problems. Our approach carries on with the parallelization of each one of these arisen subproblems giving way to a multilevel parallelization. We combine both shared memory (OpenMP) and distributed memory (MPI) programming tools to obtain what we call a *hybrid* parallel algorithm. Our parallel algorithm is efficient over different parallel configurations exploiting as far as possible the memory hierarchy as the multilevel configuration of the computational units.

We organize our paper as follows. In Sect. 2, we expose some mathematical preliminaries and our own derivations of all the triangularization algorithms. The sequential algorithm is described in Sect. 3 with a detailed analysis of each step that will be useful in the description of its parallelization. In Sect. 4, we present the description of each part of the hybrid parallel algorithm with experimental results that illustrates the improvements performed on each step and a comparison with the different approaches that can be used depending on the underlying hardware. A experimental section (Sect. 5) shows the global performance that can be expected with different hardware configurations. Finally, the paper ends with a section on our conclusions.

2 Mathematical background

2.1 Rank displacement and Cauchy-like matrices

It is said that a matrix of order n is *structured* if its *displacement representation* has a lower rank regarding n. The *displacement representation* of a symmetric Toeplitz matrix T (1) can be defined in several ways depending on the form of the displacement matrices. A useful form for our purposes is

$$\nabla_F T = FT - TF = \mathcal{G}\mathcal{H}\mathcal{G}^T; \tag{2}$$

where F is the $n \times n$ symmetric Toeplitz matrix

$$F = \begin{pmatrix} 0 & 1 & 0 & & & \\ 1 & 0 & 1 & 0 & & \\ 0 & 1 & \ddots & \ddots & & \\ & & \ddots & & & \\ & & & & & 1 \\ & & & & 1 & 0 \end{pmatrix},$$

called *displacement matrix*, $\mathcal{G} \in \mathbb{R}^{n \times 4}$ is the *generator* matrix and $\mathcal{H} \in \mathbb{R}^{4 \times 4}$ is a skew-symmetric *signature* matrix. The rank of $\nabla_F T$ is 4, that is, quite lower than n and independent of n.

It is easy to see that the displacement of T with respect to F is a matrix of a considerable sparsity from which it is not difficult to obtain an analytical form of $\mathcal{G}$ and $\mathcal{H}$.

A symmetric *Cauchy-like* matrix C is a structured matrix that can be defined as the unique solution of the displacement equation

$$\nabla_\Lambda C = \Lambda C - C\Lambda = \hat{\mathcal{G}}\mathcal{H}\hat{\mathcal{G}}^T, \tag{3}$$

being $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$, where $\text{rank}(\nabla_\Lambda C) \ll n$ and independent of n.

Now we use the normalized Discrete Sine Transformation (DST) $\mathcal{S}$ as defined in [32]. Since $\mathcal{S}$ is symmetric, orthogonal and $\mathcal{S}F\mathcal{S} = \Lambda$ [26, 33], we can obtain from (3),

$$\mathcal{S}(FT - TF)\mathcal{S} = \mathcal{S}(\mathcal{G}\mathcal{H}\mathcal{G}^T)\mathcal{S} \rightarrow \Lambda C - C\Lambda = \hat{\mathcal{G}}\mathcal{H}\hat{\mathcal{G}}^T,$$

where $C = \mathcal{S}T\mathcal{S}$ and $\hat{\mathcal{G}} = \mathcal{S}\mathcal{G}$. This shows how (2) can be transformed into (3).

In this paper, we solve the Cauchy-like linear system $C\hat{x} = \hat{b}$, where $\hat{x} = \mathcal{S}x$ and $\hat{b} = \mathcal{S}b$, by performing the triangular decomposition $C = LDL^T$, being L unit lower triangular and D diagonal. The solution of (1) is obtained by solving $Ly = \hat{b}$ and by computing $y \leftarrow D^{-1}y$, $L^T\hat{x} = y$ and $x = \mathcal{S}\hat{x}$.

Solving a symmetric Toeplitz linear system by transforming it into a symmetric Cauchy-like system has an interesting advantage due to the symmetric Cauchy-like matrix which has an important sparsity that can be exploited. Matrix C has the following form (x only denotes nonzero entries),

$$C = \begin{pmatrix} x & 0 & x & 0 & \ldots \\ 0 & x & 0 & x & \ldots \\ x & 0 & x & 0 & \ldots \\ 0 & x & 0 & x & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}.$$

We define the odd-even permutation matrix P_{oe} as the matrix that after applied to a vector, groups the odd entries in the first positions and the even entries in the last

ones,

$$P_{oe}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ \vdots \end{pmatrix} = \begin{pmatrix} x_1 \\ x_3 \\ x_5 \\ \cdots \\ x_2 \\ x_4 \\ x_6 \\ \vdots \end{pmatrix}.$$

Applying transformation $P_{oe}(.)P_{oe}^T$ to a symmetric Cauchy-like matrix, C gives

$$P_{oe}CP_{oe}^T = \begin{pmatrix} C_0 & \\ & C_1 \end{pmatrix}, \tag{4}$$

where C_0 and C_1 are symmetric Cauchy-like matrices of order $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively. In addition, it can be shown that matrices C_0 and C_1 have a displacement rank of 2, as opposed to C that has a displacement rank of 4 [34].

