UNIVERSIDAD DE COSTA RICA
SISTEMA DE ESTUDIOS DE POSGRADO

IMPLEMENTACIÓN DE UNA PLATAFORMA COMPUTACIONAL PARA LA SOLUCIÓN DE PDE'S ELÍPTICAS, MEDIANTE FEM Y DESCOMPOSICIÓN DE DOMINIOS BDDC.

Tesis sometida a la consideración de la Comisión del Programa de Posgrado en Matemática para optar al grado y título de Maestría Académica en Matemática con énfasis en Matemática Aplicada

FABIÁN MORA CORDERO

Ciudad Universitaria Rodrigo Facio, Costa Rica

2019

Dedicatoria

A mi familia,

Gracias

Agradecimientos

Deseo expresar mi gratitud con mi profesor tutor Juan Gabriel Calvo, por todo su apoyo durante toda esta etapa. También quiero agradecer al profesor Francisco Siles, por todas las oportunidades, apoyo y consejos que me ha dado a lo largo de todos estos años; al igual que al PRIS-Lab por todas las oportunidades que me ha brindado y permitirme usar el clúster Serké en las pruebas de este trabajo.

Esta tesis fue aceptada por la Comisión del Programa de Estudios de Posgrado en Matemática de la Universidad de Costa Rica, como requisito parcial para optar al grado y título de Maestría Académica en Matemática con énfasis en Matemática Aplicada.

Ph.D. Pedro Méndez Hernández
Representante del Decano
Sistema de Estudios de Posgrado

Ph.D. Juan Gabriel Calvo Alpizar
Director de Tesis

Dr. rer. nat. Francisco Siles Canales
Asesor

Ph.D. Esteban Segura Ugalde
Asesor

Ph.D. Fabio Sánchez Peña
Representante
Programa de Posgrado en Matemática

Fabián Mora Cordero
Candidato

Tabla de contenido

Resumen

En este trabajo se desarrolla una plataforma computacional en *C++-CUDA*, la cual introduce una serie de estructuras de datos que simplifican el manejo de memoria en *CUDA*. En la plataforma también se introducen elementos de programación funcional con rendimiento nativo por medio de las capacidades de meta-programación de *C++*.

Haciendo uso de esta plataforma, se desarrollan métodos de elementos finitos para la solución de la ecuación de *Poisson* en dos dimensiones en *CUDA*. Además se implementa una versión del precondicionador *(BDDC)* para *GPU*, usando tanto métodos iterativos como métodos directos para la solución de los sistemas lineales. Las pruebas computacionales se realizan en una tarjeta NVIDIA *Tesla V100* y una tarjeta *GTX 960M*. Se ha encontrado que el uso de métodos directos permite una convergencia con mayor rapidez a la solución real de la ecuación, y si se conocen las factorizaciones LU y de *Cholesky*, entonces los métodos directos son varios órdenes de magnitud más rápidos que los métodos iterativos. Finalmente se compara la plataforma desarrollada contra una implementación en *Matlab* del precondicionador *BDDC*, donde la plataforma desarrollada es hasta 11 veces más rápida que la implementación en *Matlab*.

Abstract

In this work we develop a computational platform written on *C++-CUDA* introducing a series of data structures that handle memory management, simplyfing this task on *CUDA*. With the help of the metaprogramming capabilities of *C++*, we introduce elements of functional programming with native performance to *CUDA*.

With this platform, we develop finite element methods to solve the Poisson equation in two-dimensional domains, and to ease this task we implement a version of the *BDDC* preconditioner for *GPU*, using both iterative and direct methods. The computational tests are performed on a NVIDIA *Tesla V100* and a *GTX 960M*. We have found that using direct methods accelerates the convergence to the solution, and if the LU and *Cholesky* decompositions are known, then direct methods are several orders of magnitude faster than iterative methods. Finally, we compare the developed platform to a *Matlab* implementation of the *BDDC* preconditioner, where the developed platform is up to 11 times faster than the *Matlab* implementation.

Lista de tablas

Lista de figuras

Lista de Algoritmos

Notación

Ω	Dominio de $\mathbb{R}^n$.
$\partial\Omega$	Frontera de Ω.
$int\Omega$	Interior topológico de Ω.
$\overline{\Omega}$	Clausura topológica de Ω.
$\overline{convex}(\Omega)$	Clausura convexa de Ω.
$\text{diam}(\Omega)$	Diámetro de Ω.
$\mathbb{R}^{n\times m}$	Espacio vectorial de las matrices $n \times m$ sobre $\mathbb{R}$.
GL_n	Grupo de matrices $n \times n$ invertibles sobre $\mathbb{R}$.
P_m	Anillo de polinomios de grado menor o igual a m.
V^*	Dual del espacio vectorial V.
$C^k(\Omega)$	Espacio de funciones k veces derivables en Ω.
$L^p(\Omega)$	Espacio de funciones p-integrables en Ω.
$W_p^k(\Omega)$	Espacio de Sobolev k veces derivables y p-integrables.
$H^k(\Omega)$	Espacio de Sobolev k veces derivables cuadrado integrables.
$\|v\|$	Norma cuadrática del vector v.
$\|f\|_V$	Norma del vector f en el espacio normado V.
$\|f\|_V$	Semi norma del vector f en el espacio normado V.
$D_w^\alpha f$	Derivada débil de f.
$(\Omega, \mathcal{P}, \mathcal{N})$	Elemento finito.
$\kappa(A)$	Número de condición de la matriz A.

$\mathcal{I}$	Operador interpolador.
$\mathcal{S}$	Complemento de *Schur*.
M	Operador precondicionador.
Γ	Interfaz de la región.

Acrónimos

PDE	Ecuación diferencial en derivadas parciales.
FEM	Método de elementos finitos.
BDDC	Balancing Domain Decomposition by Constraints.
NSD	Cantidad de subdominios.
NN	Cantidad de nodos.
CG	Gradiente conjugado.
MINRES	Método de mínimos residuos.
SMpV	Matriz poco densa por vector denso.
NUMA	Arquitectura de memoria no uniforme.
SISD	Una instrucción un dato.
SIMD	Una instrucción múltiples datos.
MISD	Múltiples instrucciones un dato.
MIMD	Múltiples instrucciones múltiples datos.
CUDA	Compute Unified Device Architecture.
CPU	Unidad de procesamiento central.
GPU	Tarjeta de procesamiento gráfico.
GPGPU	Tarjeta de procesamiento gráfico de propósito general.
SM	Streaming Multiprocessors.
SIMT	Una instrucción múltiples threads.

Capítulo 1

Introducción

El electromagnetismo es una de las áreas de la física que más ha contribuido a la sociedad contemporánea, ya que ha permitido la invención de toda la tecnología electrónica moderna, introduciendo dispositivos que sirven desde producir más conocimiento humano, como lo es una computadora, o un telescopio electrónico, hasta dispositivos que salvan vidas como un dispositivo de resonancia magnética. Por lo que estudiar el fenómeno electromagnético, aparte de ser interesante por sí solo, es vital para continuar el desarrollo humano.

En especial uno de los problemas más importantes para el electromagnetismo, es el de determinar el potencial eléctrico de objetos en el espacio. La importancia de resolver este problema con exactitud se entiende mejor en la dimensión de la vida cotidiana, ya que por ejemplo es responsable de la transmisión del fluido eléctrico desde las plantas generadoras de energía eléctrica hasta los lugares de consumo de esta energía, como un hospital. Otro ejemplo es la interacción de un potencial eléctrico con materiales semiconductores, ya que esta interacción tan particular es el ingrediente principal que permite la creación de los transistores -la unidad electrónica básica responsable de las computadoras.

La principal herramienta para estudiar los fenómenos electromagnéticos son las ecuaciones de Maxwell, que consisten en una serie de ecuaciones diferenciales en derivadas parciales que modelan la generalidad del comportamiento electromagnético (Griffiths, 2012). Estas ecuaciones diferenciales son difíciles de solucionar explícitamente, por ejemplo la ecuación de Poisson:

$$\begin{cases} -\Delta V = \rho/\epsilon \ \text{ en } \Omega \\ V = g \ \text{ en } \partial\Omega, \end{cases}$$

que corresponde a encontrar los potenciales eléctricos V en una región isotrópica -donde ρ es la distribución de carga en la región y ϵ es la permitividad de la región, es sumamente complicada de solucionar explícitamente hasta en dominios básicos, ya que para poder encontrar la solución explícita usualmente es necesario construir una función de Green lo cual no es sencillo (Evans, 2010).

Entonces la mejor forma de tratar estos problemas físicos que ocupan soluciones prácticas, pero que requieren un grado importante de exactitud, es el análisis numérico. Dentro de los métodos numéricos existentes para resolver ecuaciones diferenciales, uno de los que más sobresale es el método de elementos finitos (FEM), ya que genera una abstracción matemática para la creación de algoritmos, que permiten computar la solución de una ecuación diferencial en dominios con geometría complicada, con la ventaja de que las propiedades teóricas de estos algoritmos se pueden estudiar con las herramientas del análisis funcional (Brenner & Scott, 2007).

El método de elementos finitos consiste en crear una aproximación de un problema continuo en un cierto espacio de funciones de dimensión finita, es decir se busca aproximar el valor de una función en una cantidad finita de puntos del dominio mediante la combinación lineal de un conjunto finito de funciones. En la figura 1.1 se observan los resultados de utilizar el método de elementos finitos para resolver una ecuación diferencial usando 10×10 nodos, donde a la izquierda está la solución exacta, en el centro el resultado de la aproximación y a la derecha el error de la aproximación de orden 10^{-2}.

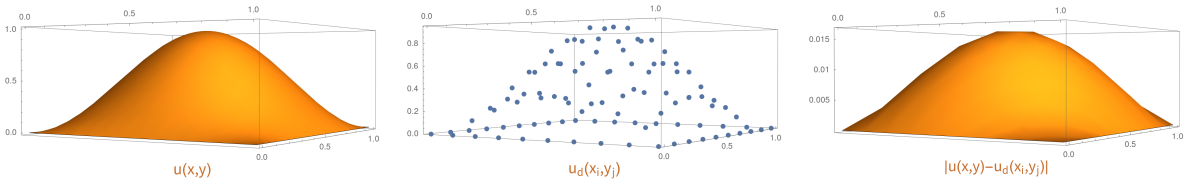

Figura 1.1: Solución de una ecuación diferencial mediante el método de elementos finitos usando una malla de 10×10 nodos.

Una de las características del método de elementos finitos es que posibilita mejorar la aproximación incrementando la cantidad de nodos y elementos utilizados. En la figura 1.2 se muestra la solución del mismo problema de la figura 1.1, pero con 100×100 nodos, lo cual mejora dramáticamente la aproximación, en consecuencia el error en la aproximación también baja y es de orden 10^{-4}.

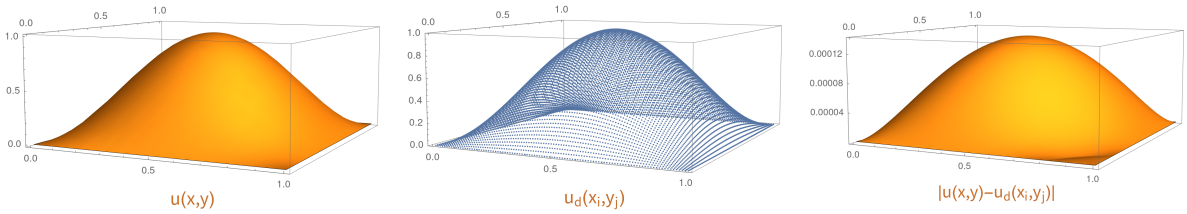

Figura 1.2: Solución de una ecuación diferencial mediante el método de elementos finitos usando una malla de 100×100 nodos.

Las unidades de procesamiento gráfico (GPU), son dispositivos con capacidades de paralelización masiva, que consisten en arreglos de miles de núcleos optimizados para operaciones numéricas (NVIDIA, 2018b). En particular, CUDA es una arquitectura de GPU introducida en el 2007 por la empresa NVIDIA, que ha sacado al mercado unidades de GPU especiales para procesamiento científico como la Tesla K20 o la Tesla V100 mostrada en la figura 1.3. En particular estos dispositivos son especialmente rápidos para algoritmos numéricos que exhiben gran nivel de paralelismo como las operaciones típicas de álgebra lineal, transformadas de Fourier, entre otras, lo que faculta que programas en GPU puedan tener *speedups* de hasta 100 órdenes de magnitud respecto a sus contrapartes en un CPU.

Figura 1.3: Tesla V100 PCIe, imagen tomada de (Nvidia, 2019).

Es importante mencionar que en la actualidad la super computadora número 1 del mundo (figura 1.4) obtiene la mayoría de su poder computacional de la tarjetas V100 de NVIDIA (TOP500, 2019),

lo cual evidencia la relevancia de las tarjetas GPU en el contexto de la computación científica a nivel mundial.

Figura 1.4: Super computadora Summit OakRidge, imagen tomada de (Wikimedia-Commons, 2018).

Estas características que tienen las unidades GPU, las hacen idóneas para la solución de ecuaciones diferenciales elípticas usando el método de elementos finitos con descomposición de dominios, ya que la solución de la ecuación en cada subdominio puede ser computada en paralelo -ya que son independientes entre ellas, y la gran mayoría de operaciones a realizar para computar la solución son operaciones de álgebra lineal, y estas operaciones las tarjetas GPU las manejan con superioridad frente a un CPU.

La idea de estudiar específicamente la ecuación diferencial de Poisson, aparte de que resuelve el problema de potenciales eléctricos en superficies isotrópicas, es que la ecuación diferencial ya ha sido ampliamente estudiada en forma teórica, existiendo numerosas implementaciones que computan la solución en CPU. Esto la convierte en el candidato ideal para probar los algoritmos desarrollados en GPU, ya que se puede comparar y validar resultados contra CPU. Además permite concentrarse en el estudio de estrategias de paralelización en GPU como: uso de memoria compartida, cantidad de transferencias de memoria de CPU a GPU, accesos a la memoria principal, puntos de sincronización, entre otros, ya que el rendimiento de un algoritmo en CUDA depende fuertemente de estos aspectos, como se menciona en (NVIDIA, 2018a).

1.1. Estado de la cuestión

Desde que se introdujo el precondicionador BDDC en (Dohrmann, 2003) éste ha permanecido como un tema de investigación relevante tanto desde el punto de vista teórico como práctico. En

(Li & Widlund, 2006) se reintrodujo el precondicionador desde otra perspectiva teórica proveyendo varias simplificaciones técnicas y relacionándolo con otros métodos anteriores de descomposición de dominios. La investigación desde el punto teórico se concentra esencialmente en una mejor escogencia de los parámetros del precondicionador y extender el precondicionador a otros elementos finitos. Por ejemplo en (Calvo, 2015) se introduce un algoritmo BDDC para resolver problemas en el espacio $H(rot)$ usando elementos finitos de Nédelec, permitiendo precondicionar sistemas que involucran el rotacional de un campo vectorial. La investigación desde el punto de vista práctico, se ha enfocado en los límites del precondicionador en sistemas computacionales. En (Kůs & Šístek, 2017) se investiga la combinación de métodos adaptativos de refinamiento de mallas con el precondicionador BDDC, en el cual se logran resolver problemas de hasta 10^9 grados de libertad usando 2048 cores.

El estudio del método de elementos finitos en la arquitectura GPU CUDA, se ha concentrado principalmente en varios aspectos: en el desarrollo y optimización de algoritmos para el cómputo del producto de matriz poco densa con un vector denso (SMpV) en GPU, en el ensamblado de la matriz que resulta de la discretización de elementos finitos, y en el precondicionado de estos sistemas. El estudio de algoritmos que ejecutan SMpV en CUDA se consideran cruciales, ya que esta operación domina la complejidad algorítmica de cada iteración de los métodos iterativos basados en subespacios de Krylov. El estudio de estos algoritmos inició en (Bell & Garland, 2008) y (Bell & Garland, 2009), en donde se analizaron varios formatos para el almacenamiento de matrices poco densas, y los rendimientos que proveían cada uno de estos formatos en la operación SMpV. El desarrollo de nuevos algoritmos para SMpV todavía sigue siendo un tema activo de investigación, algunos ejemplos de nuevos algoritmos que se han desarrollado en los últimos años corresponden a (Wong, Kuhl & Darve, 2015), (Weber, Bender, Schnoes, Stork & Fellner, 2013), (He & Gao, 2016), (Liu & Schmidt, 2015), (Torp, 2009), (Steinberger, Derlery, Zayer & Seidel, 2016), (Steinberger, Zayer & Seidel, 2017). Para un *review* de una gran cantidad de estos algoritmos, donde se discuten las fortalezas y debilidades de cada uno de los formatos presentados, véase (Filippone, Cardellini, Barbieri & Fanfarillo, 2017).

Respecto al estudio del ensamblado de matrices poco densas, en (Aspnäs, Signell & Westerholm, 2006) se establece que una buena alternativa corresponde a usar *hashtables* para el almacenamiento eficiente y modificación de entradas de la matriz. En (Alcantara, 2011) se estudian diferentes tipos de *hashtables* con sus respectivas ventajas y desventajas. En (Wong et al. 2015) también se discute el ensamblaje de estas matrices, sin embargo se discute mayoritariamente desde el punto de vista del ensamblaje de cada una de las matrices locales a los elementos.

Uno de los problemas que existe actualmente como expone (Filippone et al. 2017), es la falta de liberación de códigos fuente por parte de los desarrolladores, lo cual produce un entorpecimiento en el desarrollo de algoritmos en GPU, ya que usualmente los desarrolladores tienen que reimplementar operaciones que podrían estar consolidadas en bibliotecas. Otro de los problemas que existe, es que hay una gran cantidad de bibliotecas que se descontinuaron, por ejemplo en una gran cantidad de artículos se hace referencia a las bibliotecas CUSP y ModernGPU -desarrolladas por (Dalton, Bell, Olson & Garland, 2015) y (Baxter, 2016) respectivamente, como referentes de comparación, sin embargo estas no se actualizan en los respectivos repositorios de Github desde principios del 2018. Finalmente otro de los problemas con la mayoría de artículos es que el modelo de ejecución de la arquitectura de CUDA cambió con la introducción del *Independent thread scheduler* en la arquitectura Volta, que según (NVIDIA, 2018b) puede quebrar el funcionamiento de algunos códigos; lo cual significa que los códigos propuestos en cada uno de los artículos antes del 2017 pueden no ser inmediatamente compatibles con la arquitectura Volta en adelante.

1.2. Organización del documento

La organización del presente documento se detalla a continuación. En el capítulo 2 se introducen los elementos teóricos básicos para entender la solución de sistemas de ecuaciones por medio de métodos iterativos y directos, se presentan conceptos de espacios de Sobolev para justificar la existencia y unicidad de la ecuación de Poisson, se introducen los métodos de elementos finitos y finalmente se presenta el precondicionador *BDDC*. En el capítulo 3 se describen las arquitecturas computacionales en donde se ejecutó la plataforma computacional desarrollada, así como las pruebas realizadas para medir el rendimiento de la plataforma. En el capítulo 4 se presenta la plataforma computacional desarrollada junto con sus principales características y funciones para resolver el problema de Poisson. En el capítulo 5 se presentan y discuten los resultados obtenidos por las pruebas. Finalmente en el capítulo 6 se presentan las principales conclusiones del trabajo, junto con una serie de recomendaciones para trabajo futuro.

Capítulo 2

Marco teórico

En este capítulo se presentan los conceptos necesarios para el sustento teórico del trabajo. Las primeras dos secciones de este capítulo presentan métodos para la solución de sistemas de ecuaciones lineales con enfoques diametralmente opuestos. En la tercera sección se introducen los conceptos básicos del análisis funcional para justificar en la cuarta sección la existencia y unicidad de la solución de la ecuación diferencial. En la quinta sección se presenta el método de elementos finitos, junto con algunos resultados teóricos que garantizan la convergencia de la solución discretizada a la solución exacta de la ecuación. En la penúltima sección se presenta el precondicionador BDDC, junto con las cotas para el número de condición del operador precondicionado, y la descripción algorítmica de este precondicionador. En la última sección se introduce algunos conceptos básicos de computación paralela.

2.1. Métodos Directos

En esta sección se presentan dos métodos clásicos para la solución de sistemas ecuaciones, los cuales están inspirados en el bien conocido método de eliminación Gaussiana. Estos métodos tienen la particularidad que resuelven el sistema, primero encontrando una factorización de la matriz en términos de sistemas triangulares, los cuales son sencillos de resolver.

Definición 2.1.1. Sea $A \in \mathbb{R}^{n\times n}$ una matriz invertible, y sea $b \in \mathbb{R}^n$ un vector. Se dice que un algoritmo es un método directo para resolver el problema $Ax = b$, si bajo aritmética exacta el algoritmo produce la solución exacta del sistema de ecuaciones.

