Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Una implementación distribuida del método de gradientes conjugados aplicado al método de elementos finitos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Facundo F. Ciccioli

Director: Alejandro D. Otero

Buenos Aires, 2015

UNA IMPLEMENTACIÓN DISTRIBUIDA DEL MÉTODO DE GRADIENTES CONJUGADOS APLICADO AL MÉTODO DE ELEMENTOS FINITOS

El método de los elementos finitos es ampliamente utilizado en diversas áreas de ingeniería para calcular aproximaciones a ecuaciones diferenciales en derivadas parciales. El último paso del método es la resolución de un sistema de ecuaciones lineales ralo, cuya cantidad de incógnitas crece proporcionalmente al área o al volumen del dominio del problema. Uno de los algoritmos más usados para la resolución de estos sistemas lineales es el método de gradientes conjugados. Es sabido que en cualquier aplicación práctica se debe precondicionar el problema para disminuir sustancialmente la cantidad de iteraciones necesarias para alcanzar la convergencia. El círculo virtuoso que se da entre la tecnología y sus aplicaciones empuja constantemente el tamaño máximo de los sistemas lineales que se pueden resolver en tiempos razonables. Este límite se alcanza muy rápidamente si se restringen los cálculos a una única unidad de procesamiento, y para superarlo se recurre a la utilización de muchos procesadores que colaboran de forma simultánea para obtener una solución. En este trabajo se construye una implementación del método de gradientes conjugados donde todas las operaciones se realizan en paralelo sobre una pequeña parte de los datos de entrada, manteniendo la comunicación entre procesos al mínimo. El diseño admite la aplicación de un precondicionador. Se concluye que la implementación lograda es escalable utilizando un problema modelo de solución conocida y que el diseño es flexible realizando una segunda implementación con algunas optimizaciones.

Palabras claves: Método de elementos finitos, método de gradientes conjugados, ecuación diferencial en derivadas parciales, computación de alto rendimiento, precondicionador, MPI

AGRADECIMIENTOS

A Gaby, el amor de mi vida, cuya paciencia sobrepasó todo lo que yo me podía imaginar. Todo lo que tuviste que aguantar, mi mal humor, mis ridículamente erróneas estimaciones de tiempo, aprender a ser mamá conmigo cursando y trabajando, es algo que no estoy seguro de poder devolverte.

A Lucía, que con su metro cero cinco de altura y sus dieciseis kilos es la personita con más densidad de alegría que conozco. Espero que puedas entender mis ausencias y mis pocas paciencias. Por momentos tus demandas fueron una fuerza enorme empujándome a terminar.

Mi familia atravezó momentos muy difíciles durante la última etapa de mi carrera. Aún así el apoyo siempre estuvo, ya sea mi vieja haciéndole compañía a mi nena y mi mujer mientras yo me sentaba a hacer esto que leen, o no diciéndome que largue todo y me deje de hinchar. Allá hace miles de años, mi viejo me prestaba la oreja mientras yo me rompía la cabeza tratando de resolver los problemas de análisis del CBC. También me ayudó a decidir empezar a cursar también Ciencias Matemáticas y tiempo más tarde a dejarla.

A Marta, José y Abel, por cuidar a Lucía dándome horas para avanzar con la tesis.

A mis compañeros de cursada, algunos devenidos en amigos, que allá por el año 2004 le hablaron a un tipo re tímido en ese entonces, dando comienzo a mi vida social en la facultad. Me costó bastante entender que con la ayuda de los compañeros de cursada el paso por las materias se hace *mucho* más tolerable. Y como si eso fuera poco, muchas veces uno se cruza con gente muy copada. Pablo Terlisky, Juan Cruz Rojas Pin, Leandro Montero, Pachi, Cristián Archilla, El Pocho son algunos de los que me acompañaron durante los primeros años pero con quienes ya no tengo mucho contacto. Matu, Román, Tommy, enormes compañeros de grupo que tuvieron que aguantar mis obsesiones y mis "pará que primero leo el manual" y así y todo hoy en día siguen siendo grandes amigos míos. Fran Laborda, con quien solo cursé algunas de las primeras materias pero que por suerte nos seguimos viendo en salidas nocturnas y viajes hasta hace no mucho. Los mellizos Peloroso, grandes amigos con quienes cursé toda la secundaria pero recién nos llegamos a conocer el último año, y luego cursamos e hicimos algunos TPs y compartimos cafés y salidas a correr.

Mientras escribo esto se me ocurre que capaz los agradecimientos se tendrían que escribir justo *antes* de empezar con la tesis. Después de todo el proceso me resulta bastante difícil ver más allá del cansancio y el hartazgo que dejan atrás más de dos años de tener este asunto en la cabeza, con lo cual muy fácilmente, escritas ahora, estas palabras resulten injustas. Rememorando un poco, igual, logro recordar los momentos en los que cruzar el puente sobre la Lugones arriba del 107 y ver los pabellones de Ciudad Universitaria me dibujaban una sonrisa en la cara, o las siestas dormidas bajo el gomero entre los pabellones uno y dos entre materia y materia. Creo que sería injusto no agradecerle a la facultad y a toda la gente que hace posible que siga funcionando, con todos los problemas y todas las cosas buenas que tiene. Algunos docentes que me marcaron hasta el día de hoy son Fernando Schapachnik, Mariano Moscato y Nico Rosner.

A Ale, mi director, un gran tipo, que estuvo *siempre* totalmente predispuesto a ayudarme cuando se lo pedía, cuando no se lo pedía, cuando me bloqueaba totalmente o cuando me tenía que explicar la misma cosa por vez número mil.

A Lucía.

Índice general

1. INTRODUCCIÓN

1.1. Contexto

El presente trabajo intenta resolver un problema que se encuentra inmerso en un amplio contexto de teorías y tecnologías. De lo general a lo particular, se trata de resolver una ecuación diferencial en derivadas parciales con condiciones de frontera. El método utilizado para aproximar la solución es el de los elementos finitos (FEM), un método que comienza partiendo el dominio del problema y termina resolviendo un sistema lineal considerablemente grande.

Este trabajo ataca la última parte del proceso, es decir, la resolución del sistema de ecuaciones obtenido a partir de aplicar el FEM a un problema particular. En la figura 1.1 se puede ver un diagrama en bloques con las distintas etapas que involucran la aplicación del FEM. En este capítulo se presenta una breve introducción a los temas relacionados, y se describe cuál fue el punto de partida.

Existen antecedentes durante los últimos años en torno a brindar herramientas para que no sea necesaria una recodificación total cada vez que se quiere resolver un problema nuevo mediante el FEM. Se puede mencionar [19], donde se brindan detalles de la implementación de diversos problemas de la mecánica del continuo en clusters Beowulf, o [5], que permite flexibilidad en la configuración del programa y los métodos de resolución mediante un poderoso intérprete de comandos. En [8] se propone un marco genérico de resolución a diversos problemas de la mecánica computacional, dejando al usuario la programación de los métodos de resolución sin preocuparse por la paralelización y otros detalles técnicos. En [21] se presenta un software completo, extensible y altamente portable para resolver PDEs, el cual permite la máxima flexibilidad brindando la posibilidad de elegir la plataforma a utilizar en tiempo de ejecución. Por último, [12] introduce un framework para resolver instancias del FEM utilizando una red LAN de PCs.

1.2. Ecuaciones diferenciales

Una ecuación diferencial es una ecuación cuya incógnita no es un valor sino una función, en donde también aparece alguna derivada de la función incógnita [9]. Este tipo de ecuaciones surge naturalmente en una gran variedad de ámbitos (física, biología, química, ingeniería, etc.) dado que es muy común que cierta cantidad dependa de la velocidad de cambio de cierta otra cantidad. En física por ejemplo, una versión unidimensional de la

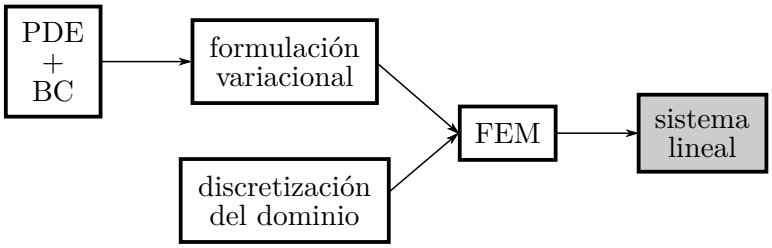

Fig. 1.1: El contexto del FEM y dónde se enfoca este trabajo

segunda ley de Newton puede expresarse mediante la ecuación diferencial:

$$m\frac{\mathrm{d}^2x}{\mathrm{d}t^2}(t) = F(x(t)), \tag{1.1}$$

donde F es una función *conocida* dependiente de la posición $x(t)$ del objeto de masa m, que a su vez depende del tiempo. Si lo que se está modelando es un resorte con uno de sus extremos fijos, y $x(t)$ es el desplazamiento del otro extremo respecto de su posición de reposo, entonces

$$F(x) = -kx,$$

donde k es una constante que depende del resorte. En este caso la solución a la ecuación diferencial resulta:

$$x(t) = A\cos t + B\sin t,$$

con los valores A y B determinados una vez que se establecen las *condiciones iniciales* del problema, o sea, la extensión del resorte cuando $t = 0$. Esta solución puede verificarse reemplazándola en 1.1.

Las ecuaciones diferenciales como 1.1 se denominan ordinarias, y lo que las distingue es que la función incógnita depende de una sola variable. En contraste, las *ecuaciones en derivadas parciales* (PDE) tienen como incógnita una función de dos variables o más, y por ende en su expresión aparecen derivadas parciales de la función buscada. Un ejemplo de una ecuación de este tipo es la siguiente forma de la ecuación del calor:

$$\frac{\partial u}{\partial t} - \alpha\frac{\partial^2 u}{\partial x^2} = 0.$$

Esta ecuación modela la variación de la temperatura en cada punto de una varilla unidimensional a medida que pasa el tiempo, en ausencia de fuentes de calor externas. Esto es, $u(x,t)$ representa la temperatura de la varilla en la posición x en el tiempo t. La constante α es la difusividad térmica del material de la varilla.

Lo esperable para una solución $u(t)$ a la ecuación del calor es que para $t = 0$ coincida con las condiciones iniciales dadas, y a medida que pase el tiempo la temperatura se vaya distribuyendo de manera uniforme hasta que sea prácticamente la misma en toda la varilla. La primera etapa donde la función u depende del tiempo se denomina *transitoria* y la segunda etapa donde la temperatura es esencialmente constante se llama *estado estacionario*. Es posible resolver la ecuación para conocer directamente el estado estacionario, sin necesidad de hacer avanzar t hasta que la temperatura deje de variar. Esto se logra buscando soluciones tales que la derivada de u respecto del tiempo sea idénticamente cero, que es lo que caracteriza al estado estacionario. Bajo esta suposición, la ecuación se convierte en:

$$\frac{\partial^2 u}{\partial x^2} = 0.$$

En este caso las condiciones iniciales dejan de ser relevantes, pues lo que interesa es el comportamiento de u cuando $t \to \infty$, pero no así las *condiciones de borde*, que se definen como el valor de la incógnita y sus derivadas involucradas en los bordes del dominio (en este caso, los extremos de la varilla).

Existen diversas técnicas que se pueden utilizar para intentar hallar una solución analítica a una ecuación diferencial, pero ninguna las puede resolver todas. Incluso en muchos casos el resultado de estos métodos provee una solución analítica que no se puede

expresar mediante funciones elementales, y en cambio la solución obtenida está expresada como una serie o una integral.

Entre las aplicaciones más importantes de las ecuaciones diferenciales se encuentran el modelado y simulación de distintos fenómenos. Por ejemplo, si se desea conocer las propiedades de disipación térmica de la estructura de un satélite, se puede utilizar esa estructura como dominio de las ecuaciones que rigen el fenómeno de disipación que se quiere analizar. Se resuelven esas ecuaciones utilizando como condiciones de borde y/o iniciales distintas fuentes de calor, como pueden ser los componentes electrónicos o el Sol, y con la solución se observa si la estructura es adecuada para disipar la cantidad de calor necesaria. De encontrar anomalías se itera sobre el diseño y se repite el proceso.

En el modelado se puede decir que la idea es la inversa que en la simulación. Lo que se pone a prueba en este caso es la ecuación diferencial, que se supone modela determinado fenómeno. Por ejemplo para realizar pronósticos meteorológicos se utilizan modelos que en base a las condiciones observadas en cierto momento, predicen el comportamiento de la atmósfera en un futuro cercano. Dichos modelos en general tienen la forma de ecuaciones diferenciales, y deben ser validados en base a comparar lo que las ecuaciones predijeron y mediciones de la evolución real de las distintas variables. Cuando se encuentran grandes divergencias o se descubren nuevos patrones, se modifican las ecuaciones.

Dado el modo de trabajo dentro de las aplicaciones mencionadas, surgió la necesidad hace mucho tiempo de métodos que cumplan con ciertas características:

- ser lo suficientemente flexibles como para que se puedan adaptar con facilidad a una gran variedad de ecuaciones,
- que sea relativamente sencillo especificar dominios de alta complejidad como lo pueden ser un avión entero o la tobera del motor de un cohete, y la misma o similares ecuaciones puedan ser resueltas sobre diferentes dominios sistemáticamente, sin ser necesario replantear el problema,
- proporcionar la solución expresada de una manera que permita su rápida representación y análisis.

Existen varios métodos que cumplen con estos criterios en mayor o menor medida, y lo logran resignando la precisión infinita de los métodos exactos: el resultado obtenido es una aproximación a la solución, sobre una aproximación del dominio. El método de los elementos finitos forma parte de esta categoría.

1.3. Método de elementos finitos

El método de elementos finitos (FEM) es una técnica que permite aproximar la solución de ecuaciones diferenciales en derivadas parciales con condiciones de frontera. La aplicación del método convierte una PDE en un sistema lineal. Para lograrlo se deben realizar una serie de pasos, uno de los cuales es una reformulación del problema. Esto excluye al FEM de la categoría de algoritmo, pues la transformación no es algo que pueda llevar acabo una máquina.

Debido a que es necesaria esta reformulación, el FEM no es fácil de abstraer. Es por esto que se optó por presentarlo aplicándolo a la resolución de un problema concreto, y para simplificar las cosas aún un poco más, el problema se planteó en una dimensión. Para

una presentación más detallada sobre los fundamentos del FEM se puede recurrir a [2]. El problema que se utilizará como ejemplo es:

hallar u tal que cumpla la ecuación

$$-u''(x) + u(x) = x \quad \forall x \in (0,1), \tag{1.2}$$

sujeta a las condiciones de borde

$$u(0) = u(1) = 0. \tag{1.3}$$

La sección 1.3.5 muestra las implicancias de utilizar un dominio de más dimensiones.

1.3.1. Formulación débil

Antes de poder aplicar el FEM a la resolución de una ecuación diferencial, se la debe convertir de una formulación clásica o fuerte a una formulación débil. El problema de ejemplo presentado en la sección anterior se encuentra en su formulación clásica.

Una formulación débil de una ecuación diferencial no solo permite el desarrollo del FEM sino que también trae consigo otras ventajas. La ecuación 1.2 tiene muy buenas propiedades que la hacen fácil de tratar. Su parte no-homogenea es infinitamente derivable así como también lo son sus coeficientes. En este caso, es posible calcular una solución exacta (utilizando la transformada de Laplace, por ejemplo) que satisfaga la ecuación en todos los puntos del dominio. En muchas aplicaciones, sin embargo, se encuentran ecuaciones cuyas propiedades no son tan buenas, lo cual puede redundar en que no exista ninguna solución a la ecuación, o sí existe pero es imposible expresarla de manera cerrada mediante funciones elementales (debiendo recurrir al uso de series o integrales, por ejemplo). La ecuación 1.4 es un ejemplo simple de un caso donde la parte no-homogenea no es ni siquiera una función en el sentido usual:

$$-u''(x) + u(x) = \delta(x - \frac{1}{2}) \quad \forall x \in (0,1). \tag{1.4}$$

La función δ es conocida como la *delta de Dirac*, y es en realidad lo que se denomina una *función generalizada*. La teoría de funciones generalizadas es la que fundamenta formalmente el uso y abuso que ingenieros y físicos hacen de las funciones clásicas cuando se refieren a cosas como la densidad de una masa puntual, o modelan el golpe de un martillo contra una superficie mediante una fuerza de magnitud infinita pero de duración cero. Para representar tales situaciones se puede suponer la existencia de una función δ que cumpla estas propiedades:

$$\delta(x) = 0 \quad \forall x \in \mathbb{R}_{\neq 0}$$
$$\int_{-\infty}^{\infty} \delta(x)\mathrm{d}x = 1$$

Con esta definición, la función $p\delta$ se puede pensar como la distribución de densidad de un problema donde en $x = 0$ se encuentra una masa puntual de masa p. Es claro que una función que cumpla con esas propiedades no existe, pues la integral de una función cuyo soporte es finito es cero. Las funciones generalizadas formalizan las nociones detrás de la idea de la función δ, validando el planteo de problemas como el de la ecuación 1.4.

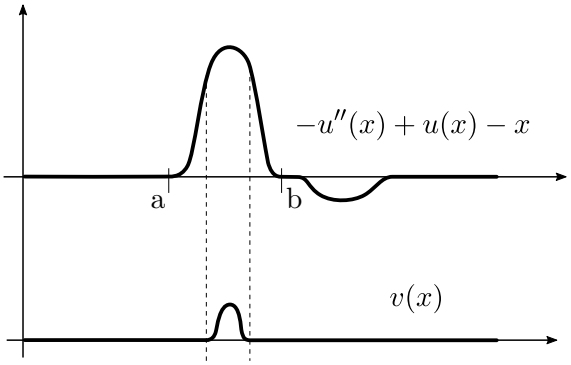

Fig. 1.2: El error producido por una posible solución al planteo clásico de la ecuación 1.2 y una función de prueba que la descartaría como solución a la formulación débil

La ecuación 1.4 se puede resolver de manera exacta y cerrada. La solución que se obtiene muestra una discontinuidad en u' cuando $x = \frac{1}{2}$ y por lo tanto u'' no existe en ese punto. Luego, no queda claro el significado de 1.4 para $x = \frac{1}{2}$. El problema radica en que el requirimiento de que la ecuación sea satisfecha en *todo* punto del dominio es demasiado fuerte. De esta idea surge la necesidad de una formulación menos exigente.

La formulación débil de la ecuación 1.2 se puede expresar como encontrar una función $u \in H$ tal que:

$$\int_0^1 (-u'' + u)v\mathrm{d}x = \int_0^1 xv\mathrm{d}x \quad \forall v \in H \tag{1.5}$$
$$u(0) = u(1) = 0,$$

donde H es una familia de funciones que garantice que las integrales involucradas tengan sentido. Las funciones v se denominan *funciones de prueba*. Con esta formulación se relaja la pretensión de que la ecuación se cumpla en todo punto, reemplazándola por la condición de que se cumpla *en promedio*. Las funciones v juegan el rol de especificar de qué manera se va a pesar un determinado promedio. Por ejemplo, cuando $v(x) = 1$ la formulación débil establece que la ecuación se debe cumplir en promedio sobre todo el dominio.

Es claro que una solución al problema original es también solución de la formulación débil, pero en principio puede parecer que se han introducido soluciones de más. Para entender por qué esto no es así se debe considerar que la familia H de funciones de prueba puede contener miembros con soporte tan pequeño como se desee. Teniendo esto en cuenta, se puede pensar en que si el error $-u'' + u - x$ es mayor a cero en cierto intervalo (a, b), existirá una $v \in H$ cuyo soporte esté contenido en (a, b) y sea toda positiva. Es claro que el producto del error por la función de prueba v será positivo y de integral no nula sobre todo el dominio, descartando a u como posible solución de la formulación débil. La figura 1.2 muestra conceptualmente esta situación.

Esta formulación permite expresar naturalmente problemas cuyas funciones involucradas presentan irregularidades como las de la ecuación 1.4, pues por ejemplo ciertos tipos de discontinuidades aún se pueden integrar, pero no derivar. Además otra ventaja reside en que representa de manera más fidedigna la forma en la que se perciben los fenómenos físicos. Para observar cierto comportamiento lo que se hace es tomar mediciones en puntos específicos del dominio, y dado que todo instrumento de medición tiene una superficie no nula y finita, en realidad se debe asumir que el valor obtenido representa el promedio de

valores reales sobre una pequeña parte del dominio. La formulación débil asegura que su solución parezca correcta ante cualquier posible medición de este tipo.

1.3.2. Aproximaciones de Galerkin

El método de las aproximaciones de Galerkin es el que permite transformar una formulación débil de una PDE en un sistema lineal de ecuaciones. La idea consiste en tomar como H un espacio vectorial de funciones, que como tal puede ser de dimensión infinita. Un ejemplo de este tipo es el conjunto de todas las funciones infinitamente derivables que se anulan en los extremos del intervalo $[0, 1]$. Utilizando las series de Fourier, toda función $v \in H$ se puede expresar como

$$v(x) = \sum_{i=1}^{\infty} a_i \psi_i(x)$$

donde para cada i la función $\psi_i(x)$ está dada por

$$\psi_i(x) = \sqrt{2} \sin(i\pi x).$$

Se puede probar que el conjunto de todas las funciones ψ_i forma una *base* de H, y al ser infinito determina que el espacio vectorial es de dimensión infinita.

Si H es de dimensión infinita, entonces para determinar una función en él es necesario calcular infinitos coeficientes a_i, lo cual no resulta muy práctico a la hora de querer hacer cálculos con una máquina finita en tiempo finito. Lo que hace la técnica de las aproximaciones de Galerkin es forzar a que H tenga dimensión finita cortando la serie en algún índice N. De esta forma, si $\{\phi_1, \phi_2, \dots\}$ es una base de H, $\{\phi_1, \dots, \phi_N\}$ es una base del subespacio H_N de H y toda función en él queda determinada por N coeficientes β_i de manera que

$$v_N(x) = \sum_{i=1}^{N} \beta_i \phi_i(x). \tag{1.6}$$

La solución u se puede expresar análogamente utilizando coeficientes α_i:

$$u_N(x) = \sum_{i=1}^{N} \alpha_i \phi_i(x). \tag{1.7}$$

Ahora, si se reemplaza u y v en 1.5 por las expresiones 1.6 y 1.7 se obtiene

$$\int_0^1 \left[\left(-\sum_{i=1}^{N} \alpha_i \phi_i'' + \sum_{i=1}^{N} \alpha_i \phi_i \right) \sum_{i=1}^{N} \beta_i \phi_i \right] \mathrm{d}x = \int_0^1 x \sum_{i=1}^{N} \beta_i \phi_i \, \mathrm{d}x$$

la cual utilizando la linealidad de la integral y reorganizando las sumas se puede reescribir como

$$\sum_{i=1}^{N} \beta_i \left[\sum_{j=1}^{N} \left(\int_0^1 \left(-\phi_j + \phi_j'' \right) \phi_i \, \mathrm{d}x \right) \alpha_j \right] = \sum_{i=1}^{N} \beta_i \int_0^1 x \phi_i \, \mathrm{d}x. \tag{1.8}$$

Es conveniente expresar esta ecuación de manera más compacta mediante las siguientes definiciones:

$$K_{ij} = \int_0^1 \left(-\phi_j + \phi_j'' \right) \phi_i \, \mathrm{d}x \tag{1.9}$$

$$F_i = \int_0^1 x \phi_i \, \mathrm{d}x. \tag{1.10}$$

Así, 1.8 resulta:

$$\sum_{i=1}^{N} \beta_i \left(\sum_{j=1}^{N} K_{ij}\alpha_j \right) = \sum_{i=1}^{N} \beta_i F_i. \tag{1.11}$$

Recordando que los β_i representan los coeficientes de la función de prueba v, y que 1.5 se tiene que satisfacer para toda v, 1.11 debe valer para toda elección posible de los β_i. Esto determina un sistema lineal de ecuaciones, pues tomando $\beta_1 = 1$ y $\beta_i = 0$ para $i \neq 1$, 1.11 es:

$$\sum_{i=1}^{N} K_{1i}\alpha_i = F_1,$$

y tomando $\beta_2 = 1$ y $\beta_i = 0$ para $i \neq 2$,

$$\sum_{i=1}^{N} K_{2i}\alpha_i = F_2.$$

Siguiendo así hasta considerar $\beta_N = 1$ y $\beta_i = 0$ para $i \neq N$ se obtiene el sistema lineal de ecuaciones representado por la matriz K cuyos coeficientes son los K_{ij} y la parte no homogénea es el vector F con los coeficientes F_i:

$$K\alpha = F \tag{1.12}$$

Los α_i son los coeficientes que determinan la aproximación a la solución a 1.5 mediante la fórmula 1.7.