The two submatrices arising in (4) have the displacement representation

$$\Lambda_j C_j - C_j \Lambda_j = G_j H_j G_j^T, \quad j = 0, 1, \tag{5}$$

where $\begin{pmatrix} \Lambda_0 & \\ & \Lambda_1 \end{pmatrix} = P_{oe}\Lambda P_{oe}^T$ and $H_0 = H_1 = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$. As it is shown in [35], given vector $u = (0 \quad t_2 \quad t_3 \quad \cdots \quad t_{n-2} \quad t_{n-1})^T$ and the first column of the identity matrix e_0, the generators of (5) can be computed as

$$G = (G_{:,0}, G_{:,1}) = \begin{pmatrix} G_0 \\ G_1 \end{pmatrix} = \sqrt{2} P_{oe}\mathcal{S}\,(u \quad e_0)\,. \tag{6}$$

The odd-even permutation matrix is used to decouple the symmetric Cauchy-like matrix arising from a real symmetric Toeplitz matrix into the following two Cauchy-like systems of linear equations

$$C_j \hat{\hat{x}}_j = \hat{\hat{b}}_j, \quad j = 0, 1, \tag{7}$$

where $\hat{\hat{x}} = (\hat{\hat{x}}_0^T \quad \hat{\hat{x}}_1^T)^T = P_{oe}\mathcal{S}x$ and $\hat{\hat{b}} = (\hat{\hat{b}}_0^T \quad \hat{\hat{b}}_1^T)^T = P_{oe}\mathcal{S}b$.

Each one of both linear systems is half the size and half the displacement rank than the original linear system, so this yields a substantial saving over the nonsymmetric forms of the displacement equation. Furthermore, it can be exploited in parallel by solving each of the two independent subsystems into two different processors.

2.2 Triangular decomposition of symmetric Cauchy-like matrices

For the discussion of this section, we will start with the displacement representation of a general symmetric Cauchy-like matrix $C \in \mathbb{R}^{n \times n}$,

$$\Lambda C - C\Lambda = GHG^T, \tag{8}$$

where Λ is diagonal, $G \in \mathbb{R}^{n\times r}$ is the generator, and $H \in \mathbb{R}^{r\times r}$ is the signature matrix (in our case $r = 2$).

Generally, the displacement representation (8) arises from other displacement representations, e.g., the displacement representation of a symmetric Toeplitz matrix or another symmetric structured matrix. Matrix C is not formed explicitly in order to avoid any computational cost. Matrix C is implicitly known by means of the generator pair. A triangular decomposition algorithm of C works on the generator pair and can be derived in an easy way as follows.

From (8), it is clear that any column $C_{:,j}$, $j = 0, \ldots, n-1$, of C can be equated from the Sylvester equation

$$\Lambda C_{:,j} - C_{:,j}\lambda_{j,j} = GH(G_{j,:})^T,$$

so the (i, j)th element of C can be computed as

$$C_{i,j} = \frac{G_{i,:}HG_{j,:}^T}{\lambda_{i,i} - \lambda_{j,j}}, \quad \text{iff } \lambda_{i,i} \neq \lambda_{j,j}, \tag{9}$$

that is, for all elements off the main diagonal. If C is a symmetric Cauchy-like matrix, only the elements in the main diagonal cannot be computed by means of (9). The main diagonal elements of C must be computed prior to the start of the factorization. This computation can be performed in $O(n \log n)$ operations using the DST. The following algorithm makes this computation [36].

Algorithm 1 Computation of the diagonal of a symmetric Cauchy-like matrix Let $(t_0, t_1, \ldots, t_{n-1})^T$ be the first column of a symmetric Toeplitz matrix T and $\mathcal{S}$ the DST transformation matrix; this algorithm returns the diagonal of the Cauchy-like matrix $C = \mathcal{S}T\mathcal{S}$ in array c.

$r_0 = nt_0$
$r_1 = 2(n-1)t_1$
$w_0 = -t_0$
$v_0 = -2t_1$
for $k = 2, \ldots, \lfloor n/2 \rfloor$
  $r_{2k-2} = 2(n-2k+2)t_{2k-2} + 2w_{k-2}$
  $w_{k-1} = w_{k-2} - 2t_{2k-2}$
  $r_{2k-1} = 2(n-2k+1)t_{2k-1} + 2v_{k-2}$
  $v_{k-1} = v_{k-2} - 2t_{2k-1}$
end for
if n is odd
  $r_{n-1} = 2t_{n-1} + 2w_{((n-1)/2)-1}$
end if
$c = \mathcal{S}r$

We assume in the following that all elements of C are known or they can be computed with (9). Let the following partition of C and Λ be

$$C = \begin{pmatrix} d & c^T \\ c & \hat{C} \end{pmatrix}, \qquad \Lambda = \begin{pmatrix} \lambda & 0 \\ 0 & \hat{\Lambda} \end{pmatrix},$$

where $d, \lambda \in \mathbb{R}$, $c \in \mathbb{R}^n$ and $\hat{\Lambda}, \hat{C} \in \mathbb{R}^{(n-1)\times(n-1)}$, then we define the following matrix X,

$$X = \begin{pmatrix} 1 & 0 \\ l & I \end{pmatrix}, \quad X^{-1} = \begin{pmatrix} 1 & 0 \\ -l & I \end{pmatrix},$$

where $l = c/d$. Let us premultiply by X^{-1} and post-multiply by X^{-T} the displacement equation (8),

$$\begin{aligned} & X^{-1}(\Lambda C - C\Lambda)X^{-T} \\ & = (X^{-1}\Lambda X)(X^{-1}CX^{-T}) - (X^{-1}CX^{-T})(X^T\Lambda X^{-T}) \\ & = \begin{pmatrix} \lambda & 0 \\ \hat{\Lambda}l - \lambda l & \hat{\Lambda} \end{pmatrix}\begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix} - \begin{pmatrix} d & 0 \\ 0 & C_{sc} \end{pmatrix}\begin{pmatrix} \lambda & l^T\hat{\Lambda} - \lambda l^T \\ 0 & \hat{\Lambda} \end{pmatrix} \\ & = (X^{-1}G)H(X^{-1}G)^T, \end{aligned}$$

where $C_{sc} = \hat{C} - (cc^T)/d$ is the Schur complement of C with respect to d. At this step, we know the first column of L, $(1\ l)^T$, and the first diagonal entry of D, d, of the LDL^T decomposition of $C = LDL^T$ for a lower unit triangular factor L and a diagonal factor D.