Solución de sistemas triangulares

Proposición 2.1.2. *Sea $U \in \mathbb{R}^{n\times n}$ una matriz triangular superior e invertible, la solución del sistema de ecuaciones $Ux = b$ está dada por:*

$$x_j = \left(b_j - \sum_{k=j+1}^{n} x_k u_{jk} \right) / u_{jj},\ j = n, n-1, \ldots, 1.$$

La complejidad algorítmica de encontrar la solución de este sistema es $\mathcal{O}\left(n^2\right)$.

Demostración. Véase (Trefethen & Bau III, 1997) págs 121-122. ■

Descomposición LU

Teorema 2.1.3. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz no singular, entonces existe una única matriz triangular inferior $L \in \mathbb{R}^{n\times n}$ con unos en la diagonal, y una única matriz triangular superior $U \in \mathbb{R}^{n\times n}$ tales que:*

$$A = LU.$$

Demostración. Véase (Trefethen & Bau III, 1997) págs 147-148. ■

Algoritmo 1. Sea $A \in \mathbb{R}^{n\times n}$ una matriz no singular. Se define el algoritmo para encontrar la descomposición LU como:

Algoritmo 1 Descomposición LU, véase algoritmo 20.1 en (Trefethen & Bau III, 1997)

```
    procedure LU(A)
Require: U ← A, L ← I
        for k := 1 do n − 1
            for j := k + 1 do n
                l_jk ← u_jk/u_kk
                for m := k do n
                    u_jm ← u_jm − l_jk u_km
                end for
            end for
        end for
    return L, U
    end procedure
```

Proposición 2.1.4. *La complejidad algorítmica de computar la descomposición LU es $\mathcal{O}\left(\frac{2}{3}n^3\right)$.*

Demostración. Véase (Trefethen & Bau III, 1997) págs 151-152. ■

Descomposición de Cholesky

Teorema 2.1.5. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica y definida positiva, entonces existe una única matriz triangular superior $U \in \mathbb{R}^{n\times n}$ tal que:*

$$A = U^T U.$$

Demostración. Véase teorema 23.1 en (Trefethen & Bau III, 1997). ■

Algoritmo 2. Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica y definida positiva. Se define el algoritmo para encontrar la descomposición de Cholesky como:

Algoritmo 2 Descomposición de Cholesky, véase algoritmo 23.1 en (Trefethen & Bau III, 1997)

```
    procedure CHOLESKY(A)
Require: U ← A
        for k := 1 do n
            for j := k + 1 do n
                for i := j do n
                    u_ji ← u_ji − u_ki u_kj / u_kk
                end for
            end for
            for j := k do n
                u_kj ← u_kj / √u_kk
            end for
        end for
    return U
    end procedure
```

Proposición 2.1.6. *La complejidad algorítmica de computar la descomposición de Cholesky es $\mathcal{O}\left(\frac{1}{3}n^3\right)$.*

Demostración. Véase (Trefethen & Bau III, 1997) pág 175. ■

Observación 2.1.7. La descomposición de Cholesky tiene una complejidad algorítmica 2 veces menor que la descomposición LU. Esto la hace preferible como algoritmo frente a la descomposición LU cuando la matriz es simétrica y definida positiva.

2.2. Métodos Iterativos

En esta sección se van a introducir dos métodos numéricos modernos para encontrar la solución de un sistema de ecuaciones, que a diferencia de los métodos directos, los métodos iterativos de esta sección

solo buscan aproximar la solución de la ecuación, reduciendo en muchas ocasiones la complejidad algorítmica respecto a los métodos directos.

Definición 2.2.1. Sea $A \in \mathbb{R}^{n\times n}$ una matriz, y sea $b \in \mathbb{R}^n$ un vector. Si $x^* \in \mathbb{R}^n$ es una solución del problema

$$Ax = b,$$

se dice que un algoritmo $\mathcal{A}$ es un método iterativo para resolver el problema $Ax = b$, si:

$$\lim_{k\to\infty} \|x_k - x^*\|_2 = 0,$$

donde $x_k = \mathcal{A}(A, b, x_0, x_1, \ldots, x_{k-1})$ es el resultado de la $k-$ésima iteración.

Definición 2.2.2. Sea $A \in \mathbb{R}^{n\times n}$ una matriz, se define el número de condición de A como:

$$\kappa(A) = \|A\|_2\|A^{-1}\|_2,$$

donde $\|A\|_2 = \sup_{\|x\|_2=1} \|Ax\|_2$. Además se dice que A está mal-condicionada, si su número de condición es alto.

Observación 2.2.3. En las siguientes secciones, se evidenciará la importancia del número de condición $\kappa(A)$.

Gradiente conjugado

El método de gradiente conjugado introducido originalmente por (Hestenes & Stiefel, 1952), es considerado como uno de los pilares de la computación científica, ya que revolucionó la forma de resolver ciertos tipos de sistemas lineales, los cuales serían computacionalmente imposibles de resolver por métodos directos.

Algoritmo 3. Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica y definida positiva. Se define el método iterativo gradiente conjugado (CG) como:

Algoritmo 3 Gradiente conjugado, véase algoritmo 38.1 en (Trefethen & Bau III, 1997)

procedure CG(A, b, k_{max}, ϵ)
Require: $x_0 \leftarrow \vec{0},\ r_0 \leftarrow b,\ p_0 \leftarrow r_0,\ k \leftarrow 0$
  for $k \leq k_{max}$ **do**
    $\alpha_k \leftarrow (r_k^T r_k)/(p_k^T A p_k)$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
    if $\|r_k\| \leq \epsilon$ **then**
      return x_{k+1}
    end if
    $\beta_k \leftarrow (r_{k+1}^T r_{k+1})/(z_k^T r_k)$
    $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
    $k \leftarrow k + 1$
  end for
return x_k
end procedure

Observación 2.2.4. Por el resto de la subsección se asume que $A \in \mathbb{R}^{n\times n}$ es simétrica y definida positiva, y x_k, p_k, r_k denotan los vectores definidos en el Algoritmo 3.

Definición 2.2.5. Se define la A-norma de $x \in \mathbb{R}^n$ como:

$$\|x\|_A = \sqrt{x^T A x}.$$

Definición 2.2.6. Se define el l-ésimo espacio de Krylov generado por A y b como:

$$\mathcal{K}_l = \left\langle b, Ab, \ldots, A^{l-1}b \right\rangle.$$

Teorema 2.2.7. *Suponga que $r_{l-1} \neq 0$, entonces se tiene que:*

$$\begin{aligned}\mathcal{K}_l &= \langle x_1, x_2, \ldots, x_l \rangle \\ &= \langle p_0, p_1 \ldots, p_{l-1} \rangle \\ &= \langle r_0, r_1, \ldots, r_{l-1} \rangle\end{aligned}$$

Más aún:

$$r_l^T r_j = 0 \ (j < l),$$
$$p_l^T A p_j = 0 \ (j < l).$$

Demostración. Véase teorema 38.1 en (Trefethen & Bau III, 1997). ■

Teorema 2.2.8. *Suponga que $r_{l-1} \neq 0$, entonces se tiene que x_l es el único punto de $\mathcal{K}_l$ que minimiza $\|e_l\|_A = \|x^* - x_l\|_A$. Más aún:*

$$\|e_l\|_A \leq \|e_{l-1}\|_A,$$

y además existe $m \leq n$ tal que $e_m = 0$.

Demostración. Véase teorema 38.2 (Trefethen & Bau III, 1997). ■

Teorema 2.2.9. *Si A tiene k autovalores distintos, entonces el método de Gradiente Conjugado converge en a lo sumo k iteraciones bajo aritmética exacta.*

Demostración. Véase teorema 38.4 (Trefethen & Bau III, 1997). ■

Teorema 2.2.10. *Sea κ el número de condición de la matriz, entonces:*

$$\frac{\|e_l\|_A}{\|e_0\|_A} \leq 2\left[\left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^l + \left(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1}\right)^{-l}\right]^{-1} \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^l.$$

Demostración. Véase teorema 38.5 (Trefethen & Bau III, 1997). ■

Proposición 2.2.11. *La complejidad algorítmica del método gradiente conjugado es $\mathcal{O}(k\,2nzv)$, donde k es la cantidad máxima de iteraciones permitidas en el método, y nzv es la cantidad de valores no cero de la matriz A.*

Demostración. Véase (Trefethen & Bau III, 1997) pág 295. ■

MINRES

La revolución causada por el método de gradiente conjugado, impulsó la búsqueda de algoritmos similares para encontrar iterativamente la solución de sistemas de ecuaciones lineales sin la restricción de que la matriz sea definida positiva. El método *MINRES* introducido en (Paige & Saunders, 1975), remueve la restricción de que la matriz sea definida positiva, pero restringe a que el sistema sea simétrico. En (Saad & Schultz, 1986) se introduce el método *GMRES* el cual es una generalización de *MINRES*, y permite resolver sistema generales de ecuaciones. En esta sección se presenta el algoritmo

de *MINRES* en lugar de *GMRES*, ya que es menos costoso computacionalmente y para efectos del presente trabajo el sistema más general a resolver es simétrico.

Algoritmo 4. Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica. Se define un paso de la iteración de Lanczos como:

Algoritmo 4 Iteración de Lanczos, véase algoritmo 36.1 en (Trefethen & Bau III, 1997)

function LANCZOS(A, β_k, v_k, v_{k-1})
  $p = Av_k$
  $\alpha_k = v_k^T p$
  $v_{k+1} = p - \beta_k v_{k-1} - \alpha_k v_k$
  $\beta_{k+1} = \|v_{k+1}\|$
  if $\beta_{k+1} \neq 0$ **then**
    $v_{k+1} = v_{k+1}/\beta_{k+1}$
  end if
return $\alpha_k, \beta_{k+1}, v_{k+1}$
end function

Algoritmo 5. Sean a y b números reales, se define la función auxiliar *SymOrtho* como:

Algoritmo 5 Función *SymOrtho*, véase (Sou-Cheng, 2006)

function SYMORTHO(a, b)
  if $b = 0$ **then**
    $s \leftarrow 0$
    $r \leftarrow |a|$
    if $a = 0$ **then** $c \leftarrow 1$ **else** $c \leftarrow \text{sign}(a)$
  else if $a = 0$ **then**
    $c \leftarrow 0$
    $s \leftarrow \text{sign}(b)$
    $r \leftarrow |b|$
  else if $|b| > |a|$ **then**
    $\tau \leftarrow a/b$
    $s \leftarrow \text{sign}(b)/\sqrt{1+\tau^2}$
    $c \leftarrow s\tau$
    $r \leftarrow b/s$
  else if $|a| > |b|$ **then**
    $\tau \leftarrow b/a$
    $c \leftarrow \text{sign}(a)/\sqrt{1+\tau^2}$
    $s \leftarrow c\tau$
    $r \leftarrow a/c$
  end if
return c, s, r
end function

Algoritmo 6. Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica, y $b \in \mathbb{R}^n$ un vector. Se define el método iterativo MINRES como:

Algoritmo 6 MINRES, véase (Sou-Cheng, 2006)

procedure MINRES(A, b, k_{max})
  $\beta_1 \leftarrow \|b\|$, $\phi_0 \leftarrow \beta_1$, $\delta_1^{(1)} \leftarrow 0$, $c_0 \leftarrow -1$, $s_0 \leftarrow 0$, $v_1 \leftarrow b/\beta_1$, $d_0 \leftarrow d_{-1} \leftarrow x_0 \leftarrow v_0 \leftarrow \vec{0}$, $k \leftarrow 1$
  for $k \leq k_{max}$ **do**
    $\{\alpha_k, \beta_{k+1}, v_{k+1}\} \leftarrow$ LANCZOS$(A, \beta_k, v_k, v_{k-1})$
    $\delta_k^{(2)} \leftarrow c_{k-1}\delta_k^{(1)} + s_{k-1}\alpha_k$
    $\gamma_k^{(1)} \leftarrow s_{k-1}\delta_k^{(1)} - c_{k-1}\alpha_k$
    $\epsilon_{k+1} \leftarrow s_{k-1}\beta_{k+1}$
    $\delta_{k+1}^{(1)} \leftarrow -c_{k-1}\beta_{k+1}$
    $\{c_k, s_k, \gamma_k^{(2)}\} \leftarrow$ SYMORTHO$(\gamma_k^{(1)}, \beta_{k+1})$
    $\tau \leftarrow c_k\phi_{k-1}$
    $\phi_k \leftarrow s_k\phi_{k-1}$
    if $\gamma_k^{(2)} \neq 0$ **then**
      $d_k \leftarrow (v_k - \delta_k^{(2)}d_{k-1} - \epsilon_k d_{k-2})/\gamma_k^{(2)}$
      $x_k \leftarrow x_{k-1} + \tau d_k$
    end if
    $k \leftarrow k + 1$
  end forreturn x_k, ϕ_k
end procedure

Definición 2.2.12. Se define el espacio de polinomios $\tilde{P}_k = \{p \in P_k[x] : \deg(p) \leq k,\ p(0) = 1\}$.

Teorema 2.2.13. *El método MINRES resuelve en la k-ésima iteración el problema de encontrar* $p_k \in \tilde{P}_k$ *tal que:*

$$\|p_k(A)b\| = \min\{\|q_k(A)b\| : q_k \in \tilde{P}_k\}.$$

Demostración. Véase (Trefethen & Bau III, 1997) pág 269. ■

Teorema 2.2.14. *Sea* $A \in \mathbb{R}^{n\times n}$ *una matriz simétrica, y sea* $r_k = b - Ax_k$ *el k-ésimo residuo producido por MINRES. Entonces:*

$$\begin{aligned} \|r_{k+1}\| &\leq \|r_k\|, \\ \|r_n\| &= 0, \\ \frac{\|r_n\|}{\|b\|} &\leq \inf_{p_k \in \tilde{P}_k} \|p_k(A)\|. \end{aligned}$$

Demostración. Véase (Trefethen & Bau III, 1997) pág 270. ■

Teorema 2.2.15. *Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica y diagonalizable, con $A = C\Lambda C^{-1}$, entonces*

$$\frac{\|r_n\|}{\|b\|} \leq \kappa(C) \inf_{p_k \in \tilde{P}_k} \sup_{x\in\Lambda(A)} \|p_k(x)\|.$$

Demostración. Véase teorema 35.2 en (Trefethen & Bau III, 1997). ■

Observación 2.2.16. Del teorema anterior se deduce que si A tiene k autovalores distintos, entonces MINRES converge en a lo sumo k iteraciones en aritmética exacta.

Precondicionadores

De las secciones de *Gradiente Conjugado* y *MINRES*, se puede observar que hay una estrecha relación entre la velocidad de convergencia de los métodos iterativos y el número de condición de la matriz A. Entonces la pregunta natural es, cómo resolver sistemas mal-condicionados sin tener que ejecutar una gran cantidad de iteraciones para obtener una aproximación tolerable. La respuesta es resolver un sistema auxiliar, que provea la misma solución y esté mejor condicionado.

Definición 2.2.17. Se define un precondicionador izquierdo como una matriz $M \in \mathbb{R}^{n\times n}$ que cumple:

- Los autovalores de $M^{-1}A$ están agrupados cerca del 1 y $\|M^{-1}A - I\|_2 \approx 0$.
- Dado un vector y, $M^{-1}y$ es fácil de computar.

Observación 2.2.18. Sea $A \in R^{n\times n}$ una matriz invertible, y suponga que M es un precondicionador izquierdo, entonces para resolver el sistema:

$$Ax = b,$$

se resuelve:

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

donde este sistema auxiliar posee la misma solución que el sistema original. Además observe que si M es un precondicionador adecuado, el sistema modificado va a tener un mejor número de condición, convergiendo en una menor cantidad iteraciones.

Proposición 2.2.19. *Sea A una matriz simétrica y definida positiva, sea M un precondicionador para el sistema lineal. Para resolver el sistema $Ax = b$, se utiliza gradiente conjugado en el sistema:*

$$M^{-1/2}AM^{-1/2}v = M^{-1/2}b, \text{ con } v = M^{1/2}x$$

donde $M^{1/2}$ cumple $M = M^{1/2}M^{1/2^T}$.

Demostración. Véase (Toselli & Widlund, 2006) pág 406. ■

Algoritmo 7. Sea $A \in \mathbb{R}^{n\times n}$ una matriz simétrica y definida positiva, y sea M un precondicionador. El método precondicionado de gradiente conjugado de la proposición 2.2.19 corresponde a:

Algoritmo 7 Gradiente conjugado precondicionado, véase (Toselli & Widlund, 2006) pág 406.

procedure PCG(A, b, k_{max}, ϵ)
Require: $x_0 \leftarrow \vec{0},\ r_0 \leftarrow b,\ z_0 \leftarrow M^{-1}r_0,\ p_0 \leftarrow z_0,\ k \leftarrow 0$
  for $k \leq k_{max}$ **do**
    $\alpha_k \leftarrow (r_k^T z_k)/(p_k^T A p_k)$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
    if $\|Ax_k - b\| \leq \epsilon$ **then**
      return x_{k+1}
    end if
    $z_{k+1} \leftarrow M^{-1}r_{k+1}$
    $\beta_k \leftarrow (z_{k+1}^T r_{k+1})/(z_k^T r_k)$
    $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
    $k \leftarrow k + 1$
  end for
return x_k
end procedure

2.3. Espacios de Sobolev

Definición 2.3.1. Sean Ω un dominio en $\mathbb{R}^n$ y $\alpha \in \mathbb{N}^n$ un multi-índice, se dice que $f \in L^1_{\text{loc}}(\Omega)$ tiene derivada débil $D^\alpha_w f$ si $\exists g \in L^1_{\text{loc}}(\Omega)$ tal que:

$$\int_\Omega g(x)\phi(x)dx = (-1)^{|\alpha|}\int_\Omega f(x)D^\alpha\phi(x)dx, \quad \forall\phi \in \mathcal{C}_0^\infty(\Omega).$$

En tal caso se dice que $D^\alpha_w f = g$.

Definición 2.3.2. Sean Ω un dominio en $\mathbb{R}^n$, $k \in \mathbb{N}$ y $1 \leq p \leq \infty$. Se define el espacio de Sobolev $W_p^k(\Omega)$ como:

$$W_p^k(\Omega) = \{f \in L^1_{\text{loc}}(\Omega) : \|f\|_{W_p^k} < \infty\}$$

donde para $1 \leq p < \infty$ se define:

$$\|f\|_{W_p^k(\Omega)} = \left(|f|^p_{W_p^k(\Omega)} + \sum_{|\alpha|<k} \|D_w^\alpha f\|^p_{L^p_\Omega} \right)^{1/p},$$

$$|f|_{W_p^k(\Omega)} = \left(\sum_{|\alpha|=k} \|D_w^\alpha f\|^p_{L^p_\Omega} \right)^{1/p},$$

y para $p = \infty$:

$$\|f\|_{W_\infty^k(\Omega)} = \max_{|\alpha|<k} \left\{ \|D_w^\alpha f\|_{L^\infty_\Omega}, |f|_{W_\infty^k(\Omega)} \right\},$$

$$|f|_{W_\infty^k(\Omega)} = \max_{|\alpha|=k} \left\{ \|D_w^\alpha f\|_{L^\infty_\Omega} \right\}.$$

Proposición 2.3.3. *Sea Ω un dominio en $\mathbb{R}^n$, $k \in \mathbb{N}$ y $1 \leq p \leq \infty$, entonces:*

1. $|\cdot|_{W_p^k(\Omega)}$ *es una seminorma en* $W_p^k(\Omega)$.
2. $\|\cdot\|_{W_p^k(\Omega)}$ *es una norma en* $W_p^k(\Omega)$.
3. $W_p^k(\Omega)$ *es un espacio de Banach con* $\|\cdot\|_{W_p^k(\Omega)}$.

Demostración. Véase teorema 1.3.2 en (Brenner & Scott, 2007). ■

Proposición 2.3.4. *Sea Ω un conjunto abierto de $\mathbb{R}^n$, $k \in \mathbb{N}$ y $1 \leq p < \infty$, entonces:*

$$C^\infty(\Omega) \bigcap W_p^k(\Omega) \text{ es denso en } W_p^k(\Omega).$$

Demostración. Véase teorema 1.3.4 en (Brenner & Scott, 2007). ■

Definición 2.3.5. Sea Ω un dominio en $\mathbb{R}^n$, se dice que su frontera $\partial\Omega$ es Lipschitz, si existe una colección de conjuntos abiertos $\{O_i, i \in I\}$, un $\epsilon > 0$, un $N \in \mathbb{N}$ y $0 \leq M \leq \infty$, tal que:

1. $\forall x \in \partial\Omega \; \exists i \in I$ tal que: $B(x, \epsilon) \subset O_i$.