Es importante destacar que lo que se acaba de hacer rompe la equivalencia de soluciones. Es decir, la función u_N *no es una solución* de la ecuación 1.5, sino que es una aproximación. Se espera que cuando mayor sea el N, menor será el error respecto de la solución real, pero también crece cuadráticamente el tamaño del sistema que hay que resolver. Por otro lado, este proceso de manipulación algebráica de las integrales para llegar a una expresión elemental de los coeficientes de K debe hacerse para cada ecuación distinta que se quiera resolver. Este es el paso que proscribe al FEM de la denominación de algoritmo.

1.3.3. Elección de funciones base

Como ya se ha dicho, el FEM no es simplemente un algoritmo sino más bien una serie de pasos y heurísticas que requieren intervención humana para llegar a un planteo del problema cuya solución se pueda programar. Por esta razón, es complicado presentar el método de manera abstracta sin recurrir a un ejemplo concreto. Para alivianar un poco esta situación, esta sección presenta *primero* un ejemplo de lo que haría esta fase del FEM, para luego derivar las ventajas y propiedades esenciales que lo caracterizan.

El método parte de la técnica de las aproximaciones de Galerkin, y establece una forma particular de elegir las funciones base ϕ_i que determinan el conjunto H donde reside la aproximación a la solución. En esencia, el principal objetivo del FEM es que los coeficientes 1.9 y 1.10 se puedan calcular metódicamente.

Para facilitar la construcción de las ϕ_i es de utilidad aplicar la fórmula de integración por partes al integrando $\phi_j''\phi_i$ de la definición de K_{ij}. De esta manera, y luego de considerar

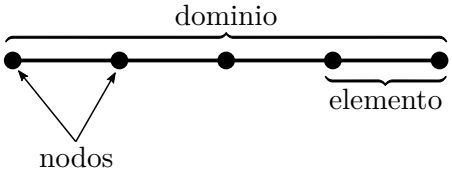

Fig. 1.3: Relación nodos–elementos

las condiciones de contorno de las ϕ_i, se obtiene la siguiente expresión que no involucra derivadas segundas:

$$K_{ij} = -\int_0^1 \phi_i\phi_j + \phi_i'\phi_j' \,\mathrm{d}x \tag{1.13}$$

Se comienza dividiendo el intervalo de integración ($\Omega = [0,1]$) en pequeñas partes denominadas *elementos*. Además, se determinan sobre todo Ω una serie de puntos llamados *nodos* y a cada elemento se le asocia un conjunto de ellos. Como en el ejemplo de la figura 1.3, en general elementos adyacentes comparten nodos, con lo cual los nodos de los elementos no son disjuntos dos a dos.

A cada nodo i excepto los de los extremos del intervalo se le asigna una ϕ_i. Para este ejemplo se usarán como ϕ_i funciones "sombrero", las cuales se ven representadas en la figura 1.4 junto con sus derivadas. La punta de cada sombrero se encuentra en el nodo correspondiente con un valor igual a 1.

Para comprender mejor las características que va a tener la función que aproxime a la solución y también la importancia que tiene la elección de la forma de las funciones base, la figura 1.5 muestra una función cualquiera f y una posible aproximación f_H utilizando las ϕ_i que se están considerando.

Ahora, con las ϕ_i de la figura 1.4, se puede volver a la fórmula 1.13 e intentar calcular el valor de la integral. Lo primero que se puede observar es que gracias a la aditividad de la integral y a la división de Ω en elementos, se tiene

$$\int_0^1 \phi_i\phi_j + \phi_i'\phi_j' \,\mathrm{d}x = \sum_{k=1}^{4} \int_{\Omega_k} \phi_i\phi_j + \phi_i'\phi_j' \,\mathrm{d}x, \tag{1.14}$$

con lo cual tiene sentido considerar una matriz K^e para cada elemento, definida como

$$K_{ij}^e = \int_{\Omega_e} \phi_i\phi_j + \phi_i'\phi_j' \,\mathrm{d}x. \tag{1.15}$$

Con esto se puede enfocar la atención en las características de cada *matriz elemental* K^e. Observando que el soporte de cada ϕ_i es la unión de los dos elementos a los cuales pertenece el nodo i, se puede ver que $K_{ij}^e = 0$ si los nodos i y j no pertenecen al elemento e, pues ambos productos $\phi_i\phi_j$ y $\phi_i'\phi_j'$ se anulan. Luego se puede reducir el análisis a calcular K_{ij}^e cuando los nodos i y j pertenecen a e, que son, en el ejemplo, solo 3: $i = j$, i un extremo y j el otro, y viceversa. Incluso se puede notar que todas las ϕ_i son análogas si se las abstrae de la posición del pico dentro de Ω y están construidas a partir de dos formas: una función lineal creciente (ψ_A) y una función lineal decreciente (ψ_B). A cada una de estas funciones se las denomina *funciones de forma* del elemento. Esta propiedad de las funciones base elegidas permite abstraerse de un elemento concreto y visualizar uno genérico que solo

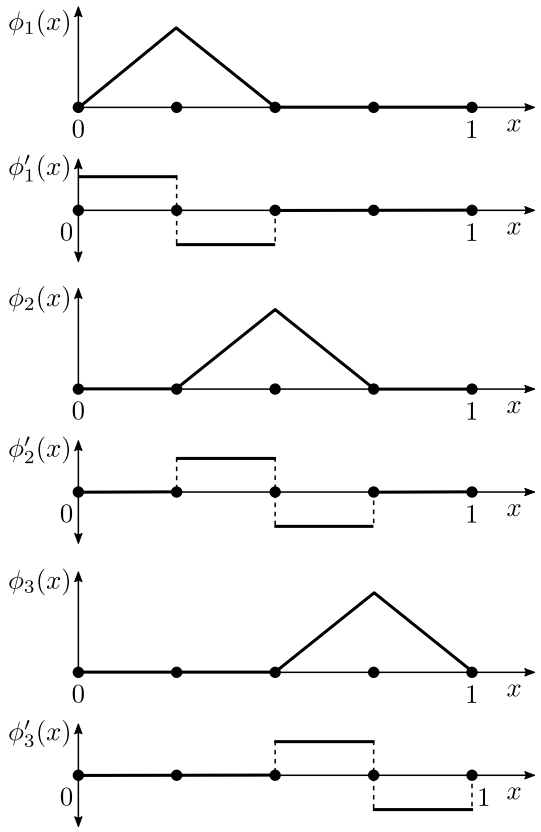

Fig. 1.4: Las funciones ϕ y sus derivadas para el dominio de la figura 1.3

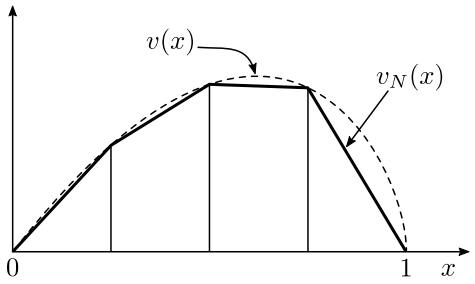

Fig. 1.5: La función v es aproximada por un elemento de H, v_N

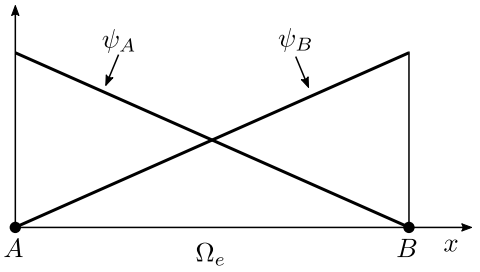

Fig. 1.6: Representación del elemento genérico junto con las funciones de forma

tenga en cuenta las partes relevantes de todo elemento, esto es: sus dos nodos A y B, y las dos funciones de forma asociadas a cada uno ψ_A y ψ_B.

A este elemento genérico también le corresponde una matriz elemental, cuyos coeficientes son

$$K^e_{AA} = \int_A^B {\psi'_A}^2 + {\psi_A}^2 \,\mathrm{d}x$$
$$K^e_{BB} = \int_A^B {\psi'_B}^2 + {\psi_B}^2 \,\mathrm{d}x$$
$$K^e_{AB} = K^e_{BA} = \int_A^B \psi'_A \psi'_B + \psi_A \psi_B \,\mathrm{d}x,$$

los cuales se pueden calcular fácilmente pues las funciones de forma son lineales y el desplazamiento en x no modifica el valor de la integral.

Estos son los coeficientes que calcula el algoritmo del FEM, y luego los ensambla en cada matriz elemental concreta las cuales sumadas dan como resultado la matriz global K.

De este ejemplo se pueden desprender las propiedades esenciales que requiere el FEM de las funciones base para que sean de utilidad:

- deben estar definidas por partes sobre cada elemento;
- debe ser posible abstraerlas del nodo al que pertenecen, para lo cual es necesario que cada parte sea un elemento de un conjunto reducido de funciones de forma;
- es necesario que pertenezcan al conjunto H de funciones de prueba adminisibles pues la integral 1.13 tiene que tener sentido.

El último punto justifica la transformación algebráica que se le hizo a 1.9, pues de no haberla hecho no se podrían haber utilizado las funciones sombrero como funciones base. La derivada segunda de estas funciones es una delta de Dirac (ver 1.3.1), cuyo cuadrado no es integrable. En líneas generales, cuanto de menor grado sean las derivadas que aparecen en la formulación débil de problema mejor será, pues se amplía el conjunto de funciones donde se permite buscar la solución.

Como punto extra, que no es esencial pero facilita la interpretación de la solución al sistema 1.12, se pide que las funciones base sean tales que el valor de una combinación lineal de ellas en uno de los nodos sea igual al coeficiente correspondiente de esa combinación lineal. Es decir, que

$$\left(\sum_{i=1}^{N} \beta_j \phi_i\right)(x_j) = \beta_j,$$

donde x_j es la coordenada del j-ésimo nodo. Esto se garantiza fácilmente haciendo valer 1 a cada ϕ_i sobre su nodo y cero sobre todos los demás.

La implementación práctica del FEM incluye muchas otras aristas. Se puede mencionar como ejemplo, que las integrales de 1.10 y 1.15 se realizan numéricamente utilizando una regla de cuadratura apropiada [2, 3]. Además en esta sección se utilizaron como ejemplo elementos con funciones de forma lineales. En la práctica y dependiendo de la aplicación se pueden utilizar funciones de mayor orden dando lugar a elementos finitos con otras propiedades. De hecho, en el FEM existen dos estrategias para mejorar la aproximación a la solución de un problema aumentando la cantidad de grados de libertad incógnita.

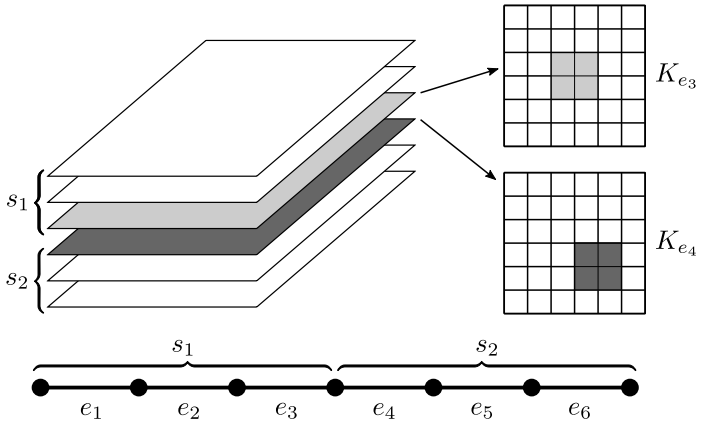

Fig. 1.7: Matriz tridimensional donde cada capa representa la matriz restringida a un elemento

La primera es el llamado refinamiento h, donde se disminuye el tamaño de los elementos dejando el orden de sus funciones de forma constante. La segunda llamada refinamiento p, consiste en aumentar el orden de las funciones trayendo aparejado un aumento de la cantidad de nodos por elemento.

1.3.4. Técnica de subestructurado

La técnica del subestructurado provee una forma de distribuir la matriz del sistema de ecuaciones lineales que resulta del FEM en dos o más procesos. Esto abre la posibilidad a maneras de resolver el sistema paralelamente en cada proceso, con la esperanza de disminuir el tiempo de ejecución o resolver problemas más grandes.

La idea es agregar un nivel intermedio de subdivisión: los elementos se agrupan en conjuntos disjuntos llamados *subdominios*. Esta separación induce otra sobre los nodos, a saber los que están sobre el borde de uno o más subdominios, que llamaremos *externos*, y los que no, que llamaremos *internos*.

Para ayudar a comprender la estructura inducida por esta organización de los elementos, es conveniente visualizar la matriz K como una matriz en tres dimensiones. Como se ve en la figura 1.7, lo que se quiere decir con esto es una "pila" de matrices, donde cada matriz de la pila se calcula de la misma forma que K pero con las integrales restringidas a un solo elemento. Entonces cada matriz de la pila se corresponde con un elemento, y cada posición dentro de una matriz estará asociada a un par de nodos (o lo que es equivalente, funciones ϕ): uno para la fila y otro para la columna. Se llamará *pilar* a considerar una misma posición para todas las matrices de la pila. De esta forma, la matriz K se obtiene sumando todas las capas.

Con esta idea, elegir un subdominio equivale a elegir un conjunto de capas. Si se considera ahora la elección de un par de nodos internos de ese subdominio, entonces se está aumentando la restricción a un pilar. Dadas las propiedades de las funciones ϕ presentadas en la sección 1.3.3, puede pasar una de dos cosas:

- Los conjuntos de elementos a los que pertenecen sendos nodos son disjuntos
- Ambos nodos corresponden a un elemento en común

En el primer caso el pilar estará íntegramente conformado por ceros; en el segundo, habrá algunas posiciones no nulas, pero la característica a destacar es que estas posiciones solo pueden corresponder a elementos del subdominio elegido. Esta propiedad indica que se puede extraer la submatriz correspondiente a los nodos internos de un subdominio y procesarla de manera totalmente independiente al resto.

Si se hace lo mismo que se acaba de hacer pero tomando un nodo interno y uno externo, se llega a la misma conclusión: la submatriz asociada a un subdominio y a nodos internos en sus filas y externos en sus columnas es independiente de todo lo demás. En cambio, tomando dos nodos externos esta independencia se pierde. En el caso unidimensional que se está considerando esto se puede apreciar tomando el pilar que le corresponde a elegir dos veces el mismo nodo. Un nodo externo tiene la distinción de que pertenece a elementos de subdominios distintos. Esto impacta en que habrá posiciones no nulas del pilar tanto en capas del subdominio como en otras correspondientes a otros subdominios, como lo evidencia el hecho de que en la figura 1.7 las matrices K_{e_3} y K_{e_4} comparten una posición no nula. Por lo tanto, al contrario que en los otros casos, no se puede extraer la submatriz correspondiente a un subdominio y sus nodos externos. Todo lo que se puede hacer es extraer ese sub-bloque y asociarlo a un proceso, pero se debe tener presente que ese proceso solo posee información parcial sobre la submatriz deseada. Cada vez que se quiera hacer una operación sobre esta submatriz será necesario considerar la información que tienen los otros procesos.

En la práctica se realiza una reorganización más para poder expresar operaciones sobre las submatrices que se acaban de describir. Esto es, se permutan las filas y columnas de K de manera tal que todas las correspondientes a nodos externos queden consecutivas. De esta forma se llega a una matriz en bloques así:

$$K = \begin{bmatrix} K_{ee} & K_{ei} \\ K_{ie} & K_{ii} \end{bmatrix}$$

donde el primer subíndice indica el tipo de nodos de las filas y el segundo el de las columnas, siendo e externos e i internos. Si se ordenan los nodos de manera tal de respetar el mismo orden topológico de izquierda a derecha que en la figura 1.7, entonces la estructura de K es como la que se ve en la figura 1.8.

1.3.5. El FEM en más dimensiones y el problema modelo

El FEM despliega su máximo potencial cuando se lo utiliza para resolver problemas en dominios de 2 o 3 dimensiones. El uso de elementos para especificar la superficie o el volumen donde se quiere resolver una ecuación le otorga una enorme versatilidad, facilitando su aplicación a problemas con dominios muy complicados. Más aún, los elementos no tienen por qué ser todos iguales, lo cual permite crear mallas con mayor densidad de nodos en lugares donde se desea hacer un análisis más detallado de la solución. Un ejemplo de malla bidimensional compleja se puede apreciar en la figura 1.9.

Las diferencias entre el planteo unidimensional y los de más dimensiones no son muchas. En un plano, los elementos ahora pueden ser cualquier polígono que permita cubrir el dominio (particularmente se utilizan triángulos y cuadrilateros) o incluso pueden estar delimitados por curvas. Al igual que en el caso unidimensional, no existe restricción sobre dónde colocar los nodos de los elementos. La figura 1.10 muestra un ejemplo de elementos cuadrados con 8 nodos, que es el tipo de elementos usado para la solución del problema modelo en este trabajo. Situaciones análogas se dan en 3 dimensiones.

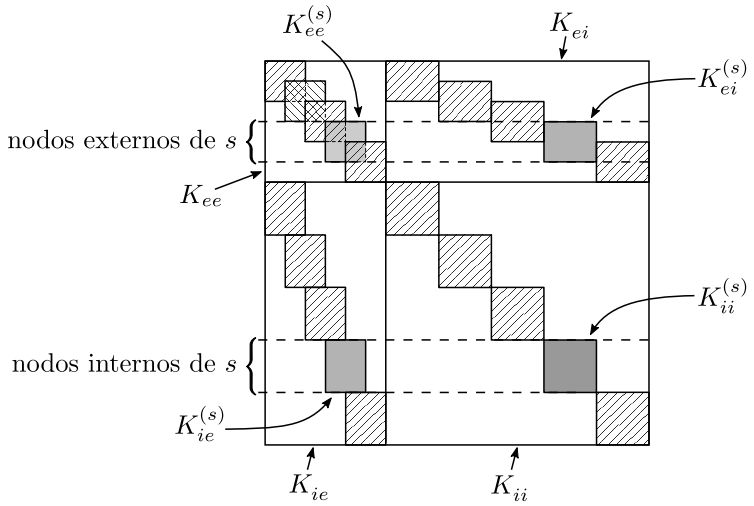

Fig. 1.8: Estructura de la matriz K

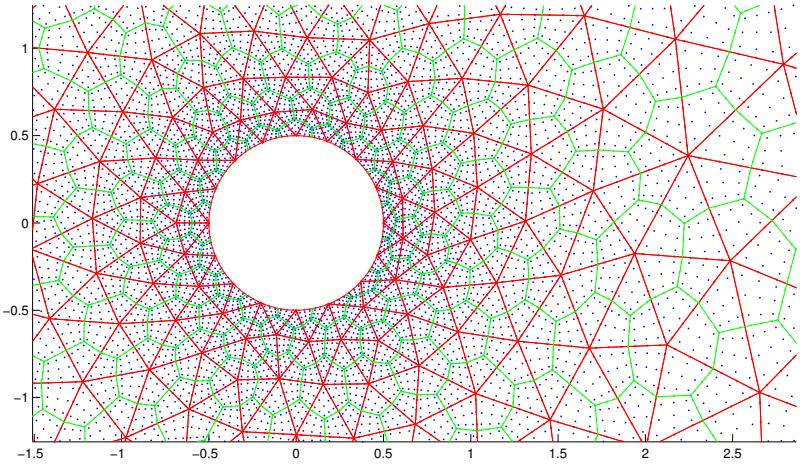

Fig. 1.9: Ejemplo de una malla en dos dimensiones, con elementos no uniformes y nodos dentro de los mismos

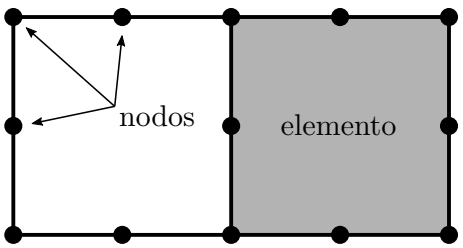

Fig. 1.10: Relación nodos–elementos en dos dimensiones

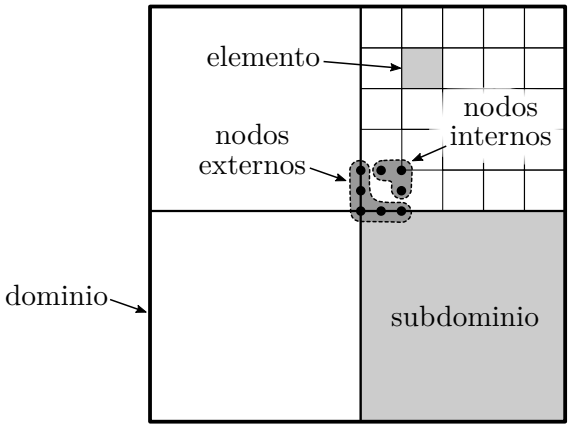

Fig. 1.11: Subdominios en dos dimensiones y su relación con los elementos

La figura 1.11 muestra la relación entre un subdominio, los nodos externos y los nodos internos si se utiliza la técnica de subestructurado en un dominio bidimensional. Observando esta figura se puede apreciar que los subdominios son al dominio lo que los elementos a los subdominios, es decir, la técnica de subestructurado se puede pensar como aplicar el FEM directamente sobre un dominio dividido en elementos que son los subdominios. Los nodos de estos elementos son la unión de los nodos de cada elemento de la malla original que conforma el subdominio.

A lo largo de todo este trabajo se utilizó un problema modelo para poder corroborar la correctitud de los algoritmos implementados. Dicho problema está basado en la ecuación de Poisson sobre un dominio cuadrado de lado unitario, con condiciones de Dirichlet homegéneas:

$$-\nabla^2\phi = f, \tag{1.16}$$

sobre el dominio $\Omega = [0,1]^2$, con $\phi(\partial\Omega) = 0$. La función f es tal que la solución ϕ es conocida y expresable:

$$\phi(x,y) = e^{xy}\sin(\pi x)\sin(\pi y).$$

Imponiendo esta solución resuta que f tiene que ser

$$\begin{aligned} f(x,y) = -\, e^{xy}\left\{\left(y^2 - 2\pi^2 + x^2\right)(\sin(\pi x)\sin(\pi y)) + \right. \\ \left. 2\pi\left(y\cos(\pi x)\sin(\pi y) + x\sin(\pi x)\cos(\pi y)\right)\right\}. \end{aligned}$$

Conociendo la solución exacta a la ecuación 1.16 fue posible corroborar que los resultados obtenidos por el código implementado eran los correctos por medio del cálculo del error respecto de la solución exacta.

1.4. Punto de partida: resolución parcialmente distribuida

Este trabajo se realizó utilizando un framework de propósito general implementado en C++, desarrollado en [16] y extendido en [4]. Una adaptación de este framework para que soporte la resolución paralela se encuentra en [14].

En [16] se presenta un paquete de clases con el objetivo de facilitar y sistematizar la formulación de un problema a ser resuelto mediante el FEM. Se encapsulan conceptos

como nodo, dominio y elemento para permitir una descripción relativamente sencilla de cada faceta del problema. Por ejemplo, se pueden implementar diversos tipos de elementos, representando las distintas configuraciones que éstos pueden tener respecto de sus nodos y/o sus funciones ϕ.

Al mismo tiempo que describe el problema, el código provee las herramientas para resolverlo. Para esto se vale de clases que representan matrices ralas y densas, cuyo objetivo principal es desacoplar al resto de las clases de las bibliotecas de resolución de sistemas lineales. El resto de las clases se encarga de construir el sistema lineal, teniendo en cuenta el tipo de elemento, la distribución de los nodos en el dominio y la ecuación que se está resolviendo. También se incluyen facilidades para resolver numéricamente las integrales que son parte esencial del método.

La necesidad constante de resolver problemas más y más grandes en tiempos razonables empuja la optimización del código al límite. Sin embargo, suponiendo que exista un grado de optimización máximo, una vez alcanzado la única manera de disminuir el tiempo de ejecución para un mismo problema sería modificar el hardware: acelerar el reloj del procesador, incrementar los caches, aumentar la velocidad del bus de memoria, etc. Una alternativa potencialmente de menor costo y que no precisa esperar que se den los avances tecnológicos necesarios para contar con hardware más rápido, es modificar el código para que sea capaz de resolver el problema utilizando más de un procesador. Esta técnica se conoce como *paralelización*, y en general consiste en partir el problema en partes más pequeñas de alguna forma conveniente, y hacer que cada procesador resuelva una parte. Idealmente un problema que se resuelve en tiempo T en un solo procesador se resolvería en tiempo $\frac{T}{n}$ en n procesadores.

Como un primer paso hacia la paralelización de la resolución del sistema lineal del FEM, [14] presenta una extensión al trabajo previo. La idea central reside en aplicar la técnica del subestructurado y explorar el uso del complemento Schur de una matriz para paralelizar la resolución del sistema lineal.