Equating the $(2, 2)$ position in the above equation, we have the displacement representation of the Schur complement of C with respect to its first element,

$$\hat{\Lambda}C_{sc} - C_{sc}\hat{\Lambda} = G_1 H G_1^T,$$

where G_1 is the portion of $X^{-1}G$ from the second row down. The process can now be repeated on the displacement equation of the Schur complement C_{sc} to get the second column of L and second element of D of the LDL^T factorization of C. Repeating this process, we can obtain after n steps, the LDL^T factorization of C.

The steps described above are summarized in the following algorithm for the case in which the Cauchy-like matrix is symmetric and the rank of its displacement is 2.

Algorithm 2 LDL^T **decomposition of symmetric Cauchy-like matrices** Let $G \in \mathbb{R}^{n\times 2}$ be the generator, $H \in \mathbb{R}^{2\times 2}$ be the signature matrix, λ be an array with the diagonal entries of $\Lambda \in \mathbb{R}^{n\times n}$ of the displacement of a symmetric Cauchy-like matrix C of the form (8), and the diagonal entries of C ($c_{k,k}$, for $k = 0, \ldots, n-1$) obtained from Algorithm 1. This algorithm returns a unit lower triangular matrix L and the diagonal entries of a diagonal factor D, stored in the diagonal of L, of the LDL^T decomposition of C.

for $k = 0, \ldots, n-1$

1. for $i = k+1, \ldots, n-1$

$c_{i,k} = (G_{i,:} H G_{k,:}^T)/(\lambda_i - \lambda_k)$.
end for
2. $d = c_{k,k}$.
3. for $i = k+1, \ldots, n-1$
$l_{i,k} = c_{i,k}/d$.
end for
4. $l_{k,k} = d$.
5. for $i = k+1, \ldots, n-1$
$g_{i,0} \leftarrow g_{i,0} - g_{k,0}\, l_{i,k}$.
$g_{i,1} \leftarrow g_{i,1} - g_{k,1}\, l_{i,k}$.
end for
6. for $i = k+1, \ldots, n-1$
$c_{i,i} \leftarrow c_{i,i} - d\, l_{i,k}^2$.
end for
end for

The above algorithm can be modified in order to not perform all the iterations for k. This modification can be used to develop a block version of the algorithm.

The two columns of the generator are updated from the $(i+1)$th row down in step 5 of Algorithm 2. The ith row of the generator does not need to be updated. Updating such a row gives a zero row, so it is not referenced in the next iterations.

Diagonal entries of C must be updated before the next iteration, as well as the generator. This is performed in step 6 of Algorithm 2.

Algorithm 2 also returns the ith row of the generator in the i iteration. Thus, on the return of the algorithm, we will have an $n \times 2$ matrix in which the ith row is the first row of the generator of the displacement representation of the ith Schur complement of C with respect to its principal leading submatrix of order $(i-1)$. This is useful in the blocking algorithm used in the parallel algorithm.

3 Sequential algorithm

For the exposition of Algorithm 3, we first define

$$\hat{b} = \begin{pmatrix} \hat{b}_0 \\ \hat{b}_1 \end{pmatrix} = P_{oe} \mathcal{S} b, \tag{10}$$

where P_{oe} (2.1) is the odd-even permutation, $\mathcal{S}$ is the DST used in (4) to translate the Toeplitz displacement representation to a Cauchy-like one and b is the independent vector.

Algorithm 3 (Algorithm for the solution of a symmetric Toeplitz system with Cauchy-like transformation) Given $T \in \mathbb{R}^{n \times n}$ a symmetric Toeplitz matrix and $b \in \mathbb{R}^n$ an independent term vector, this algorithm returns the solution vector $x \in \mathbb{R}^n$ of the linear system $Tx = b$.

1. "Previous computations."
2. Obtain $C_0 = L_0 D_0 L_0^T$ and $C_1 = L_1 D_1 L_1^T$ (4).
3. Solve $L_0 D_0 L_0^T \hat{x}_0 = \hat{b}_0$ and $L_1 D_1 L_1^T \hat{x}_1 = \hat{b}_1$.
4. Compute $x = \mathcal{S} P_{oe}^T \begin{pmatrix} \hat{x}_0 \\ \hat{x}_1 \end{pmatrix}$.

The first step is up to the translation of the Toeplitz linear system into the Cauchy-like linear system. The operations performed in this step involve the computation of G_0 and G_1 (6), the computation of Λ_0 and Λ_1, where

$$\begin{pmatrix} \Lambda_0 & \\ & \Lambda_1 \end{pmatrix} = P_{oe} \Lambda P_{oe}, \tag{11}$$

the computation of the diagonal of C, and the computation of $\hat{b}$ (10).

In the second step, the triangularization of the two independent Cauchy-like linear systems is carried out by means of Algorithm 2. The main workload of the algorithm falls into this step. The third step consists of the solution of several triangular linear systems whereas in the last step, the solution of the Toeplitz linear system is obtained.

Although the previous algorithm lists four main steps, we have split them into ten different stages that will give a more clear idea of which operations are performed. Let us enumerate them.

1. $G_{:,0} = \sqrt{2} P_{oe} \mathcal{S} u$ (6)
2. $G_{:,1} = \sqrt{2} P_{oe} \mathcal{S} e_0$ (6)
3. $\hat{b} = P_{oe} \mathcal{S} b$
4. Compute $(\Lambda_0 \oplus \Lambda_1)$ (11)
5. Compute diagonal entries of C
6. Compute $L_0 D_0 L_0^T$, the LDL^T decomposition of C_0
7. Solve $L_0 D_0 L_0^T \hat{x}_0 = \hat{b}_0$
8. Compute $L_1 D_1 L_1^T$, the LDL^T decomposition of C_1
9. Solve $L_1 D_1 L_1^T \hat{x}_1 = \hat{b}_1$
10. $x = \mathcal{S} P_{eo} \hat{x}$.

The first 5 stages correspond to step 1 of Algorithm 3. The second step includes stages 6 and 8, which are computed by means of Algorithm 2. Stages 7 and 9 correspond to step 3. Finally, stage 10 is equivalent to step 4.