2. No más de N de los O_i se intersecan de manera no trivial.

3. $O_i \cap \Omega = O_i \cap \Omega_i$, donde Ω_i es el gráfico de una función Lipschitz ϕ_i, que satisface $\|\phi_i\|_{Lip(\mathbb{R}^{n-1})} \leq M$, donde $\|f\|_{Lip(\Omega)} = \|f\|_{L_\infty} + \sup\left\{\dfrac{|f(x)-f(y)|}{|x-y|} : x, y \in \Omega, x \neq y\}\right\}$.

Teorema 2.3.6. (Desigualdad de Sobolev). *Sea Ω un dominio en $\mathbb{R}^n$ con frontera Lipschitz, y sean $1 \leq p < \infty$ y $k \in \mathbb{N}^*$ tal que:*

$$k \geq n \quad \text{para } p = 1,$$
$$k > n/p \quad \text{para } p > 1.$$

Entonces $\exists C > 0$ tal que $\forall f \in W_p^k$ se tiene que:

$$\|f\|_{L^\infty(\Omega)} \leq C\|f\|_{W_p^k(\Omega)}.$$

Más aún $\exists u \in C^0(\Omega) \bigcap L^\infty(\Omega)$ tal que u pertenece a la misma clase de equivalencia que f en $L^\infty(\Omega)$.

Demostración. Véase teorema 1.4.6 en (Brenner & Scott, 2007). ■

Observación 2.3.7. La desigualdad de Sobolev establece que dadas suficientes condiciones de regularidad sobre las derivadas débiles las funciones se pueden considerar como acotadas y continuas. Este hecho convierte a estos espacios en lugares atractivos para buscar soluciones de ecuaciones diferenciales, ya que usualmente se busca la continuidad de la solución.

Definición 2.3.8. Sea Ω un dominio en $\mathbb{R}^n$, se define:

$$\mathring{W}_p^k(\Omega) = \left\{v \in W_p^k(\Omega) \ : \ D_w^\alpha v|_{\partial\Omega} = 0 \text{ en } L^2(\partial\Omega),\ \forall |\alpha| < k\right\}$$

2.4. Formulación variacional de un problema de valores frontera

Notación 2.4.1. *Cuando $p = 2$, se denota $W_2^k(\Omega)$ por $H^k(\Omega)$.*

Proposición 2.4.2. *$H^k(\Omega)$ es un espacio de Hilbert.*

Definición 2.4.3. Sea $a(\cdot,\cdot)$ una forma bilineal acotada y definida positiva en un espacio normado H, se dice que $a(\cdot,\cdot)$ es coerciva en $V \leq H$, si $\exists \alpha > 0$ tal que:

$$a(v,v) \geq \alpha \|v\|_H^2, \, \forall v \in V.$$

Proposición 2.4.4. *Sea H un espacio de Hilbert, $a(\cdot,\cdot)$ una forma bilineal en H, simétrica y coerciva en $V \leq H$, donde $V < H$ subespacio cerrado, entonces $(V, a(\cdot,\cdot))$ es un espacio de Hilbert.*

Demostración. Véase proposición 2.5.3 en (Brenner & Scott, 2007). ■

Definición 2.4.5. Sea $a(\cdot,\cdot)$ una forma bilineal en un espacio de Hilbert H, con $a(\cdot,\cdot)$ simétrica y coerciva, se define la norma energía como:

$$\|v\|_E = \sqrt{a(v,v)}.$$

Problema variacional simétrico

Sea H un espacio de Hilbert, $a(\cdot,\cdot)$ una forma bilineal simétrica y coerciva en $V \leq H$, donde $V < H$ subespacio cerrado, el problema variacional simétrico se define como:

$$\text{Dado } F \in V^*, \text{ encuentre } u \in V \text{ tal que: } a(u,v) = F(v), \, \forall v \in V.$$

Teorema 2.4.6. *Suponga que se cumplen las hipótesis de un problema variacional simétrico, entonces existe un único $u \in V$ que resuelve el problema variacional.*

Demostración. Véase teorema 2.5.6 en (Brenner & Scott, 2007). ■

Problema de aproximación de Ritz-Galerkin

Suponga las hipótesis del problema variacional simétrico, y sea $V_h < V$, donde $\dim V_h < \infty$. El problema de Ritz-Galerkin es:

$$\text{Dado } F \in V^*, \text{ encuentre } u_h \in V_h \text{ tal que: } a(u_h,v) = F(v), \, \forall v \in V_h.$$

Corolario 2.4.7. *El problema de Ritz-Galerkin, dadas las hipótesis del problema variacional simétrico, tiene solución única.*

Proposición 2.4.8. (Ortogonalidad fundamental de Galerkin). *Sean u y u_h soluciones a los problemas variacional simétrico y de Ritz-Galerkin, entonces:*

$$a(u - u_h, v) = 0, \ \forall v \in V.$$

Además se tiene que:

$$\|u - u_h\|_E = \min_{v \in V_h} \|u - v\|_E.$$

Teorema 2.4.9. (Céa). *Sean u y u_h soluciones a los problemas variacional simétrico y de Ritz-Galerkin, entonces:*

$$\|u - u_h\|_V = \frac{C}{\alpha} \min_{v \in V_h} \|u - v\|_V,$$

donde α es la constante de coercividad de la forma bilineal $a(\cdot, \cdot)$.

Demostración. Véase teorema 2.8.1 en (Brenner & Scott, 2007). ■

Ecuación de Poisson

Definición 2.4.10. Sea Ω un dominio en $\mathbb{R}^n$, la ecuación de Poisson con condiciones de frontera de Dirichlet corresponde al problema:

$$\begin{cases} -\Delta u = f \ \text{ en } \Omega \\ u = g \ \text{ en } \partial\Omega \end{cases}$$

donde $f \in L^2(\Omega)$, y $g \in C(\partial\Omega)$.

Teorema 2.4.11. *Sea $a(v, w) = \int_\Omega \nabla v \cdot \nabla w \, dx$. Si u resuelve*

$$a(u, v) = \int_\Omega f \, v \, dx \qquad \forall v \in \{v : v \in H^1, v|_{\partial\Omega} = 0\},$$

entonces u es una solución débil al problema de Poisson con condiciones de frontera nulas.

Demostración. Véase proposición 5.1.7 en (Brenner & Scott, 2007). ■

Corolario 2.4.12. *La existencia y unicidad para el problema de Poisson con condiciones de frontera nulas se sigue de la existencia y unicidad de la solución del problema variacional simétrico.*

Observación 2.4.13. Suponga que $g \in H^1(\Omega)$, entonces la solución del problema:

$$a(\tilde{u}, v) = \int_\Omega f\, v\, dx - a(g, v) \qquad \forall v \in \{v : v \in H^1, v|_{\partial\Omega} = 0\},$$

corresponde a la solución del problema de Poisson con condiciones frontera g.

2.5. Espacio de elementos finitos

Definición 2.5.1.

- Se define el dominio de los elementos $\Omega \subset \mathbb{R}^n$, como un conjunto compacto con $int\Omega \neq \emptyset$, y frontera suave a trozos.
- Se define el espacio de funciones forma como un espacio vectorial finito dimensional $\mathcal{P}$, que consiste de funciones definidas en Ω.
- Se define el conjunto de variables nodales $\mathcal{N} = \{N_1, \ldots, N_k\}$ como una base de $\mathcal{P}^*$.

Al triplete $(\Omega, \mathcal{P}, \mathcal{N})$ se le llama un elemento finito.

Observación 2.5.2. Pese a que no se dice en la definición, en general se asume que las variables nodales pertenecen a un espacio de Sobolev.

Definición 2.5.3. Sea $(\Omega, \mathcal{P}, \mathcal{N})$ un elemento finito, a la base $\{\phi_1, \cdots, \phi_k\}$ de $\mathcal{P}$ dual a $\mathcal{N}$, se le llama la base nodal de $\mathcal{P}$.

Definición 2.5.4. Sea $(\Omega, \mathcal{P}, \mathcal{N})$ un elemento finito, y $\{\phi_1, \ldots, \phi_k\}$ la base nodal, se define el interpolante $\mathcal{I}_\Omega$ para una función f como:

$$\mathcal{I}_\Omega f = \sum_{i=1}^{k} N_i(f)\phi_i,$$

donde se asume que f está bien definida en los N_i.

Proposición 2.5.5. *El interpolante cumple que:*

1. $\mathcal{I}_\Omega$ *es un operador lineal.*
2. $N_i(\mathcal{I}_\Omega(f)) = N_i(f),\ \forall i = 1, \ldots, k.$

Definición 2.5.6. Una subdivisión τ de un dominio Ω, es una colección de conjuntos $\{\Omega_i\}$ tales que:

1. $int\Omega_i \cap int\Omega_j = \emptyset$ si $i \neq j$
2. $\cup\Omega_i = \bar{\Omega}$.

Definición 2.5.7. Sea Ω un dominio con una subdivisión τ, donde cada dominio en la subdivisión tiene asociado un elemento finito $(\Omega_i, \mathcal{P}_i, \mathcal{N}_i)$. Se define el interpolante global en (Ω, τ) como el operador lineal:

$$\mathcal{I}_\tau f|_{\Omega_i} = \mathcal{I}_{\Omega_i} f.$$

Definición 2.5.8. Sean $(\Omega, \mathcal{P}, \mathcal{N})$, $(\hat{\Omega}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$ dos elementos finitos, estos se dicen afín-equivalentes, si $\exists F : \mathbb{R}^n \to \mathbb{R}^n$ tal que $F(x) = Ax + b$, con $A \in GL_n,\ b \in \mathbb{R}^n$, y se tiene que:

1. $F(\Omega) = \hat{\Omega}$
2. $F^*(\hat{\mathcal{P}}) = \mathcal{P}$
3. $F_*(\mathcal{N}) = \hat{\mathcal{N}}$

donde $F^*,\ F_*$ son el pullback y el pullforward respectivamente. Esta relación se denota por $(\Omega, \mathcal{P}, \mathcal{N}) \sim_F (\hat{\Omega}, \hat{\mathcal{P}}, \hat{\mathcal{N}})$.

Definición 2.5.9. Se dice que el interpolante tiene orden de continuidad r, si:

$$\mathcal{I}_\tau f \in C^r(\Omega),\ \forall f \in C^m(\Omega),$$

donde m es el orden de la mayor derivada parcial que aparece en los $\mathcal{N}_i$.

Cotas para el error de interpolación

Definición 2.5.10. Sea $\Omega \subset \mathbb{R}^n$ un dominio, se dice que tiene forma de estrella respecto a una bola $B \subset \Omega$, si $\forall x \in \Omega \Rightarrow \overline{convex}(B \cup \{x\}) \subset \Omega$.

Definición 2.5.11. Sea $\Omega \subset \mathbb{R}^n$ un dominio, y $\{\tau^h :\ h \in (0, 1]\}$ una familia de subdivisiones tales que:

$$\text{máx}\{\text{diam}T : T \in \tau^h\} \leq h\text{diam}\Omega.$$

- La familia se dice cuasi-uniforme si $\exists \rho > 0$, tal que:

$$\text{mín}\{\text{diam}B_T : T \in \tau^h\} \geq \rho\, h\, \text{diam}\Omega \ \forall h \in (0,1].$$

- La familia se dice que es regular si $\exists \rho > 0$, tal que:

$$\text{diam}B_T \geq \rho \text{diam}T \ \forall h \in (0,1], \ \forall T \in \tau^h, \tag{2.1}$$

donde B_T es la bola más grande que hace que T tenga forma de estrella.

Definición 2.5.12. Sea Ω un dominio, y sea $\rho_{\text{máx}} = \sup\{\rho : \Omega \text{ es estrellado respecto } B(x_0, \rho)\}$, se define el parámetro de regularidad de Ω como:

$$\gamma = \frac{\text{diam}\Omega}{\rho_{\text{máx}}}.$$

Proposición 2.5.13. *Una familia de subdivisiones $\{\tau^h : \ h \in (0,1]\}$ es regular si y solo si $\gamma_{T,h} \leq M$, $\forall T \in \tau^h$, $\forall h \in (0,1]$ y $M \in \mathbb{R}^+$.*

Teorema 2.5.14. *Sea $\{\tau^h : \ h \in (0,1]\}$ una familia regular de subdivisiones de un dominio poliédrico Ω de $\mathbb{R}^n$. Sea $(K, \mathcal{P}, \mathcal{N})$ un elemento finito de referencia que cumple:*

- *K es estrellado respecto a una bola.*
- *$P_{m-1} \subset \mathcal{P} \subset W^m_\infty$, donde P_{m-1} son los polinomios de grado menor o igual a m.*
- *$\mathcal{N} \subset C^l(\bar{\Omega})^*$,*

para algunos l, m, p. Además suponga que para todo $T \in \tau^h$, $0 < h \leq 1$, $(T, \mathcal{P}_T, \mathcal{N}_T) \sim_F (K, \mathcal{P}, \mathcal{N})$. Entonces existe $C > 0$ una constante que depende solamente en n, m, p, ρ (donde ρ es el de la ecuación 2.1) y el elemento de referencia tal que para $0 \leq s \leq m$ se tiene que:

$$\left(\sum_{T \in \tau^h} \|v - \mathcal{I}^h v\|^p_{W^s_p(T)} \right)^{1/p} \leq C h^{m-s} |v|_{W^m_p(\Omega)}$$

para todo $v \in W^m_p(\Omega)$.

Demostración. Véase teorema 4.4.20 (Brenner & Scott, 2007). ■

Elemento lineal de Lagrange en $\mathbb{R}^2$

Para dos dimensiones el elemento de Lagrange sobre elementos triangulares está dado por:

- K un triángulo, con v_1, v_2, v_3 sus vértices.
- $\mathcal{P}$ los polinomios lineales a trozos definidos en Ω.
- $\mathcal{N} = \{N_1, N_2, N_3\}$ donde $N_i(f) = f(v_i)$ con $f : \mathbb{R}^2 \to \mathbb{R}$.

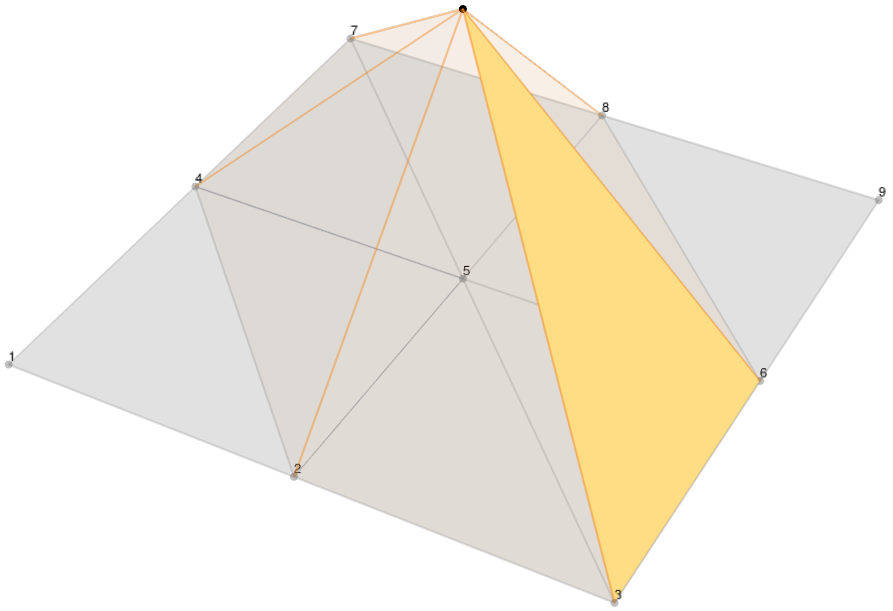

Figura 2.1: Elemento de la base de las funciones forma, con elemento de Lagrange.

Proposición 2.5.15. *El elemento de Lagrange produce un interpolante con orden de continuidad* 0.

Proposición 2.5.16. *Todo elemento de Lagrange es afín-equivalente entre sí.*

Discretización mediante elementos finitos

Definición 2.5.17. Sea $a(\cdot, \cdot) : V \times V \to \mathbb{R}$ una forma bilineal, y sea $F : V \in \mathbb{R}$ un funcional lineal, donde el espacio V contiene al espacio de funciones forma $\mathcal{P}$. Sea $\{\phi_1, \ldots, \phi_k\}$ una base del espacio de funciones forma, se define la discretización del problema de Ritz-Galerkin como:

$$Au = f,$$

donde $A_{ij} = a(\phi_i, \phi_j)$, y $f_i = F(\phi_i)$.

Observación 2.5.18. De los teoremas 2.4.9 y 2.5.14, se puede garantizar que disminuyendo el tamaño de la malla, la solución del problema discretizado converge a la solución del problema variacional simétrico.

Teorema 2.5.19. *Si la forma bilineal $a(\cdot,\cdot)$ proviene de un problema variacional elíptico de segundo orden (e.g., ecuación de Poisson), y se tiene una malla regular de tamaño h de un dominio Ω, entonces el número de condición de la discretización cumple que:*

$$\kappa(A) = \mathcal{O}(h^{-2}).$$

Demostración. Véase teorema B.32 en (Toselli & Widlund, 2006). ■

2.6. BDDC

Descomposición de Dominios

Definición 2.6.1. Sea Ω una región en $\mathbb{R}^n$, se define una partición de la región como una familia de conjuntos $\mathcal{F} = \{\Omega_1, \ldots, \Omega_N\}$, que cumple:

- $\Omega_i \subset \Omega$
- $\cup\Omega_i = \Omega$
- $\operatorname{int}\Omega_i \cap \operatorname{int}\Omega_j = \emptyset, \quad \forall i \neq j$

Se define la interfaz de la región Ω como:

$$\Gamma = \bigcup \Gamma_i, \text{ donde}$$
$$\Gamma_i = \bigcup_{i \neq j} \left(\partial\Omega_i \cap \partial\Omega_j\right).$$

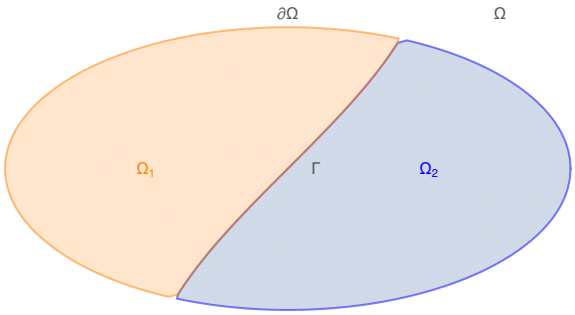

Figura 2.2: Región Ω particionada en 2, donde Ω_1, Ω_2 son los subdominios y Γ es la interfaz.

Notación 2.6.2. *Los nodos interiores al subdominio se van a identificar por un subíndice I, y los nodos en la interfaz se van a identificar por Γ.*

Observación 2.6.3. Por el resto de la sección se va a asumir que se está trabajando con el problema de Poisson homogéneo, y que N es la cantidad de subdominios.

Definición 2.6.4. Se definen las contribuciones locales de cada subdominio, que resultan de discretizar el problema variacional en Ω_i como:

$$A^{(i)} = \begin{pmatrix} A_{II}^{(i)} & A_{I\Gamma}^{(i)} \\ A_{\Gamma I}^{(i)} & A_{\Gamma\Gamma}^{(i)} \end{pmatrix}, \qquad f^{(i)} = \begin{bmatrix} f_I^{(i)} \\ f_\Gamma^{(i)} \end{bmatrix}.$$

Notación 2.6.5. *Considere las siguientes proyecciones entre espacios:*

- $R^{(i)} : \Omega \to \Omega_i$.
- $R_\Gamma^{(i)} : \Gamma \to \Gamma_i$.
- $R_\Gamma : \Omega \to \Gamma$.

Proposición 2.6.6. *Sean A y f las discretizaciones de la sección 2.5, entonces se tiene que:*

$$A = \sum {R^{(i)}}^T A^{(i)} R^{(i)},$$
$$f = \sum {R^{(i)}}^T f^{(i)}.$$

Demostración. Véase sección 1.3.1 de (Toselli & Widlund, 2006). ■

Definición 2.6.7. Se define el i-ésimo complemento local de Schur como:

$$\mathcal{S}^{(i)} = A_{\Gamma\Gamma}^{(i)} - A_{\Gamma I}^{(i)}(A_{II}^{(i)})^{-1}A_{I\Gamma}^{(i)}.$$

Adicionalmente se define:

$$g_{\Gamma}^{(i)} = f_{\Gamma}^{(i)} - A_{\Gamma I}^{(i)}(A_{II}^{(i)})^{-1}f_{I}^{(i)}.$$

Definición 2.6.8. Se define el complemento de Schur del problema discretizado $Au = f$, por:

$$\mathcal{S} = \sum {R_{\Gamma}^{(i)}}^{T} \mathcal{S}^{(i)} R_{\Gamma}^{(i)}.$$

Adicionalmente se define:

$$g_{\Gamma} = \sum {R_{\Gamma}^{(i)}}^{T} g_{\Gamma}^{(i)}.$$

Teorema 2.6.9. *El problema $Au = f$ es equivalente a resolver:*

$$\begin{cases} \mathcal{S}u_{\Gamma} = g_{\Gamma}, \\ A_{II}^{(i)}u_{II}^{(i)} = f_{I}^{(i)} - A_{I\Gamma}^{(i)}R_{\Gamma}^{(i)}u_{\Gamma}. \end{cases} \tag{2.2}$$

Demostración. Véase (Toselli & Widlund, 2006). ■

Observación 2.6.10. La ventaja de resolver el sistema 2.2, es que después de resolver el problema sobre la interfaz, el resto de subsistemas se pueden resolver de manera embarazosamente paralela.