El complemento Schur provee una manera de resolver un sistema lineal de ecuaciones en dos pasos, valiéndose del uso de sistemas más pequeños en cada uno. Si se tiene el sistema de ecuaciones expresado en bloques:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \end{Bmatrix},$$

realizando la multiplicación se obtienen las siguientes dos ecuaciones:

$$Ax_1 + Bx_2 = b_1 \tag{1.17}$$
$$Cx_1 + Dx_2 = b_2 \tag{1.18}$$

Despejando x_2 de 1.18, reemplazándola en 1.17 y luego despejando x_1 se obtiene un sistema que permite calcular los valores de x_1 y x_2 por separado:

$$\left(A - BD^{-1}C\right) x_1 = b_1 - BD^{-1}b_2 \tag{1.19}$$
$$Dx_2 = b_2 - Cx_1 \tag{1.20}$$

Se define el *complemento Schur* del bloque D como la matriz asociada a 1.19: $A - BD^{-1}C$.

Se puede demostrar que el complemento Schur del bloque K_{ii} compuesto por los bloques K_{ii}^s de cada subdominio (como en la figura 1.8) es la suma de los complementos Schur

de las matrices restringidas a cada subdominio:

$$K_{Sch} = \sum_{i=1}^{N} K_{ee}^{s_i} - K_{ei}^{s_i} {K_{ii}^{s_i}}^{-1} K_{ie}^{s_i} = \sum_{i=1}^{N} K_{Sch}^{s_i} \tag{1.21}$$

Utilizando este hecho es posible operar paralelamente con cada $K_{Sch}^{s_i}$ para obtener la solución sobre los nodos externos, y luego calcular la de los nodos internos por medio de 1.20. En este proceso es necesario resolver dos sistemas de ecuaciones, lo cual da la posibilidad de experimentar con distintas combinaciones de solvers para cada sistema. Este trabajo está basado en lo realizado en [14], en donde se exploran dos de estas posibilidades:

- Un *solver* directo para ambos pasos
- Un *solver* iterativo para el paso que resuelve los nodos externos y uno directo para calcular los internos

La paralelización en el caso de dos solvers directos se aplicó únicamente para el cáculo de $K_{Sch}^{s_i}$, concentrando luego la K_{Sch} global en un solo proceso y resolviendo allí para los nodos externos. Luego se envía esta solución a todos los otros procesos, los cuales se encargan de resolver el sistema para sus nodos internos.

Para el caso del *solver* iterativo se probó con la implementación del método de gradientes conjugados (CG) que provee la biblioteca MKL de Intel [13]. Esta biblioteca si bien no cuenta con soporte para paralelización, sí provee un mecanismo denominado *Reverse Call Interface* (RCI) que le permite al usuario tener cierto grado de control sobre las operaciones que hace el algoritmo en cada iteración. En este caso se aprovecha esta inversión de control para realizar la multiplicación entre la matriz del sistema y la solución actual de forma simultánea y por separado en cada proceso. Los resultados son nuevamente concentrados en el proceso que está ejecutando el CG para que continúe con la iteración.

Las mediciones muestran que la utilización de un *solver* iterativo puede llegar a ser hasta 4 veces más rápida que si se utiliza un *solver* directo en ambas. Otra ventaja radica en que en ningún momento de la ejecución es necesario almacenar la matriz K_{Sch} global entera en un único proceso. Sin embargo, la implementación del *solver* iterativo paralelo que utiliza la técnica de RCI no presenta una escalabilidad muy buena. Esto encuentra justificación en que al existir un proceso maestro que coordina al CG, sigue siendo necesario realizar una reducción global de una gran cantidad de datos. Dicha reducción presenta un cuello de botella esencial a este mecanismo de control. Además, si bien las otras operaciones que realiza el proceso central (como multiplicar un vector por un escalar, o sumar dos vectores) son de un orden menor que multiplicar una matriz por un vector, se está desperdiciando la posibilidad de paralelizarlas.

1.5. Resolución de sistemas lineales

En general, los métodos de resolución de sistemas lineales se pueden separar en dos grandes grupos: directos e iterativos [3]. Lo métodos directos se valen de la manipulación algebráica de las ecuaciones para calcular eventualmente el valor de cada componente de la incógnita x en un paso. La factorización LU es un ejemplo de esta categoría. Por otro lado, los métodos iterativos toman una estimación de la solución y la mejoran a lo largo de una serie de iteraciones, hasta que se considera que una aproximación es suficientemente buena. Cada grupo tiene sus propias ventajas y desventajas, y la elección de un método

de uno u otro depende en gran medida de las características del problema y de la matriz. Sin embargo, es posible enunciar caractetísticas que los diferencian más allá de la forma en la que llegan a una solución.

Los métodos directos llegan a la solución en un número fijo de pasos y son capaces de lograr precisiones en general más elevadas que los iterativos, siempre y cuando no empiece a pesar la acumulación de errores en las operaciones de punto flotante. Para problemas bien condicionados llegan fácilmente a la precisión de la máquina. También resultan muy útiles si es necesario resolver el mismo sistema varias veces, variando únicamente la parte no-homogénea. Es posible realizar una primera etapa de orden alto, que prepara el camino para una sucesión de segundas etapas más rápidas. En contrapartida, los requerimientos de memoria de los métodos directos suelen ser mucho más exigentes que los de los iterativos. Como ejemplo de esto se tiene que la factorización LU de una matriz rala resulta en matrices densas L y U.

Por otro lado, los métodos iterativos cuentan con una versatilidad inexistente en los directos. La precisión deseada es un parámetro de configuración del algoritmo, en general necesitando menos iteraciones para precisiones más bajas, lo cual redunda en un menor tiempo de ejecución. Entonces, por ejemplo, si la solución del sistema representa tensiones sobre distintos puntos de un circuito, y el instrumento de medición tiene una precisión de 4 dígitos, se puede ajustar el algoritmo para que provea dicha precisión y resolver el sistema en un tiempo mucho menor que utilizando un método directo. Además, cuentan con un gran potencial de optimización del uso de memoria (y también del tiempo de ejecución real, aunque no cambie el orden) si se aplican a matrices ralas, debido a que están basados principalmente en operaciones sencillas sobre matrices y vectores. Los métodos iterativos pierden la ganancia que obtienen sobre los directos cuando se resuelve varias veces el mismo sistema con distintos vectores del lado derecho b. Y además no suelen ser una buena elección si la matriz involucrada es densa, pues el tiempo de ejecución puede igualar (o superar) al de los métodos directos, aún obteniendo una precisión mucho menor.

A continuación se presenta una breve introducción al método de gradientes conjugados, que es el elegido para este tipo de problemas pues es el más utilizado cuando se trata de matrices simétricas. Si se desea una explicación más detallada de sus propiedades se puede recurrir a [11]. La siguiente sección está basada en [18], donde se presenta una justificación de por qué el método funciona.

1.5.1. Método de las direcciones conjugadas

El método de las direcciones conjugadas (DC) es un método iterativo con dos características particulares: si bien realiza iteraciones para obtener un resultado, en ausencia de errores de redondeo llega a la solución exacta en n pasos; además es en realidad un pseudo-algoritmo, pues es necesario determinar un aspecto que el método deja libre si se quiere hacer una implementación concreta. El hecho de que sea en esencia un método directo no lo proscribe del mundo de los métodos iterativos, ya que se lo suele usar para resolver sistemas tan grandes que no tiene sentido hacer las n iteraciones. En cambio, se le impone un criterio de corte para determinar que cierta aproximación es adecuada.

El DC se puede ver en Alg. 1. Una matriz A es *definida positiva* si $x^tAx > 0$ para todo $x \neq 0$, y dos vectores x e y se dicen *A-conjugados* si $x^tAy = 0$. Un conjunto de vectores $v_1, \ldots, v_n$ es A-conjugado si sus elementos lo son dos a dos y ${v_i}^tAv_i \neq 0$. El vector $r_x = b - Ax$ es conocido como el *residuo*, y el vector $e_x = \hat{x} - x$ es el *error* si $\hat{x}$ es la solución del sistema. El residuo se puede ver como el error transformado por A al mismo

espacio donde reside b, pues $Ae_x = r_x$. En Alg. 1 se ve claramente que este método sin ninguna modificación no se puede implementar, pues no establece una manera de obtener las n direcciones conjugadas necesarias d_i.

Alg. 1 Método de las direcciones conjugadas

Pre $A \in \mathbb{R}^{n\times n}$ definida positiva, $d_0, \ldots, d_{n-1}$ A-conjugados, $x_0, b \in \mathbb{R}^n$
Post $Ax_n = b$
  $r_0 \leftarrow b - Ax_0$
  for $i \leftarrow 0 \ldots n-1$ **do**
    $\alpha_i \leftarrow \dfrac{{d_i}^t r_i}{{d_i}^t A d_i}$
    $x_{i+1} \leftarrow x_i + \alpha_i d_i$
    $r_{i+1} \leftarrow r_i - \alpha_i A d_i$
  end for

Esencialmente el método parte de una estimación inicial x_0 provista por el usuario, y calcula su residuo r_0. Luego procede moviéndose hacia los puntos $x_1, \ldots, x_n$ en la dirección de búsqueda d_{i-1} en cada paso. Cada x_i está más cerca de la solución $\hat{x}$ que el anterior, como se verá a continuación. El valor r_i siempre representa el residuo de la estimación actual, actualizándose a partir del residuo anterior para disminuir los errores por cancelación.

Se puede demostrar que el DC calcula la solución del sistema en a lo sumo n pasos escribiendo el error inicial e_0 como combinación lineal de las direcciones de búsqueda[1]:

$$e_0 = \sum_{i=0}^{n-1} \gamma_i d_i.$$

Utilizando el hecho de que los d_i son A-conjugados, se puede expresar el coeficiente γ_j en función del error e_j. Se comienza premultiplicando la expresión anterior por $d_j A$ para obtener el valor de γ_j en función del error inicial:

$$\begin{aligned} {d_j}^t A e_0 &= \sum_i \gamma_i {d_j}^t A d_i = \gamma_j {d_j}^t A d_j \\ \gamma_j &= \frac{{d_j}^t A e_0}{{d_j}^t A d_j} \end{aligned} \tag{1.22}$$

El error en el paso j se puede escribir como el error inicial sumado a los desplazamientos que se hacen posteriormente, esto es:

$$e_j = e_0 + \sum_{i=0}^{j-1} \alpha_i d_i$$

Ahora nuevamente considerando que los d_i son A-conjugados, se le puede sumar cero a

[1] Se puede demostrar fácilmente que un conjunto de n vectores A-conjugados forman una base de $\mathbb{R}^n$

1.22 para obtener lo que se buscaba:

$$\begin{aligned}\gamma_j &= \frac{{d_j}^t A e_0}{{d_j}^t A d_j} + \frac{{d_j}^t A \left(\sum_{i=0}^{j-1} \alpha_i d_i\right)}{{d_j}^t A d_j} \\ &= \frac{{d_j}^t A \left(e_0 + \sum_{i=0}^{j-1} \alpha_i d_i\right)}{{d_j}^t A d_j} \\ &= \frac{{d_j}^t A e_j}{{d_j}^t A d_j}\end{aligned}$$

Sabiendo que $Ae_j = r_j$, se tiene que la cantidad que se avanza en la dirección d_i según el Alg. 1 es:

$$\alpha_i = -\gamma_i \tag{1.23}$$

Es decir, en cada paso se cancela la coordenada del error inicial correspondiente a la dirección utilizada en ese paso. Es claro que después de n pasos todas las coordenadas se habrán cancelado con lo que el error e_n será cero.

Una propiedad que será de importancia en la siguiente sección es que el residuo es siempre ortogonal a todas las direcciones de búsqueda anteriores. Como se acaba de demostrar, el error en el paso j tiene cero en las primeras $j-1$ coordenadas de su expresión en la base de direcciones A-conjugadas. O sea

$$e_j = \sum_{k=j}^{n-1} \gamma_k d_k \tag{1.24}$$

Multiplicando a izquierda a 1.24 por ${d_i}^t A$ con $i < j$, y de nuevo usando que las d_i son A-conjugadas, se tiene que

$${d_i}^t r_j = {d_i}^t A e_j = \sum_{k=j}^{n-1} \gamma_k {d_i}^t A d_k = 0 \qquad \text{si } i < j \tag{1.25}$$

Es importante resaltar que en la práctica no es el caso que se use aritmética exacta, con lo cual debido a errores de redondeo durante la ejecución del algoritmo el error final será distinto de cero aún si se hicieran las n iteraciones.

1.5.2. Método del gradiente conjugado

El método del gradiente conjugado (CG) establece una manera específica de construir las direcciones de búsqueda. Para hacerlo utiliza la versión A-conjugada del conocido proceso Gram-Schmidt de ortogonalización de bases [17]. Este algoritmo toma un conjunto de vectores linealmente independientes y construye iterativamente otro conjunto de vectores A-conjugados.

El algoritmo de conjugación Gram-Schmidt se presenta en Alg. 2. En principio, para una elección cualquiera de los vectores $u_0, \ldots, u_{n-1}$, este algoritmo requiere que se recuerden todas las direcciones calculadas previamente, pues éstas se utilizan para calcular la siguiente dirección conjugada. Esta característica lo hace bastante inadecuado para aplicarlo al cálculo de las direcciones de búsqueda del CG, pues los requerimientos de memoria cuando se trabaja con miles de componentes pueden ser muy elevados.

Alg. 2 Proceso de Gram-Schmidt para A-conjugación

Pre $u_0, \ldots, u_{n-1}$ linealmente independientes
Post $d_0, \ldots, d_{n-1}$ son A-conjugados
  $d_0 \leftarrow u_0$
  for $i \leftarrow 1 \ldots n-1$ **do**
    for $j \leftarrow 0 \ldots i-1$ **do**
      $\beta_{ij} \leftarrow -\dfrac{{u_i}^t A d_j}{{d_j}^t A d_j}$
    end for
    $d_i \leftarrow u_i + \sum_{j=0}^{i-1} \beta_{ij} d_j$
  end for

A continuación se verá que este problema desaparece si se usan los residuos r_i como los vectores que Gram-Schmidt va a conjugar (ie, los u_i). Esta elección hace que la mayor parte de los coeficientes β_{ij} valgan cero. Para ver por qué esto es así primero es necesario demostrar que cada residuo también es ortogonal a los u_j usados para obtener las direcciones de búsqueda anteriores. Tomando la expresión de d_i que se usa Alg. 2 y aplicando 1.25, se tiene

$$\begin{aligned} {d_i}^t r_j &= {u_i}^t r_j + \sum_{k=0}^{i-1} \beta_{ik} {d_k}^t r_j, \\ 0 &= {u_i}^t r_j \qquad \text{ya que } i < j. \end{aligned} \tag{1.26}$$

Para poder aplicar Gram-Schmidt a los residuos hace falta verificar que estos son linealmente independientes. Como primera dirección de búsqueda se usa r_0, tal como dicta el paso de inicialización de Gram-Schmidt. Como se vio en la sección anterior (ecuación 1.25), el residuo es ortogonal a toda dirección de búsqueda previa, por ende, $r_1 \perp r_0$ ya que $d_0 = r_0$. Esto permite hacer un paso más de conjugación usando $u_1 = r_1$, calcular d_1 para avanzar hacia x_2 y obtener r_2. Ahora, gracias a 1.26, también $r_2 \perp r_1$ y $r_2 \perp r_0$. Es decir, $\{r_0, r_1, r_2\}$ es linealmente independiente. Nuevamente esto permite usar $u_2 = r_2$ para construir la tercer dirección de búsqueda. Siguiendo de esta manera, usando la propiedad 1.26 en cada paso, es posible aplicar Gram-Schmidt a todos los residuos.

Ahora se está en condiciones de ver la ventaja de utilizar los residuos como vectores a conjugar. Recordando que $r_{j+1} = r_j - \alpha_j A d_j$, se puede obtener una expresión del numerador de los β_{ij}:

$$\begin{aligned} {r_i}^t r_{j+1} &= {r_i}^t r_j - \alpha_j {r_i}^t A d_j \\ {r_i}^t A d_j &= \frac{1}{\alpha_j} \left({r_i}^t r_j - {r_i}^t r_{j+1} \right) \end{aligned}$$

Como cada residuo es ortogonal a los u_j anteriores y el CG utiliza como u_j a los residuos, se tiene que

$${u_i}^t A r_j = \begin{cases} -\frac{{r_i}^t r_i}{\alpha_{i-1}} & \text{si } j+1 = i \\ 0 & \text{si } j+1 < i \end{cases}$$

Es decir que $\beta_{ij} = 0$ si $j+1 < i$, lo cual permite calcular cada dirección de búsqueda nueva solo en función de la dirección anterior, eliminando la necesidad de almacenar todas

las direcciones previas. Dado que β_{ij} solo es distinto de cero cuando $i = j + 1$, se puede descartar uno de los subíndices y definir simplemente β_j. Expandiendo el valor de α_j se obtiene el valor de β_j:

$$\beta_j = -\frac{{r_{j+1}}^t r_{j+1}}{{r_j}^t r_j}$$

Juntando el DC con el proceso de Gram-Schmidt aplicado sobre los residuos y usando el resultado que se acaba de obtener sobre los β_{ij}, en Alg. 3 se muestra el método del gradiente conjugado.

Alg. 3 Método de gradientes conjugados

Pre $A \in \mathbb{R}^{n \times n}$ simétrica definida positiva, $x_0, b \in \mathbb{R}^n$
Post $Ax_n = b$
  $r_0 \leftarrow b - Ax_0$
  $d_0 \leftarrow r_0$
  for $i \leftarrow 0 \dots n-1$ **do**
    $\alpha_i \leftarrow \dfrac{{r_i}^t r_i}{{d_i}^t A d_i}$
    $x_{i+1} \leftarrow x_i + \alpha_i d_i$
    $r_{i+1} \leftarrow r_i - \alpha_i A d_i$
    $\beta_i \leftarrow \dfrac{{r_{i+1}}^t r_{i+1}}{{r_i}^t r_i}$
    $d_{i+1} \leftarrow r_{i+1} + \beta_i r_i$
  end for

1.6. Precondicionadores

Si bien el CG calcula la solución al sistema en ausencia de errores de redondeo, en la práctica se le impone un criterio de corte que en cada iteración evalúa qué tan buena es la aproximación actual. Mediante una tolerancia especificada por el usuario, cuando la solución se juzga adecuada, el método se detiene. Por este motivo es importante analizar la velocidad de convergencia del CG.

Un análisis de convergencia de un método iterativo en general establece una cota superior para el error en cada paso. Para el caso del CG, en [18] se demuestra que el error e_i satisface la siguiente desigualdad:

$$\|e_i\|_A \leq 2 \left(\frac{\sqrt{\kappa} - 1}{\sqrt{\kappa} + 1} \right)^i \|e_0\|_A, \tag{1.27}$$

donde $\|\cdot\|_A$ es la norma inducida por la matriz A, definida como $\|x\|_A = \sqrt{x^t A x}$, y κ es el *número de condición* de A, que se define como el cociente entre el mayor y el menor autovalor de A. La figura 1.12 se muestra la base de la cota en función de κ. Se puede apreciar que cuanto más grande es el número de condición de la matriz, más lenta es la convergencia lograda por el CG.

La cota 1.27 apunta en la dirección de buscar una manera de disminuir el número de condición de la matriz A antes de aplicar el CG. Una forma de lograr esto es utilizando una matriz M simétrica y definida positiva que tenga la propiedad de que $M^{-1}A$ cuenta

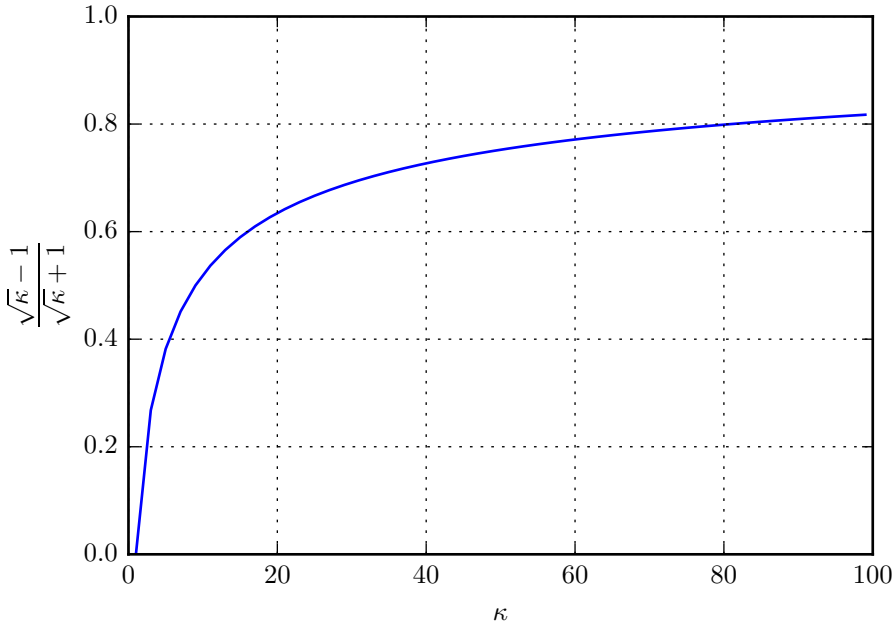

Fig. 1.12: Base de la cota 1.27 en función del número de condición de la matriz

con un número de condición menor que el de A. Luego se usa el hecho de que la solución al sistema

$$M^{-1}Ax = M^{-1}b$$

es la misma que la de $Ax = b$, y se resuelve el primero esperando que sean necesarias menos iteraciones para lograr la misma aproximación. Hacer esto se llama *precondicionar* el sistema, y la matriz M se dice que es un *precondicionador*.

El problema de este enfoque reside en que la matriz $M^{-1}A$ puede no ser simétrica ni definida positiva, aún cuando M y A lo sean. Esto se resuelve ya que al ser M simétrica y definida positiva, se puede hallar una matriz E tal que $EE^t = M$. Es sencillo demostrar que $M^{-1}A$ y $E^{-1}AE^{-t}$ tienen los mismos autovalores, y esta última matriz también es simétrica y definida positiva. Además, $Ax = b$ es equivalente a

$$E^{-1}AE^{-t}\hat{x} = E^{-1}b$$
$$E^t x = \hat{x}$$

Por lo tanto se puede aplicar el CG a $E^{-1}AE^{-t}$ obteniendo los mismos beneficios en términos de cantidad de iteraciones que se obtendrían al usar M, y luego resolver un sistema más para hallar x.

Es claro que aplicar el CG ingenuamente sobre la matriz $E^{-1}AE^{-t}$ es computacionalmente muy costoso, pues no solo requiere calcular la matriz E, sino que ahora hay que resolver dos sistemas. Por suerte haciendo algunas sustituciones de variables se consigue una expresión del CG precondicionado que solo depende de saber multiplicar un vector por M^{-1}. Este algoritmo se presenta en Alg. 4 y se denomina *método del gradiente conjugado precondicionado* (PCG), y es el que se implementa en este trabajo.

La elección de un precondicionador adecuado debe sopesar el tiempo necesario para construirlo y para aplicarlo en cada iteración. Es importante notar que al igual que con la matriz A, no hace falta contar con la matriz M, sino solo con una forma de calcular

Alg. 4 Método de gradientes conjugados precondicionado

Pre $A, M \in \mathbb{R}^{n \times n}$ simétricas definidas positivas, $x_0, b \in \mathbb{R}^n$
Post $Ax_n = b$
$r_0 \leftarrow b - Ax_0$
$d_0 \leftarrow r_0$
$z_0 \leftarrow M^{-1} r_0$
for $i \leftarrow 0 \dots n-1$ **do**
$\alpha_i \leftarrow \frac{{r_i}^t u_i}{{d_i}^t A d_i}$
$x_{i+1} \leftarrow x_i + \alpha_i d_i$
$r_{i+1} \leftarrow r_i - \alpha_i A d_i$
$z_{i+1} \leftarrow M^{-1} r_{i+1}$
$\beta_i \leftarrow \frac{{r_{i+1}}^t z_{i+1}}{{r_i}^t z_i}$
$d_{i+1} \leftarrow z_{i+1} + \beta_i r_i$
end for

su efecto sobre un vector. En la práctica está generalmente aceptado que para sistemas relativamente grandes, el PCG debe utilizarse en lugar del CG.

1.7. Objetivos

El objetivo de este trabajo es reemplazar el *solver* iterativo presentado brevemente en la sección 1.4 y en detalle en [14], por uno que presente mejores características de escalabilidad. Se parte de la hipótesis de que la falta de escalabilidad de la implementación previa encuentra su origen en la reducción global necesaria para transportar las aproximaciones sucesivas entre el proceso coordinador y el resto. Es claro que cierta comunicación es necesaria para la resolución de un sistema de las características del producido por el FEM con la técnica del subestructurado, pero la centralización de la totalidad del vector solución dista de ser esencial. El origen de los datos compartidos se encuentra en que subdominios adyacentes comparten nodos, es por esto que la comunicación necesaria debería poder reducirse, al menos en parte, a procesos correspondientes a subdominios vecinos.