This list will help us to present the parallel implementations of the algorithm in next section.

4 Parallel algorithm

First of all, we have studied which parts of the sequential algorithm are more time-expensive in order to focus our parallelization efforts on the most expensive steps. Table 1 shows the results of this study. These results and all the following have been ob-

Table 1 Per-part analysis of the sequential algorithm ($n = 29{,}999$)

	Time (s)	% of total
Previous computations	0.01	0.12
LDL^T decomposition	6.12	71.83
$LDL^T x = b$ solution	2.39	28.05
Other computations	< 0.01	< 0.01
Total	8.52	

tained in a cluster of 4 tetra-processor nodes. Each node consists of 4 Alpha EV68CB running at 1000 MHz with 4 GBytes of RAM.

The next three subsections pay attention to each one of the three parts of the algorithm ordering by its weight in the total time. The following is a subsection that explains the different alternatives used to implement the parallel algorithm.

4.1 LDL^T decomposition

As Table 1 shows, the LDL^T decomposition is the most expensive step of Algorithm 3 ($> 70\%$ of the total time).

As we mentioned in previous sections, we solve the $Tx = b$ system transforming it into *two* Cauchy-like systems $C_0\hat{x}_0 = \hat{b}_0$ and $C_1\hat{x}_1 = \hat{b}_1$. Therefore, the LDL^T decomposition step actually includes *two* decompositions: $C_0 = L_0 D_0 L_0^T$ and $C_1 = L_1 D_1 L_1^T$ (5). Since they are independent, we can compute them concurrently. We call this splitting of the LDL^T decomposition the *first level* of parallelism. It has been implemented with both MPI and OpenMP standards, so it is possible to use it on the most suitable architecture.

Obviously, this solution cannot take advantage of more than two processors. To avoid this limitation, we have implemented a *second level* of parallelism. It consists of the parallelization of the subroutine that performs each one of the two LDL^T decompositions. A previous analysis of Algorithm 2 shows that there exists a data dependency among loops 1, 3, 5, and 6. However, it is possible to get them all together in only one inner loop with the appropriate order in the operations flow. Therefore, we have rewritten Algorithm 2 into Algorithm 4.

Algorithm 4 (Reordered version of Algorithm 2) Let $G \in \mathbb{R}^{n\times 2}$ be the generator, $H \in \mathbb{R}^{2\times 2}$ be the signature matrix, λ be an array with the diagonal entries of $\Lambda \in \mathbb{R}^{n\times n}$ of the displacement of a symmetric Cauchy-like matrix C of the form (8) and the diagonal entries of C ($c_{k,k}$, for $k = 0, \ldots, n-1$); this algorithm returns a unit lower triangular matrix L and the diagonal entries of a diagonal factor D, stored in the diagonal of L, of the LDL^T decomposition of C. (It has been respected the numeration of the steps as they appear in Algorithm 2 in order to show the rearrangement process.)

for $k = 0, \ldots, n-1$
  2. $d = c_{k,k}$.
  4. $l_{k,k} = d$.
     for $i = k+1, \ldots, n-1$

Table 2 Sequential vs. parallel implementations of LDL^T decomposition ($n = 29{,}999$)

	Time (s)	Speed-up
Sequential	6.12	
OpenMP ($p = 2$)	3.21	1.90
MPI ($p = 2$)	3.27	1.87
MPI+OpenMP ($p = 4$)	1.73	3.54
MPI+OpenMP ($p = 8$)	0.98	6.25

1. $c_{i,k} = (G_{i,:} H G^T_{k,:})/(\lambda_i - \lambda_k)$.
3. $l_{i,k} = c_{i,k}/d$.
6. $c_{i,i} \leftarrow c_{i,i} - d\, l^2_{i,k}$.
5. $g_{i,0} \leftarrow g_{i,0} - g_{k,0}\, l_{i,k}$.
   $g_{i,1} \leftarrow g_{i,1} - g_{k,1}\, l_{i,k}$.
   end for
end for

In this reordered version, there is no data dependency among iterations of the inner loop (i-loop). Therefore, this loop can be parallelized with OpenMP directives.

In order to implement both levels of parallelism, we have used two MPI processes for the first level of parallelism and OpenMP threads for the second one. Nowadays, no compiler supports OpenMPs *nested parallelism*. However, in [37], a technique is presented that can be used to avoid this limitation. The technique is based on the creation of a sufficient number of threads at the first level of parallelism as it will be needed in all steps of the algorithm. Using an optimal workload distribution algorithm that assigns work to each active thread, an optimal workload balance can be achieved. Furthermore, we propose a hybrid solution in order to achieve the suitability of running the algorithm on a wider range of different machines, exploiting the particular advantages of each different hardware configuration.

Three different implementations have been written: one MPI implementation of the first level of parallelism, one OpenMP implementation of the first level as well and a more complex MPI+OpenMP implementation of both levels of parallelism.

Table 2 shows the execution time of the LDL^T decomposition step obtained with each one of these different implementations. Also, the ratio between the sequential and each of the parallel versions (speed-up) is provided to show the growth in speed achieved. As it can be seen, the most expensive step of the algorithm is highly reduced giving up a great impact in the overall performance since this step takes more than 70% of the sequential time.

4.2 $LDL^T x = b$ solution

The third step of Algorithm 3 is the solution of the systems $L_0 D_0 L_0^T\ \hat{x}_0 = \hat{b}_0$ and $L_1 D_1 L_1^T\ \hat{x}_1 = \hat{b}_1$. Here we have followed a similar approach for the LDL^T decomposition. We introduce a *first level* of parallelism simply by concurrently solving each system. A *second level* of parallelism is also introduced by solving each system with a multithreaded $dtrsv$ subroutine of BLAS.