Teorema 2.6.11. *Sea Ω una región con una malla regular de tamaño h, y sea $\mathcal{F} = \{\Omega_1, \ldots, \Omega_k\}$ una partición regular de la región Ω. Sea $H = \max diam\,\Omega_i$, entonces se tiene que:*

$$\kappa(\mathcal{S}) \leq \frac{C}{hH},$$

donde C es una constante independiente de h y H.

Demostración. Véase lema 4.11 (Toselli & Widlund, 2006). ■

Observación 2.6.12. De la proposición anterior se observa que el sistema $\mathcal{S}u_\Gamma = g_\Gamma$ va a estar mal condicionado para mallas finas, sin embargo observe que tiene mejor número de condición que el sistema original $Au = f$.

BDDC

Dado que el complemento de Schur está mal condicionado, la pregunta natural es cómo precondicionarlo. Clark Dohrmann en (Dohrmann, 2003) propuso el precondicionador de descomposición de dominios balanceado por restricciones *(BDDC)*, el cual consiste en resolver un problema global sobre un subconjunto de nodos de la interfaz llamados variables primales, y calcular el resto de grados de libertad sobre la interfaz de manera independiente en cada subdominio, restaurando la continuidad de la solución global mediante un operador de promediado.

Existen dos formas para derivar el precondicionador, la forma original de Dohrmann, y la propuesta por Jing Li y Olof Widlund en (Li & Widlund, 2006). Se decide seguir la derivación de Widlund porque tiene varias simplificaciones técnicas desde el punto de vista matemático, sin embargo es importante recalcar que la derivación de Dohrmann es más apta para implementaciones en computadoras. Para poder introducir el método *BDDC*, tenemos que primero considerar una serie de subespacios.

En el espacio de elementos finitos de Ω_i, denote el conjunto de funciones continuas por W^i, donde cada W^i se descompone en una parte interior al subdominio, y una parte en la interfaz, es decir:

$$W^{(i)} = W_I^{(i)} \bigoplus W_\Gamma^{(i)}.$$

Adicionalmente considere los siguientes espacios producto:

$$W = \prod W^{(i)}, \qquad W_I = \prod W_I^i, \qquad W_\Gamma = \prod W_\Gamma^i.$$

Denote el subespacio de funciones continuas en W_Γ por $\widehat{W}_\Gamma$. Además defina el espacio $\widetilde{W}_\Gamma$, dado por:

$$\widetilde{W}_\Gamma = W_\Delta \bigoplus \widehat{W}_\Pi = \left(\prod W_\Delta^{(i)}\right) \bigoplus \widehat{W}_\Pi,$$

donde $\widehat{W}_\Pi$ es el espacio *coarse* de funciones continuas sobre las variables primales de la interfaz. Y $W_\Delta^{(i)}$ es el espacio de variables duales de la interfaz, que es formado por las funciones que se anulan sobre las variables primales.

Considere los siguientes operadores de restricción y extensión entre espacios. $R_{\Gamma\Delta}$ y $R_{\Gamma\Pi}$ las restricciones del espacio $\widetilde{W}_\Gamma$ a los espacios W_Δ y $\widehat{W}_\Pi$ respectivamente. Adicionalmente defina $R_\Gamma^{(i)}$: $\widehat{W}_\Gamma \to W_\Gamma^{(i)}$, $R_\Delta^{(i)} : W_\Delta \to W_\Delta^{(i)}$, $R_\Pi^{(i)} : \widehat{W}_\Pi \to W_\Pi^{(i)}$ operadores que mapean variables de la interfaz global a cada subinterfaz. Adicionalmente defina $R_\Gamma : \widehat{W}_\Gamma \to W_\Gamma$ como la suma directa de los $R_\Gamma^{(i)}$, y el operador $\widetilde{R}_\Gamma : \widehat{W}_\Gamma \to \widetilde{W}_\Gamma$ como la suma directa de los $R_{\Gamma\Pi}$ y los $R_\Delta^{(i)} R_{\Gamma\Delta}$.

Sea δ una partición de la unidad sobre Γ, es decir $\sum_{j\in\mathcal{N}_x} \delta_j(x) = 1$, donde $\mathcal{N}_x$ es el conjunto de índices de los subdominios que contienen al nodo x. Defina los operadores de promediado $R_{D\Delta}^{(i)}$ y $R_{D\Pi}^{(i)}$, que corresponden a $R_\Gamma^{(i)}$, $R_\Delta^{(i)}$ escalados en cada nodo por δ_i.

Finalmente como es necesario asegurar la continuidad de la solución sobre la interfaz, en vez de resolver $\mathcal{S}u_\Gamma = g_\Gamma$, se resuelve:

$$\widehat{\mathcal{S}}u_\Gamma = \left(R_\Gamma^T \mathcal{S} R_\Gamma\right) u_\Gamma = \left(\sum_{i=1}^{N} R_\Gamma^T g_\Gamma^{(i)}\right),$$

ya que ese sistema corresponde a la restricción del operador $\mathcal{S}$ a $\widehat{W}_\Gamma$, forzando que $u_\Gamma \in \widehat{W}_\Gamma$.

El precondicionador *BDDC* para este nuevo sistema corresponde a:

$$M_{BDDC}^{-1} = R_{D\Gamma}^T (T_0 + T_{sub}) R_{D\Gamma},$$

donde T_0 es el operador de correción *coarse*:

$$T_0 = \Psi(\Psi^T \mathcal{S} \Psi)^{-1} \Psi^T$$

con:

$$\Psi = \begin{bmatrix} \Psi^{(1)} \\ \vdots \\ \Psi^{(N)} \end{bmatrix},$$

y cada $\Psi^{(i)}$ satisface:

$$\begin{bmatrix} S^{(i)} & {C^{(i)}}^T \\ C^{(i)} & 0 \end{bmatrix} \begin{bmatrix} \Psi^{(i)} \\ V^{(i)} \end{bmatrix} = \begin{bmatrix} 0 \\ R_\Pi^{(i)} \end{bmatrix}.$$

Además, T_{sub} es el operador de corrección de cada subdominio dado por:

$$T_{sub} = \sum^{N} \begin{bmatrix} {R_{\Gamma}^{(i)}}^T & 0 \end{bmatrix} \begin{bmatrix} S^{(i)} & {C^i}^T \\ C^i & 0 \end{bmatrix}^{-1} \begin{bmatrix} R_{\Gamma}^{(i)} \\ 0 \end{bmatrix}.$$

Teorema 2.6.13. *Para una escogencia apropiada de las variables primales y de la partición de la unidad, el número de condición del operador precondicionado cumple que:*

$$\kappa(M_{BDDC}^{-1}\widehat{\mathcal{S}}) \leq C(1 + \log(H/h))^2.$$

Demostración. Véase teorema 7.8.19 en (Brenner & Scott, 2007). ■

Observación 2.6.14. El conjunto de variables primales se escoge de tal manera que no existan subdominios flotantes, es decir que cada problema tenga solución única, y que la solución sea consistente en la interfaz.

Observación 2.6.15. De acuerdo a (Dohrmann, 2003) para el problema de Poisson en 2D, se puede tomar el conjunto de variables primales como los nodos sobre la interfaz que están en más de dos subdominios, y la partición de la unidad como la cantidad de subdominios en la que aparece un nodo.

Implementación algorítmica de BDDC

Definición 2.6.16. Se define la base local del espacio *coarse* $\Phi^{(i)}$ por medio de:

$$\begin{bmatrix} A^{(i)} & {C^{(i)}}^T \\ C^{(i)} & 0 \end{bmatrix} \begin{bmatrix} \Phi^{(i)} \\ \Lambda^{(i)} \end{bmatrix} = \begin{bmatrix} 0 \\ I \end{bmatrix},$$

donde I es la matriz identidad.

Observación 2.6.17. El sistema de ecuaciones para obtener los $\Phi^{(i)}$ es simétrico e indefinido, y se le conoce como un sistema de punto de silla.

Definición 2.6.18. Se define la matriz del problema *coarse*, como:

$$\mathcal{S}_C = \sum {R_C^{(i)}}^T \mathcal{S}_C^{(i)} R_C^{(i)},$$

con $\mathcal{S}_C^{(i)} = -\Lambda^{(i)}$, y los $R_C^{(i)}$ las proyecciones del espacio de variables primales sobre la interfaz en las variables primales que pertenecen al i-ésimo subdominio.

Algoritmo 8. Multiplicar por el operador precondicionador BDDC corresponde a:

Algoritmo 8 Multiplicación por el operador precondicionador BDDC, véase (Dohrmann, 2003).

procedure APPLYBDDC(y)

$r^{(i)} \leftarrow {R_B^{(i)}}^T W^{(i)} R_\Gamma^{(i)} y$

$$\begin{bmatrix} u_i \\ \mu_i \end{bmatrix} \leftarrow \begin{bmatrix} A^{(i)} & {C^{(i)}}^T \\ C^{(i)} & 0 \end{bmatrix}^{-1} \begin{bmatrix} r^{(i)} \\ 0 \end{bmatrix}$$

$r_C \leftarrow \sum {R_C^{(i)}}^T {\Phi^{(i)}}^T r^{(i)}$

$u_C \leftarrow \mathcal{S}_C^{-1} r_C$

$u_C^{(i)} \leftarrow \sum \Phi^{(i)} {R_C^{(i)}}^T u_C$

$z_\Gamma \leftarrow \sum {R_\Gamma^{(i)}}^T W^{(i)} R_B^{(i)} (u_i + u_C^{(i)})$

return z_Γ
end procedure

En este algoritmo $R_B^{(i)} : \Omega_i \to \Gamma_i$ es el operador restricción, y $W^{(i)}$ corresponde a la partición de la unidad.

2.7. Computación paralela

La principal idea de la computación paralela según (Sinnen, 2007) consiste en tomar una tarea, dividirla en subtareas relativamente independientes entre sí y procesar la mayor cantidad de subtareas al mismo tiempo, donde es necesario garantizar que el resultado final de todas las subtareas ejecutadas en paralelo, sea equivalente al resultado final de ejecutar las tareas secuencialmente.

De acuerdo con (Sinnen, 2007) los procesadores se clasifican de acuerdo a la taxonomía de *Flynn*, la cual establece cuatro categorías:

	Un dato	Múltiples Datos
Una instrucción	SISD	SIMD
Múltiples instrucciones	MISD	MIMD

Tabla 2.1: Taxonomía de Flynn

En el ambiente de computación de alto rendimiento, lo más común es encontrar tanto procesadores *MIMD* como *SIMD*, donde los procesadores *MIMD* corresponden a procesadores multi-núcleo, o varios

nodos de procesamiento en el caso de clústers computacionales. Los procesadores *SIMD* corresponden a unidades de cómputo vectorial, como lo son las tarjetas gráficas *GPGPU*.

Para mejorar la clasificación de los procesadores se extiende la clasificación al tipo de memoria que utilizan, donde se distinguen dos tipos de memoria:

1. Memoria centralizada, como es el caso de la memoria compartida y los caché en los procesadores multi-núcleo, donde el costo de acceder a un dato en otro núcleo es prácticamente nulo.

2. Memoria distribuida, como es el caso de las computadoras multi-procesador con arquitectura *NUMA*, o nodos de un clúster conectados por una conexión de red, donde el costo de acceder un dato en otro procesador es alto.

De acuerdo con (Cormen, Leiserson, Rivest & Stein, 2009), el rendimiento computacional de un algoritmo paralelo se mide respecto a la cantidad T_P, que corresponde al tiempo de ejecución en segundos del algoritmo en P procesadores, donde naturalmente T_1 es el tiempo secuencial del algoritmo. A partir de T_P se define el *speedup* de un algoritmo paralelo respecto a su versión secuencial como:

$$speedup = \frac{T_1}{T_P},$$

que representa el incremento de velocidad respecto a la versión secuencial. De la definición del *speedup* se puede observar que:

$$speedup \leq P,$$

donde esta última desigualdad limita las expectativas de mejora en el tiempo de ejecución de un algoritmo paralelo a P veces más rápido que su versión secuencial. Otro límite teórico importante según (Kirk & Hwu, 2010) corresponde a la ley de *Amdahl*, que establece que para una carga de trabajo fija, y un algoritmo con una porción no paralelizable se tiene que:

$$speedup = \frac{1}{(1-p) + \frac{p}{s}},$$

donde p representa la fracción del código que es paralelizable, y s es el *speedup* de esas porciones paralelas. Gráficamente la ley de *Amdahl* se puede visualizar en la figura 2.3:

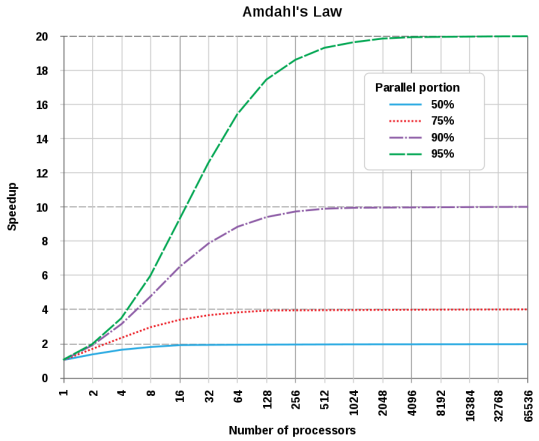

Figura 2.3: Ley de *Amdahl*, imagen tomada de (Wikimedia-Commons, 2008)

Una cuestión importante a notar de la gráfica es que, el *speedup* después de una cierta cantidad de procesadores deja de crecer y se estabiliza, esto debido a que la porción del algoritmo no paralelizable se vuelve predominante en el tiempo de ejecución.

Por el momento solo se ha discutido el rendimiento computacional desde el punto de vista del procesador, sin embargo un aspecto crucial para el buen rendimiento de cualquier programa, es el rendimiento que puedan tener los diferentes tipos de memoria en una computadora, ya que en muchas ocasiones los accesos de memoria terminan siendo verdaderos cuellos de botella para el programa.

En (Cormen et al. 2009) explican que el análisis teórico de un algoritmo -tomando en consideración la memoria, es tremendamente más complicado y complejo que el análisis tomando en consideración solamente el procesador, en consecuencia usualmente no se realiza un análisis teórico de la memoria. Sin embargo existen métricas para analizar el rendimiento de memoria desde el punto de vista práctico, tal vez la más importante de ellas es el ancho de banda utilizado por el programa, la cual se mide en (GB/s), y representa la velocidad con que los datos se están cargando desde la memoria principal.

CUDA

CUDA es un entorno de programación paralela producido por la compañía *NVIDIA* para sus tarjetas *GPU*. El entorno provee un compilador para código C++-CUDA, herramientas de *profiling*, y una serie de bibliotecas programadas para GPU, que van desde una implementación de *BLAS* hasta algoritmos de redes neuronales.

En general para poder programar código eficiente para las tarjetas de *NVIDIA*, es necesario

describir la arquitectura de estas tarjetas.

La arquitectura de ejecución utilizada por las tarjetas de *NVIDIA* se denomina *SIMT* "una instrucción, múltiples *threads*", que se clasifica según la taxonomía de *Flynn* como un procesador *SIMD*. Esta arquitectura comienza cuando el multiprocesador *(SM)* calendariza y ejecuta grupos de 32 *threads* llamados *warps*. En un *warp* todos los *threads* inician ejecución en la misma instrucción, y a menos que ocurra un condicional como un *if*, todos los *threads* siempre van a ejecutar cada instrucción del programa al mismo tiempo, maximizando la utilización de recursos.

Ante la aparición de un condicional en el código se produce una condición de divergencia de *threads*, serializando la ejecución de cada una de las ramas del condicional. Esto significa la desactivación de los *threads* del *warp* que no cumplen el condicional, y los *threads* que sí la cumplen ejecutan esa rama del condicional hasta completarla. Posteriormente se ejecuta la siguiente rama del condicional de la misma manera hasta completar todas las ramas, reconvergiendo a la misma instrucción fuera del condicional para todos los *threads*. Es importante mencionar que desde la arquitectura *Volta* introducida en el 2017, existe un calendarizador con capacidad de ejecutar *threads* individualmente, significando que cada rama de la divergencia no tiene que ejecutar hasta completarse en un solo momento, lo cual permite un nivel de paralelismo de mayor finesa.

Adicionalmente a la agrupación de *threads* en *warps*, existe una macro jerarquía de *threads*, donde los *warps* se agrupan en bloques, y finalmente estos se agrupan en grillas de bloques, como se muestra en la figura 2.4.

Otro aspecto crucial a considerar en el diseño de programas para *GPU* es la jerarquía de memoria de estas tarjetas la cual se muestra en la figura 2.4. De esta figura se puede observar que cada thread tiene sus propios registros, cada bloque tiene una memoria compartida accesible por todos los *threads* del bloque, y existe una memoria global accesible por todos los *threads* en una grilla. Adicionalmente existe la memoria constante que es de solo lectura para los *threads*, los caches L2 y L1 en cada SM, y la memoria de textura que también es accesible para todos los *threads*.

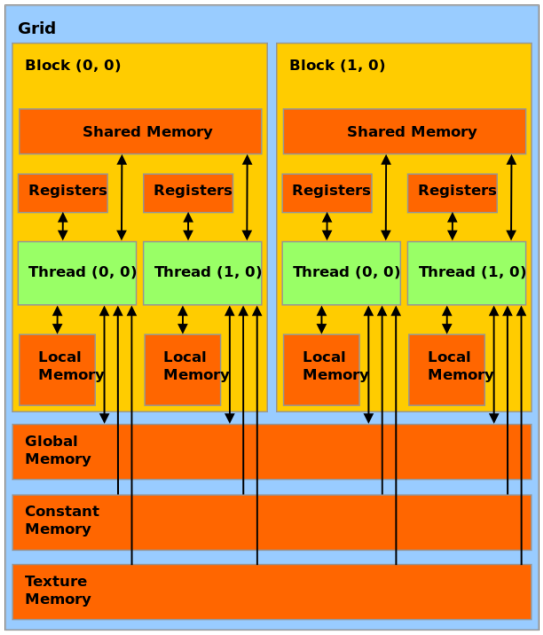

Figura 2.4: Agrupación de *threads* en *CUDA*, imagen tomada de (Wikimedia-Commons, 2010)

Uno de los cuidados más grandes que se debe tener cuando se realiza un programa en CUDA, es la forma en la que se accede la memoria global, ya que existen una serie de reglas impuestas por la arquitectura del procesador, las cuales si no se respetan usualmente disminuyen considerablemente el rendimiento del programa. Para obtener el máximo rendimiento de la memoria las reglas impuestas por la arquitectura se pueden resumir de la siguiente manera:

Todas las posiciones de memoria accedidas por los threads *del* warp *necesitan ser contiguas en la memoria global, y la primera de estas posiciones tiene que ser un múltiplo de* 32 *bytes, donde no importa el orden de acceso a lo interno de los* warps.

De no seguir estas reglas, el multiprocesador *SM* tiene que emitir más de un pedido de acceso a la memoria global, significando un desperdicio del ancho de banda de la memoria.

La figura 2.5 es el caso ideal de acceso a memoria, donde todas las posiciones son contiguas y están alineadas, por lo que el *SM* emite una cantidad mínima de pedidos a memoria. En el caso de la figura 2.6 se observa que pese a que solo se corrieron las posiciones de la figura 2.5 a la derecha en una unidad, como los accesos no están alineados, el *SM* tiene que realizar una petición de memoria extra.