En este trabajo se intenta mejorar la escalabilidad modificando la implementación del *solver* iterativo. Se plantea dejar de lado la implementación incluida en la biblioteca MKL y construir en su lugar un *solver* diseñado desde el comienzo para la ejecución paralela. Las operaciones básicas que se deben proveer de manera distribuida son la multiplicación matriz-vector y el producto punto entre dos vectores, siendo la hipótesis que si estas escalan, también lo hará el *solver*. La idea fundamental para implementar esas operaciones de manera realmente distribuida es comunicar solo lo que sea necesario. En el caso del FEM con la técnica del subestructurado, por ejemplo, es evidente que los nodos externos son compartidos por algunos subdominios, no por todos. Contando con una manera de expresar esto, se puede pensar en implementaciones de las operaciones en las que solo se comuniquen subdominios vecinos entre sí.

La ventaja de una tal implementación radicaría en que la cantidad de subdominios vecinos permanece constante con el aumento de los subdominios totales. Se espera lograr así una escalabilidad que solo dependa de la capacidad de la red para realizar comunicaciones simultáneas.

También se resolvería el otro cuello de botella, que es la no-homogeneidad de los requerimientos de memoria de los procesos involucrados. Mediante el modo de comunicación planteado, ningún proceso tendría un rol especial, y todos deberían ser capaces de contener solamente la parte de la matriz del problema y de los vectores que le corresponde a su subdominio.

2. DESARROLLO

2.1. Diseño

Para matener el objetivo de modularidad y flexibilidad planteado en [16], el diseño del *solver* implementado en este trabajo también intentó seguir las prácticas de la programación orientada a objetos. Para tal fin, el primer paso fue identificar las distintas abstracciones involucradas, creando interfaces que encapsulen a cada una de ellas. Las siguientes secciones presentan los resultados.

2.1.1. Encapsulamiento del algoritmo

Al momento de comenzar el trabajo la base de código contaba con un único *solver*: el que utiliza la biblioteca MKL de Intel, que implementa la versión iterativa de la idea presentada en 1.4. Este *solver* se encontraba implementado como una función. Ya que iba a ser necesario comparar la nueva implementación con la anterior, era un requerimiento que ambos solvers se pudieran intercambiar con facilidad. Para tal fin, se creó la interfaz `IterativeSolver` que se ve en la figura 2.1.

Esta interfaz representa cualquier implementación de un *solver* iterativo, ya sea secuencial o paralela. Una tal implementación debe proveer la operación `solve(A,b)` para resolver el sistema lineal representado por sus parámetros. Como se verá más adelante, el parámetro `A` representa la matriz del sistema como una forma de multiplicarla por un vector. El método `getCurrentX()` debe devolver el actual candidato a solución. Este valor puede no ser la solución del sistema, no solo por la naturaleza inherentemente inexacta de un *solver* iterativo, sino también porque una implementación puede proveer medios para limitar la cantidad de iteraciones o para avanzar de a una iteración. Estas opciones resultan de gran utilidad a la hora de buscar errores o analizar la convergencia del algoritmo. El método `iterationsCount()` debe devolver la cantidad de iteraciones realizadas hasta el momento.

Dado que la idea de la interfaz `IterativeSolver` es abstraer los métodos iterativos de resolución de sistemas lineales, cualquier aspecto que sea particular a un método concreto debe ser especificado a través de métodos en la clase que la implementa. En el caso del PCG, el precondicionador a utilizar es un ejemplo de un tal aspecto. Se optó por indicarlo al momento de construir la instancia particular de *solver*, por medio del constructor.

La interfaz es paramétrica sobre el tipo `T`, que corresponde a la clase concreta utilizada

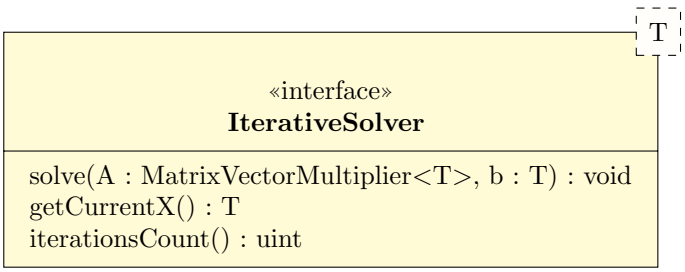

Fig. 2.1: Interfaz `IterativeSolver`.

para representar las matrices y vectores. La razón principal del uso de un *template* en esta interfaz es permitir a una implementación particular la creación de vectores auxiliares que generalmente son necesarios en un *solver* iterativo. Los detalles de esta decisión se pueden ver en 2.1.4.

El nuevo *solver* se creó como una implementación de la interfaz `IterativeSolver`, y la versión anterior se modificó para que también se ajuste a ella. Esta manera de organizar el código también va a favor del objetivo de extensibilidad del framework. La implementación de un tercer *solver* sin la necesidad de modificar código preexistente, es evidencia de que ese objetivo fue alcanzado en cuanto a solvers se refiere.

2.1.2. Encapsulamiento de las operaciones

Un algoritmo iterativo necesita realizar un conjunto de operaciones usando sus matrices y sobre sus vectores de entrada. Muchas de estas son operaciones simples como multiplicar un vector por un escalar, o sumar dos vectores. Casi todas ellas ya se encontraban implementadas en las clases `Matrix` y `SparseMatrix` del framework. Otras, como el producto punto entre dos vectores y el producto de una matriz por un vector, son sensiblemente más complejas y admiten una mayor variedad de maneras de realizarlas. Por ejemplo, un producto matriz-vector puede llevarse acabo sin calcular nunca explícitamente la matriz, como es el caso del producto contra el complemento Schur de un bloque de una matriz. Por otro lado, ambas operaciones pueden tener implementaciones muy distintas dependiendo de la manera en la que se encuentren distribuidos sus datos de entrada.

Para permitir explorar todas estas posibilidades de manera más o menos sencilla, se crearon las interfaces `MatrixVectorMultiplier` y `DotProduct`. Una implementación de la primera solo debe proveer el método `times(v)`, que reemplace al vector `v` por el producto entre la matriz representada por la instancia y ese mismo vector. En Alg. 5 se muestra la implementación del multiplicador por complemento Schur de la ecuación 1.19 que se utiliza en este trabajo. Contando con una instancia de un tal multiplicador, una invocación a su método `times(v)` dejará en `v` el resultado de multiplicar el complemento Schur del bloque D por `v`. La manera indirecta de expresar la multiplicación intenta resaltar el hecho de que no es necesario calcular la matriz del complemento ni invertir el bloque D.

Alg. 5 Implementación de la multiplicación por el complemento Schur

Class SchurMultiplier **implements** MatrixVectorMultiplier
  function CONSTRUCTOR(A, B, C, D)
    Store parameters
  end function
  function TIMES(v)
    $\hat{v} \leftarrow Cv$
    $\hat{v} \leftarrow x$ tal que $Dx = \hat{v}$
    $\hat{v} \leftarrow B\hat{v}$
    $v \leftarrow Av - \hat{v}$
  end function
End

El caso de la interfaz `DotProduct` es similar. Se debe proveer la operación `calculate(v,w)` que devuelve el valor del producto punto entre los vectores `v` y `w`. Como se puede ver en

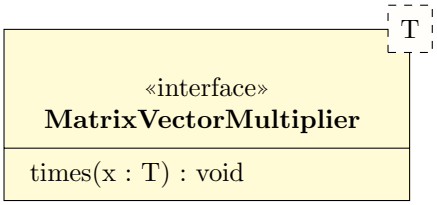

Fig. 2.2: Interfaz `MatrixVectorMultiplier`.

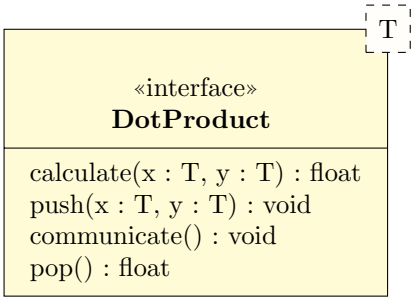

Fig. 2.3: Interfaz `DotProduct`

la figura 2.3, esta interfaz cuenta con tres métodos adicionales para realizar series de operaciones de manera asincrónica. La API soporta asincronismo usando como modelo una pila: el método `push(v,w)` apila el par de vectores `v` y `w` que representa un producto punto que se quiere realizar. En las llamadas a `push()` se puede realizar la parte sincrónica del cálculo de haber alguna. Luego el método `communicate()` realiza la parte que se puede hacer asincrónicamente, conceptualmente convirtiendo los pares (v, w) que se encuentran en la pila en los productos $\langle v, w \rangle$. En esencia, la idea es desacoplar el cálculo de la obtención del resultado. Es por esto que el método `communicate()` debe ser no bloqueante, para dar la oportunidad de hacer otros cálculos en simultáneo con el del producto punto inmediatamente después de llamarlo. Por último, el método `pop()` tiene como objetivo desapilar un resultado de la pila, debiendo bloquearse en caso de que aún no se encuentren disponibles. En Cód. 1 se muestra cómo se usa la API de asincronismo.

En la sección 2.3.2 se encuentra la razón de la necesidad del asincronismo para productos punto.

Cód. 1 Ejemplo de utilización de la API de asincronismo de `DotProduct`

```
1 /* ... */
2 dot_product.push(v1, v2)
3 dot_product.push(w1, w2)
4 dot_product.communicate()
5 /* ... Cálculos que no dependen del resultado de ⟨v1, v2⟩ ni de ⟨w1, w2⟩ */
6 res_w = dot_product.pop()        /* res_w ← ⟨v1, v2⟩ */
7 res_v = dot_product.pop()        /* res_v ← ⟨w1, w2⟩ */
8 /* ... */
```

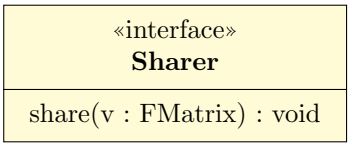

Fig. 2.4: Interfaz `Sharer`.

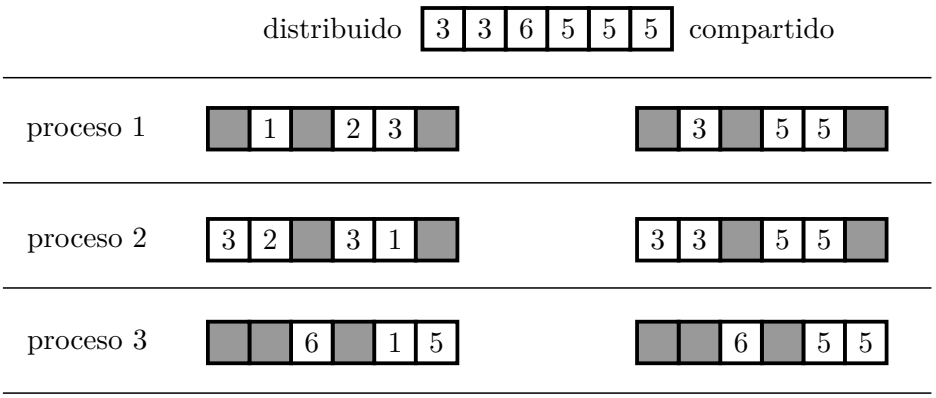

Fig. 2.5: Diferencia entre un vector distribuido y uno compartido en 3 procesos.

2.1.3. Encapsulamiento de la comunicación

Cuando se toma la decisión de implementar un algoritmo de manera que pueda aprovechar el uso de múltiples máquinas al mismo tiempo, inevitablemente aparece la necesidad de mover datos entre los distintos procesos involucrados. Esta necesidad es inherente al cómputo paralelo, y es un derivado de uno de los costos que conlleva el aumento del poder de cómputo: la complejidad, tanto de aprehensión como de implementación de la tarea una vez paralelizada.

Hasta el momento, las armas con las que se cuenta para combatir esta complejidad son las mismas que para el código secuencial: la abstracción y la "separación de intereses" (del inglés *separation of concerns*). Es por esto que para favorecer la modularidad y abstraer el hecho de que el código que se lee en realidad debe cooperar con otras copias de sí mismo, se creó la interfaz `Sharer` (Fig. 2.4). La responsabilidad de un objeto que implementa esta interfaz es garantizar que todos los procesos cuenten con la misma información para realizar sus cálculos. Para precisar qué quiere decir esto es necesario distinguir entre dos maneras de separar datos en distintos procesos: los datos pueden estar compartidos o distribuidos. En la figura 2.5 se puede ver un ejemplo de un vector de seis elementos, en un caso distribuido en tres procesos y en otro compartido por ellos.

Como es usual, a cada proceso se le asigna un subconjunto de los datos con los cuales va a trabajar. Estos subconjuntos no necesariamente tienen intersección nula, y existen dos posibilidades para los elementos que se encuentran en varios procesos. Si todos los procesos guardan el mismo valor, se dirá que los datos están *compartidos*. En cambio si cada proceso tiene su propio aporte al valor final de los elementos repetidos, se dirá que los datos se encuentran *distribuidos*.

Cabe destacar que la propiedad de los datos de pertenecer a una u otra categoría no es invariante a lo largo de toda la ejecución de un programa. Por ejemplo, en este trabajo los datos son matrices y vectores, y la distribución es tal que para obtener el valor real de un

determinado elemento se debe sumar el aporte que tiene cada proceso. Para simplificar el análisis, se puede suponer que todos los procesos tienen todos los elementos, caso extremo pero válido. Bajo este supuesto, se tiene que una matriz A se encuentra distribuida en varios procesos de manera que cada uno tiene una matriz A_i y

$$A = \sum_i A_i.$$

Se puede observar que multiplicar la matriz A por un vector compartido v distribuye el resultado. Esto es así dado que cada proceso realizará el producto $A_i v$, que ahora es el vector Av distribuido, pues

$$Av = \left(\sum_i A_i \right) v = \sum_i A_i v.$$

Si se desea seguir operando con el resultado de Av es necesario convertirlo nuevamente en un vector compartido, ya que el producto entre dos vectores distribuidos no es de ninguna utilidad global ni localmente.

Dicho todo esto resulta muy sencillo expresar la responsabilidad del método `share(v)`: tomar un vector v distribuido y convertirlo en uno compartido.

2.1.4. Adaptación para prescindir del complemento Schur

Como ya se mencionó en 1.4, la versión del framework de la que se partió para realizar este trabajo combina la técnica de subestructurado con el complemento Schur. El uso de la técnica de subestructurado implica que el código crea el sistema separado en bloques correspondientes a pares de nodos que son los dos externos, los dos internos, o uno interno y uno externo (ver 1.3.4). Asimismo, la utilización del complemento Schur para resolver el sistema requiere de dos pasos: primero se resuelven los nodos externos y luego los internos.

En principio, el uso del complemento Schur no es esencial para este trabajo dado que no presenta ninguna ventaja en cuanto a cantidad de comunicación necesaria entre procesos. Sin embargo, la resolución de un sistema lineal mediante el complemento Schur puede considerarse como una forma de precondicionarlo, potencialmente disminuyendo la cantidad de iteraciones necesarias para obtener una buena aproximación. Por este motivo, la implementación del nuevo *solver* debía posibilitar el uso opcional del complemento Schur, y para esto se reutilizó el código de creación del sistema en bloques. Con estas decisiones y requerimientos, el nuevo *solver* tenía que ser capaz de trabajar con:

- Matrices y vectores sin estructura, para la resolución del primer paso cuando se utiliza el complemento Schur
- Matrices y vectores separados en bloques, para la resolución del sistema entero de una sola vez

Resultaba claro que la realización de dos implementaciones separadas no era una opción, pues hubiera implicado el mantenimiento de dos algortimos funcionalmente idénticos. Además, contando con las interfaces que abstraen las operaciones que realiza el CG sobre sus datos, la doble implementación parecía innecesaria. El problema que persistía era el hecho de que el CG necesita crear vectores auxiliares para hacer su trabajo, y estos deben

ser compatibles con los vectores x y b del sistema $Ax = b$. La solución a este problema se encontró en los *templates* de C++.

La clase abstracta `IterativeSolver` se hizo depender de un tipo T, que representa el tipo de los vectores con los que va a trabajar el algoritmo. Este tipo necesita tener implementadas algunas operaciones básicas, como ser multiplicarlo por un escalar y sumarle un múltiplo de otro elemento del mismo tipo. Pero este requerimiento no es esencial, pues lo ideal sería encapsular estar operaciones en clases como se hizo con el producto punto y el producto matriz-vector. El uso más importante del *template* T es la creación de los vectores auxiliares que necesita el CG para operar.

La introducción de un tipo paramétrico posibilitó el uso del mismo código fuente para resolver el sistema precondicionado con Schur y sin él. Para la resolución mediante Schur sencillamente se usa `Matrix` como T, y se provee un multiplicador matriz-vector que sabe multiplicar por el complemento Schur, como se mencionó en 2.1.2. La instancia de `DotProduct` provista solo debe saber realizar el producto punto entre dos objetos `Matrix`.

Para la resolución sin el complemento Schur fue necesaria la implementación de una clase que represente un vector partido en dos: externa e interna. Esta clase, llamada `SplittedVector`, es una clase muy sencilla que solo sirve como una suerte de envoltorio alrededor de dos instancias de `Matrix` (la parte interna y la parte externa), delegando a ambas cada operación que se realice sobre él. Estas son solo las operaciones simples como sumar, multiplicar por un escalar, etc., pues el producto punto y el producto matriz-vector se implementan como una clase aparte, como ya se describió anteriormente.

Una de las grandes desventajas de la solución mediante *templates* es que las dos clases resultantes de instanciar a T como `Matrix` y como `SplittedVector` no son polimórficas. Esto revela que la doble implementación no desapareció, sino que se cargó sobre el código del usuario, el cual debe ser distinto si se quiere resolver usando el complemento Schur o no.

2.2. Implementación de funciones comunes

En esta sección se describen las implementaciones concretas que se hicieron de las interfaces correspondientes a las operaciones vistas en la sección previa. Tales operaciones son comunes a ambas implementaciones de `IterativeSolver` que se realizaron.

2.2.1. Patrón de comunicación

Una de las ideas centrales a este trabajo es la reducción de la cantidad de comunicación necesaria entre procesos. Una manera de lograr esto es utilizar el hecho de que los nodos externos de un subdominio solo son compartidos con los subdominios vecinos a él, y no con todos. Para poner a prueba esta idea, se realizó una implementación de `Sharer` a modo de prueba de concepto. Se parte de la suposición de que todo proceso tiene los elementos de las matrices y vectores que pertenecen a su subdominio, y no más. Desde esa base, basta definir cómo se debe manejar cada nodo que está en más de un subdominio (los que están en uno solo, en los bordes del dominio total, no es necesario comunicarlos).

La clase `SendRecvSharer` implementa un patrón de comunicación en el que cada nodo perteneciente a más de un subdominio se envía al subdominio de menor *id*. En esencia, el objetivo es un *all reduce* sobre los nodos compartidos usando la operación suma. Esta operación se implementó en dos etapas: una de reducción y una de broadcast. Para ambas etapas se utilizaron las funciones punto a punto `MPI_Isend()` y `MPI_Irecv()` junto con

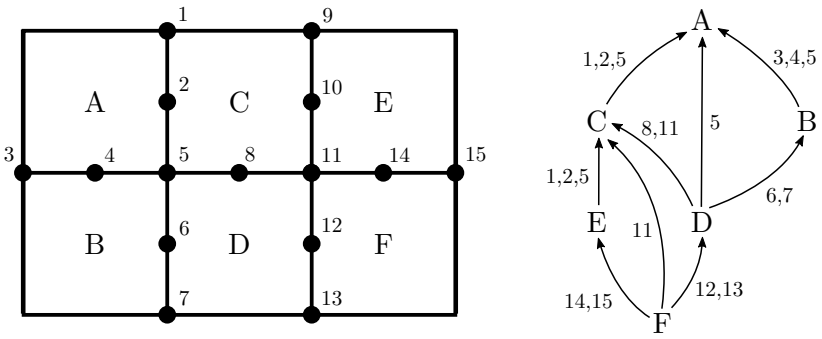

Fig. 2.6: Patrón de comunicación (el orden de los subdominios es alfabético)

dos mapas, *send* y *recv*, que definen el árbol de comunicación para la etapa de reducción. Durante el broadcast se usan los mismos mapas pero con sus roles invertidos.

Los mapas tienen la siguiente semántica: si send_A es el mapa send del subdominio A, entonces $\text{send}_A(B)$ es la lista de nodos que A le tiene que enviar a B. Análogamente, $\text{recv}_A(B)$ es la lista de nodos que A tiene que recibir de B. La figura 2.6 muestra un posible dominio dividido en seis subdominios. Marcados se encuentran los nodos externos, y el grafo dirigido representa los mapas send: un eje $S_1 \xrightarrow{N} S_2$ significa que $\text{send}_{S_1}(S_2) = N$, donde N es una lista de nodos.

Estos mapas se crean al momento de parsear el subdominio, utilizando el formato preexistente de especificación del dominio y subdominios. La implementación actual tiene un orden $O(sn_e^2)$ en cada proceso, donde s es la cantidad de subdominios y n_e la cantidad de nodos externos por subdominio. Si e_s es la cantidad de elementos por subdominio, y consideramos elementos de 8 nodos, entonces $8\sqrt{e_s}$ aproxima a n_e. Con lo cual, la creación de los mapas *send* y *recv* tiene un orden $O(se_s) = O(e)$, con e la cantidad total de elementos.

Este orden agrega una gran componente secuencial a todo el proceso de resolución de la PDE, pues e es independiente de la cantidad de procesos que se utilicen. Dado que el orden de la creación de los mapas está fuertemente ligado al formato del archivo de especificación del dominio y subdominios, la reducción y/o distribución del mismo queda fuera del alcance de este trabajo.

Este patrón se implementó para probar la correctitud del *solver*, manteniendo la característica de escalabilidad que se quería lograr (sin comunicaciones globales). Como tal debe considerarse como una versión preliminar plausible de ser optimizada ampliamente, pero que el momento resultó eficiente para los objetivos del trabajo.

2.2.2. Producto matriz-vector

La interfaz que representa un producto matriz-vector es `MatrixVectorMultiplier` (ver 2.1.2). Se implementaron dos clases concretas de esta interfaz: `BlockMultiplier` y `SchurMultiplier`. La primera sabe multiplicar una matriz que se encuentra separada en 4 bloques por una instancia de `SplittedVector`, y es la que permite utilizar el nuevo *solver* de manera directa, sin mediar el complemento Schur (ver 2.1.4). El código de esta clase es muy sencillo, siendo que solo debe calcular el lado derecho de

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \end{Bmatrix} = \begin{Bmatrix} Ax_1 + Bx_2 \\ Cx_1 + Dx_2 \end{Bmatrix}.$$

En esta ecuación, los vectores x_1 y x_2 representan la parte externa e interna, respectivamente, de un `SplittedVector`. Los bloques A, B, C y D son las matrices del sistema del FEM subestructurado K_{ee},K_{ei}, K_{ie} y K_{ii} respectivamente. Dado que el bloque K_{ee} es distribuido, se debe utilizar un `Sharer` para convertir Ax_1 en un vector compartido antes de realizar la suma (ver 2.1.3).

Por otro lado, la clase `SchurMultiplier` es la que permite utilizar el complemento Schur al resolver el sistema mediante el nuevo *solver*, y lo que hace es multiplicar a un vector por el complemento Schur del último bloque suministrado en el momento de su construcción. A una nueva instancia se le proveen 4 matrices, representando los 4 bloques de una matriz más grande, y cada llamado a `times(v)` deja en `v` el producto del complemento Schur de uno de ellos por `v` según Alg. 5. En el CG implementado se utiliza el complemento del bloque K_{ii}y, dado que la matriz K_{Sch} de la ecuación 1.21 se encuentra distribuida, el resultado debe ser compartido antes de finalizar.

2.2.3. Producto punto

Implementar el producto punto de forma distribuida presentó un pequeño desafío. La operación debe hacerse con cuidado pues todos los vectores con los que se trabaja se encuentran compartidos. Como ya se ha dicho anteriormente, las operaciones se pueden realizar localmente y luego compartir los resultados siempre y cuando uno de los dos operandos sea compartido y el otro no. Por ejemplo, en el caso del producto punto, si v es un vector compartido y w es distribuido, con aportes w_1 y w_2, se tiene que

$$v^t w = v^t(w_1 + w_2) = v^t w_1 + v^t w_2$$

Con lo cual cada proceso podría hacer su producto local $v^t w_i$, y luego la suma de todos los resultados sería el producto punto global.