Table 3 Sequential vs. parallel implementations of the $LDL^T x = b$ solution step ($n = 29{,}999$)

	Time (s)	Speed-up
Sequential	2.39	
OpenMP ($p = 2$)	1.45	1.65
MPI ($p = 2$)	1.19	2.01
MPI+OpenMP ($p = 4$)	0.83	2.88
MPI+OpenMP ($p = 8$)	0.69	3.46

Table 4 Previous computations. Execution time of the 5 tasks

	Task 1	Task 2	Task 3	Task 4	Task 5
29,999	2.92×10^{-3}	9.75×10^{-4}	1.95×10^{-3}	9.75×10^{-4}	4.88×10^{-3}
30,000	0.176	9.76×10^{-4}	0.175	9.76×10^{-4}	0.177

Again, two different versions implement the first level of parallelism: an MPI version and an OpenMP version, while an MPI+OpenMP version implements both levels of parallelism. Table 3 shows the execution time and the speed-up of these implementations.

Using our own multithreaded version of the BLAS $dtrsv$ subroutine, we have obtained a good speed-up up to 4 processors. For more than two processors concurrently solving a triangular linear system of equations, a further development must be made. However, if there is a native multithreaded version available, even better results for bi- and tetra-processor boards can be expected than that shown in Table 3.

4.3 Previous computations

We call the "previous computations" part of the algorithm where the Toeplitz matrix is transformed into a Cauchy-like matrix by means of discrete trigonometric transformations in order to perform its LDL^T decomposition. This previous step consists of the first five steps enumerated in Sect. 3.

The "previous computations" step involves a total of three Discrete Sine Transformations (DSTs), each one in tasks 1, 3 and 5. We have used routine `dsint` of the `fftpack` package to apply a DST. This function uses fast algorithms that use different radix of low primes in order to obtain an asymptotic cost of $O(n \log n)$ operations. However, the algorithm highly depends on the size of the primes in which the value $n + 1$ is decomposed. If some of the mentioned primes are large, the algorithm can be significantly expensive. Table 4 shows the execution time of each of the five tasks for two different problem sizes.

Only a unit of difference between the two problem sizes in Table 4 makes it more than 50 times slower the execution of the previous computations step of one problem with respect to the other. As it is expected, this fact affects the weight of each step in the algorithm. Table 5 is an analog to Table 1 for the case $n = 30{,}000$ in which it can be seen how the weight of the first and the last computations increase while the others decrease. Time in Table 5 is more than 11% larger than the time obtained in Table 1.

Table 5 Per-part analysis of the sequential algorithm ($n = 30{,}000$)

	Time (s)	% of total
Previous computations	0.54	5.69
LDL^T decomposition	6.27	66.07
$LDL^T x = b$ solution	2.4	25.29
Other computations	0.28	2.95
Total	9.49	

Table 6 Execution times of the "previous computations" step in one processor for different problem sizes

Problem size	Prime decomposition of $n+1$	Time (s) with `dsint`	Time (s) with *Chirp-z*
10,000	$1 \times 73 \times 137$	1.17×10^{-2}	0.23
13,000	$1 \times 13{,}001$	3.22	0.23
16,000	$1 \times 16{,}001$	4.87	0.24
19,000	$1 \times 19{,}001$	6.88	0.55
22,000	$1 \times 7 \times 7 \times 449$	5.17×10^{-2}	0.55
25,000	$1 \times 23 \times 1{,}087$	0.18	0.55
28,000	$1 \times 28{,}001$	15.00	0.55

Two main proposals are used to solve this problem. The first one deals with the use of the *Chirp-z* factorization [32]. The other one deals with parallelism.

The use of the *Chirp-z* factorization is a technique proposed to solve the same problem when the DFT is used. We have adapted the factorization to the case of the DST. Basically, the *Chirp-z* factorization consists of transforming the problem of applying a DST of size n into a Toeplitz-matrix by a vector product of size m ($m > n$). As it is well known, the Toeplitz-matrix by a vector product can be carried out in a very fast way ($O(m \log m)$) by using the DFT, so we turn the DST problem of size n into a DFT problem of size m. The freedom to choose the size of the Toeplitz matrix involved in the *Chirp-z* factorization allows us to select $m = 2^t$, being t, the minimum value that stands for $m > n$. Although m can be quite large, the DFT algorithm runs so fast that it does not constitute a problem. The final result is an algorithm with a computational cost independent of the size of the largest prime in which $n + 1$ is decomposed and faster than routine `dsint` in many cases (Table 6).

The new factorization to apply the DST to an array involves a new problem to solve, which is the choice between both of the two routines to be used for a given problem size. Many applications involving a symmetric Toeplitz linear system are characterized because there exists a certain degree of freedom of choosing the size of the problem, i.e., in digital signal processing where the problem size is closely related to the length of a sampled signal. In addition, the problem size never changes while the application involving the symmetric Toeplitz linear system is running, so once the size of the problem to solve is known, the suitable transformation algorithm to be used can be chosen.

Table 7 Sequential vs. parallel implementations of the "previous computations" step

	$n = 29{,}999$		$n = 30{,}000$	
	Time (s)	Speed-up	Time (s)	Speed-up
Sequential	1.07×10^{-2}		0.54	
OpenMP ($p = 2$)	1.07×10^{-2}	1.00	0.54	1.00
MPI ($p = 2$)	1.07×10^{-2}	1.00	0.54	1.00
MPI+OpenMP ($p = 4$)	8.79×10^{-3}	1.21	0.37	1.45
MPI+OpenMP ($p = 8$)	6.83×10^{-3}	1.53	0.21	2.57

We have developed a simple but effective tuning program that allows us to make the suitable choice. The tuning program executes both routines to different problem sizes n, such that $n + 1$ are prime numbers. There always exists a threshold prime number from which the *Chirp-z* runs faster than routine `dsint`. This threshold prime number is platform dependent, so the tuning program will be executed only once to figure it out. With a given problem size and a known threshold prime, the suitable DST routine can be chosen to solve the problem as efficient as possible.