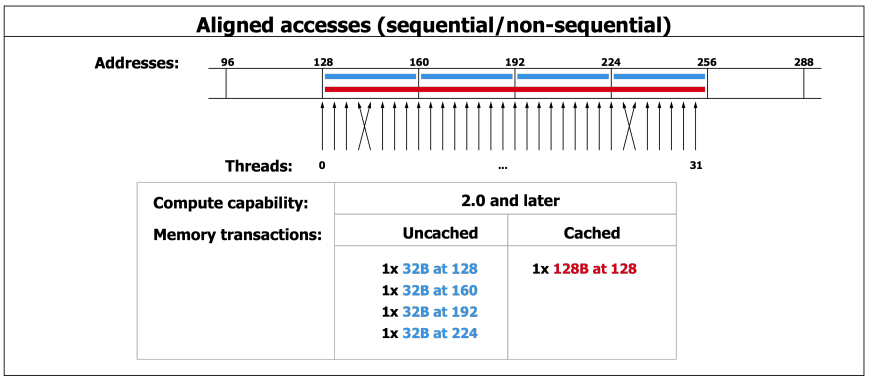

Figura 2.5: Accesos alineados a la memoria global, imagen tomada de (NVIDIA, 2018b)

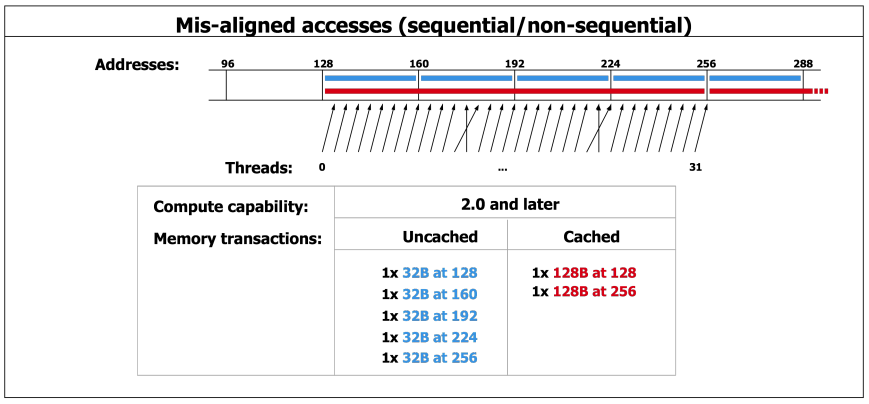

Figura 2.6: Accesos desalineados a la memoria global, imagen tomada de (NVIDIA, 2018b)

Capítulo 3

Metodología

En este capítulo se describirán las dos arquitecturas en donde se corrieron los algoritmos desarrollados, junto con las pruebas para evaluar su desempeño y validar su funcionamiento.

3.1. Descripción de las arquitecturas

Las pruebas fueron ejecutadas en dos arquitecturas computacionales, una laptop de uso personal, la otra es un nodo del clúster de alto rendimiento Serké del PRIS-Lab de la Escuela de Ingeniería Eléctrica de la Universidad de Costa Rica. En la tabla 3.1 se describe la configuración de Hardware y Software de ambas arquitecturas, donde se puede observar que la diferencia entre las arquitecturas es esencialmente la configuración del Hardware.

Computadora	Computadora personal	Nodo de procesamiento del clúster Serké
Procesador	i7 6700HQ	2 Xeon Platinum 8180M
RAM (GB)	16 DDR3	640 DDR4
GPU	GTX 960M	2 Tesla V100 PCIE
Sistema operativo	Debian testing	CentOS 7.6
Kernel	4.19.37-3	3.10.0-957.12.2.el7
GLIBC	2.28	2.28
GCC	8.3.0	8.3.0
CUDA Driver	418.67	418.67
CUDA Toolkit	10.1	10.1

Tabla 3.1: Configuración de las arquitecturas usadas en las pruebas.

Debido a que las arquitecturas computacionales son muy diferentes, en la tabla 3.2 se detallan las especificaciones de los dos tipos de tarjetas GPU usadas en las pruebas. De esta tabla es importante recalcar que la tarjeta GTX 960 no tiene soporte de operaciones atómicas de doble punto flotante y que el rendimiento de las operaciones de doble punto flotante es sumamente inferior en esa tarjeta respecto al rendimiento de punto flotante simple, debido a que la tarjeta GTX 960M pertenece al mercado de consumo doméstico.

Tarjeta	GeForce GTX 960M	Tesla V100 PCIE
CUDA cores	640	5120
Frecuencia base (MHz)	1096	1370
Rendimiento FP32 (TFLOPS)	1.5	14
Rendimiento FP64 (TFLOPS)	0.047	7
Ancho de bando (GB/s)	80	900
RAM (GB)	4	32
Arquitectura	Maxwell	Volta
Soporte de operaciones atómicas FP64	No	Sí

Tabla 3.2: Tarjetas GPU usadas en las pruebas.

En la tabla 3.3 se detallan las especificaciones de los CPU de ambas arquitecturas, donde la diferencia más explotada en la plataforma es la cantidad de cores del 8180M.

Procesador	i7 6700HQ	Xeon Platinum 8180M
Núcleos	4	28
Threads	8	56
Frecuencia base (GHz)	2.60	2.50
Caché (MB)	6	38.5
Ancho de bando (GB/s)	34.1	119
Extensiones de conjunto de instrucciones de Intel	SSE4.1, SSE4.2, AVX2	SSE4.2, AVX, AVX2, AVX-512

Tabla 3.3: CPU usados en las pruebas.

3.2. Pruebas de la plataforma computacional

Para medir el rendimiento de cada una de las partes del programa, se ejecutó el programa con las configuraciones que se muestran en las tablas 3.4, 3.5, 3.6 y 3.7. El mismo programa genera un archivo donde se registran los tiempos de ejecución, consumo de memoria, entre otros; véase el apéndice para ejemplos de este archivo. Para comparar el rendimiento de la plataforma desarrollada se utilizó una implementación de *BDDC* hecha por Juan Gabriel Calvo usando partes de IFem (Chen, s.f.) en Matlab. Las pruebas ejecutadas en Matlab se muestran en la tabla 3.8, las cuales solo se ejecutaron

en la laptop de uso personal. Finalmente en las pruebas se resolvieron los problemas de Poisson:

$$\begin{cases} -\Delta u = 2\pi^2 \sin(\pi x)\sin(\pi y) \ \text{ en } [0,1]^2, \\ u = 0 \ \text{ en } \partial[0,1]^2, \end{cases}$$

$$\begin{cases} -\Delta u = \sin(xy)(x^2+y^2) \ \text{ en } [0,1]^2, \\ u(x,y) = \sin(xy) \ \text{ en } \partial[0,1]^2, \end{cases}$$

cuyas soluciones exactas son conocidas.

Cantidad de nodos	$\sqrt{NSD}$	Cantidad de threads
129	4	1,2,4
	8	1,2,4,8
	16	1,2,4,8,16
	32	1,2,4,8,16
257	4	1,2,4
	8	1,2,4,8
	16	1,2,4,8,16
	32	1,2,4,8,16
513	4	4
	8	4,8
	16	4,8,16
	32	4,8,16,32
	64	4,8,16,32,64
1025	8	8
	16	8,16
	32	8,16,32
	64	8,16,32,64
2049	16	16
	32	16,32
	64	32,64
	128	32,64,128
4097	64	64

Tabla 3.4: Pruebas de la plataforma usando métodos directos en el clúster Serké.

Cantidad de nodos	$\sqrt{NSD}$	Cantidad de threads
129	2	2
	4	2,4
	8	2,4,8
	16	2,4,8,16
257	2	2
	4	2,4
	8	2,4,8
	16	2,4,8,16
513	2	2
	4	2,4
	8	4,8
1025	2	2
	4	2,4
	8	4,8
	16	4,8,16
2049	4	4

Tabla 3.5: Pruebas de la plataforma usando métodos iterativos en el clúster Serké.

Cantidad de nodos	$\sqrt{NSD}$	Cantidad de threads
129	4	1,2,4
	8	1,2,4,8
	16	1,2,4,8,16
	32	1,2,4,8,16
257	4	2,4
	8	2,4,8
	16	2,4,8,16
	32	2,4,8,16
513	4	4
	8	4,8
	16	4,8
	32	4,8
	64	4,8
1025	8	4,8
	16	4,8
	32	4,8

Tabla 3.6: Pruebas de la plataforma usando métodos directos en la Laptop.

Cantidad de nodos	$\sqrt{NSD}$	Cantidad de threads
129	2 4 8 16	2 2,4 2,4,8 2,4,8,16
257	2 4 8 16	2 2,4 2,4,8 2,4,8,16
513	2 4	2,4 4
1025	2	2

Tabla 3.7: Pruebas de la plataforma usando métodos iterativos en la Laptop.

Cantidad de nodos	$\sqrt{NSD}$
129	2,4,8,16,32
257	2,4,8,16,32
513	2,4,8,16,32

Tabla 3.8: Pruebas realizadas en Matlab para comparar el rendimiento de la plataforma.

Capítulo 4

Plataforma computacional

En este capítulo se presenta la plataforma computacional desarrollada en *C++-CUDA*, la cual según el programa *cloc* (Danial, 2019) posee alrededor de 24000 líneas de código. Específicamente se discuten las principales características de la plataforma, y cómo con ella se implementó el precondicionador *BDDC* para la solución de la ecuación de Poisson con condiciones de Dirichlet usando elementos finitos.

4.1. Características de la plataforma

Los lenguajes de programación de bajo nivel como *C* o *Fortran* son muy usados dentro de la comunidad científica debido a su excelente rendimiento computacional, sin embargo este tipo de lenguajes no son tan fáciles de programar como lo son los lenguajes de alto nivel como *Matlab* o *Python*, ya que en los lenguajes de bajo nivel el programador tiene que estar pendiente del manejo de memoria, tambien usualmente son necesarias más lineas de código, etc.

C++ es un lenguaje de programación con la misma eficiencia que *C* u otros lenguajes de bajo nivel, pero introduce conceptos de más alto nivel, que no solo facilita el desarrollo de programas, si no que también permite crear códigos más generales y abstractos. En especial en *C++* existe una biblioteca estándar *(STL)*, la cual introduce una gran cantidad de funciones y estructuras de datos que facilitan el desarrollo de programas. Desafortunadamente *CUDA* no tiene una biblioteca estándar que provea estructuras de datos que permitan abstraer al programador del manejo de memoria, entorpeciendo el desarrollo de programas.

Para tratar este vacío en *CUDA*, la plataforma desarrollada introduce los siguientes tipos de memoria: *cpu_memory*, *gpu_memory*, *pinned_memory* y *managed_memory*, estos tipos definen la forma y lugar en que las estructuras de datos de la plataforma van a asignar memoria. Por ejemplo *gpu_memory* define que la memoria tiene que ser asignada en la memoria de la tarjeta *GPU*. Es necesario hace una mención especial a *pinned_memory*, ya que esta define un tipo de memoria en la

RAM del *CPU* la cual no es paginable, permitiendo transferencias de memoria asíncronas entre el *CPU* y la *GPU*, mejorando el rendimiento de las transferencias de memoria en general.

La plataforma introduce dos estructuras de datos básicas que facilitan el manejo de memoria: *Array<T,memoryType>* y *Tensor<T,memoryType>*, con *T* es el tipo de datos de los elementos almacenados y *memoryType* es uno de los tipos de memoria definidos. Estás estructuras permiten la creación y destrucción de memoria, usando los constructores o el método *reserve_memory* para construir memoria, y el destructor o el método *free_memory* para destruir memoria. Una característica importante es que las estructuras permiten la copia de datos entre *CPU-GPU*, *GPU-CPU*, *CPU-CPU* y *GPU-GPU*, con el operador "=", o bien con alguno de los métodos de importar o copiar memoria. Es crucial indicar que todas las operaciones de memoria indicadas son compatibles con arquitecturas con múltiples *GPU*. La diferencia entre la estructura *Array* y *Tensor*, es que la segunda provee mejor rendimiento cuando el tipo de datos es vectorial.

Usando las clases *Array* y *Tensor* se crearon las siguientes estructuras de datos: *dual_memory*, *CXS* y *Hash_table*. La clase *dual_memory* crea memoria tanto en *CPU* como en *GPU*, permitiendo la replica de bloques de memoria entre la *RAM* del *CPU* y la *RAM* del *GPU*. La clase *CXS* puede representar matrices poco densas en los formatos *COO*, *CRS*, *CCS*, o general cualquier formato basado en 3 vectores de datos. Finalmente la clase *Hash_table* implementa una tabla hash con *open-addressing* donde la función hash se puede pasar como parámetro *template* a la clase. Es crucial indicar que esta estructura en el mejor de los casos permite la inserción de elementos en tiempo constante, lo cual la convierte en una buena candidata para ensamblar matrices poco densas.

Una de las características principales de la plataforma desarrollada es la posibilidad de realizar programación funcional, mediante el uso de las características de meta-programación de *C++*. En especial se introducen los siguientes funcionales: *apply*, *apply_meta*, *reduce*, *reduce_apply* y *reduce_meta*, los cuales pueden generar código ejecutable secuencialmente en el *CPU*, paralelamente en el *CPU* mediante el uso de *OpenMP*, y código paralelo para la *GPU*. Estos funcionales reciben la función a ejecutar en un conjunto de datos mediante un parámetro *template*. También reciben como parámetro *template* el lugar en el que se quiere ejecutar el código (*CPU* o *GPU*), y los parámetros de paralelización, es decir la cantidad de threads en *OpenMP*, el tamaño de los bloques en *CUDA*. Es crucial mencionar que debido a la forma en que se programó y las optimizaciones que permiten los compiladores modernos, los funcionales mencionados tienen rendimiento nativo, es decir, casi no existe penalización en el tiempo de ejecución por no haber programado directamente la función.

En la plataforma se incluyen una serie de funciones básicas compatibles con los funcionales que permiten realizar una gran variedad de operaciones. Por ejemplo, existe el operador *scalar_ma* el cual permite realizar la operación $ax + y$. Es importante mencionar que en la mayoría de arquitecturas modernas de hardware, esa operación está optimizada para calcularse en una menor cantidad de instrucciones que multiplicar y después sumar, por lo que *scalar_ma* llama a esa optimización directamente en el caso de código en *GPU*.

La figura 4.1 presenta un ejemplo con la mayoría de las características discutidas de la plataforma. En este ejemplo se puede observar que la plataforma simplifica y abstrae el desarrollo de código.

```
// Se define un operador compatible con el funcional apply
struct operador {
    template <typename V> __host_device__ static V fn(V x,V y) {
        return x+y;
    }
    template <typename V> __host_device__ static V fn(V x,V y,Vector<V,2> alpha) {
        return alpha(0)*x+alpha(1)*y;
    }
};
int main() {
    size_t vectorSize=1024;
    // Se crea un vector de tamaño vectorSize, el cual está presente tanto en la GPU como la RAM del CPU
    dual_memory<double> x(vectorSize);
    // Se crean 3 vectores de doubles de tamaño vectorSize en la memoria de la GPU
    Array<double,gpu_memory> wGPU(vectorSize),vGPU(vectorSize),uGPU(vectorSize);
    // Se crean 2 vectores de doubles de tamaño vectorSize en la memoria RAM del CPU
    Array<double,cpu_memory> yCPU(vectorSize),wCPU;
    // Se realiza la operación wGPU=-vGPU+3.1415uGPU en paralelo en la tarjeta GPU
    apply<operador,DEVICE_GPU>(wGPU.data(),vGPU.data(),uGPU.data(),vectorSize,double_2({-1,3.1415}));
    // Se copian los datos del vector wGPU en la GPU al vector wCPU en la memoria RAM del CPU
    wCPU=wGPU;
    // Se realiza la operación xCPU=wCPU+yCPU en paralelo en el CPU
    apply<operador,DEVICE_CPU>(x.data(DEVICE_CPU),wCPU.data(),yCPU.data(),vectorSize);
    // Se replica el bloque de memoria de x en la GPU
    x.refresh(DEVICE_GPU);
    // Se calcula la norma al cuadrado del vector x y se almacena en wGPU[0]
    reduce_apply<scalar_add<double>,norma<double>,CTA(double,0),DEVICE_GPU>(wGPU.data(),x.data(DEVICE_GPU),n
        );
}
```

Figura 4.1: Código de ejemplo de la plataforma computacional desarrollada.

4.2. Solución de PDE's

Para el ensamblado de la matriz de elementos finitos se desarrolló un kernel de *CUDA* llamado *___assemble_SM___*. Esta función recibe una tabla hash que corresponde a la matriz de elementos finitos, el vector f que corresponde al lado derecho del sistema lineal $Au = f$, un *Tensor* con las coordenadas cartesianas de los nodos de cada elemento, un *Array* con los índices de los nodos, y un *Array* indicando cuales nodos pertenecen a la frontera. Adicionalmente esta función recibe por

parámetro *template* la acción de la forma bilineal, y las condiciones frontera y de Poisson.

Es importante mencionar que la forma bilineal que corresponde a Poisson se define en el *struct Lagrange_2D*. En la figura 4.2 se muestra un ejemplo del formato para definir las condiciones frontera para la ecuación de Poisson, donde *poisson* es el lado derecho de la ecuación de Poisson y *boundary* son las condiciones frontera.

```
template <typename T> struct boundary_conditions {
    __attr_opfihd__ static T poisson(C_Q(T) x,C_Q(T) y) {
        return __sin__(x*y)*(__pow_2__(x)+__pow_2__(y));
    }
    __attr_opfihd__ static T boundary(C_Q(T) x,C_Q(T) y) {
        return __sin__(x*y);
    }
};
```

Figura 4.2: Ejemplo de como definir las condiciones frontera y el lado derecho de la ecuación de Poisson.

Para solucionar sistemas de ecuaciones se definen las clases *CGSolver*, *PCGSolver*, *MinResSolver*, *CholeskySolver* y *LUSolver*, las cuales se inicializan con el método *init*, y resuelven los sistemas con la función *solve*. En el caso de *CholeskySolver* y *LUSolver* antes de ejecutar *solve* es necesario llamar al método *analyze*, también es importante mencionar que estas dos clases son *wrappers* de los *solvers* triangulares de la biblioteca *CUSPARSE* de *NVIDIA*, además una vez creados solo funcionan para un sistema lineal. En el caso de *CGSolver*, *PCGSolver* y *MinResSolver* el método *solve* se puede ejecutar después de la inicialización de recursos, los argumentos del método son la matriz A, un *Array* con el lado derecho y, y un *Array* para almacenar la solución x, es decir se resuelve $Ax = y$.

Respecto a *BDDC*, se crearon las clases *BDDCMesh* la cual almacena la subdivisión de la malla y la interfaz, *BDDCMatrices* que almacenan las matrices de elementos finitos de cada subdominio, las proyecciones, la partición de la unidad y las descomposiciones de *Cholesky* y LU en el caso de métodos directos. También se introducen las clas *BDDCSolversCGMINRES* y *BDDCSolversCGLU*, las cuales almacenan los recursos de los solvers, entre otras cosas. Finalmente en los *namespaces* *BDDCDirect* y *BDDC*, se incluyen las funciones *primalSpaceBase*, *computegGamma*, *computeSchurC*, *computeInterface* y *computeSubdomains*, las cuales computan cada uno de los pasos necesarios para encontrar la solución usando el precondicionador *BDDC*. Finalmente también se incluye la funciones *verifySolution* y *bddcIO*, la primera calcula el error de la solución contra la solución real de la ecuación, y *bddcIO* almacena en archivos *csv* todas las matrices y vectores necesitados por el precondicionador.

Capítulo 5

Resultados y discusión

En este capítulo se presentan los resultados generados por la plataforma computacional desarrollada en las pruebas establecidas en el capítulo 3. La mayor parte del análisis se lleva a cabo respecto a las pruebas ejecutadas en el clúster Serké, donde se analiza memoria, tiempos de ejecución y convergencia de la solución. Al final del capítulo se describen brevemente los resultados obtenidos en la laptop de uso personal y se comparan con una implementación existente del precondicionador BDDC escrita en Matlab.

5.1. Consumo de memoria

Como se observa de las figuras 5.1 y 5.2 el consumo de memoria en los métodos directos es considerablemente mayor que en los métodos iterativos, donde para ciertas pruebas existe un incremento de más de 4 veces en el consumo de memoria, como se observa en el caso de una malla de 2049×2049 nodos. Este incremento en el consumo es debido a que los métodos directos necesitan almacenar las matrices L y LU de cada uno de los sistemas lineales, las cuales no están presentes en los métodos iterativos. Además se tiene que los solvers triangulares también requieren de memoria adicional, ya que necesitan almacenar la información de cómo resolver los sistemas y de un lugar para hacer cálculos temporales.