En el caso de dos vectores compartidos, la operación no resulta tan clara. La figuran 2.7 ilustra la situación si no se tiene cuidado cuando se hacen las operaciones locales. Se cuenta con dos procesos, A y B, que comparten vectores con la siguiente topología: A tiene asignados prefijos de cierto tamaño y B sufijos, de manera tal que la componente del medio de los vectores se encuentra en ambos. Como se ve en la figura, si se hacen los productos locales y luego se suma el resultado, el valor ab está sumado dos veces, por lo tanto el resultado de la suma está excedido del producto punto en ab.

La solución que se ideó para este problema fue hacer el producto entre componentes compartidas en uno solo de los procesos que las compartan. Para esto se aprovecharon los mismos mapas utilizados para compartir un vector descriptos en 2.2.1, pues estos mapas ya identifican un proceso único entre un grupo de procesos que comparten una componente: el que la recibe de todos los demás. Tomando a ese como el único que realiza la multiplicación de esa componente, queda resuelto el problema de la repetición de sumandos.

La implementación del producto punto distribuido como se acaba de describir también se hizo a modo de prueba de concepto. Aún así se tuvo el cuidado de priorizar la utilización de las funciones de la biblioteca MKL. Esencialmente, el cálculo consta de dos pasos: durante el primero cada proceso calcula su aporte al producto global, y luego cada uno de esos aportes se hace llegar a todos los otros procesos. Si se recuerda de la sección 2.1.4, todas las operaciones, en particular el producto punto, deben tener una versión para trabajar con instancias de `Matrix` y otra con instancias de `SplittedVector`. Sin embargo, los dos pasos necesarios para calcular el producto global son comunes al manejo de ambos tipos

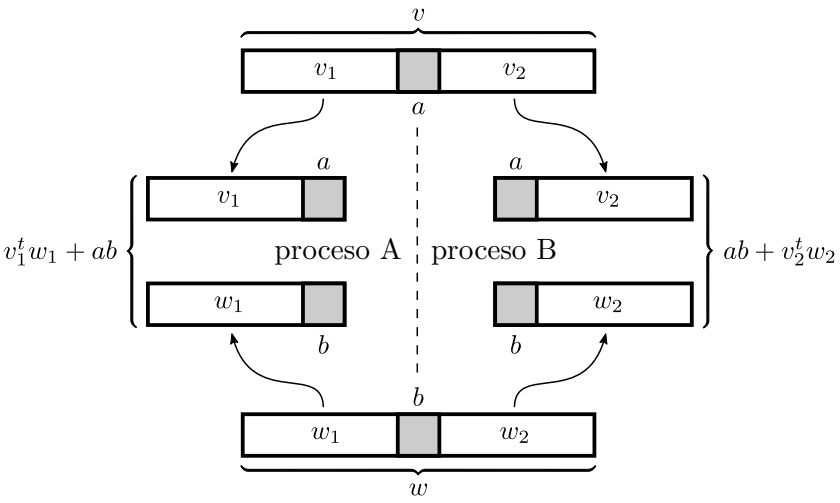

Fig. 2.7: Producto punto distribuido incorrecto

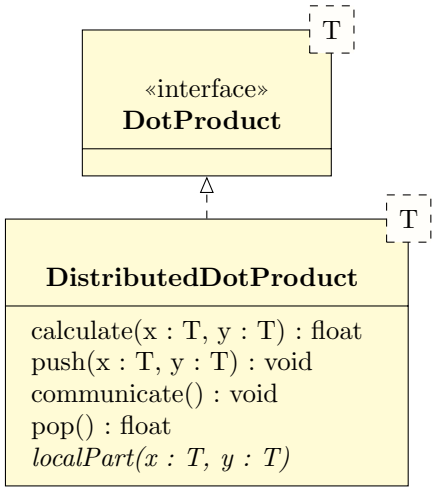

Fig. 2.8: Clase abstracta de `DotProduct` con soporte para paralelismo.

de vectores. Por esta razón se creó la clase abstracta `DistributedDotProduct` como una implementación de `DotProduct` que encapsula el segundo paso y deja libre a implementar el primero (Fig. 2.8).

El método abstracto `localPart` de la clase `DistributedDotProduct` es el que las clases que hereden de ella deben implementar para llevar adelante el cálculo del aporte local al product. El método `calculate()` es muy sencillo, pues solo debe obtener el resultado de `localPart()` y luego se vale de MPI para hacer la reducción global. Los métodos que soportan la API de asincronismo también son comunes a cualquier implementación, pues la única diferencia con el método `calculate()` es que los dos pasos están separados en dos funciones. Entonces `push()` invoca a `localPart()` para obtener la parte local e introduce en una pila un par con sus dos argumentos, x e y. Luego el método `communicate()` es el encargado de hacer la reducción global, solo que esta vez utilizando la versión asincrónica de la operación.

Ahora sí se pueden introducir las implementaciones específicas para vectores de tipo

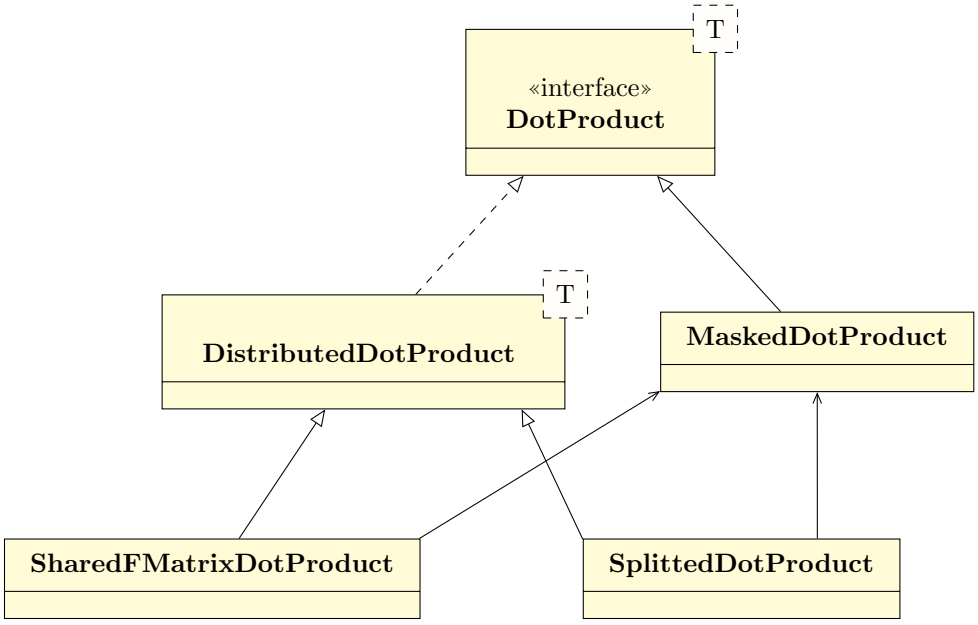

Fig. 2.9: Jerarquía de clases entera de los productos punto.

`Matrix` y de tipo `SplittedVector`. Aquí también se encuentran similitudes y la posibilidad de reutilización de código. Los vectores de tipo `Matrix` son vectores enteramente compartidos, en el sentido de que no se puede asumir a priori que alguna parte no se encuentra también en otro proceso. Por otro lado, un `SplittedVector` consta de dos partes, una de las cuales es puramente local pero la otra es análoga a un vector de tipo `Matrix`. Tanto la parte compartida de un `SplittedVector` como un vector de tipo `Matrix` se debe tratar como se describió anteriormente, teniendo cuidado de no repetir cálculos.

La clase `MaskedDotProduct` es la responsable de calcular el aporte al producto punto de las componentes asociadas a cierto proceso. Como su nombre lo indica, esta clase realiza el producto punto entre dos instancias de `Matrix`, previamente aplicándoles la misma máscara a cada una con el objetivo de ignorar ciertas componentes: las componentes que ya están siendo consideradas en otro proceso. Como se dijo antes, para esto se le provee el mapa send al momento de creación y se calcula en base a él una máscara, marcando con ceros las posiciones que no se deben tener en cuenta. Un llamado a `calculate()` utiliza la función `vdPackM()` de la biblioteca MKL, cuyo objetivo es copiar de un lugar a otro los elementos indicados por la máscara. Finalmente, se usa nuevamente la biblioteca MKL para hacer el producto punto entre las componentes copiadas.

Una vez que se cuenta con la clase `MaskedDotProduct`, la implementación de las clases que multiplican `Matrix` y `SplittedVector` resulta trivial. Ambas clases mantienen una referencia a una instancia de `MaskedDotProduct`, la cual es utilizada para calcular el aporte de sus partes compartidas. En el caso de `Matrix`, la parte compartida es la instancia misma, y en el caso de `SplittedVector` la parte compartida es la correspondiente a los nodos externos. Al valor calculado por `MaskedDotProduct` se le suma el producto punto usual de las componentes asociadas a nodos internos.

2.2.4. Utilización extensiva de bibliotecas matemáticas

Al momento de comenzar este trabajo, algunas de las operaciones sobre matrices se encontraban implementadas de manera ingenua mediante ciclos. Estas operaciones se reemplazaron por llamadas a funciones de la biblioteca MKL de Intel, presuponiendo que éstas se encuentran fuertemente optimizadas. Se observó una reducción de aproximadamente un 30 % en el tiempo por iteración luego de la introducción de las funciones de la biblioteca. Este resultado refuerza la idea de que se deben usar funciones especializadas siempre que sea posible.

Las operaciones que se reimplementaron usando funciones de la MKL fueron:

- la suma y resta de matrices,
- la multiplicación de un vector por un escalar,
- la copia de una matriz,
- el producto interno (local) entre dos vectores.

Para implementar de manera eficiente el CG fue necesario crear un par de métodos para sumar a un vector un múltiplo de otro y viceversa, sin hacer una copia. Estas dos operaciones se implementaron directamente con las funciones de la MKL. El producto punto enmascarado descripto en 2.2.3 también utiliza las funciones de la biblioteca.

2.3. Implementación de algoritmos de PCG

Se realizaron dos implementaciones del PCG, una que utiliza directamente Alg. 4, llamada *solver distribuido*, y otra que implementa las optimizaciones sugeridas en [10] denominada *solver distribuido pipelined*. En esta sección se presenta el código de ambas versiones junto con algunos detalles de su implementación.

2.3.1. Solver distribuido

La primera implementación que se hizo de PCG es equivalente al algoritmo 4 con la adición del criterio de corte. Dado que el manejo de la comunicación de los datos se encuentra encapsulado dentro de las operaciones, el código final no es excesivamente distinto del pseudocódigo.

En Cód. 2 se muestra el preámbulo de la implementación del PCG. Aquí se ve cómo se utiliza el tipo paramétrico T para crear los vectores auxiliares del tipo correcto. El método `copyShape()` se creó para hacer un poco más clara la lectura del código, y sólo reinicializa la instancia para que tenga el mismo tamaño que el parámetro.

El fragmento que se ve en Cód. 3 sólo inicializa las variables auxiliares a los valores necesarios previo comienzo de las iteraciones. El caso trivial en el que $b = 0$ se salva aquí pues el residuo inicial es necesario de todas formas para el criterio de corte. La variable de instancia `M_inv` representa el precondicionador, y debe ser provisto al momento de crear el *solver* como una instancia de `MatrixVectorMultiplier`.

El ciclo principal del PCG se puede ver en Cód. 4. En la línea 8 se utiliza la función `addMultiple`, que le suma a la instancia un múltiplo de un vector; y en la línea 16, la función `multiplyAndAdd` multiplica a la instancia por un escalar y luego le suma un vector. Se puede observar en la línea 9 el producto punto extra (y por ende, la reducción global) necesario para establecer la condición de corte.

Cód. 2 Creación de vectores auxiliares

```
template <typename T>
void SimplePCG<T>::operator()(MatrixVectorMultiplier<T> &A, T &b)
{
    T r("r"); r.copyShape(b);
    T d("d"); d.copyShape(b);
    T Ad("A.d"); Ad.copyShape(b);
    T z("z"); z.copyShape(b);
    this->current_x.copyShape(b);
    /* ... */
```

Cód. 3 Inicialización del PCG

```
    /* ... */
    double initial_residual = this->dot_product(b, b);
    if (initial_residual == 0) return; /* b = 0, la solución es x = 0 */
    r = b;                             /* r = b - Ax_0, asumiendo x_0 = 0 */
    z = r;
    this->M_inv.times(z);
    double rz = this->dot_product(r, z);
    d = z;
    /* ... */
```

2.3.2. Solver distribuido pipelined

Para demostrar la flexibilidad del encapsulamiento de las operaciones se implementó otra versión del PCG. Se trata del *PCG pipelined* presentado en [10], y es un algoritmo matemáticamente equivalente al PCG presentado en Alg. 4, pero que cuenta con una propiedad que lo hace atractivo para implementarlo de forma paralela.

Al igual que el algoritmo original, la versión pipelined necesita calcular dos productos punto en cada iteración, pero con la diferencia de que en éste esos dos productos se encuentran consecutivos. Esto permite ocultar la latencia incurrida por uno de ellos en la del otro, pues la reducción global de ambos se puede hacer en una sola vez. Además, el paso inmediatamente posterior al cálculo de los dos productos es una multiplicación matriz-vector que no depende de ellos. Con lo cual, es posible solapar el tiempo durante el cual se está haciendo la reducción global con el tiempo de la parte local de la multiplicación siguiente.

La forma mediante la que se logra agrupar los productos punto es descubriendo nuevas relaciones de recurrencia que se desprenden de las originales. Esto tiene el costo de requerir vectores auxiliares extra para almacenar los términos de las nuevas relaciones. A modo de ejemplo, el algoritmo original utiliza la siguiente relación:

$$r_{i+1} = r_i - \alpha_i \underbrace{Ad_i}_{s_i} \tag{2.1}$$

Tomando $s_i = Ap_i$ como muestra la ecuación y multiplicándola por M^{-1}, se obtiene

$$M^{-1}r_{i+1} = M^{-1}r_i - \alpha_i \underbrace{M^{-1}s_i}_{q_i}, \tag{2.2}$$

o equivalentemente dado que ya se tenía la relación $u_i = M^{-1}r_i$:

$$u_{i+1} = u_i - \alpha_i q_i. \tag{2.3}$$

Cód. 4 Ciclo del PCG

```
1     /* ... */
2     for (int i = 0; i < this->max_iterations; i++) {
3         Ad = d;
4         A.times(Ad);
5         double dAd = this->dot_product(d, Ad);
6         double a = rz / dAd;
7         this->current_x.addMultiple(a, d);
8         r.addMultiple(-a, Ad);              /* r ← r − αAd */
9         double rr = this->dot_product(r, r);
10        if (rr <= this->tolerance * initial_residual)
11            break;
12        z = r;
13        this->M_inv.times(z);
14        double new_ru = this->dot_product(r, z);
15        double b = new_rz / rz;
16        d.multiplyAndAdd(b, z);             /* d ← z + βd */
17        rz = new_rz;
18    }
19 }
```

Esta ecuación establece una relación de recurrencia para el residuo precondicionado que antes no existía, y es la que se puede ver en la línea 29 de Cód. 7. Por otro lado, las ecuaciones 2.1 y 2.2 parecen necesitar un producto matriz-vector adicional para el cálculo de s_i y q_i respectivamente. Sin embargo, esos productos se pueden eliminar usando las relaciones de recurrencia que ya se tenían para d_i para hallar otras que permitan mantener q_i y s_i actualizados. Tales relaciones son las implementadas por las líneas 24 y 25 de Cód. 7.

En Cód. 5 se ve la primer parte del código de la versión pipelined del PCG. Esta versión del algoritmo necesita muchos más vectores auxiliares para mantener los valores que se introducen con el objetivo de evitar nuevos productos matriz-vector.

Cód. 5 Creación de vectores auxiliares

```
1  template <typename T>
2  void PipelinedPCG<T>::operator()(MatrixVectorMultiplier<T> &A, T &b)
3  {
4      T r("r"); r.copyShape(b);
5      T u("u"); u.copyShape(b);
6      T w("w"); w.copyShape(b);
7      T m("m"); m.copyShape(b);
8      T n("n"); n.copyShape(b);
9      T z("z"); z.copyShape(b);
10     T q("q"); q.copyShape(b);
11     T s("s"); s.copyShape(b);
12     T p("p"); p.copyShape(b);
13     double alpha          = 0
14     double beta           = 0;
15     double previous_gamma = 0;
16     double current_gamma;
17     double delta;
18     this->current_x.copyShape(b);
```

En Cód. 6 se muestra la inicialización de los vectores, y en Cód. 7 el ciclo principal. Lo importante a destacar aquí es el uso de la API de asincronismo del producto punto. La

llamada a `communicate()` es no-bloqueante, y sólo inicia la reducción global y mientras tanto se pueden hacer las operaciones de las líneas 6 a 9.

Cód. 6 Inicialización de variables auxiliares, versión pipelined del PCG

```
1    double initial_residual = this->dot_product(b,b);
2    if (initial_residual == 0) return;    /* b = 0, la solución es x = 0 */
3    r = b;                                /* r = b − Ax0, asumiendo x0 = 0 */
4    u = r;
5    this->M_inv.times(u);
6    w = u;
7    A.times(w);
```

2.3.3. Criterio de corte

Dado que el enfoque de este trabajo se encuentra en realizar la primer implementación de un algoritmo de CG propio, no se llevó acabo una investigación profunda acerca de los distintos criterios de corte posibles y sus impactos en la cantidad de iteraciones ni la precisión de la solución. Es por esto que el criterio de corte utilizado es el mismo que el que usa por defecto la biblioteca MKL de Intel. Siendo r_i el residuo actualizado por el PCG, este consiste en detener las iteraciones cuando se satisface la siguiente condición:

$$\|r_i\| \leq tol \cdot \|b\| \tag{2.4}$$

El valor de *tol* que utiliza Intel por defecto es 10^{-6}. Todas las corridas que se presentan aquí utilizan esa tolerancia. Este criterio garantiza la siguiente cota para el error e_i en cada iteración:

$$\|e_i\| \leq tol \cdot \|b\| \cdot \|A^{-1}\|$$

Utilizar el valor de $\|r_i\|$ para determinar si seguir iterando o no tiene sentido pues equivale a la norma del residuo $Ax_i - b$: si este valor es pequeño, x_i es una buena aproximación a la solución del sistema. Sin embargo, se debe tener en cuenta que los errores de redondeo introducidos por el uso de aritmética finita terminan separando a estos dos valores. Para analizar la importancia del error introducido en el cálculo de $\|r_i\|$ se instrumentó el código para que también calcule este valor pero mediante la expresión directa $Ax_i - b$.

Como se puede observar en la figura 2.10, el residuo calculado por el algoritmo decrece hasta un valor muy inferior al residuo real (en la figura se muestra un detalle de las primeras 700 iteraciones, pero $\|r_i\|$ decrece constantemente hasta un orden de 10^{-300} y luego se hace 0). El estancamiento del residuo real solo se puede deber a que x_i ya no cambia más. La razón por la que x_i deja de cambiar pero r_i no es que $||r_i||$ forma una sucesión que tiende a 0, mientras que $||x_i||$ tiende a $||A^{-1}b|| \neq 0$ si $b \neq 0$. En algún punto de ambas sucesiones, éstas comienzan a variar solo en decimales que están más allá de la precisión de los números de punto flotante utilizados por la máquina. La diferencia en el punto a partir del cual esto comienza a suceder se debe a que se pueden representar muchos más números alrededor del 0 que alrededor de un número no nulo.

La figura 2.11 muestra el error relativo entre el valor de $\|r_i\|$ y $\|Ax_i - b\|$. Se puede apreciar que, como era esperable, el error relativo aumenta a medida que se itera debido a los errores de redondeo. Lo que se hace en general para intentar apaciguar este problema es calcular explícitamente $Ax_i - b$ cada cierta cantidad de iteraciones, lo que se conoce como *restart* del residuo. En este trabajo no se consideró necesario tomar dicho recaudo.

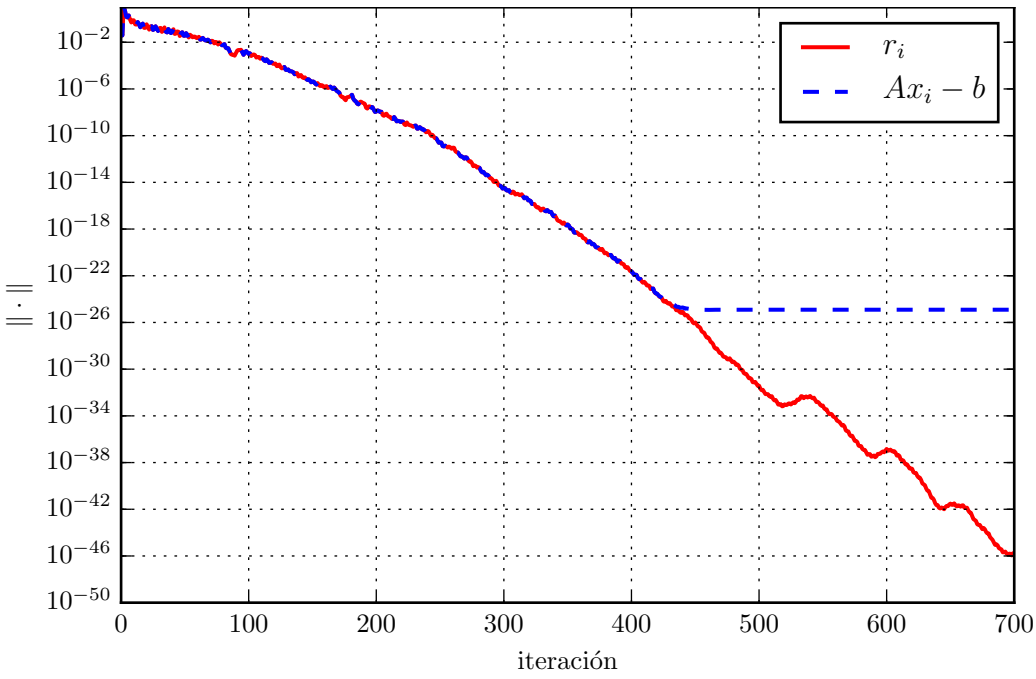

Fig. 2.10: El residuo real comparado con el residuo calculado por el CG

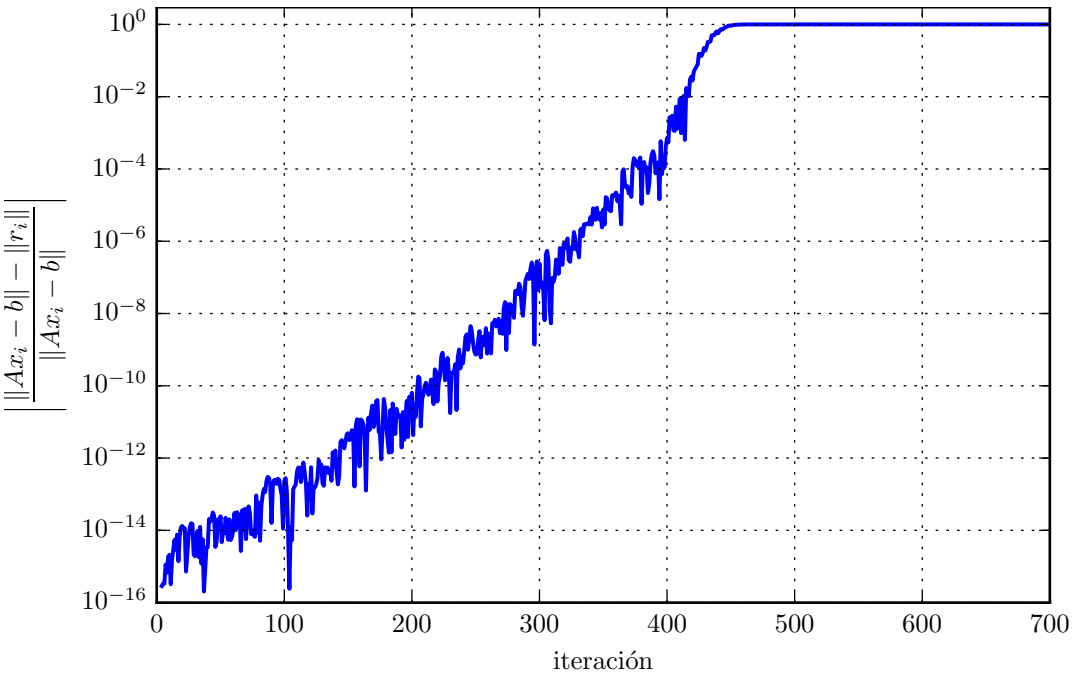

Fig. 2.11: Error relativo de r_i respecto del residuo real $Ax_i - b$

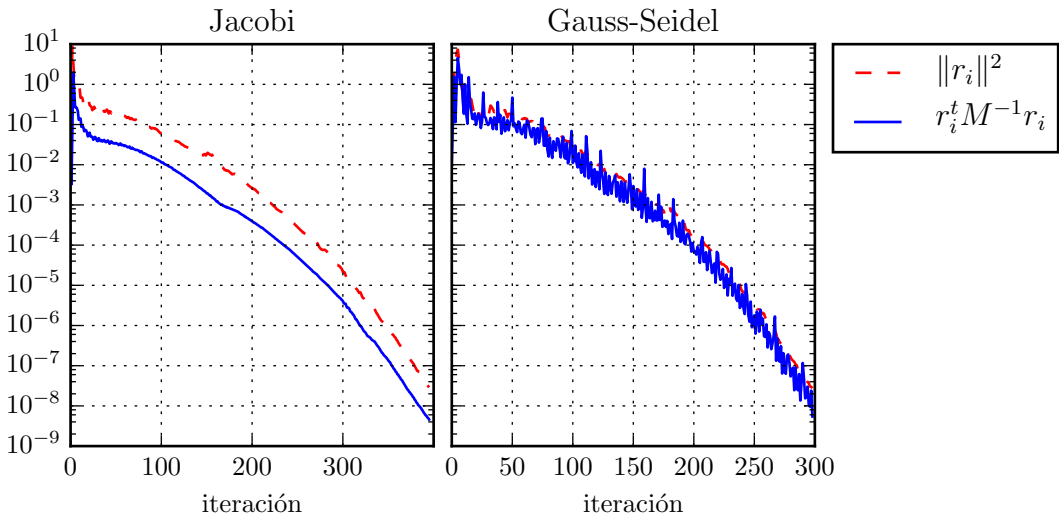

Fig. 2.12: Comparación entre la norma canónica del residuo y su norma según el precondicionador M.