Furthermore, we have improved our algorithm with a routine that automatically chooses the DST routine to use. This routine receives the threshold prime number and makes a prime decomposition of $n + 1$. The routine divides $n + 1$ by prime numbers obtained from a sufficiently large ordered table of primes and whether the reminder is smaller than the threshold prime or whether the maximum prime number of the prime decomposition of $n + 1$ is larger than the threshold prime the routine stops. Thus, the algorithm can choose in the few milliseconds that spends this process the best choice in runtime. Results shown in Sect. 5 are all obtained with this improvement.

The other proposal to reduce the impact of the "previous computations" step in the overall cost of the algorithm consists of executing the 5 tasks concurrently by means of the OpenMP standard. In order to avoid the communications cost, we only allow the concurrency inside each multiprocessor board being replicated from these previous computations on the two different MPI processes as it will be further explained in the next subsection. Obviously, this second improvement is complementary to the previous one.

Table 7 shows the execution time and speed up of the three implementations for the "previous computations" step for the two problem size cases.

4.4 Parallel implementations

Once the parallelization of the three first steps of Algorithm 3 have been analyzed, we present the three parallel implementations performed (OpenMP version, MPI version and MPI+OpenMP version) from a global point of view.

The first one, the OpenMP version, implements the first level of parallelism of the three main steps of Algorithm 3 with a share-memory approach. The algorithm is suitable for running in biprocessor boards. Figure 1 shows the algorithm graphically. Numbers correspond to the stages enumerated in Sect. 3. Boxes in the same row are executed concurrently and bold lines represent synchronization points.

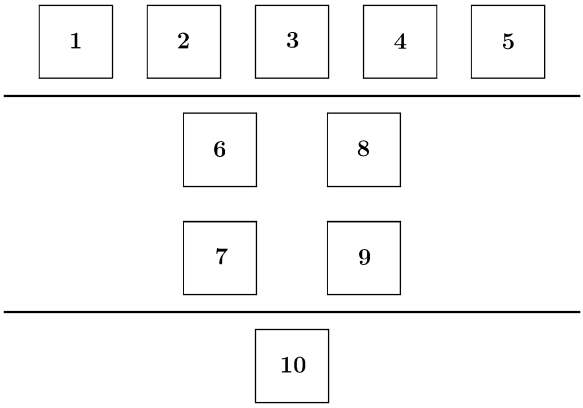

Fig. 1 OpenMP version flow chart

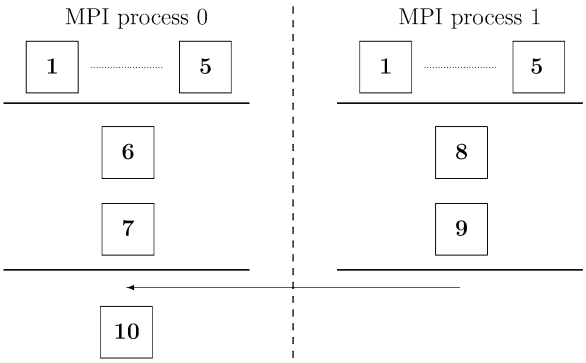

Fig. 2 MPI version flow chart

The second one, the MPI version, implements the first level of parallelism. Also, the distributed-memory approach is used in the "previous computations" step. The algorithm is suitable for two interconnected monoprocessor boards or biprocessor boards without the OpenMP capability. The concurrent computation of the "previous computations" step can be easily deactivated in the second case if necessary. Figure 2 shows the algorithm graphically. Arrows represent communication messages between processes.

Finally, the MPI+OpenMP version implements both levels of parallelism with a multilevel programming approach. The first level is implemented via MPI processes and the second level with OpenMP threads. This version represents the most versatile version of the algorithm and is suitable for more complex configurations like those we used in our experiments. Figure 3 shows the algorithm graphically. Boxes numbered from 1 to t represent the t threads executing the second level of parallelism in the LDL^T decomposition and the $LDL^T x = b$ solution stages, respectively.

Algorithm 5 shows the most general case (the *hybrid* algorithm) represented in Fig. 3. Algorithm 5 is the parallel version of Algorithm 3 using MPI processes and concurrent threads.

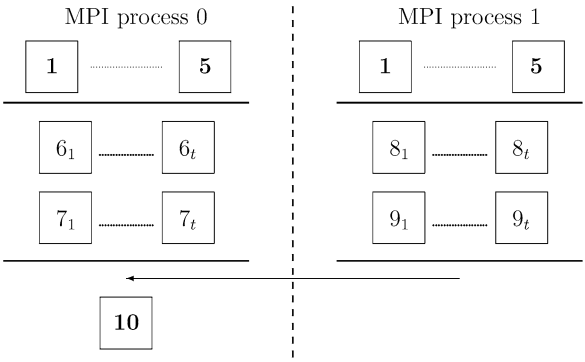

Fig. 3 MPI+OpenMP version flow chart

Algorithm 5 (Hybrid algorithm for the solution of a symmetric Toeplitz system with Cauchy-like transformation) Given $T \in \mathbb{R}^{n \times n}$, a symmetric Toeplitz matrix and $b \in \mathbb{R}^n$, an independent term vector, this algorithm returns the solution vector $x \in \mathbb{R}^n$ of the linear system $Tx = b$.

Launch two MPI processes P_0 and P_1. Each MPI process $p = P_i$, $i = 0, 1$, do

1. Compute Tasks 1–5 by means of 5 OpenMP threads (parallel sections).
2. Obtain $C_i = L_i D_i L_i^T$ by means of a modified version of Algorithm 4, in which the inner loop (i) becomes an OpenMP **parallel for**.
3. Solve $L_i D_i L_i^T \hat{x}_i = \hat{b}_i$ by using our threaded routine `dtrsv` (Sect. 4.2).
4. If $p = P_1$, then send x_1 to P_0;

   else P_0 receives x_1 and computes $x = \mathcal{S} P_{oe}^T \begin{pmatrix} \hat{x}_0 \\ \hat{x}_1 \end{pmatrix}$.

5 Experimental results

The target cluster used sets up a good scenario for our experiments since it gives us the chance to use the distributed memory paradigm, the shared memory paradigm and the multi-level programming paradigm.