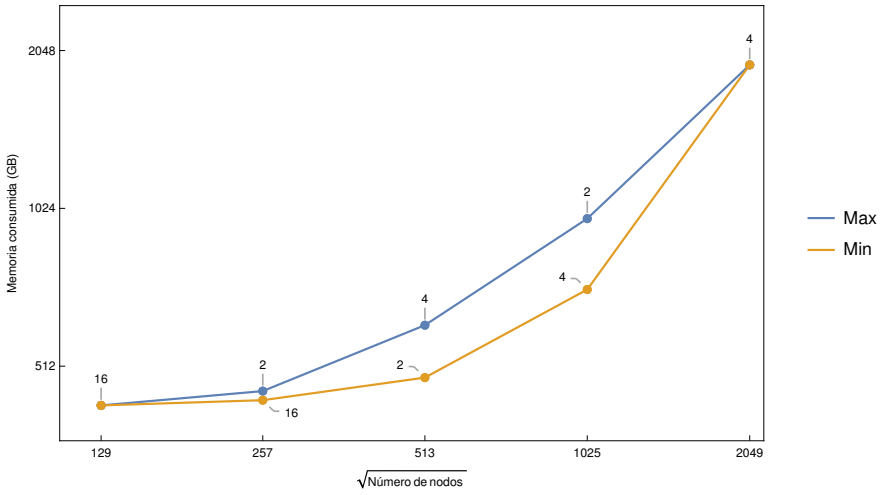

Figura 5.1: Consumo de memoria usando métodos iterativos. En cada punto se indica $\sqrt{NSD}$.

En la figura 5.2 se puede observar que aumentar la cantidad de subdominios en los métodos directos disminuye el consumo de memoria hasta en factores de 2. Esto se debe a que las matrices de los subsistemas son más pequeñas, por lo que es más sencillo reducir el fill-in en las descomposiciones LU y de Cholesky, y además se tiene que los solvers triangulares ocupan un espacio menor para su funcionamiento. De esta gráfica también se observa la razón por la cual 4097×4097 nodos es la malla más grande tratada en las pruebas, ya que para esta malla se consumen alrededor de $22(GB)$ de memoria, y dado que las tarjetas Tesla V100 tienen un máximo de memoria de $32(GB)$, es imposible resolver problemas más grandes usando métodos directos con una sola tarjeta Tesla V100.

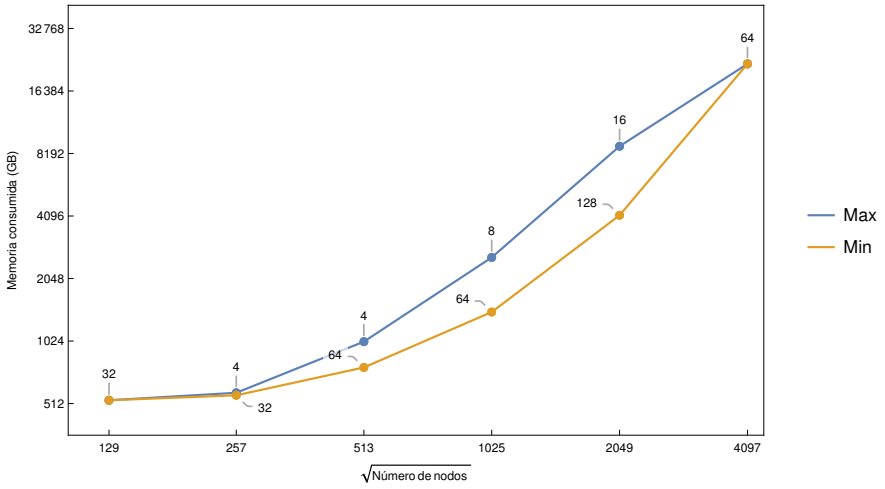

Figura 5.2: Consumo de memoria usando métodos directos. En cada punto se indica $\sqrt{NSD}$.

5.2. Ensamblaje de los subsistemas

En la figura 5.3 se muestra el tiempo de ejecución requerido para ensamblar las matrices de cada subsistema. En esta gráfica se observa que en general aumentar la cantidad de subdominios y threads incrementa considerablemente el tiempo de ejecución como se observa para la malla de 2049×2049 nodos, donde existe un incremento de 70 veces en el tiempo de ejecución respecto a la configuración con la menor cantidad de subdominios para esa misma malla. El incremento se debe a que el kernel de CUDA responsable por el ensamblaje utiliza una *hashtable* con open-addressing para almacenar los resultados, por lo que al disminuir el tamaño de los subsistemas, se disminuye el tamaño de la *hashtable*, aumentando la probabilidad de riesgo de colisiones en la inserción de elementos, en consecuencia se disminuye el rendimiento de la estructura (Alcantara, 2011).

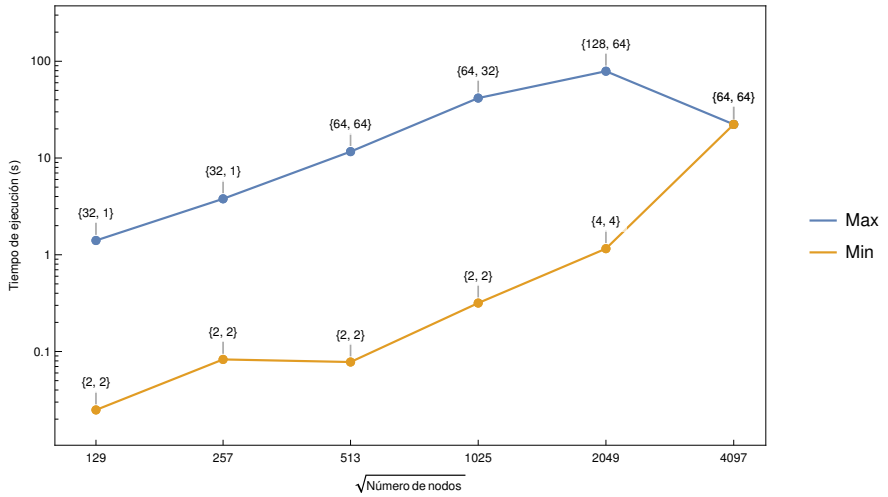

Figura 5.3: Tiempo de ensamblaje de los subsistemas. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

En la figura 5.4 se muestra una gráfica más detallada del tiempo de ensamblaje de los subsistemas para una malla de 1025×1025 nodos, donde el número de threads representa cuántos subsistemas se están intentando calcular simultáneamente. De esta y la figura 5.3 se observa que el aumento en la cantidad de threads después de una cierta cantidad también afecta negativamente el rendimiento del ensamblaje. Esto ocurre por un *oversubscription* de la tarjeta, ya que se calendarizan muchos kernels de ensamblaje lo cual disminuye el rendimiento individual de los kernel de ensamblaje. Entonces pese a que en el caso de 64 subdominios se pueden calcular los 64 subdominios simultáneamente, el oversubscription de la tarjeta produce que esa combinación no sea favorable para el tiempo de ejecución.

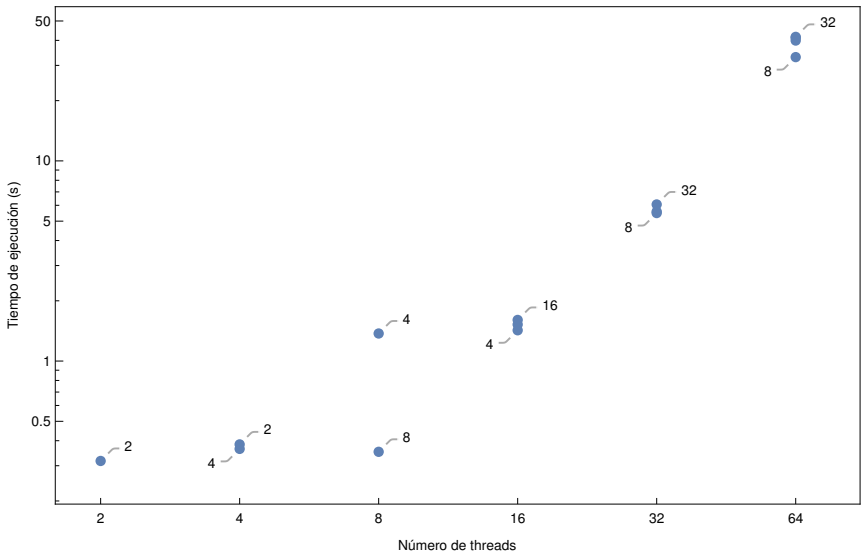

Figura 5.4: Tiempo de ensamblaje para una malla de 1025×1025 nodos. En cada punto se indica $\sqrt{NSD}$.

5.3. Solución de los sistemas lineales

En la figura 5.5 se presentan los tiempos de ejecución para encontrar las descomposiciones LU y de *Cholesky* de los sistemas lineales de punto de silla y $A_{II}^{(i)}$ de la sección 2.6, las cuáles son necesarias únicamente para la solución por medio de métodos directos. Estas descomposiciones se computan en el CPU ya que los algoritmos presentados en la sección 2.1 no se paralelizan bien en una GPU. A diferencia de los tiempos de ensamblaje, aumentar la cantidad de threads y la cantidad de subdominios ayuda a disminuir el tiempo de ejecución. Esto se debe principalmente a que estas factorizaciones son algoritmos con complejidad $\mathcal{O}(n^3)$, por lo que se benefician de que el tamaño de la matriz sea pequeño. Además como cada descomposición se realiza en un solo core, aumentar el número de threads también beneficia el rendimiento ya que se pueden computar muchas de estas en paralelo.

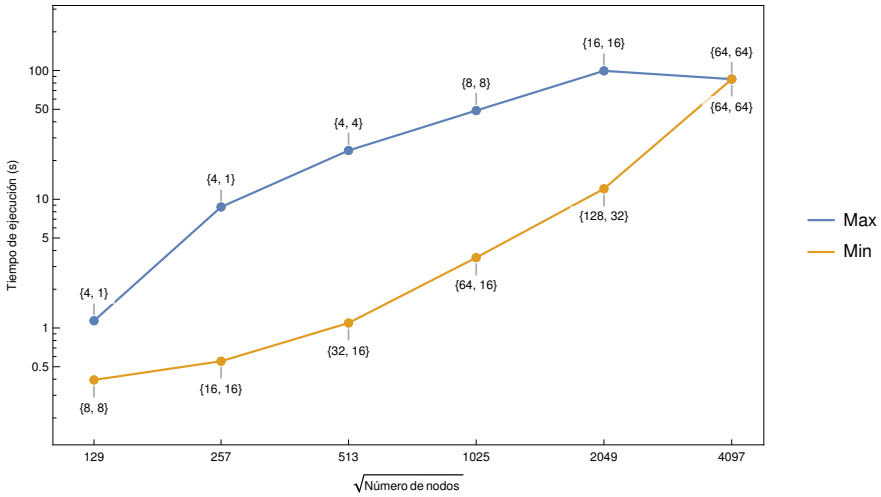

Figura 5.5: Tiempo para encontrar las descomposiciones LU y de *Cholesky* de los sistemas de cada subdominio. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

Como se observa en las figuras 5.6, 5.7, los métodos directos son varios órdenes de magnitud superiores a los métodos iterativos para encontrar la solución de sistemas lineales, esto sucede principalmente por el hecho de que resolver un sistema triangular tiene complejidad algorítmica $\mathcal{O}(n^2)$, mientras que cada iteración de gradiente conjugado o MINRES tiene complejidad $\mathcal{O}(n^2)$. Una cuestión importante a notar, es que en ambas figuras el tiempo mínimo para la solución de los sistemas crece de manera lineal respecto al tamaño de la malla, lo cual es deseable para cualquier algoritmo. En el caso particular de la figura 5.6 existe un segundo motivo para la superioridad de los métodos directos. Este es que a lo largo del algoritmo *PCG* es necesario resolver varias veces un mismo sistema lineal para diferentes lados derechos, y los métodos iterativos no son particularmente buenos para la solución de múltiples lados derechos.

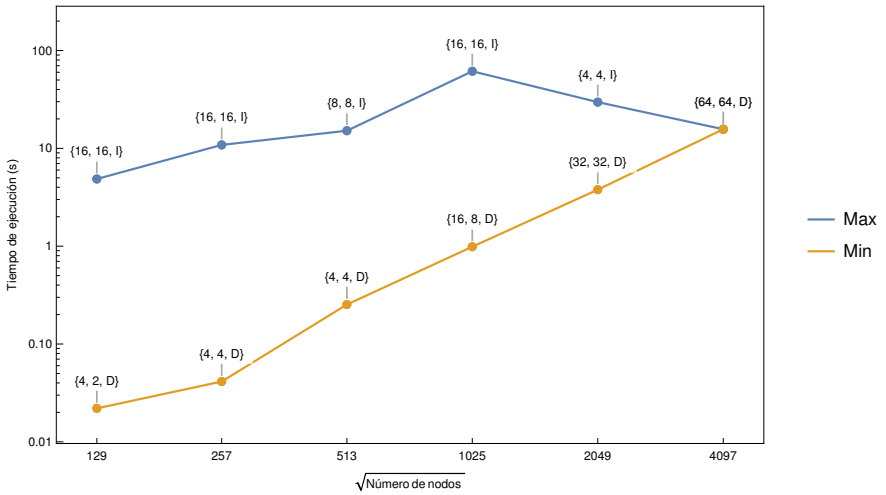

Figura 5.6: Tiempo para encontrar la solución de la interfaz. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

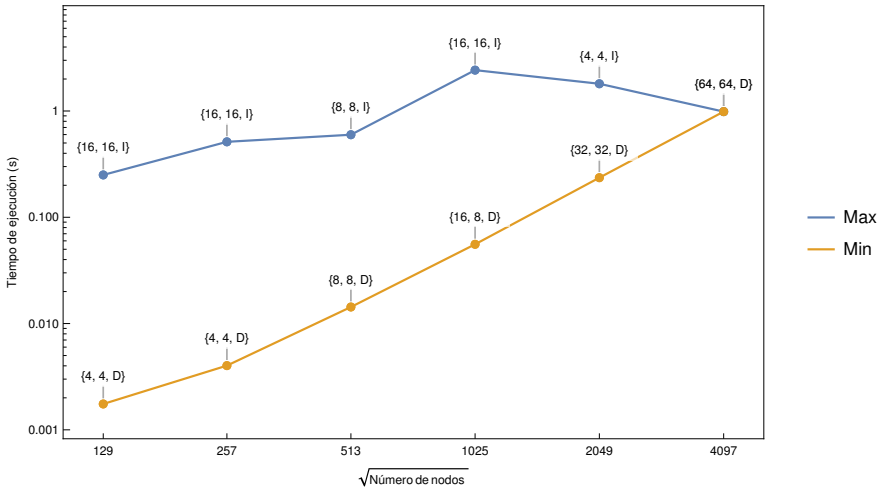

Figura 5.7: Tiempo para encontrar la solución de los subdominios. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

De las figuras 5.8 y 5.9 se puede observar que en general aumentar tanto la cantidad de subdominioscomo la cantidad de threads es perjudicial para el rendimiento de la plataforma. El aumento en el número de subdominios perjudica el rendimiento porque las tarjetas GPU son más eficientes entre más grande sea el problema a resolver, ya que se reduce la candelarización de kernels optimizando el uso de recursos de la tarjeta. El aumento de la cantidad de threads también perjudica el rendimiento por el

oversubscribing de los recursos. Es importante observar que los métodos iterativos son más perjudicados que los métodos directos por el incremento en la cantidad de subdominios, ya que como se observa en la figura 5.8 el menor tiempo de ejecución siempre sucede con la menor cantidad de subdominios, esto se debe a que los métodos iterativos son más eficientes conforme el tamaño del problema crece.

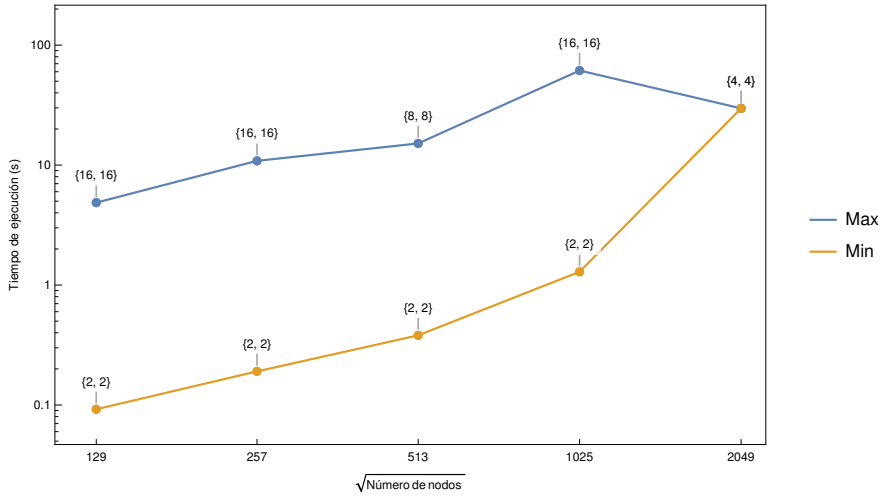

Figura 5.8: Tiempo para encontrar la solución de la interfaz con métodos iterativos. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

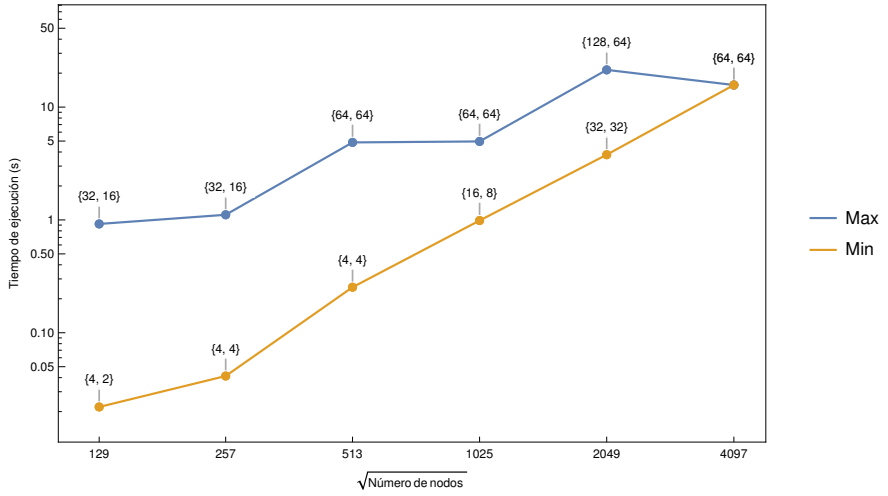

Figura 5.9: Tiempo para encontrar la solución de la interfaz con métodos directos. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

En la figura 5.10 se presentan los tiempos requeridos para resolver el problema en la interfaz, encontrar la solución en los subdominios, y el tiempo requerido para ensamblar el problema de la

interfaz sin tomar en cuenta el tiempo para encontrar las descomposiciones. En ésta se observa que los métodos directos son hasta 100 veces más rápidos que los métodos iterativos, con la ventaja adicional que mantienen un crecimiento lineal en el tiempo de ejecución respecto al tamaño de la malla. En la figura 5.11 se muestra el speedup mínimo de los métodos directos respecto a los métodos iterativos, de donde se observa que por lo menos siempre son al menos dos veces más rápido, y por lo general 6 veces más rápidos.

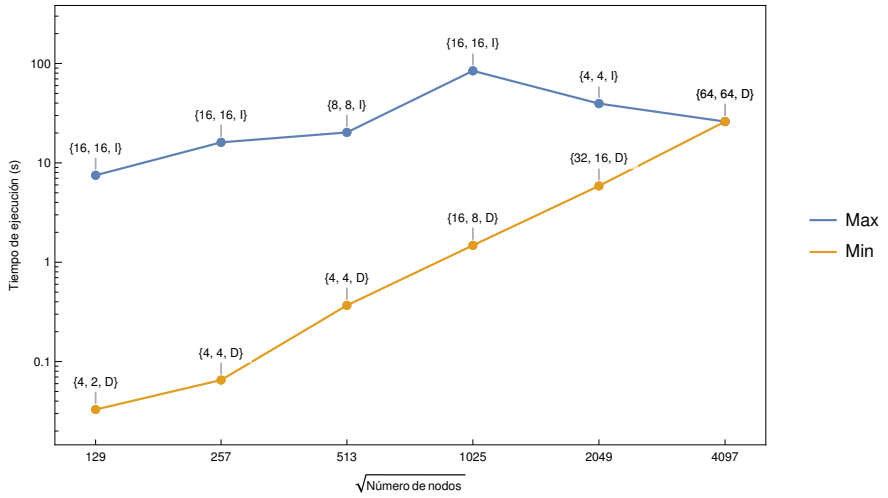

Figura 5.10: Tiempo total para encontrar la solución de la ecuación sin tomar en cuenta el tiempo de descomposición. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

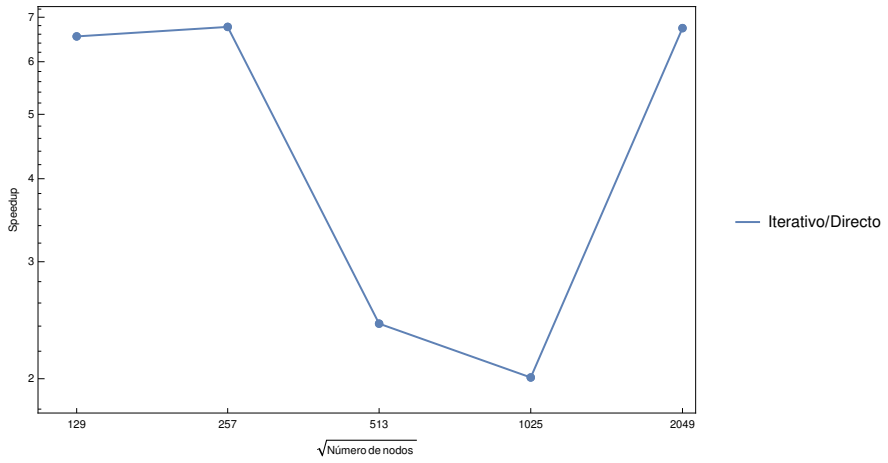

Figura 5.11: Speedup de los métodos directos respecto a los métodos iterativos sin tomar en cuenta los tiempos de descomposición.