En [1] se presenta una breve reseña sobre distintos criterios de corte posibles y las garantías que cada uno da sobre el error de la solución. El texto advierte acerca de lo inadecuado de este criterio si la matriz A se encuentra muy mal condicionada y x_0 se encuentra muy cerca del espacio nulo de A. En ese caso, el criterio puede resultar muy difícil de satisfacer, iterando como consecuencia hasta la cantidad máxima de iteraciones.

Es importante destacar que para utilizar este criterio de corte es necesario contar con $\|r_i\|$. Este valor se puede calcular mediante el producto $r_i \cdot r_i$, pero esta operación implica una reducción global. En el CG sin precondicionar (Alg. 3), la norma del residuo se encuentra disponible en cada iteración pues es usada por el algoritmo para calcular los coeficientes α y β. En cambio en el PCG (Alg. 4), el valor que se utiliza es $r_i^t M^{-1} r_i$. Es decir que usando el criterio de corte mencionado se agrega una reducción global adicional, en principio no esencial al método. Se cree que esta reducción se puede evitar utilizando el valor de $r_i^t M^{-1} r_i$ en lugar de $\|r_i\|$ en la ecuación 2.4. La figura 2.12 muestra el progreso de los dos valores para los precondicionadores implementados en este trabajo. Se puede observar que ambos decrecen de la misma forma, solo siendo necesario ajustar la tolerancia *tol* para lograr cortar en el mismo punto.

2.4. Implementación de precondicionadores

2.4.1. Precondicionador de Jacobi

Uno de los dos precondicionadores que se implementaron fue el de Jacobi [1], que es el más simple de todos. Consiste en tomar a M como la diagonal de la matriz del sistema que se quiere resolver:

$$M_{ij} = \begin{cases} A_{ii} & \text{si } i = j \\ 0 & \text{si no} \end{cases}$$

La implementación de este precondicionador es muy simple y requiere sólo almacenar un vector para todos los elementos de la diagonal. Los valores que se guardan son los recíprocos de los originales, para evitar hacer la división en cada iteración. Dado que se

están extrayendo componentes de la matriz K_{ee}, que se encuentra distribuida, es preciso compartir la diagonal. Esta comunicación es necesaria solo cuando se construye el precondicionador, pues luego durante las iteraciones la operación de aplicar el precondicionador se hace entre dos datos compartidos, con lo cual cada proceso que tiene una misma componente hace la misma cuenta.

Cód. 7 Ciclo del PCG pipelined

```
    for (int i = 0; i < this->max_iterations; i++) {
        this->dot_product.push(r, u);
        this->dot_product.push(w, u);
        this->dot_product.communicate();

        m = w;
        this->M_inv.times(m);
        n = m;
        A.times(n);

        delta = this->dot_product.pop();          /* w · u */
        current_gamma = this->dot_product.pop();  /* r · u */

        if (i != 0) {
            beta = current_gamma / previous_gamma;
            alpha = current_gamma / (delta - beta * current_gamma / alpha);
        } else {
            beta = 0;
            alpha = current_gamma / delta;
        }
        previous_gamma = current_gamma;

        z.multiplyAndAdd(beta, n);
        q.multiplyAndAdd(beta, m);
        s.multiplyAndAdd(beta, w);
        p.multiplyAndAdd(beta, u);
        this->current_x.addMultiple(alpha, p);
        r.addMultiple(-alpha, s);
        u.addMultiple(-alpha, q);
        w.addMultiple(-alpha, z);

        double rr = this->dot_product(r,r);
        if (rr <= this->tolerance * initial_residual)
            break;
    }
}
```

2.4.2. Precondicionador de Gauss-Seidel

El precondicionador de Gauss-Seidel se define como:

$$M = (D + L)D^{-1}(D + U), \tag{2.5}$$

donde D, L y U son la diagonal, y los triángulos inferior y superior respectivamente [1]. La inversa de la matriz M está dada por:

$$M^{-1} = (D + U)^{-1}D(D + L)^{-1}. \tag{2.6}$$

En el caso de la matriz K, en $D+U$ están involucrados el triángulo superior de K_{ee} y de K_{ii}, y la matriz K_{ei}. Sucede lo mismo con $D+L$, pero con los triángulos inferiores y la matriz K_{ie}.

Las matrices K_{ei} y K_{ie} son construidas por el código que construye el sistema, pero los triángulos inferiores y superiores de K_{ee} y K_{ii}deben ser calculados. Puesto que se trata de matrices gigantescas, extraerlos ingenuamente mediante un ciclo resulta extremadamente costoso en tiempo.

Para el caso de los triángulos superiores se utilizó el hecho de que K_{ee} y K_{ii} son simétricas. Las instancias de la clase `SparseMatrix` cuentan con un flag que indica si la matriz que representan es simétrica o no, y en caso de serlo solo almacenan el triángulo superior. Cualquier acceso al triángulo inferior es debidamente transpuesto y referido al superior, previa verificación del flag de simetría. Aprovechando este truco se puede extraer muy rápidamente el triángulo superior de una matriz simplemente copiando todas las estructuras internas a otra matriz que tiene su flag de simetría apagado. Cabe notar que esta operación deja en la nueva matriz la diagonal.

La extracción del triángulo inferior se realizó usando una función de la biblioteca MKL. Si se tiene una matriz rala almacenada en *Compressed Row Storage*, su transpuesta se puede obtener cambiando la representación a *Compressed Column Storage* sin cambiar la manera de acceder a las estructuras. La función `mkl_dcsrcsc()` se encarga de hacer el cambio.

En Cód. 8 se muestra el código que aplica este precondicionador. La secuencia es multiplicar a v por $(D+L)^{-1}$, luego por D y finalmente por $(D+U)^{-1}$. La multiplicación por D se hace exactamente igual que en el precondicionador de Jacobi, y el primer y último paso son análogos. Para evitar calcular y almacenar la inversa de $D+L$, se resuelve el sistema $(D+L)x = v$. Suponiendo que $A^{\downarrow}$ es el triángulo inferior de la matriz A, se tiene que el sistema que se quiere resolver es:

$$\begin{bmatrix} K_{ee}^{\downarrow} & 0 \\ K_{ie} & K_{ii}^{\downarrow} \end{bmatrix} \begin{Bmatrix} x_e \\ x_i \end{Bmatrix} = \begin{Bmatrix} v_e \\ v_i \end{Bmatrix}$$

Haciendo el producto se llega que se deben resolver estos dos sistemas secuencialmente:

$$K_{ee}^{\downarrow} x_e = v_e \tag{2.7}$$

$$K_{ii}^{\downarrow} x_i = v_i - K_{ie} x_e \tag{2.8}$$

Exactamente el mismo proceso pero usando los triángulos superiores de K_{ee} y K_{ii} junto con K_{ei} se hace para la multiplicación por $(D+U)^{-1}$.

Luego de hacer la multiplicación localmente es necesario compartir el resultado, pues en la cuenta está involucrada K_{ee}, que es una matriz distribuida. En este trabajo se utilizó una versión inexacta de este precondicionador en la cual las operaciones de las ecuaciones 2.7 y 2.8 se realizaron subdominialmente de manera de evitar comunicar las matrices distribuidas. Luego de aplicar el precondicionador de forma local se compartió el resultado, tratando al precondicionador como una matriz distribuida.

2.4.3. Complemento Schur

Si bien el complemento Schur no puede expresarse como un precondicionador en el sentido presentado en 1.6, sí se lo puede considerar una técnica de precondicionamiento.

Cód. 8 Aplicación del precondicionador de Gauss-Seidel

```
1  void GaussSeidel::times(Splitted &v)
2  {
3      DLKee->solve(v.ve.elements);       /* v_e ← K_ee^↓ v_e, como en 2,7 */
4      v.vi -= Kie * v.ve;
5      DLKii->solve(v.vi.elements);       /* v_i ← K_ii^↓ (v_i − K_ie v_e), como en 2,8 */
6
7      /* v ← D(D + L)^{-1} v */
8      vdMul(diagonal.ve.length(), v.ve.elements, diagonal.ve.elements, v.ve.elements);
9      vdMul(diagonal.vi.length(), v.vi.elements, diagonal.vi.elements, v.vi.elements);
10
11     /* Se repite lo mismo que antes pero con (D + U)^{-1} */
12     DUKii->solve(v.vi.elements);
13     v.ve -= Kei * v.vi;
14     DUKee->solve(v.ve.elements);
15
16     sharer.share(v.ve);
17 }
```

En efecto, se puede demostrar que el complemento Schur de un bloque tiene un número de condición menor que la matriz entera, con lo cual la cantidad de iteraciones necesarias será menor [22].

La aplicación de este precondicionador es la misma que la presentada en 1.4, excepto que la solución a la ecuación 1.19 se calcula utilizando el nuevo *solver*. Esto quiere decir que en realidad al resolver aplicando esta técnica de precondicionamiento se está tomando $M = id$ y $A = K_{Sch}$. El código que implementa el multiplicador por K_{Sch} se presenta en Cód. 9, y su función es dejar en v el valor de $K_{ee} - K_{ei}K_{ii}^{-1}K_{ie}v$. Al igual que en la implementación del precondicionador de Gauss-Seidel, el bloque K_{ii} nunca se invierte, sino que se resuelve el sistema $K_{ii}y = x$ para calcular $K_{ii}^{-1}x$. Dado que en la cuenta está involucrada la matriz K_{ee}, que es distribuida, es necesario compartir el resultado.

Cód. 9 Multiplicación de un vector por el complemento Schur de K_{ii}

```
1  void SchurMultiplier::times(FMatrix &v)
2  {
3          FMatrix v1(Kei.cols, 1, "v1");
4          FMatrix v2(Kee.rows, 1, "v2");
5
6          v1 = Kie * v;
7          Kii.solve(v1.elements);      /* v_1 ← K_ii^{-1} K_ie v */
8          v2 = Kei * v1;               /* v_2 ← K_ei K_ii^{-1} K_ie v */
9
10         v = Kee * v - v2;            /* v ← K_Sch v */
11
12         sharer.share(v);
13 }
```

Esta técnica de precondicionamiento requiere que el vector b de $Ax = b$ proporcionado al *solver* iterativo sea el lado derecho de la ecuación 1.19. Este vector es calculado por cada proceso al momento de construir el sistema y debe ser compartido, pues involucra las componentes correspondientes a los nodos externos del vector F de la ecuación 1.12. Adicionalmente, una vez que el *solver* iterativo ha calculado la solución a la ecuación 1.19, se debe resolver la ecuación 1.20 para obtener el valor de la aproximación sobre los nodos internos de cada subdominio. En este paso se utiliza el *solver* directo provisto por la biblioteca PARDISO, y es totalmente paralelo pues todos los datos son locales.

3. RESULTADOS

Como se planteó en la sección 1.7, el objetivo de este trabajo fue la construcción de un *solver* iterativo basado en el método de gradientes conjugados, para resolver los sistemas lineales que resultan de la implementación del FEM en el framework mencionado en la sección 1.4. La principal condición que debe cumplir este *solver* es presentar buenas características de escalabilidad. Además de la implementación del método tal y como se lo encuentra en los libros de texto, se realizó una implementación extra para poner a prueba las ideas que se presentan en [10]. Este capítulo expone los resultados obtenidos con ambos solvers y con los precondicionadores implementados.

Para evaluar las características de las diferentes implementaciones se analizó utilizar diferentes técnicas para instrumentar el código. El primer intento de instrumentación del código se hizo utilizando el framework TAU[1]. El mismo cuenta con varios mecanismos para introducir el código que realiza la instrumentación. Se puede modificar automáticamente el código fuente, cambiar el código dinámicamente en tiempo de ejecución, o introducir manualmente llamadas a macros para crear, disparar y detener timers. Para programas distribuidos, cuenta con la posibilidad de generar lo que se denominan "trazas", las cuales almacenan información sumamente detallada de los mensajes MPI enviados y recibidos por cada proceso.

El uso de TAU se descartó tempranamente por resultar excesivamente complicado de configurar. Por ejemplo, la instrumentación automática del código fuente no se pudo hacer funcionar, con lo cual hubo que recurrir a la instrumentación manual del mismo. Además la característica que más atraía del framework, que era la generación de trazas, demostró no ser de mucha utilidad pues no se pudieron hallar buenos visualizadores de la información obtenida.

Se optó por hacer una pequeña implementación *ad hoc* de clases que permitan la creación, control y persistencia de timers directamente en el código fuente. Esto eliminó la complejidad de tener que configurar TAU, manteniendo el mismo conjunto de funcionalidades que eran de interés. Lo que se quería lograr era medir intervalos de tiempo, y para esto se utilizó la función POSIX `clock_gettime`[2] que provee acceso a timers del sistema. La precisión necesaria no era mucha, pues los tiempos a medir estaban en el orden de los milisegundos.

Todos los tiempos se midieron por proceso, y a la hora de analizarlos se consideró el tiempo máximo. Para corridas de menos de unas centenas de segundos se hicieron varias ejecuciones idénticas, y de entre los máximos calculados para cada corrida se tomó el mínimo como el tiempo final para usar en el análisis.

3.1. Nociones básicas

Cuando se habla de escalabilidad, en general se está haciendo referencia a la capacidad de un sistema para resolver problemas cada vez más grandes. En el contexto de computación de alto rendimiento, esa escalabilidad se logra utilizando más de un procesador para resolver el problema, repartiendo la carga en cada uno de ellos de manera que por separado

[1] http://www.cs.uoregon.edu/research/tau/home.php
[2] http://linux.die.net/man/3/clock_gettime

se les presente un problema de tamaño menor. En un mundo ideal, si cierto código puede resolver un problema de tamaño N, entonces k procesadores podrían resolver un problema de tamaño kN. En la práctica esto no es así, pues la adaptación de un algoritmo para que pueda ser ejecutado en múltiples procesadores viene con un costo extra. Operaciones que en la versión secuencial (un procesador) no eran necesarias aparecen en la versión paralela para coordinar todos los procesos. Esta coordinación suele implicar la comunicación de datos entre procesos, algo que puede resultar extremadamente costoso en términos de tiempo. Es por esto que a medida que se utilizan más procesadores, el tiempo ahorrado es cada vez menor. El mérito de un código paralelo es mayor cuanto su comportamiento más se aproxime al ideal.

Para evaluar la calidad de un determinado código o parte del mismo se utilizan dos maneras de medir su escalabilidad llamadas fuerte y débil [15]. La primera es la que se acaba de describir, y suele ser de mayor utilidad cuando se trata con problemas que son limitados por CPU. Con esto se quiere decir que la tarea principal del código es el cálculo, con lo cual lo más deseable es escalar fuertemente para reducir la carga de procesamiento sobre cada CPU. La escalabilidad débil implica mantener fijo el tamaño del problema *por procesador*, logrando que al aumentar la cantidad de procesos se resuelvan problemas cada vez más grandes. Este tipo de escalabilidad tiene sentido para problemas que son limitados por memoria.

El *speedup* es una medidad de cuánto se acerca determinado código al mundo ideal donde la comunicación entre procesos es gratis o no existe. Si T_1 es el tiempo que le lleva resolver el problema a un solo procesador y T_n el que le lleva a n procesadores, el speedup se define como:

$$S_n = \frac{T_1}{T_n}$$

Cuando se analiza escalabilidad fuerte, el speedup ideal es $S_n = n$, significando que el mismo problema se resuelve n veces más rápido. En cuanto a escalabilidad débil, el speedup ideal es $S_n = 1$, pues implica que se logra resolver un problema de tamaño kn en el mismo tiempo que demora resolver uno de tamaño k.

3.2. Contexto

La resolución de un sistema lineal es solo uno de los pasos que comprenden la aproximación de la solución a una ecuación diferencial mediante el FEM. Previamente es necesario crear las matrices que representan al sistema que se va a resolver, en función de la malla elegida para aproximar al dominio, el tipo de elemento y las características del problema. Dada la localidad del soporte de las funciones ϕ asociadas a cada nodo, estas matrices tienen una gran cantidad de ceros, con lo cual se representan utilizando estructuras que permiten almacenar solo las posiciones no nulas. La distribución de estas posiciones se desprende directamente de la topología de la malla, con lo cual es posible crear los arreglos necesarios antes de comenzar con los cálculos. Esto además tiene sentido pues agregar una posición no nula nueva a una matriz rala es una operación muy costosa. El otro paso que siempre estará presente cuando se utilice el FEM es, naturalmente, el llenado de esas matrices ralas en función de las matrices elementales (ver 1.3.3).

En el problema modelo basta con realizar estos dos pasos una única vez, pues la topología de la malla se encuentra fija y la matriz K también. Sin embargo, este no es siempre el caso. En problemas no lineales se utiliza un método iterativo que en cada paso

Subdominios	Creación estructuras	Creación de send/recv	Ensamble
1	3,2 seg	3,6 μseg	4,8 seg
4	3,4 seg	11,3 mseg	5,0 seg
16	4,0 seg	66,5 mseg	6,1 seg
64	3,8 seg	247,7 mseg	6,1 seg
256	3,9 seg	1,2 seg	6,3 seg
576	3,6 seg	2,1 seg	4,6 seg

Tab. 3.1: Tiempos por fase según cantidad de procesos.

resuelve un sistema lineal distinto para ir aproximando así la solución. Un ejemplo de esto puede ser la ecuación del calor donde la difusividad térmica del material depende de la temperatura. Aquí la matriz depende de la solución, con lo cual en cada paso es necesario calcular nuevamente el sistema lineal en función de la solución obtenida en el paso anterior. Otro ejemplo de una situación análoga es en problemas transitorios, siendo preciso recalcular la matriz en cada paso temporal. También puede darse el caso de ambas situaciones combinadas: un problema transitorio o temporal que en cada paso de tiempo implique resolver un problema no lineal. En todas estas situaciones, la malla podría mantenerse constante por lo que la estructura de valores no nulos de la matriz se mantendría idéntica, y para llenar las matrices no sería necesario realizar la operación de definir la estructura. Solo habría que poner en cero todos los valores y comenzar de nuevo con el llenado de los valores. Esto también es así si la malla se deforma manteniendo su topología, es decir las relaciones de vecindad entre elementos.

La topología de la malla puede modificarse en medio de la aplicación del FEM. En problemas de fractura, cierto paso de tiempo puede determinar que las tensiones ejercidas sobre algunos nodos exceden las que el material es capaz de soportar. La malla se modifica separando cada uno de esos nodos en dos, efectivamente modificando la estructura y tamaño de la matriz K. En esta situación se deberán realizar ambas etapas (definir la estructura de la matriz y llenarla) en cada iteración temporal y/o dentro del *solver* no lineal.

En este trabajo además de la creación y llenado de las matrices ralas, se agrega la necesidad de crear las estructuras utilizadas para compartir los resultados locales con los procesos correspondientes. Estas estructuras codifican la relación de vecindad entre subdominios, especificando a qué proceso se debe enviar y de qué proceso recibir el resultado en cada nodo (se presentan en detalle en 2.2.1).

La tabla 3.1 muestra el tiempo para la creación de la estructura de las matrices ralas, la creación de los mapas send y recv, y el llenado de las matrices. Esta tabla se presenta principalmente para poner en perspectiva el tiempo necesario para la resolución (a ser analizado en las secciones siguientes) respecto del tiempo de los otros pasos. Además se puede observar que el tiempo extra necesario para la creación de los mapas introducidos en este trabajo recién llega a estar en el mismo orden que el tiempo de creación de las estructuras vacías en los casos de mayor cantidad de subdominios. Este resultado coincide con el análisis del orden de complejidad para la creación de esos mapas (2.2.1), donde se ve que es proporcional a la cantidad de elementos, pues las filas de la tabla corresponden todas al mismo tamaño de problema por proceso.

Todas las pruebas que se presentan en esta sección fueron realizadas en el cluster

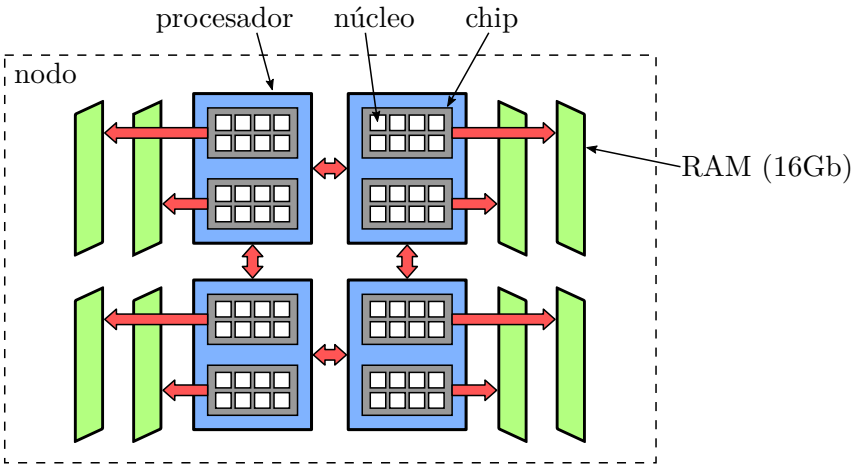

Fig. 3.1: Organización de un nodo del cluster Tupac

Tupac[3], el cual está equipado con 256 procesadores AMD Opteron 6276. Cada procesador es un pack de dos chips, con 8 núcleos de procesamiento cada uno. Esto proporciona un total de 4096 CPUs. Los procesadores están organizados en nodos de a 4, y cada nodo cuenta con 8 bancos de 16Gb de memoria cada uno. Cada uno de los dos chips de un procesador tiene acceso directo a uno de los bancos de memoria, pero todos los núcleos de un nodo pueden acceder a todos los bancos. Esto da lugar a una arquitectura que se denomina *NUMA* (Non-Uniform Memory Access), cuya característica es que un núcleo percibe velocidades de acceso distintas según a qué región de la memoria accede. La estructura de un solo nodo del cluster se puede ver en la figura 3.1 (Tupac consta de 64 nodos como estos). En cuanto a la estructura de caches, cada núcleo cuenta con su propio cache L1, cada par de núcleos comparte un mismo cache L2 y cada chip tiene su propio cache L3 compartido por todos los núcleos del chip. Los nodos están interconectados entre sí por medio de una red InfiniBand QDR. El sistema operativo instalado al momento de realizar las pruebas era RedHat 6.5.

3.3. Comparación entre las distintas implementaciones del PCG

3.3.1. Escalabilidad débil

En esta sección se presentan los resultados de los tiempos de ejecución obtenidos usando la implementación clásica y la versión pipelined (2.3.1 y 2.3.2 respectivamente). Se probó la escalabilidad débil fijando el tamaño del problema por proceso en 65k elementos, y la cantidad de subdominios se hizo variar entre 1 y 576. De esta forma se llegó a resolver problemas con aproximadamente 40M elementos totales y 113M grados de libertad (incógnitas). Para estas comparaciones entre los métodos se utilizaron mallas regulares, en las que todos los elementos son cuadrados iguales.