As we explained in the Introduction, this algorithm represents an important computational kernel in many applications, i.e., multichannel sound reproduction systems. These kinds of applications dealing with digital signal processing are expected to be run on low to middle cost hardware as fast as possible, paying more attention to the execution time (i.e., to process sampled signals) than to the efficiency. The cluster can be used to present results simulating different target systems:

- The first one is a two monoprocessors cluster. Here only the MPI version (Fig. 2) can be used (Table 8).
- The second scenario is a two-processor board. The OpenMP two-processor version is used. The sequential version shown in the first line of Table 9 has been obtained with only one processor, whereas the second line corresponds to the implementation of the OpenMP version (Fig. 1) that uses the two processors.

Table 8 Execution time on two Intel Pentium 4 boards at 1.7 GHz with 1 GB memory

Version	10,000	13,000	16,000	19,000	22,000
Sequential	2.40	4.35	6.49	9.44	19.0
MPI	1.31	2.46	3.63	5.35	6.46

Table 9 Execution time on Intel XEON boards at 2.2 GHz with 3.5 GB memory

Version	10,000	13,000	16,000	19,000	22,000	25,000
Sequential	2.12	3.85	5.66	8.22	10.2	13.3
OpenMP	1.21	2.27	3.33	4.90	5.92	7.68
MPI + OpenMP	0.69	1.37	1.97	2.89	3.12	4.08

Table 10 Execution time on the Alpha's boards (MPI+OpenMP version)

Version	10,000	13,000	16,000	19,000	22,000	25,000	28,000
Sequential	0.98	1.99	2.88	4.54	5.05	6.35	8.83
1 tetra-processor	0.36	0.86	1.15	1.84	2.38	2.94	3.34
2 tetra-processor	0.25	0.60	0.78	1.56	1.61	2.10	1.94

- The third scenario is a two two-processor cluster. Now the MPI+OpenMP version (Fig. 3) can take advantage of such hardware. The time is shown in the third line of Table 9.
- The fourth scenario is one tetra-processor board. The MPI+OpenMP version can also be used. Table 10 shows the sequential time by using only one processor of the board, whereas the second line shows the time using the four processors.
- The last scenario is a two tetra-processor cluster, the most suitable version is MPI+OpenMP again (third line of Table 10).

Table 10 shows a significant reduction in time by using our different approaches to solve the problem in parallel. The algorithm spends almost 9 seconds to solve a problem of size $n = 28{,}000$ in one processor while using 8 processors the time is reduced to less than two seconds.

6 Conclusions

Based on our mathematical approach that translates a symmetric Toeplitz linear system to a another structured linear system called Cauchy-like, we have derived a parallel algorithm that solves this problem in parallel efficiently. This is possible since the solution of a symmetric Cauchy-like linear system can be split into two independent linear systems to be solved by means of two MPI or OpenMP processes. Furthermore, we go beyond this partition to solve in parallel each of the two arisen subproblems by means of OpenMP threads. The efficiency achieved in this second level of parallelism is possible thanks to the diagonality of the displacement matrices involved in the displacement representation of Cauchy-like matrices.

As it was shown, translating a symmetric Toeplitz matrix to a Cauchy-like one is not a trivial step. The problem of using discrete transformations based on the FFT has been solved by using the so-called *Chirp-z* factorization. In addition, we provide the programmer with a tuning program to chose which type of DST routine must be used plus a fast routine that makes this choice in runtime with a millisecond cost.

The experimental results show the utility of our parallel hybrid algorithm and its versatility to be used on different hardware/software configurations. Furthermore, the algorithm fits not only the actual hardware, but the upcoming hybrid parallel architectures incorporating multicore processors on cluster boards.

Acknowledgements This work has been partially supported by the Ministerio de Educación y Ciencia of the Spanish Government, and FEDER funds of the European Commission under Grant TIC 2003-08238-C02-02 and by the Programa de Incentivo a la Investigación de la Universidad Politécnica de Valencia 2005 under the Project 005522. We also want to acknowledge to Universidad Politécnica de Cartagena and Universidad de Murcia for allowing us to use the hardware platforms to carry out our experiments.

References

1. Schur I. (1917 (1986)) On power series which are bounded in the interior of the unit circle I, II. In: Gohberg I. (ed) I. Schur methods in operator theory and signal processing. Operator theory: advances and applications, vol 18. Birkhäuser, Basel, pp 31–59
2. Kailath T, Kung SY, Morf M (1979) Displacement ranks of matrices and linear equations. J Math Anal Appl 68:395–407
3. Bareiss EH (1969) Numerical solution of linear equations with Toeplitz and vector Toeplitz matrices. Numer Math 13:404–424
4. Rissanen J (1973) Algorithms for triangular decomposition of block Hankel and Toeplitz matrices with application to factoring positive matrix polynomials. Math Comput 27(121):147–154
5. Morf M (1974) Fast algorithms for multivariable systems. PhD thesis, Stanford University
6. Levinson N (1946) The Wiener RMS (root mean square) error criterion in filter design and prediction. J Math Phys 25:261–278
7. Durbin J (1960) The fitting of time series models. Rev Int Stat Inst 28:233–243
8. Trench WF (1964) An algorithm for the inversion of finite Toeplitz matrices. J Soc Ind App Math 12(3):515–522
9. Zohar S (1969) Toeplitz matrix inversion: The algorithm of W.F. Trench. J ACM 16(4):592–601
10. Zohar S (1974) The solution of a Toeplitz set of linear equations. J ACM 21(2):272–276
11. Kailath T, Sayed AH (eds) (1999) Fast reliable algorithms for matrices with structure. SIAM, Philadelphia
12. Brent R, Luk F (1983) A systolic array for the linear time solution of Toeplitz systems of equations. J VLSI Comput Syst 1:1–22
13. Kung SY, Hu YH (1983) A highly concurrent algorithm and pipelined architecture for solving Toeplitz systems. IEEE Trans Acoust Speech Signal Process ASSP-31(1):66
14. Ipsen I (1987) Systolic algorithms for the parallel solution of dense symmetric positive-definite Toeplitz systems. Technical Report YALEU/DCS/RR-539, Department of Computer Science, Yale University, New Haven, CT, May 1987
15. Brent RP (1990) Parallel algorithms for Toeplitz matrices. In: Golub GH, Van Dooren P (eds) Numerical linear algebra, digital signal processing and parallel algorithms. Computer and systems sciences, number 70. Springer, pp 75–92
16. Huang Y, McColl WF (1999) A BSP Bareiss algorithm for Toeplitz systems. J Parallel Distributed Comput 56(2):99–121
17. de Doncker E, Kapenga J (1990) Parallelization of Toeplitz solvers. In: Golub GH, Van Doore P (eds) Numerical linear algebra, digital signal processing and parallel algorithms. Computer and systems sciences, number 70. Springer, pp 467–476