La figura 5.12 presentan los mismos tiempos de la figura 5.10 pero incluyendo el tiempo necesario para encontrar las descomposiciones LU y de *Cholesky*. En este caso se encuentra que los métodos iterativos son más rápidos que los métodos directos, debido no solo a la complejidad algorítmica de las descomposiciones que es $\mathcal{O}(n^3)$, si no también por la falta de paralelismo para encontrar la descomposición de cada submatriz.

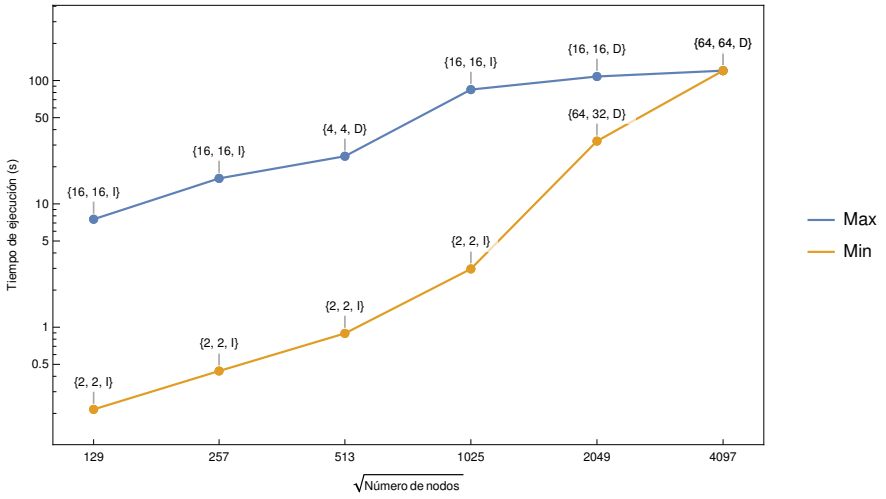

Figura 5.12: Tiempo total para encontrar la solución de la ecuación tomando en cuenta el tiempo de descomposición. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

En la figura 5.13 se muestra la cantidad de iteraciones que necesita el método de gradiente conjugado precondicionado con el precondicionador *BDDC* para converger a la solución de la interfaz. De esta gráfica se puede observar la efectividad del precondicionador, ya que se observa un máximo de 9 iteraciones para cualquier de las mallas, cantidad de subdominios y tipo de método. Es importante hacer notar que los casos donde el precondicionador ejecuta una sola iteración no fueron comunes dentro la totalidad de las pruebas.

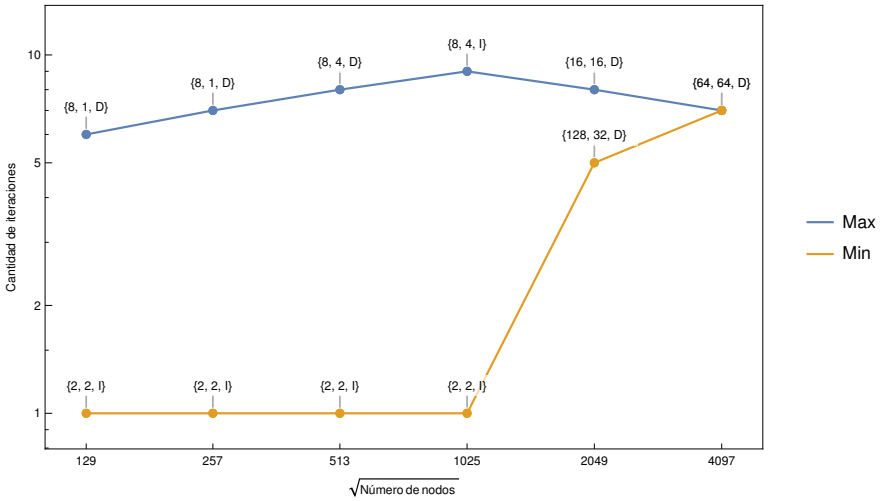

Figura 5.13: Cantidad de iteraciones en el método PCG con el precondicionador BDDC. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

En la figura 5.14 se puede observar que efectivamente la solución converge a la solución real conforme se refina la malla. Es crucial notar que los métodos directos convergen más rápido a la solución real, mientras que los métodos iterativos no convergen con tanta facilidad. Esto se debe a que los métodos iterativos solo aproximan la solución del sistema, entonces para lograr la misma precisión provista por los métodos directos hubiese sido necesario incrementar la cantidad de iteraciones, empeorando el tiempo de ejecución de los métodos iterativos. También es importante mencionar que la tolerancia usada en el método PCG fue de 10^{-10}, por lo que disminuir la tolerancia podría mejorar la aproximación de la solución global.

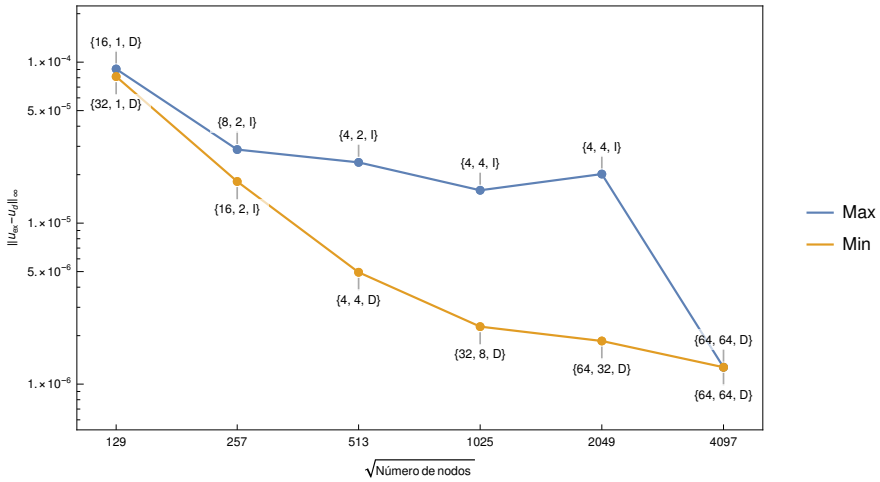

Figura 5.14: Error de la solución generada contra la solución exacta. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

5.4. Resultados en la laptop de uso personal.

En las figuras 5.15, 5.16 y 5.17, se presentan un resumen de los resultados obtenidos en la laptop de uso personal. En estos se puede observar un comportamiento similar a los resultados obtenidos en el clúster Serké, con la diferencia de que los tiempos son considerablemente más lentos respecto a los del clúster.

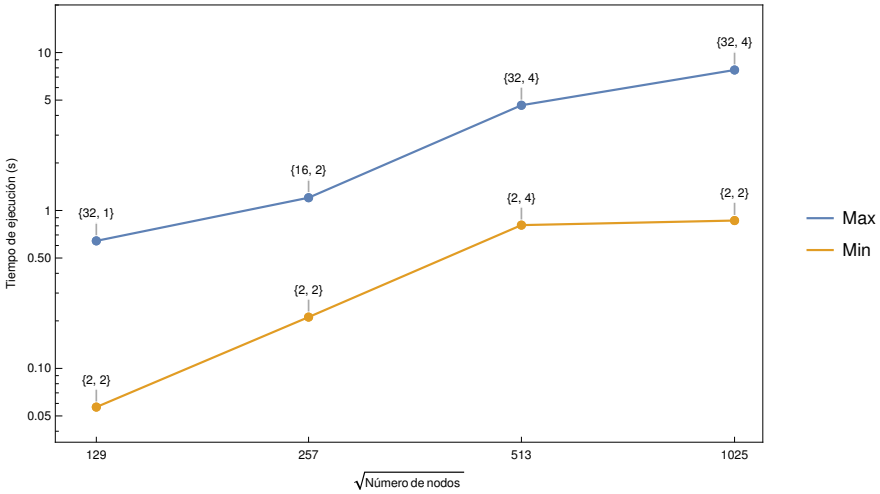

Figura 5.15: Tiempo de ensamblaje de los subsistemas en la computadora personal. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

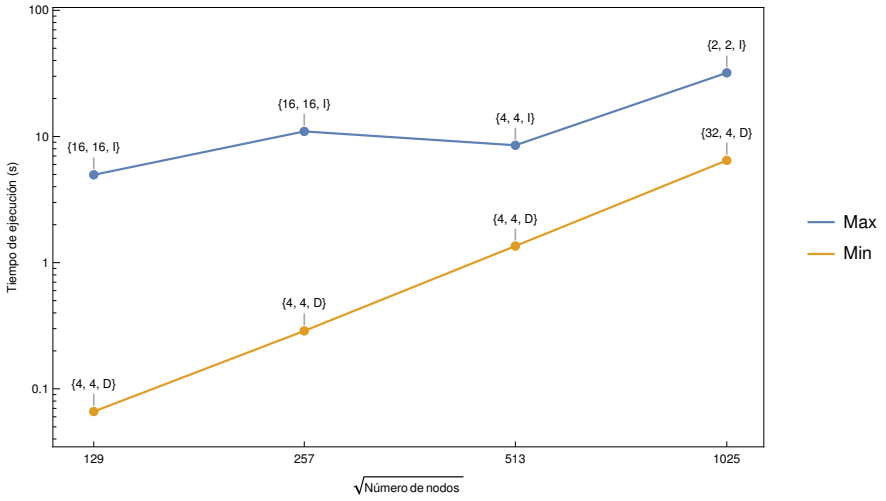

Figura 5.16: Tiempo total para encontrar la solución de la ecuación sin tomar en cuenta el tiempo de descomposición en la computadora personal. En cada punto se indica $\sqrt{NSD}$, la cantidad de threads utilizados y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

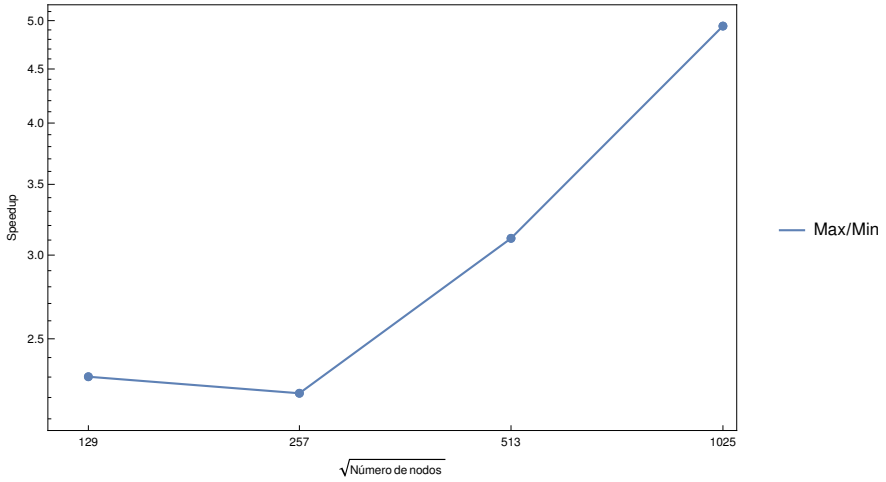

Figura 5.17: Speedup de los métodos directos respecto a los métodos iterativos sin tomar en cuenta los tiempos de descomposición en la computadora personal.

De las figuras 5.18, 5.19, se puede observar que los tiempos de ensamblaje en la plataforma desarrollada son considerablemente más rápidos que los tiempos generados por Matlab, donde se puede observar un speedup mínimo de 2 veces y un speedup máximo de 14 veces. Es importante notar de estas figuras que la plataforma desarrollada es más rápida conforme crece el tamaño de la malla, esto se debe a que la eficiencia de las tarjetas GPU crece con el tamaño del problema.

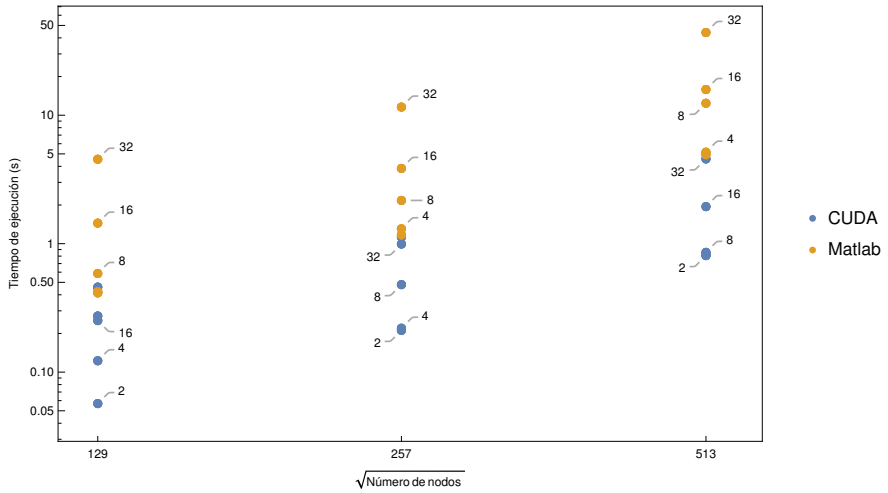

Figura 5.18: Tiempo de ensamblaje de los subsistemas en CUDA y Matlab en la computadora personal. En cada punto se indica $\sqrt{NSD}$ y la cantidad de threads utilizados.

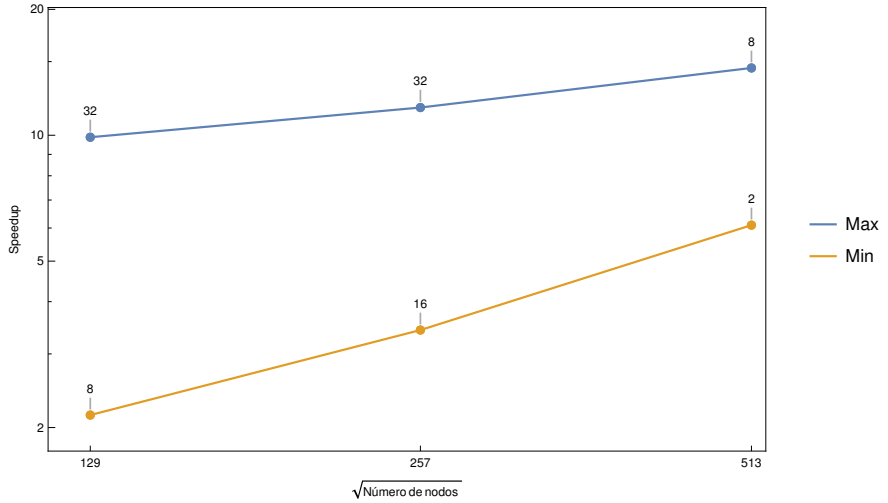

Figura 5.19: Speedup de los tiempos de ensamblaje en CUDA respecto a los tiempos de ejecución en Matlab en la computadora personal. En cada punto se indica $\sqrt{NSD}$.

Como se observa en las figuras 5.20, 5.21, la plataforma desarrollada también es más rápida que Matlab para encontrar la solución de la ecuación, donde es importante mencionar que los tiempos en CUDA sí incluyen el tiempo para encontrar las factorizaciones de la matrices en el caso de los método directos. Es necesario notar que los speedups logrados son menores a los de los tiempos de ensamblaje, donde el speedup más pequeño fue de 1.3 y el speedup más alto fue de 11 veces más rápido. Al igual que los tiempos de ensamblaje, es importante observar que el speedup también crece conforme el tamaño

de la malla crece.

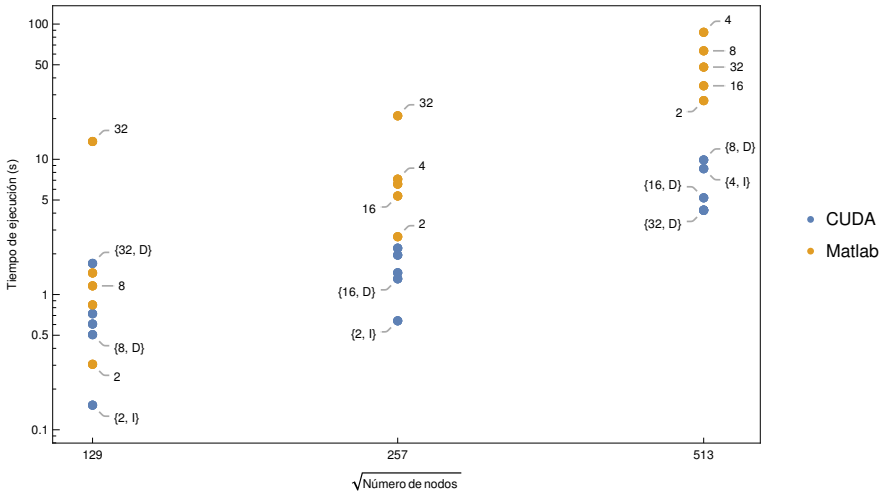

Figura 5.20: Tiempo total para encontrar la solución de la ecuación en CUDA y en Matlab en la computadora personal. En cada punto se indica $\sqrt{NSD}$ y el método utilizado, donde la letra I es para indicar métodos iterativos y una D para métodos directos.

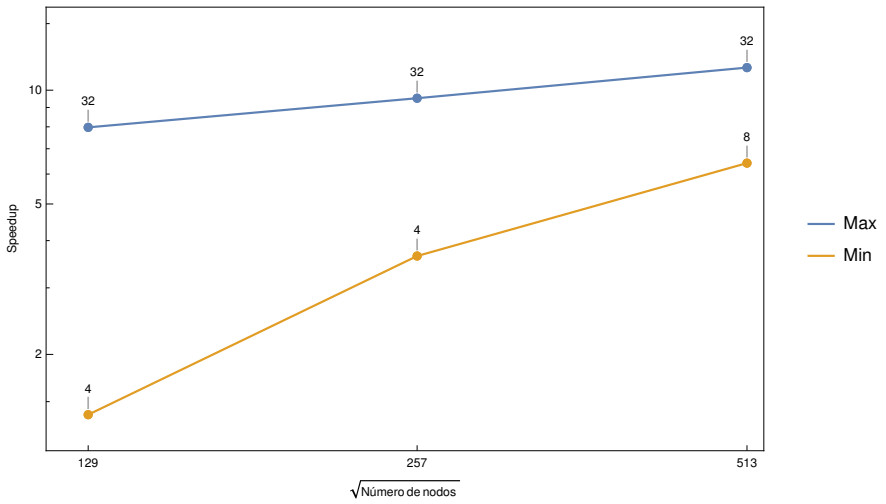

Figura 5.21: Speedup de los tiempos para encontrar la solución de la ecuación en CUDA respecto a los tiempos de ejecución en Matlab en la computadora personal. En cada punto se indica $\sqrt{NSD}$.

Capítulo 6

Conclusiones, recomendaciones y trabajo futuro

6.1. Conclusiones

En este capítulo se presenta un resumen de las principales conclusiones y contribuciones realizadas. Finalmente se realizan recomendaciones sobre el trabajo futuro, tanto desde el punto de vista de la plataforma computacional, como el de la solución de PDE's mediante el precondionador BDDC.

Se desarrolló una plataforma en C++-CUDA en la que se simplifica el manejo de memoria desde el punto de vista del programador, además se crearon estructuras de datos básicas de propósito general y de propósito específico enfocadas al análisis numérico, como lo son vectores, matrices densas y poco densas, hashtables, entre otras. Adicionalmente se creó un framework de meta-programación que permite la introducción de elementos de programación funcional a C++-CUDA, donde este permite el desarrollo de algoritmos paralelos funcionales tanto en GPU como en CPU, con la característica especial de que proveen rendimiento nativo. Además todos los algoritmos son compatibles con la arquitectura Volta y sus predecesoras.

Utilizando la plataforma desarrollada se implementaron varios algoritmos para la solución de sistemas de ecuaciones lineales, como lo son el método de gradiente conjugado, *MINRES*, y sus versiones precondicionadas. Adicionalmente se introdujeron métodos eficientes y altamente personalizables para la operación de reducción de datos, permitiendo por ejemplo calcular la p-norma de un vector de una manera sencilla y eficiente para el programador.