Las figuras 3.2 y 3.3 muestran el tiempo incurrido en la resolución del sistema sin precondicionar y el speedup respectivamente. No se aprecian diferencias significativas entre las dos versiones, siendo la versión pipelined levemente más lenta. El speedup ideal en el caso de escalabilidad débil es 1, y se puede ver cómo el speedup real decrece muchísimo en

[3] www.tupac.conicet.gov.ar

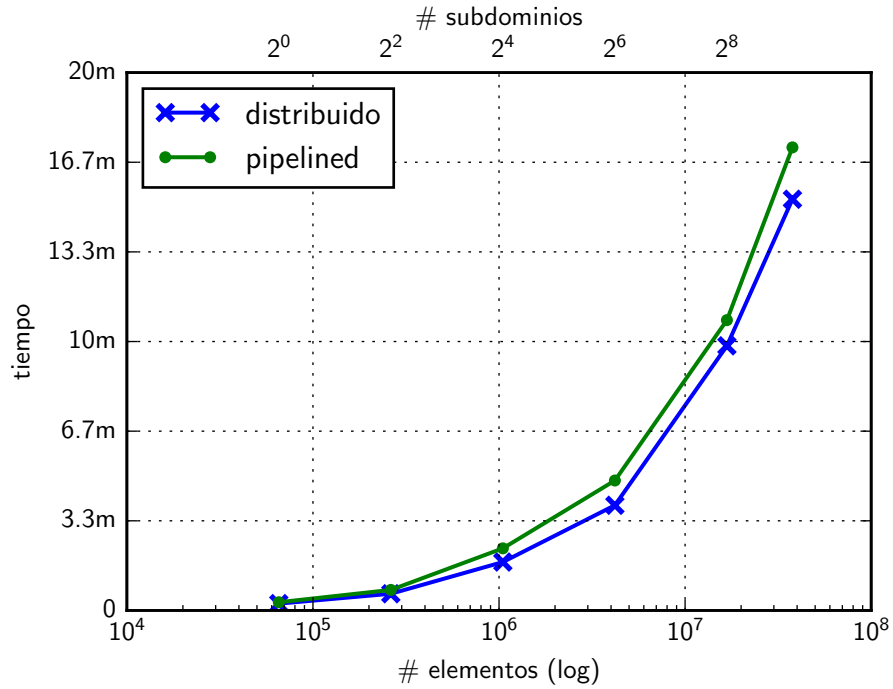

Fig. 3.2: Wall time del tiempo de resolución en escalabilidad débil.

ambos casos. También es importante notar que a medida que crece el tamaño del problema el tiempo necesario para la construcción y el llenado de las estructuras utilizadas se hace despreciable frente al tiempo de resolución.

El tiempo total necesario para resolver el sistema depende de dos factores: la cantidad de iteraciones y el tiempo incurrido en una iteración. En este caso, la cantidad de iteraciones se multiplica por 27 entre el problema más chico y el más grande. Esto es así pues el autovalor más chico de la matriz K está relacionado con la inversa de la dimensión del dominio total del problema (en este caso 1) y el autovalor más grande es proporcional a la inversa de la distancia mínima entre nodos de la malla. Dejando el tamaño del problema fijo, aumentar la cantidad de elementos acorta la distancia internodal, aumentando el número de condición de la matriz K y el PCG necesita más iteraciones para alcanzar la convergencia. De esta forma, el aumento del número de iteraciones está en línea con la disminución en 24 veces del tamaño de los elementos finitos.

Un incremento de 27 veces en la cantidad de iteraciones indica que el mejor speedup que se puede esperar para el problema más grande, ie. considerando que *cada iteración* escala idealmente, es de 0,03. La línea punteada de la figura 3.3 representa el speedup ideal que se podría lograr en caso de que cada iteración escale perfectamente. Se ve que de todas formas hay una degradación del rendimiento respecto de ese speedup ideal. Con lo cual, cabe preguntarse cómo escala el código normalizando el tiempo de todas las iteraciones. La figura 3.4 muestra el speedup considerando una sola iteración. El speedup por iteración cae a la mitad en los casos de mayor cantidad de procesos, pudiendo observarse que la versión pipelined es marginalmente más rápida en esos casos.

Por más que la cantidad de iteraciones sea lo que gobierna la pérdida de rendimiento, es preciso analizar el origen de la pobre escalabilidad por iteración. Para ello, se instrumentaron las operaciones principales y el speedup de cada una se presenta en la figura 3.5.

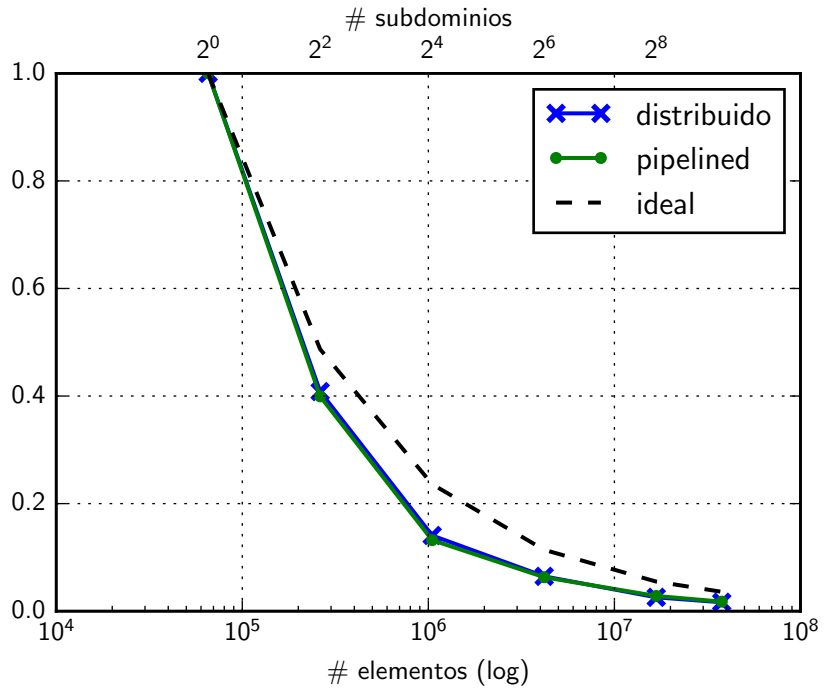

Fig. 3.3: Speedup del tiempo de resolución en escalabilidad débil.

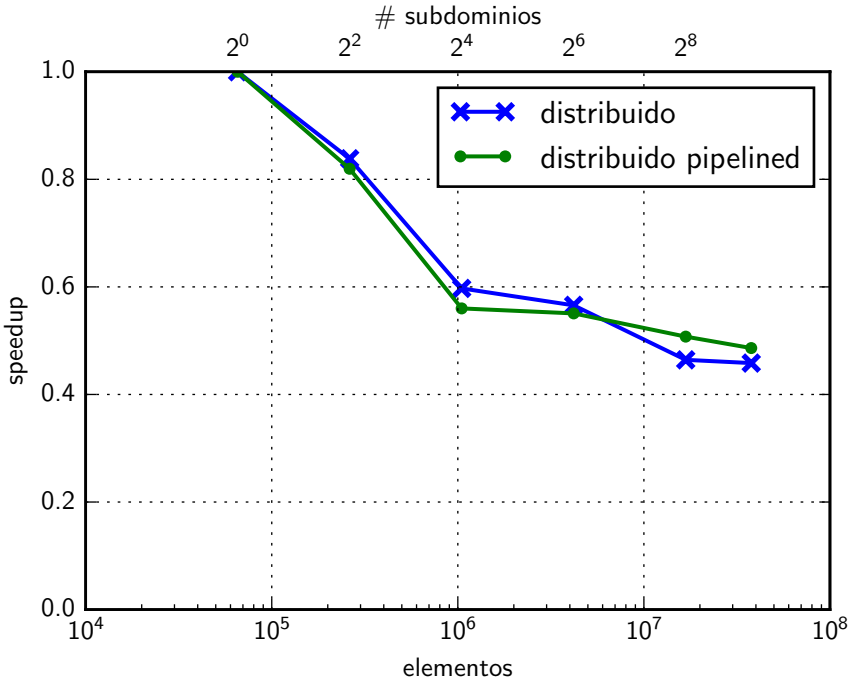

Fig. 3.4: Speedup del tiempo por iteración en escalabilidad débil.

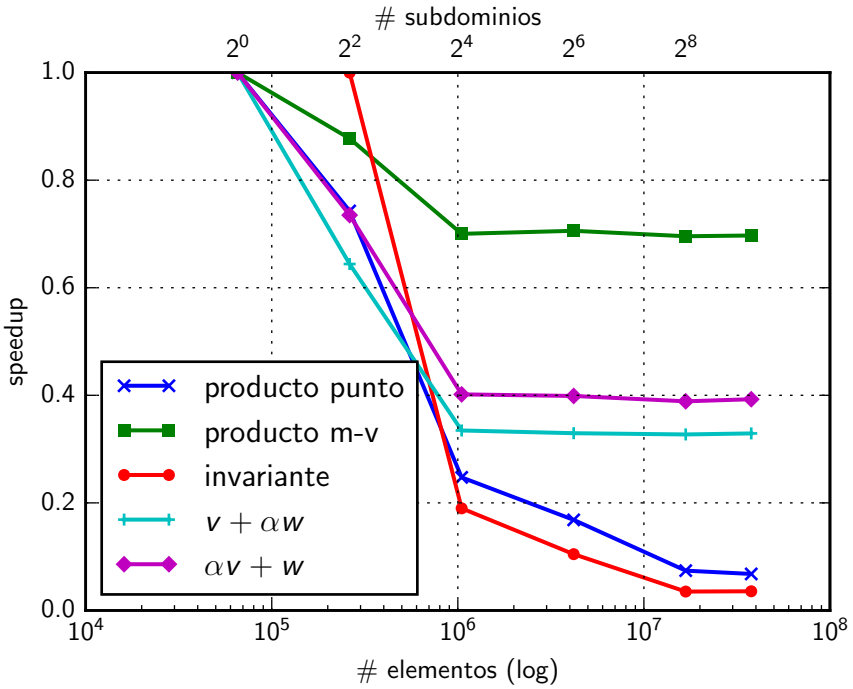

Fig. 3.5: Speedup por operación en escalabilidad débil.

La operación "sharing" se refiere a convertir un vector de distribuido a compartido, como fue visto en 2.1.3. Dado que esta operación no es necesaria cuando se resuelve el problema en un solo procesador, la curva para su speedup comienza en 4 subdominios, y los puntos siguientes son el speedup respecto de este caso base. Lo que se observa es que el producto punto y la conversión son las operaciones que peor escalan. Que el producto punto no escale bien era esperable y es inevitable, pues incluye una reducción global en la cual todos los procesos deben comunicarse. Suponiendo que la comunicación se hace en forma de árbol, aún la cantidad de pasos secuenciales necesarios está en el orden de $\log_2(n)$ donde n es la cantidad de procesos. Por más bueno que sea el patrón de comunicación, no hay forma de que la cantidad de pasos sea independiente de n. Esta es la razón de ser de la versión pipelined, ya que su principal característica es la superposición de esa reducción global con operaciones locales de cómputo.

La mala escalabilidad de la operación "share" parecería ser evitable. Dado que cada proceso se comunica únicamente con sus vecinos es esperable que exista una gran cantidad de comunicaciones que se pueden hacer simultáneamente. Para justificar esto, se presenta un posible patrón de comunicación que escalaría mejor. Un conjunto de 4 subdominios puede completar la primera fase de comunicación (la reducción) en 3 pasos, como lo indica la figura 3.6. Luego, se puede subdividir conceptualmente toda la grilla de subdominios en bloques de 2×2 y, simultáneamente, realizar la comunicación dentro de cada bloque. Esto deja afuera de la comunicación a los nodos que están sobre los bordes de la partición en bloques, para lo cual el siguiente paso es mover la división un subdominio hacia la derecha y uno hacia abajo. Volviendo a comunicar dentro de cada bloque, ahora lo único que resta por hacer es transferir en diagonal los nodos del medio de la subdivisión en bloques original, lo cual se puede hacer en dos pasos: primero se comunica horizontalmente y después verticalmente. El patrón se ilustra en la figura 3.7, donde cada cuadrado representa un

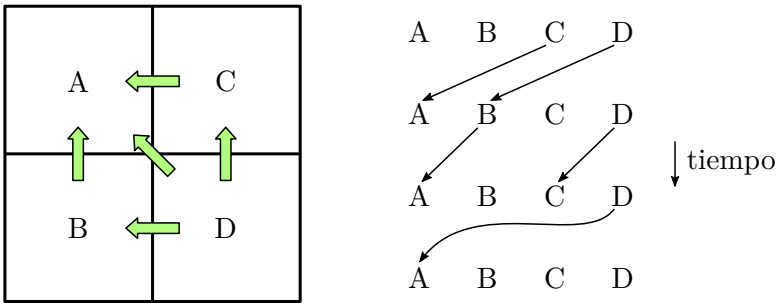

Fig. 3.6: Mínima cantidad de pasos de comunicación para reducir con 4 subdominios

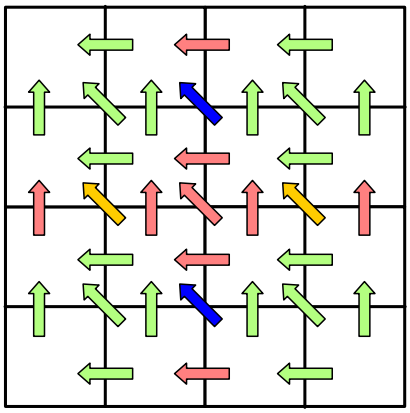

Fig. 3.7: Representación de un patrón de comunicación que no depende de la cantidad de elementos

subdominio y la idea es que las flechas del mismo color son comunicaciones que se pueden hacer de a pasos como en la figura 3.6, pero simultáneamente en toda la malla. En total se necesitan 4 pasadas secuenciales para realizar la totalidad de la comunicación, y ese número es independiente de la cantidad de subdominios. De todas formas, este patrón de comunicación solo sería efectivo en mallas con las características de las utilizadas en este capítulo, es decir, mallas regulares formadas por cuadriláteros. En mallas más generales como la de la figura 3.11 debería recurrirse a resolver el problema analizando el grafo de conectividades.

La implementación que se hizo del "share" no establece ningún orden ni estructura sobre la forma en la que los procesos se comunican entre sí. Cada proceso simplemente lanza todos los `recv()` que tiene que lanzar y luego todos los `send()`. Se cree que esta es la razón de la pérdida de speedup. Se hubiera esperado que las operaciones vectoriales y de multiplicación matriz-vector escalen casi perfectamente, pues carecen de comunicación y el tamaño del problema por proceso está fijo, pero sin embargo se observa un escalón en el rendimiento que se estabiliza a partir de 16 subdominios. Se cree que el origen de esta pérdida se encuentra en que los procesos comienzan a competir por el uso de la memoria y los caches. Conociendo la arquitectura de la máquina donde se realizaron las pruebas, se puede concluir que el escalón de 1 a 4 procesos se da porque, de a pares, los procesos utilizan el mismo banco de memoria y comparten el cache L3. De 4 a 16 pasan a ser 4 los procesos que utilizan el mismo banco. Suponiendo que este sea el caso y por ende la pérdida fuera esencial sin modificar el hardware, la ganancia en speedup (considerando todas las iteraciones) que se obtendría de conseguir hacer escalar el "share" de manera

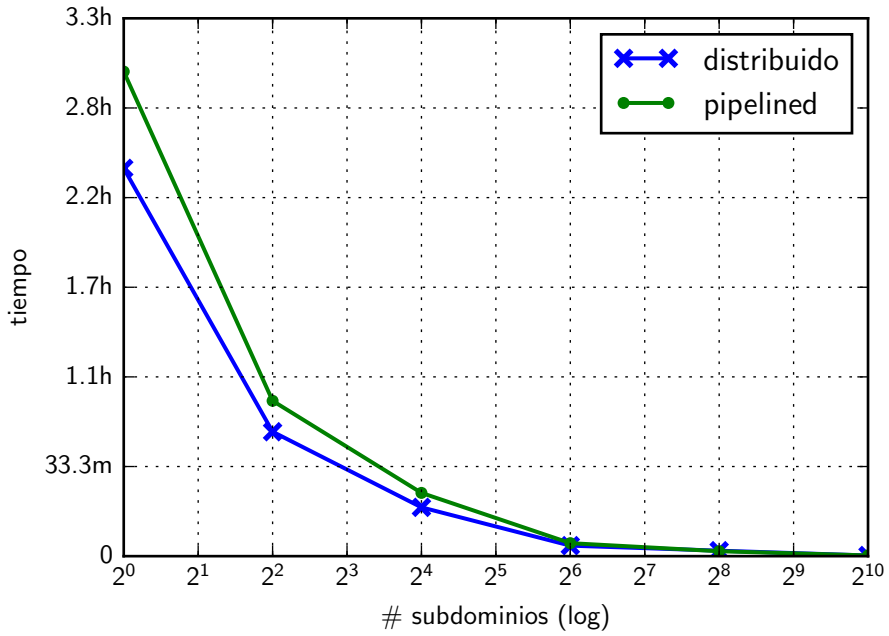

Fig. 3.8: Wall time del tiempo de resolución en escalabilidad fuerte.

ideal es de apenas un 3 %.

3.3.2. Escalabilidad fuerte

En esta sección se presentan los resultados obtenidos en escalabilidad fuerte. Se utilizó un problema de 4M de elementos y la cantidad de procesos se hizo variar entre 1 y 1024. Las figuras 3.8 y 3.9 muestran el tiempo de resolución y el speedup para las versiones implementadas.

Cuando se analiza escalabilidad fuerte el speedup ideal es la cantidad de procesos utilizados, pues el objetivo es resolver un mismo problema en menos tiempo. Se puede ver que el speedup obtenido es aproximadamente la mitad del ideal, con una pérdida aún mayor al utilizar 256 procesos. Como se está resolviendo siempre el mismo problema, la cantidad de iteraciones es la misma para todos los puntos de las figuras, con lo cual es necesario analizar cómo está escalando cada operación.

La figura 3.10 muestra el speedup desglozado por operación. Lo primero que se puede notar es que el producto matriz-vector y las operaciones vectoriales escalan superlinealmente al ejecutarlas en muchos procesadores (el extremo es la operación $\alpha v + w$ que es el doble de rápida a partir de 256 procesos). La causa más probable de este comportamiento es que a medida que se achica el tamaño del problema *por proceso*, esas operaciones pueden hacer un uso más eficiente de los caches. El producto punto se encuentra en la misma situación que al analizar escalabilidad débil: con la salvedad de que la parte local de su cálculo es más rápida pues opera sobre menos datos, la parte de comunicación es exactamente igual. La cantidad de datos a transferir sigue siendo un entero, y los procesos involucrados son los mismos.

La escalabilidad del sharing tiene la misma forma que la del producto punto, pero un factor de 0,5 debajo de este, lo mismo que sucedía con el speedup en escalabilidad débil. Esto no sorprende demasiado, pues por más que ahora la cantidad de datos a transferir

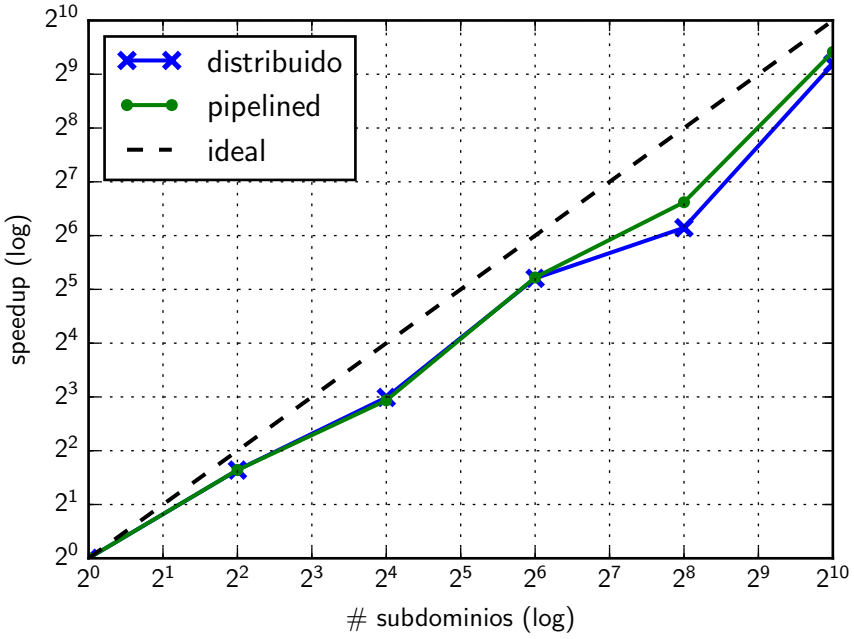

Fig. 3.9: Speedup del tiempo de resolución en escalabilidad fuerte.

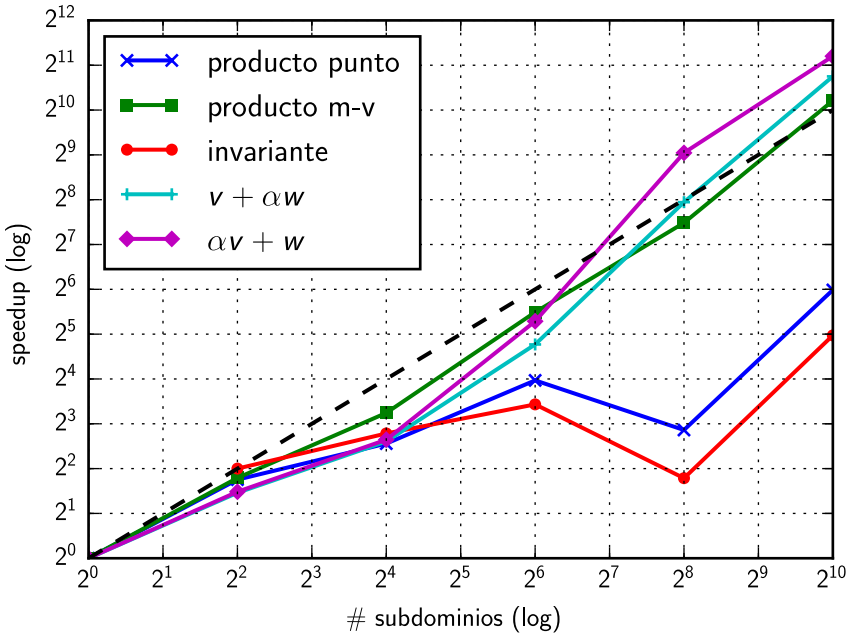

Fig. 3.10: Speedup de cada operación en escalabilidad fuerte.

es cada vez menor a medida que se agregan más procesadores, la falta de estructura en el patrón de comunicación implementado muy probablemente redunde en que los procesos se bloqueen esperando la recepción de un mensaje cuando podrían estar enviando. Además la latencia, siendo un tiempo fijo que no depende de la cantidad de datos a transferir, es una limitante en el speedup.

3.4. Precondicionadores

A continuación se presentarán los resultados obtenidos en cuanto a la aplicación de los precondicionadores implementados. Se optó por utilizar solo la versión clásica del algoritmo dado que la intención del análisis es sopesar la reducción en la cantidad de iteraciones versus el costo de crear y aplicar el precondicionador.

La primera sección presenta una comparación entre el *solver* preexistente (ver 1.4) y el implementado en este trabajo. Esta comparación se postergó hasta este punto pues la versión centralizada de Intel está basada exclusivamente en el complemento Schur. Las secciones que le siguen comparan los distintos precondicionadores (incluyendo el complemento Schur) entre sí y sin aplicar ninguno, pero lo hacen aplicando el método a mallas diferentes.

Hasta ahora las mallas utilizadas para obtener los resultados fueron grillas uniformes, en la cual todos los elementos son iguales porque los diferentes métodos de resolución no cambian la característica espectral del sistema de ecuaciones. La comparación entre precondicionadores se hizo utilizando mallas con elementos distintos como la que se ve en la figura 3.11. Precisamente, estas mallas se obtienen aplicando la transformación

$$(x, y) \mapsto (x^\alpha, y^\alpha).$$

a una malla regular. Este tipo de mallas implican no solo un número de condición mucho mayor, sino también que los autovalores de la matriz se encuentren más dispersos. Esta situación se parece mucho más a lo que se puede encontrar en la práctica. En general, la irregularidad de la malla no es lo único que afectará las características espectrales de la matriz, sino también la variedad de propiedades de los distintos elementos. Esta variación se puede dar, por ejemplo, si el dominio tiene materiales diferentes por regiones.

La irregularidad de los elementos impacta en la cantidad de iteraciones necesarias, pero las operaciones que realiza cada iteración son exactamente las mismas. Ya establecidas las características de escalabilidad por iteración, la efectividad de un precondicionador se analizó teniendo en cuenta en cuánto reduce las iteraciones y el costo extra de aplicación del mismo. Esto tiene sentido hacerlo sobre mallas que representen un poco mejor las que se espera utilizar en la práctica, al menos en cuanto a cantidad de iteraciones se refiere.