18. Gohberg I, Koltracht I, Averbuch A, Shoham B (1991) Timing analysis of a parallel algorithm for Toeplitz matrices on a MIMD parallel machine. Parallel Comput 17(4–5):563–577
19. Aho AV, Hopcroft JE, Ullman JD (1974) The design and analysis of computer algorithms. Addison-Wesley, Reading
20. Pan V (2000) Parallel complexity of computations with general and Toeplitz-like matrices filled with integers and extensions. SICOMP SIAM J Comput 30
21. Reif JH (2005) Efficient parallel factorization and solution of structured and unstructured linear systems. JCSS J Comput Syst Sci 71
22. Alonso P, Badía JM, Vidal AM (2001) A parallel algorithm for solving the Toeplitz least squares problem. In: Lecture Notes in Computer Science, vol 1981. Springer, Berlin, pp 316–329
23. Alonso P, Badía JM, Vidal AM (2005) Solving the block-Toeplitz least-squares problem in parallel. Concurr Comput Pract Experience 17:49–67
24. Alonso P, Badía JM, Vidal AM (2005) An efficient parallel algorithm to solve block-Toeplitz systems. J Supercomput 32:251–278
25. Alonso P, Badía JM, Vidal AM (2004) Parallel algorithms for the solution of Toeplitz systems of linear equations. In: Lecture Notes in Computer Science, vol 3019. Springer, Berlin, pp 969–976
26. Gohberg I, Kailath T, Olshevsky V (1995) Fast Gaussian elimination with partial pivoting for matrices with displacement structure. Math Comput 64(212):1557–1576
27. Thirumalai S (1996) High performance algorithms to solve Toeplitz and block Toeplitz systems. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign
28. Alonso P, Vidal AM (2005) The symmetric-Toeplitz linear system problem in parallel. In: Lecture Notes in Computer Science, vol 3514. Springer, Berlin, pp 220–228
29. Alonso P, Vidal AM (2005) An efficient parallel solution of complex Toeplitz linear systems. In: PPAM. Lecture Notes in Computer Science, vol 3911. Springer, Berlin, pp 486–493
30. Alonso P, Bernabeu MO, Vidal AM (2006) A parallel solution of hermitian Toeplitz linear systems. In: Computational Science—ICCS. Lecture Notes in Computer Science, vol 3991. Springer, Berlin, pp 348–355
31. Alonso P, Badía JM, Vidal AM (2005) An efficient and stable parallel solution for non–symmetric Toeplitz linear systems. In: Lecture Notes in Computer Science, vol 3402. Springer, Berlin, pp 685–692
32. Van Loan C (1992) Computational frameworks for the fast Fourier transform. SIAM, Philadelphia
33. Heinig G (1994) Inversion of generalized Cauchy matrices and other classes of structured matrices. Linear Algebra Signal Process IMA Math Appl 69:95–114
34. Thirumalai S (1996) High performance algorithms to solve Toeplitz and block Toeplitz systems. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign
35. Alonso P, Vidal AM (2005) An efficient and stable parallel solution for symmetric Toeplitz linear systems. TR DSIC-II/2005, DSIC-Univ Polit Valencia
36. Chan RH, Ng MK, Wong CK (1996) Sine transform based preconditioners for symmetric Toeplitz systems. Linear Algebra Appl 232(1–3):237–259
37. Blikberg R, Sørevik T (2001) Nested parallelism: Allocation of threads to tasks and openmp implementation. Sci Program 9(2-3):185–194

Miguel O. Bernabeu received his Engineer degree in Computer Science from the Universidad Politécnica de Valencia, Spain, in 2005.

He was a Research Fellow with the Universidad Politécnica de Valencia from 2004 through 2007. He is currently a Research Assistant with the Computing Laboratory of the University of Oxford, UK. His research interests include parallel computing and numerical linear algebra and its applications to signal processing and cardiac modeling and simulation.

Pedro Alonso was born in Valencia, Spain, in 1968. He received the Engineer degree in Computer Science from the Universidad Politécnica de Valencia, Spain, in 1994 and the Ph.D. degree from the same University in 2003. His dissertation was on the design of parallel algorithms for structured matrices with application in several fields of digital signal analysis.

Since 1996 is a full professor in the Department of Computer Science of the Universidad Politécnica de Valencia and he is a member of the High Performance Networking and Computing Research Group of the Universidad Politécnica de Valencia. His main areas of interest include parallel computing for the solution of structured matrices with applications in digital signal processing.

Antonio M. Vidal was born in Alicante, Spain, in 1949. He receives his M.S. degree in Physics from the Universidad de Valencia, Spain, in 1972, and his Ph.D. degree in Computer Science from the Universidad Politécnica de Valencia, Spain, in 1990. Since 1992 he has been in the Universidad Politécnica de Valencia, Spain, where he is currently a full professor and coordinator of the Parallel and Distributed Ph.D. studies in the Department of Computer Science. His main areas of interest include parallel computing with applications in numerical linear algebra and signal processing.