En los casos que se presentan aquí se tomó $\alpha = 1{,}5$. La elección de un α mayor en problemas grandes deja algunos nodos demasiado cerca entre sí, haciéndolos indistinguibles para la precisión del formato de archivo de entrada utilizado.

3.4.1. Comparación entre la versión distribuida y la centralizada de Intel

En la figura 3.12 se puede ver el tiempo total de resolución tanto para la versión centralizada de Intel como para el *solver* distribuido. Los tiempos considerados aquí incluyen, además de las iteraciones en sí, el tiempo de la factorización LU de K_{ii}, el tiempo de la creación del vector de la parte derecha de la ecuación 1.19 y el tiempo necesario para

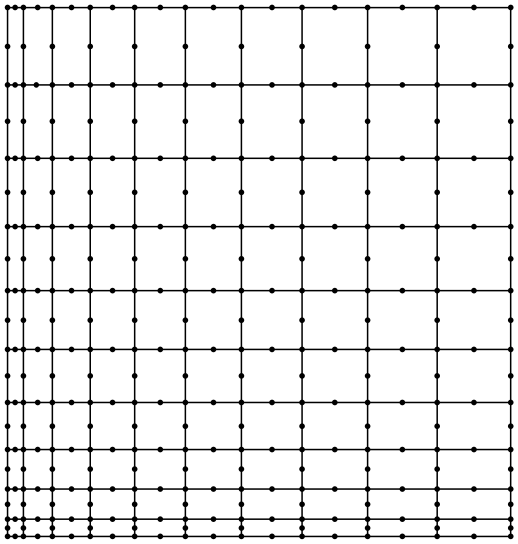

Fig. 3.11: Malla irregular análoga a la utilizada para la comparación entre los precondicionadores.

resolver el sistema de los nodos internos (1.20). La factorización de K_{ii} se realiza pues si se recuerda el algoritmo 5, la multiplicación por el complemento Schur requiere la resolución del sistema $K_{ii}x = b$. Todos estos tiempos son tenidos en cuenta pues son costos extra implicados por la utilización del complemento Schur. Son el equivalente a la creación y aplicación de un precondicionador.

El nuevo *solver* logra resolver el sistema más grande en un tercio del tiempo que el código original, lo cual representa una mejora sustancial. Por otro lado, la figura 3.13 muestra el speedup observado teniendo en cuenta el tiempo por iteración. Habiendo verificado que ambos algoritmos realizan la misma cantidad de iteraciones, el modo de realizar cada iteración es lo único que varía entre las dos versiones. Esto permite concluir que la ganancia en tiempo de ejecución se debe a que cada iteración es 3 veces más rápida. Por otro lado, el speedup por iteración es considerablemente mejor que sin utilizar el complemento Schur (ver figura 3.4). Esta mejora puede resultar contraria a la intuición, pues la única diferencia a nivel iteración entre usar el complemento Schur y no usarlo radica en la multiplicación matriz-vector, siendo mucho más costosa si se usa (un 97 % del tiempo de una iteración vs. un 73 %, respectivamente). Sin embargo, este costo extra es puramente de cálculo local en cada proceso, con lo cual la pobre escalabilidad de las operaciones que involucran comunicación tienen un impacto mucho menor en el caso del complemento Schur.

3.4.2. Comparación de los distintos precondicionadores

Aquí se presentan los resultados obtenidos de comparar los distintos precondicionadores implementados. Se utilizó un problema modelo de 4M de elementos, dividido en 4, 64 y 256 subdominios para analizar el comportamiento de cada precondicionador bajo distintas condiciones de paralelización. Las mallas utilizadas fueron irregulares como la de la figura 3.11.

La figura 3.14 muestra los tiempos totales de resolución utilizando los distintos precondicionadores. Los casos de 4 subdominios sin precondicionador y con Gauss-Seidel quedan fuera del gráfico pues demoraron ambos más de una hora. Se puede observar que la utili-

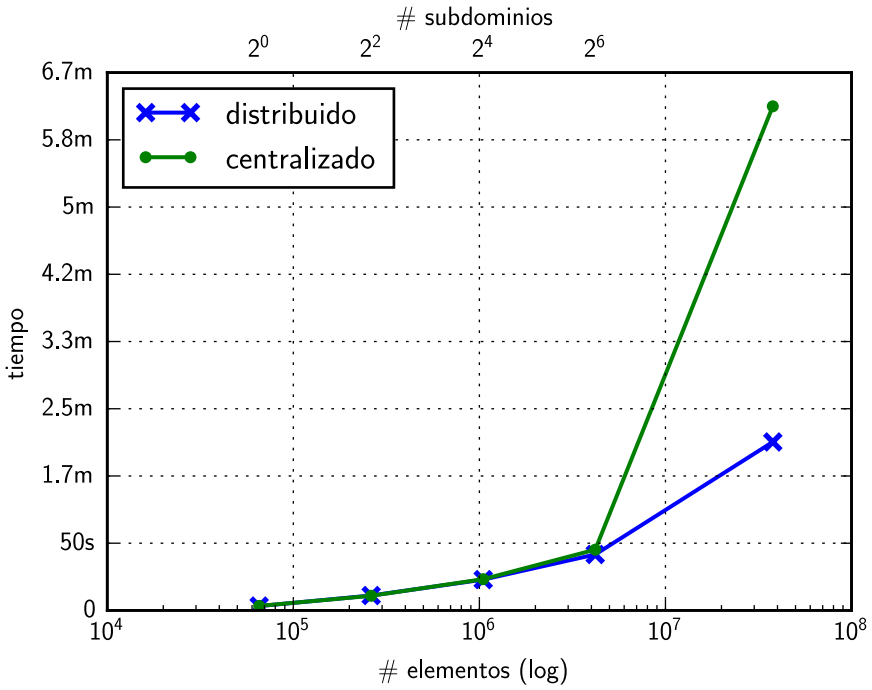

Fig. 3.12: Wall time del tiempo de resolución en escalabilidad débil.

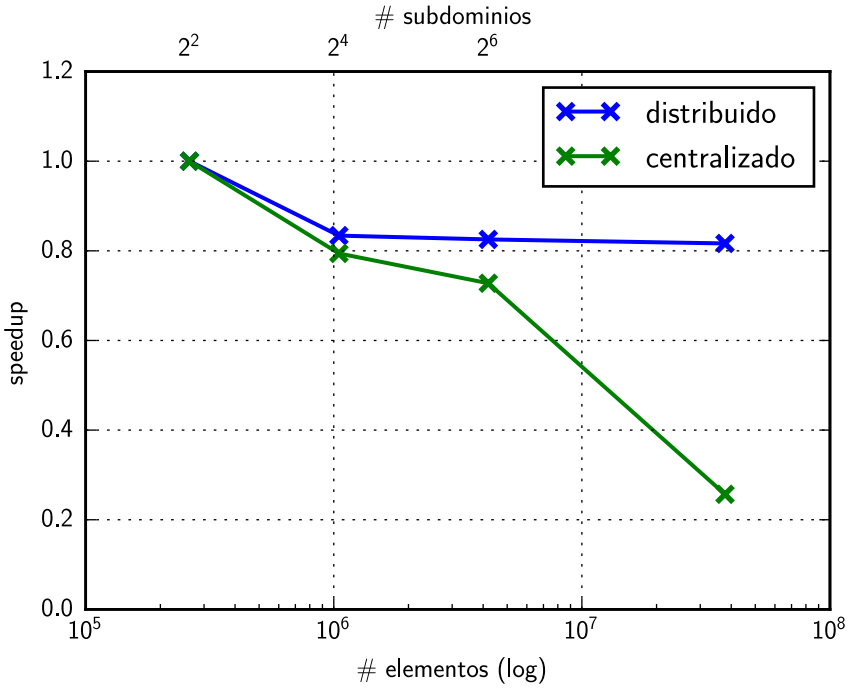

Fig. 3.13: Speedup del tiempo por iteración en escalabilidad débil.

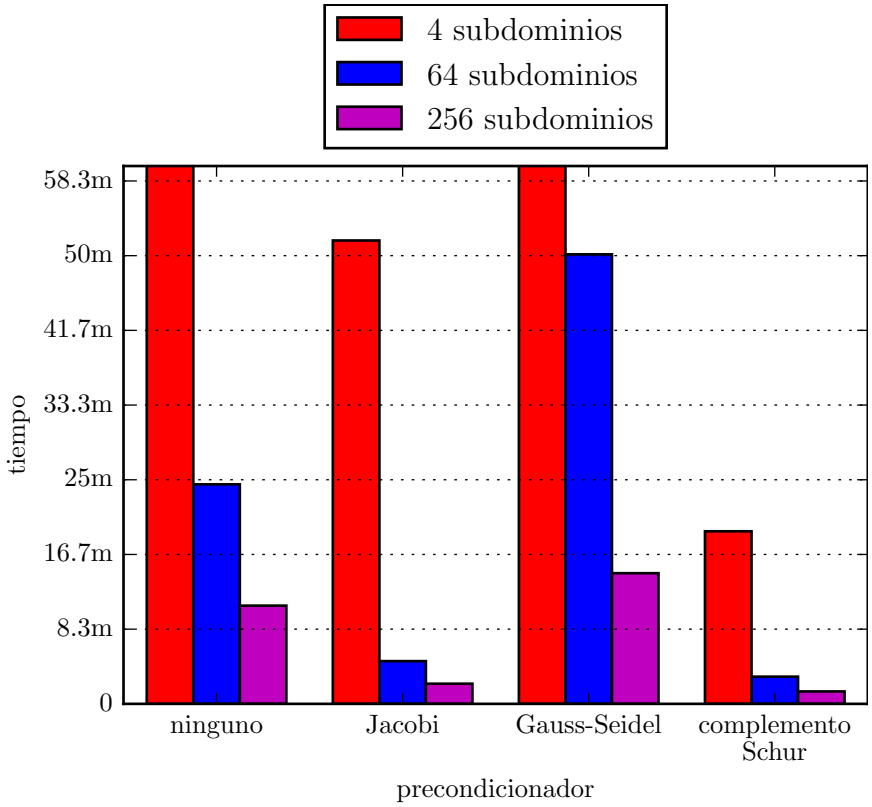

Fig. 3.14: Tiempo total de resolución usando precondicionador.

zación del complemento Schur ofrece la mayor ganancia en términos de tiempo, superando al precondicionador de Jacobi en un 38 % para el caso de 256 subdominios. El precondicionador de Gauss-Seidel necesitó aún más tiempo que sin aplicar ningún precondicionador en todos los casos.

La figura 3.15 presenta la cantidad de iteraciones que hizo el PCG usando cada precondicionador. La mayor ganancia la alcanza nuevamente el complemento Schur, realizando aproximadamente un 1 %, 3 % y 5 % de las iteraciones necesarias si no se utiliza precondicionador. Es interesante notar que si bien Jacobi y Gauss-Seidel logran una reducción semejante entre sí (variando alrededor del 20 % entre las distintas subdivisiones), la aplicación de este último es dos órdenes de magnitud más lenta, lo cual justifica el abultado aumento del tiempo total de resolución. De la misma forma, si bien el uso del complemento Schur reduce mucho más las iteraciones que Jacobi, la ganancia en tiempo no es tan notoria pues la aplicación del complemento Schur es bastante costosa según puede verse en Cód. 9.

La diferencia en cantidad de iteraciones según la subdivisión que presenta la aplicación del complemento Schur es esperable. Como se vio en la sección 1.4, cuando se aplica el complemento Schur lo que resuelve el *solver* iterativo es el sistema sobre los nodos externos. Al cambiar la cantidad de subdominios, los nodos externos varían alterando el número de condición de K_{Sch}. Por otro lado, la variación en la cantidad de iteraciones del precondicionador de Gauss-Seidel se debe a que la implementación de este precondicionador no es exacta, realizada sin comunicar componentes de la matriz distribuida K_{ee}. Esta falta de comunicación impacta en que al variar la división en subdominios se está aplicando un

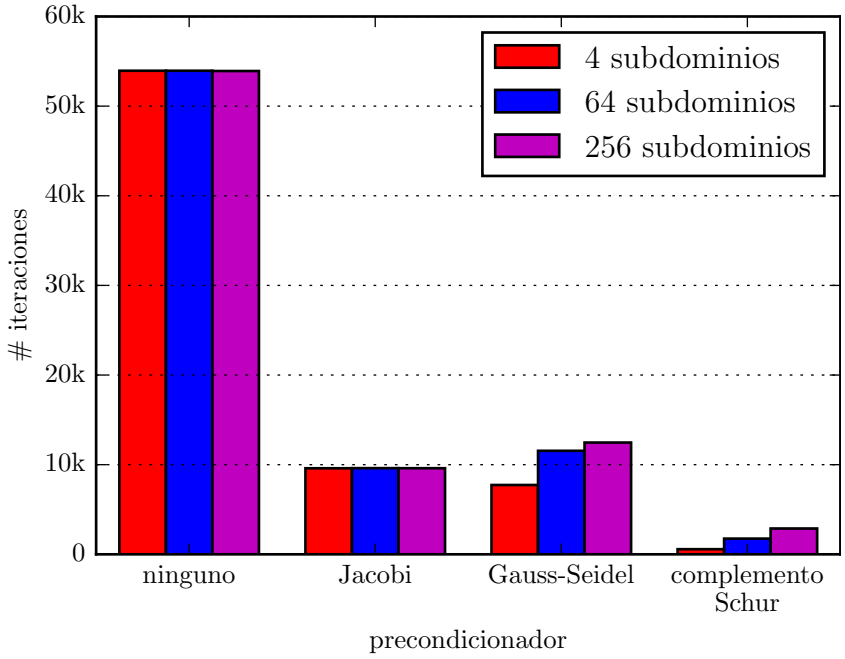

Fig. 3.15: Cantidad de iteraciones según precondicionador.

precondicionador levemente distinto.

Dado que el precondicionador de Gauss-Seidel alcanza una reducción en la cantidad de iteraciones un poco menor que el precondicionador de Jacobi, para que resulte conveniente la aplicación del primero sería necesario optimizar su aplicación hasta que sea más rápida que la de Jacobi. Una optimización posible pero aún insuficiente sería aprovechar el hecho de que los cuatro sistemas que se deben resolver para aplicar Gauss-Seidel ya son triangulares. La implementación utilizada resuelve los sistemas mediante la biblioteca PARDISO, la cual primero crea una factorización LU de la matriz y luego hace sustitución hacia atrás y hacia adelante. Sin embargo el orden de esta operación sigue siendo cuadrático, mientras que la aplicación de Jacobi es lineal. Resulta muy difícil imaginar una optimización de Gauss-Seidel que supere en tiempo de aplicación al precondicionador de Jacobi.

4. CONCLUSIONES

En este trabajo se planteó una implementación paralela de un *solver* iterativo, el PCG, para ser aplicado en la aproximación de soluciones a ecuaciones diferenciales mediante el FEM. La idea central fue mejorar la escalabilidad de la versión preexistente, reduciendo tanto la cantidad de información a comunicar entre procesos como el número de procesos que deben compartir los datos. Además se homegeneizó la memoria requerida por los distintos procesos para permitir una escalabilidad débil en términos de memoria mucho mayor. Para tal fin se realizó un diseño que permitió encerrar en abstracciones cada aspecto de un *solver* iterativo, y se programaron dos versiones del PCG: la llamada *distribuida*, implementa el PCG clásico tal cual se encuentra en los libros de texto de métodos numéricos introductorios, y la versión llamada *pipelined*, que implementa las modificaciones sugeridas en [10] para enmascarar la comunicación global inevitable con cáculo local. Esta última versión se construyó no solo para intentar disminuir el tiempo de resolución, sino también para poner a prueba la jerarquía de interfaces y clases ideada.

Para implementar un *solver* completo y funcional fue necesario implementar, además del algoritmo en sí, una operación de producto punto que funcione de forma distribuida. También para permitir que los procesos que comparten información utilicen datos consistentes entre sí, se programó un patrón de comunicación que solo involucra a los procesos que necesitan y proveen la información compartida.

A la hora de utilizar un *solver* iterativo para resolver sistemas lineales de millones de incógnitas resulta imprescindible contar con un buen precondicionador, que sea capaz de disminuir significativamente el número de iteraciones necesarias a la vez que su aplicación no incurra en un costo extra demasiado alto. Para comenzar a explorar las alternativas aplicando el framework que se está extendiendo, se implementaron algunos precondicionadores elementales a modo de primeros pasos. Los precondicionadores implementados son el de Jacobi y una versión inexacta de Gauss-Seidel. También se adaptó el *solver* para que permita resolver el sistema mediante una técnica de precondicionamiento que utiliza el complemento Schur.

Los resultados obtenidos muestran que utilizando el nuevo *solver* junto con el precondicionador de complemento Schur, se logró resolver el mismo problema (38M de elementos, 113M de incógnitas) en un tercio del tiempo que le tomó a la implementación anterior. Esta mejora corresponde a una ejecución en paralelo en 576 procesadores. En problemas más chicos (16M de elementos o menos) el tiempo de resolución se encontró que es solo marginalmente inferior con los nuevo solvers. En cuanto a la escalabilidad de una sola iteración, la versión aquí presentada cuando se usa el complemento Schur alcanzó un speedup de 0,84 para todas las cantidades de procesos que se probaron, mientras que la versión anterior se ubica siempre debajo de 0,79, llegando al mínimo de 0,25 para el caso de 576 procesadores.

Respecto de la implementación del *solver* distribuido pipelined, no se encontraron mejoras significativas en los tiempos de resolución respecto del distribuido. Al observar el speedup por iteración para escalabilidad débil y el speedup total para escalabilidad fuerte sí se evidencian mejoras del *solver* distribuido pipelined por sobre el distribuido común para los casos de mayor cantidad de procesos. En escalabilidad fuerte, por ejemplo, para los casos de 512 y 1024 procesos el speedup del pipelined es un 40 % y un 15 % mayor

respectivamente. Esto en principio es consistente con que la reducción global comienza a ser significativa cuando la cantidad de procesos es muy grande, dando lugar a una mayor solapamiento entre el cálculo local y la comunicación. Es necesario efectuar pruebas más detalladas si se desea evaluar el rendimiento del *solver* distribuido pipelined de manera más concluyente.

El speedup por iteración de la versión distribuida cae por debajo de 0,5 al utilizar la mayor cantidad de procesadores. Esto es bastante menos de lo que se esperaba, pero un análisis por operación reveló que el código que comparte las componentes de un vector entre todos los procesos relevantes (*sharing*) escala mucho peor de lo que podría hacerlo (abajo de 0,1, aún por debajo del speedup del producto punto que implica una reducción global). Si bien la multiplicación matriz-vector es la operación que más peso tiene en cuanto a tiempo, es necesario mejorar el patrón de comunicación utilizado por la operación de *sharing* si se pretende aumentar el speedup por iteración del *solver*.

Las pruebas que se hicieron con los precondicionadores mostraron que el uso del complemento Schur logra las mayores reducciones en la cantidad de iteraciones, y los menores tiempo de ejecución, si bien el precondicionador de Jacobi se le acerca mucho cuando la cantidad de procesos es elevada y los subdominios son chicos (con complemento Schur la resolución demora un 63 % lo que demora con Jacobi). Es posible hipotetizar que bajo ciertas condiciones el precondicionador de Jacobi dé mejores resultados que el complemento Schur, pues este último debe resolver un sistema para calcular $K_{ii}^{-1}v$ en cada aplicación (una operación cuadrática en el tamaño de v) mientras que Jacobi solo tiene que multiplicar un vector por otro componente a componente (operación lineal en el tamaño del vector). Podría pasar que al subdividir en muchos subdominios, cada uno de ellos muy grandes, la reducción en la cantidad de iteraciones alcanzada por el complemento Schur no logre compensar el menor costo de aplicación en el que incurre el precondicionador de Jacobi.

El hecho de que la cantidad de iteraciones necesarias para alcanzar convergencia cuando se aplica el complemento Schur aumente junto con la cantidad de subdominios es un llamado de atención. Para lograr una buena escalabilidad en aún mayor número de procesadores, será necesario la aplicación de un precondicionador global al sistema representado por K_{Sch}. En [20], [7] y [6] se proponen alternativas para esto que podrían ser exploradas en el futuro.

En cuanto al diseño, uno de sus puntos débiles resulta de que la interfaz `IterativeSolver` es un *template*, y se especializa en dos clases distintas para utilizar o no el complemento Schur. Bajo estas circunstancias, el *solver* sin complemento Schur y el *solver* con complemento Schur no son polimórficos, imponiendo sobre el usuario del framework la necesidad de crear código diferente para uno y otro caso. Se deberá evaluar si esto representa un problema real, ya que siendo que el complemento Schur disminuye notoriamente la cantidad de iteraciones es posible que la versión que no lo utiliza sea de poca utilidad más que como referencia.

En definitiva, en este trabajo se ha logrado una implementación del PCG con buena escalabilidad, la cual ha sido utilizada con éxito en problemas de más de 100M de incógnitas que fueron resueltos en paralelo utilizando más de 1000 procesadores, manteniendo la flexibilidad y modularidad del framework original. De las posibles mejoras a futuro, las dos que más impacto podrían tener en la mejora de la eficiencia del código son una implementación más cuidadosa de la comunicación de a pares en el producto matriz vector y la incorporación de un precondicionador global complementario a la técnica del complemento Schur.

Bibliografía

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, USA, 1994.

[2] E. B. Becker, G. F. Carey, and J. T. Oden. *Finite Elements: An Introduction*, volume 1 of *Texas finite element series*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1981.

[3] R. L. Burden and J. D. Faires. *Numerical analysis*. Brooks Cole, 1998.

[4] G. H. Burzstyn, A. D. Otero, and J. Quinteros. Una nueva implementación para las ecuaciones de Navier–Stokes mediante KLE y elementos espectrales. In *Mecánica Computacional*, volume 27, pages 2367–2383. AMCA, November 2008.

[5] Klapka I. Cardona A. and Geradin M. Design of a new finite element programming environment. *Engineering Computation*, 11:365–381, 1994.

[6] Luiz M Carvalho, Luc Giraud, and Patrick Le Tallec. Algebraic two-level preconditioners for the Schur complement method. *SIAM Journal on Scientific Computing*, 22(6):1987–2005, 2001.

[7] JM Cros. A preconditioner for the Schur complement domain decomposition method. *14th International Conference on Domain Decomposition Methods*, pages 373–380, 2003.

[8] Buscaglia G.C. Dari E.A. and Lew A. A parallel general purpose finite element system. *IX SIAM Conference on Parallel Processing for Scientific Computing. San Antonio, Texas, USA*, 1999.

[9] L. C. Evans. *Partial Differential Equations*, volume 19 of *Graduate studies in mathematics*. American Mathematical Society, 1998.

[10] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40:224–238, 2013.

[11] Magnus R. Hestener and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *J. of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[12] Mackie R. I. Object oriented implementation of distributed finite element analysis in .net. *Advances in Engineering Software*, 38:726–737, 2007.

[13] Intel®. Math kernel library. https://software.intel.com/en-us/intel-mkl.

[14] A. D. Otero and J. Quinteros. General parallel finite/spectral-element oriented C/C++ framework. In *Proceedings of PARENG2011 – The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, April 2011.

[15] Peter Pacheco. *Parallel Programming With MPI*. Morgan Kaufmann, October 1996.

[16] J. Quinteros, P. M. Jacovkis, and V. A. Ramos. Diseño flexible y modular de modelos numéricos basados en elementos finitos. In Sergio A. Elaskar, Elvio A. Pilotta, and Germán A. Torres, editors, *Mecánica Computacional*, volume XXVI, pages 1724–1740. AMCA, October 2007.

[17] Erhard Schmidt. Zur Theorie der linearen und nichtlinearen Integralgleichungen. I. Teil: Entwicklung willkürlicher Funktionen nach Systemen vorgeschriebener. *Mathematische Annalen*, 63:433–476, 1907.

[18] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf, 1994.

[19] V.E. Sonzogni, A.M. Yommi, N.M. Nigro, and M.A. Storti. A parallel finite element program on a Beowulf cluster. *Advances in Engineering Software*, 33(7-10):427–443, 2002.

[20] M. Storti, L. Dalcín, R. Paz, A. Yommi, V. Sonzogni, and N. Nigro. A preconditioner for the Schur complement matrix. *Advances in Engineering Software*, 37(11):754 – 762, 2006.

[21] C. Subramanian V. Heuveline, D. Lukarski and J.-P. Weiss. Parallel preconditioning and modular finite element solvers on hybrid CPU-GPU systems. *Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, 2011.

[22] Fuzhen Zhang. *The Schur Complement and Its Applications*, volume 4 of *Numerical Methods and Algorithms*. Springer, 2005.