Respecto al trabajo realizado con el método de elementos finitos, se encontró que el ensamblado de las discretizaciones es eficiente siempre y cuando la cantidad de subdominios no sea muy alta, sin embargo esa limitante es superable. Respecto a la solución de la ecuación se encontró que si se asume que se tienen las factorizaciones de cada uno de los subsistemas, los métodos directos son varios órdenes de magnitud más rápidos que los métodos iterativos, sin embargo, si no se asume que las factorizaciones

se conocen a priori, para problemas menores a 2049×2049 nodos los métodos iterativos son globalmente más rápidos. También se observó que la solución producida por métodos directos converge a mayor velocidad que la producida por métodos iterativos. Respecto al precondicionador BDDC se observa un correcto funcionamiento del mismo, ya que en todos los experimentos redujo la cantidad de iteraciones del sistema del complemento de Schur a un máximo de 9 iteraciones. Adicionalmente se encontró que la malla más grande que se puede tratar usando métodos iterativos y una sola tarjeta V100 es de 4097×4097 nodos, por una cuestión de consumo de memoria. Finalmente se encontró que la plataforma desarrollada es varios órdenes de magnitud superior a una implementación de *BDDC* en Matlab, llegando a ser hasta 11 veces más rápida que la de Matlab.

6.2. Trabajo futuro

Existen varios puntos en donde se pueden mejorar los tiempos de ejecución. En primer lugar, creando un kernel de CUDA que ensamble varias submatrices en una sola llamada, mejoraría dramáticamente los tiempos de ejecución, ya que se eliminaría el *oversubscription* de la tarjeta. Adicionalmente también se deberían probar otras funciones hash para la *hashtable*, o bien otras estructuras de datos para crear la tabla que no sean basadas en *open-addressing*.

Respecto a la solución de los sistemas de ecuaciones lineales, una posible línea de trabajo futuro sería implementar un método multifrontal (Duff & Reid, 1983) para la solución de los subsistemas, ya que estos métodos permiten computar las factorizaciones de una matriz con un gran nivel de paralelismo. Entonces este método directo podría eliminar la ventaja global en los tiempos de ejecución que tienen los métodos iterativos en el presente trabajo.

También queda como trabajo futuro extender los algoritmos para solucionar PDE's en regiones tridimensionales, elementos finitos más generales y otros tipos de ecuaciones. Finalmente también se debe considerar extender la acción del precondicionador al uso de métodos adaptativos, y BDDC multinivel (Li & Widlund, 2006).

Capítulo 7

Bibliografía

Alcantara, D. A. F. (2011). *Efficient hash tables on the gpu* (Tesis doctoral, University of California, Davis, Davis, CA).

Aspnäs, M., Signell, A. & Westerholm, J. (2006). Efficient assembly of sparse matrices using hashing. En *International Workshop on Applied Parallel Computing* (pp. 900-907).

Baxter, S. (2016). moderngpu 2.0. Recuperado desde https://github.com/moderngpu/moderngpu

Bell, N. & Garland, M. (2008). *Efficient sparse matrix-vector multiplication on CUDA*. Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

Bell, N. & Garland, M. (2009). Implementing sparse matrix-vector multiplication on throughput-oriented processors. En *Proceedings of the conference on high performance computing networking, storage and analysis* (p. 18).

Brenner, S. & Scott, R. (2007). *The mathematical theory of finite element methods*. Springer Science & Business Media.

Calvo, J. G. (2015). *Domain Decomposition Methods for Problems in H(curl)* (Tesis doctoral, Departamento de Matemáticas, Universidad de Nueva York).

Chen, L. (s.f.). *iFEM: An innovative finite element method package in Matlab*. Recuperado desde https://www.math.uci.edu/~chenlong/Papers/iFEMpaper.pdf

Cormen, T., Leiserson, C., Rivest, R. & Stein, C. (2009). *Introduction to Algorithms* (3). Cambridge, MA: MIT Press.

Dalton, S., Bell, N., Olson, L. & Garland, M. (2015). Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. Recuperado desde http://cusplibrary.github.io/

Danial, A. (2019). cloc, Count Lines of Code. Recuperado desde https://github.com/AlDanial/cloc

Dohrmann, C. (2003). A Preconditioner for Substructuring Based on Constrained Energy Minimization. *SIAM Journal on Scientific Computing, vol. 25, issue 1*, *25*, 246-258. doi:10.1137/S1064827502412887

Duff, I. S. & Reid, J. K. (1983). The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Softw. 9*(3), 302-325. doi:10.1145/356044.356047

Evans, L. (2010). *Partial Differential Equations* (2). Graduate Studies in Mathematics. Providence, RI: American Mathematical Society.

Filippone, S., Cardellini, V., Barbieri, D. & Fanfarillo, A. (2017). Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw. 43*(4), 30-1. doi:10.1145/3017994

Griffiths, D. (2012). *Introduction to Electrodynamics* (4). Glenview, IL: Pearson Education.

He, G. & Gao, J. (2016). A novel CSR-based sparse matrix-vector multiplication on GPUs. *Mathematical Problems in Engineering*, *2016*. doi:10.1155/2016/8471283

Hestenes, M. R. & Stiefel, E. (1952). Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, *49*(6), 409-436.

Kirk, D. & Hwu, W.-m. (2010). *Programming Massively Parallel Processors* (1). Burlington, MA: Elsevier.

Kůs, P. & Šístek, J. (2017). Coupling parallel adaptive mesh refinement with a nonoverlapping domain decomposition solver. *Advances in Engineering Software*, *110*, 34-54. doi:https://doi.org/10.1016/j.advengsoft.2017.03.012

Li, J. & Widlund, O. B. (2006). FETI-DP, BDDC, and block Cholesky methods. *International Journal for Numerical Methods in Engineering, vol. 66, issue 2, pp. 250-271*, *66*, 250-271. doi:10.1002/nme.1553

Liu, Y. & Schmidt, B. (2015). LightSpMV: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. En *Application-specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on* (pp. 82-89).

NVIDIA. (2018a). *CUDA C Best Practices Guide*. DG-05603-001_v10.0. NVIDIA. Recuperado desde %7Bhttp://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html%7D

NVIDIA. (2018b). *CUDA C Programming Guide*. PG-02829-001_v10.0. NVIDIA. Recuperado desde %7Bhttp://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html%7D

Nvidia. (2019). Tesla V100 PCIe. Recuperado desde https://www.nvidia.com/es-es/data-center/tesla-v100/

Paige, C. C. & Saunders, M. A. (1975). Solution of Sparse Indefinite Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, *12*(4), 617-629.

Saad, Y. & Schultz, M. H. (1986). GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, *7*(3), 856-869.

Sinnen, O. (2007). *Task scheduling for parallel systems* (1). Hoboken, NJ: John Wiley & Sons.

Sou-Cheng, C. (2006). *ITERATIVE METHODS FOR SINGULAR LINEAR EQUATIONS AND LEAST-SQUARES PROBLEMS* (Tesis doctoral, Instituto de Ingeniería Computacional y Matemática, Universidad de Stanford).

Steinberger, M., Derlery, A., Zayer, R. & Seidel, H.-P. (2016). How naive is naive SpMV on the GPU? En *High Performance Extreme Computing Conference (HPEC), 2016 IEEE* (pp. 1-8). doi:10.1109/HPEC.2016.7761634

Steinberger, M., Zayer, R. & Seidel, H.-P. (2017). Globally Homogeneous, Locally Adaptive Sparse Matrix-vector Multiplication on the GPU. En *Proceedings of the International Conference on Supercomputing* (pp. 13-1). ICS '17. Chicago, Illinois: ACM. doi:10.1145/3079079.3079086

TOP500. (2019). TOP500 LIST. Recuperado desde https://www.top500.org/lists/2019/06/

Torp, A. (2009). *Sparse linear algebra on a GPU: with Applications to flow in porous Media* (Tesis de maestría, Institutt for matematiske fag).

Toselli, A. & Widlund, O. (2006). *Domain decomposition methods-algorithms and theory*. Springer Science & Business Media.

Trefethen, L. N. & Bau III, D. (1997). *Numerical linear algebra*. Siam.

Weber, D., Bender, J., Schnoes, M., Stork, A. & Fellner, D. (2013). Efficient GPU data structures and methods to solve sparse linear systems in dynamics applications. En *Computer Graphics Forum* (Vol. 32, *1*, pp. 16-26).

Wikimedia-Commons. (2008). Amdahl's law. Recuperado desde https://en.wikipedia.org/wiki/File:AmdahlsLaw.svg

Wikimedia-Commons. (2010). Memory hierarchy. Recuperado desde https://commons.wikimedia.org/wiki/File:Memory.svg

Wikimedia-Commons. (2018). The Summit Supercomputer. Recuperado desde https://commons.wikimedia.org/wiki/File:Summit_(supercomputer).jpg

Wong, J., Kuhl, E. & Darve, E. (2015). A New Sparse Matrix Vector Multiplication GPU Algorithm Designed for Finite Element Problems. *CoRR*, *abs/1501.00324*. eprint: {1501.00324v1}. Recuperado desde %7Bhttp://arxiv.org/abs/1501.00324v1%7D

Apéndice

Apéndice A

Resultados generados

```
BDDC parameters
        NN:  513    NSD: 16 ThreadNum: 32
*************************************************************************
Creating mesh:
Boundary: 0.0986286
Partition: 0.0494236
Interface: 0.0585733
Subdomains: 0.0627774
Corners: 0.00179122
Mesh created, total etime: 0.307596, mesh creation etime: 0.303295
    Subdomains: 0.271349     Arrays: 0.0319467
*************************************************************************
*************************************************************************
Init matrices:
Init matrices time: 0.367513
*************************************************************************
*************************************************************************
Beginning assembly:
Assembly etime: 0.891541
Total memory: 32480, free memory: 31863
Assembly time: 1.30817
*************************************************************************
*************************************************************************
Beginning initBDDCMatrices:
Total memory: 32480, free memory: 32109
InitBDDC time: 0.107013
*************************************************************************
*************************************************************************
Init solvers:
Total memory: 32480, free memory: 32005
Init solvers time: 0.450281
```

```
**************************************************************************
**************************************************************************
Computing decompositions:
Total memory: 32480, free memory: 31811
Decompositions time: 1.49388
**************************************************************************
**************************************************************************
Analyzing decompositions:
Total memory: 32480, free memory: 31727
Analyzing time: 0.31988
**************************************************************************
**************************************************************************
Beginning primalSpaceBase:
Total memory: 32480, free memory: 31727
Init time: 0.149018
**************************************************************************
**************************************************************************
Beginning gGamma:
Total memory: 32480, free memory: 31721
gGamma time: 0.0410414
**************************************************************************
**************************************************************************
Beginning SchurC:
Total memory: 32480, free memory: 31721
SchurC time: 0.00199409
**************************************************************************
**************************************************************************
Beginning Interface:

    PCG it: 0
        Sxp time:   0.0255508
        Error:      0.33934
        Minvr time: 0.022937
        It time:    0.0485867

    PCG it: 1
        Sxp time:   0.0247093
        Error:      0.0299787
        Minvr time: 0.0222503
```

```
    It time:    0.0470564

PCG it: 2
    Sxp time:   0.024524
    Error:      0.00133384
    Minvr time: 0.0223661
    It time:    0.0469712

PCG it: 3
    Sxp time:   0.0244566
    Error:      0.000250496
    Minvr time: 0.0221719
    It time:    0.0467074

PCG it: 4
    Sxp time:   0.0249169
    Error:      3.82132e-05
    Minvr time: 0.0219506
    It time:    0.046966

PCG it: 5
    Sxp time:   0.0243487
    Error:      1.7224e-06
    Minvr time: 0.0219832
    It time:    0.0464116

PCG it: 6
    Sxp time:   0.0252075
    Error:      2.28071e-07
    Minvr time: 0.0223073
    It time:    0.047593

PCG it: 7
    Sxp time:   0.0246584
    Error:      2.56668e-08
    Minvr time: 0.021615
    It time:    0.0463573

PCG it: 8
```

```
        Sxp time:   0.0253321
        Error:      4.71504e-09
PCG results:
     It count: 8, error: 4.71504e-09
Total memory: 32480, free memory: 31719
Interface time: 0.433211
**************************************************************************
**************************************************************************
Beginning Subdomains:
Total memory: 32480, free memory: 31717
Subdomains time: 0.0215622
**************************************************************************
**************************************************************************
Beginning verification:
verification time: 0.0187609        error: 1.0052e-05       Interface error: 1.0052e-05
**************************************************************************
Total memory: 32480, free memory: 31717
Execution ended successfully, etime: 5.06966
```

```
BDDC parameters
        NN:  513    NSD: 16 ThreadNum: 32
**************************************************************************
Creating mesh:
Boundary: 0.0979624
Partition: 0.0491067
Interface: 0.0592489
Subdomains: 0.0628573
Corners: 0.00186064
Mesh created, total etime: 0.306684, mesh creation etime: 0.302271
    Subdomains: 0.271178     Arrays: 0.0310928
**************************************************************************
**************************************************************************
Init matrices:
Init matrices time: 0.379531
**************************************************************************
**************************************************************************
Beginning assembly:
Assembly etime: 5.23434
Total memory: 32480, free memory: 32087
Assembly time: 5.31621
**************************************************************************
**************************************************************************
Beginning initMatrices:
Total memory: 32480, free memory: 32109
Init time: 0.122489
**************************************************************************
**************************************************************************
Init solvers:
Total memory: 32480, free memory: 32005
Init solvers time: 0.45848
**************************************************************************
**************************************************************************
Beginning primalSpaceBase:
Total memory: 32480, free memory: 31999
Init time: 10.7477
**************************************************************************
**************************************************************************
Beginning gGamma:
```

```
Total memory: 32480, free memory: 31993
gGamma time: 0.489421
**************************************************************************
**************************************************************************
Beginning SchurC:
Total memory: 32480, free memory: 31993
SchurC time: 0.000722034
**************************************************************************
**************************************************************************
Beginning Interface:

    PCG it: 0
        Sxp time:   1.24294
        Error:      0.339331
        Minvr time: 3.01721
        It time:    4.26027

    PCG it: 1
        Sxp time:   1.21237
        Error:      0.0299783
        Minvr time: 2.99002
        It time:    4.20252

    PCG it: 2
        Sxp time:   1.15292
        Error:      0.00133381
        Minvr time: 2.88456
        It time:    4.03761

    PCG it: 3
        Sxp time:   1.03325
        Error:      0.0002505
        Minvr time: 2.86629
        It time:    3.89965

    PCG it: 4
        Sxp time:   0.914668
        Error:      3.82205e-05
        Minvr time: 2.82455
```

```
        It time:    3.73933

    PCG it: 5
        Sxp time:   0.882705
        Error:      1.72247e-06
        Minvr time: 2.68586
        It time:    3.56873

    PCG it: 6
        Sxp time:   0.770521
        Error:      2.28352e-07
        Minvr time: 2.61776
        It time:    3.38842

    PCG it: 7
        Sxp time:   0.668065
        Error:      2.56756e-08
        Minvr time: 2.53883
        It time:    3.207

    PCG it: 8
        Sxp time:   0.590658
        Error:      4.79825e-09
PCG results:
     It count: 8, error: 4.79825e-09
Total memory: 32480, free memory: 31991
Interface time: 33.5237
*************************************************************************
*************************************************************************
Beginning Subdomains:
Total memory: 32480, free memory: 31989
Subdomains time: 1.27148
*************************************************************************
*************************************************************************
Beginning verification:
verification time: 0.00555321       error: 1.37244e-05      Interface error: 1.37244e-05
*************************************************************************
Total memory: 32480, free memory: 31989
Execution ended successfully, etime: 52.6949
```

```
BDDC parameters
        NN:  2049   NSD: 32 ThreadNum: 128
**************************************************************************
Creating mesh:
Boundary: 2.61593
Partition: 0.884843
Interface: 1.47475
Subdomains: 1.77213
Corners: 0.0266469
Mesh created, total etime: 7.36284, mesh creation etime: 7.31306
    Subdomains: 6.77449  Arrays: 0.538577
**************************************************************************
**************************************************************************
Init matrices:
Init matrices time: 3.94435
**************************************************************************
**************************************************************************
Beginning assembly:
Assembly etime: 10.4759
Total memory: 32480, free memory: 27496
Assembly time: 15.5147
**************************************************************************
**************************************************************************
Beginning initBDDCMatrices:
Total memory: 32480, free memory: 31434
InitBDDC time: 1.40766
**************************************************************************
**************************************************************************
Init solvers:
Total memory: 32480, free memory: 31312
Init solvers time: 0.478345
**************************************************************************
**************************************************************************
Computing decompositions:
Total memory: 32480, free memory: 28588
Decompositions time: 29.5687
**************************************************************************
**************************************************************************
Analyzing decompositions:
```

```
Total memory: 32480, free memory: 26540
Analyzing time: 1.8185
**************************************************************************
**************************************************************************
Beginning primalSpaceBase:
Total memory: 32480, free memory: 26540
Init time: 1.76713
**************************************************************************
**************************************************************************
Beginning gGamma:
Total memory: 32480, free memory: 26517
gGamma time: 0.51112
**************************************************************************
**************************************************************************
Beginning SchurC:
Total memory: 32480, free memory: 26517
SchurC time: 0.00739838
**************************************************************************
**************************************************************************
Beginning Interface:

    PCG it: 0
        Sxp time:   0.239155
        Error:      0.680579
        Minvr time: 0.230801
        It time:    0.470102

    PCG it: 1
        Sxp time:   0.239815
        Error:      0.0380233
        Minvr time: 0.231465
        It time:    0.471426

    PCG it: 2
        Sxp time:   0.244016
        Error:      0.00353318
        Minvr time: 0.229591
        It time:    0.47376
```

```
PCG it: 3
    Sxp time:   0.238542
    Error:      0.000965327
    Minvr time: 0.22976
    It time:    0.468469

PCG it: 4
    Sxp time:   0.239998
    Error:      0.000134457
    Minvr time: 0.231926
    It time:    0.472081

PCG it: 5
    Sxp time:   0.24123
    Error:      6.12135e-06
    Minvr time: 0.229665
    It time:    0.471042

PCG it: 6
    Sxp time:   0.237612
    Error:      1.68994e-06
    Minvr time: 0.230152
    It time:    0.467906

PCG it: 7
    Sxp time:   0.240175
    Error:      3.22849e-07
    Minvr time: 0.22718
    It time:    0.46751

PCG it: 8
    Sxp time:   0.239816
    Error:      3.55023e-08
    Minvr time: 0.231481
    It time:    0.471448

PCG it: 9
    Sxp time:   0.24018
    Error:      4.11124e-09
```

```
PCG results:

     It count: 9, error: 4.11124e-09

Total memory: 32480, free memory: 26517

Interface time: 4.73616

**************************************************************************

**************************************************************************

Beginning Subdomains:

Total memory: 32480, free memory: 26517

Subdomains time: 0.130793

**************************************************************************

**************************************************************************

Beginning verification:

verification time: 0.0569356        error: 9.08821e-06      Interface error: 9.08821e-06

**************************************************************************

Total memory: 32480, free memory: 26517

Execution ended successfully, etime: 67.662
```

```
BDDC parameters
        NN:  2049   NSD: 4  ThreadNum: 8
**********************************************************************
Creating mesh:
Boundary: 2.46646
Partition: 2.1169
Interface: 1.1951
Subdomains: 0.736724
Corners: 0.00145488
Mesh created, total etime: 7.15781, mesh creation etime: 7.10566
    Subdomains: 6.51684  Arrays: 0.588823
**********************************************************************
**********************************************************************
Init matrices:
Init matrices time: 0.419357
**********************************************************************
**********************************************************************
Beginning assembly:
Assembly etime: 1.45316
Total memory: 32480, free memory: 29533
Assembly time: 4.20443
**********************************************************************
**********************************************************************
Beginning initMatrices:
Total memory: 32480, free memory: 31389
Init time: 0.761133
**********************************************************************
**********************************************************************
Init solvers:
Total memory: 32480, free memory: 31125
Init solvers time: 0.491833
**********************************************************************
**********************************************************************
Beginning primalSpaceBase:
Total memory: 32480, free memory: 30965
Init time: 6.7362
**********************************************************************
**********************************************************************
Beginning gGamma:
```

```
Total memory: 32480, free memory: 30803
gGamma time: 1.13191
**************************************************************************
**************************************************************************
Beginning SchurC:
Total memory: 32480, free memory: 30803
SchurC time: 0.000311886
**************************************************************************
**************************************************************************
Beginning Interface:

    PCG it: 0
        Sxp time:   1.62579
        Error:      0.0894169
        Minvr time: 2.91069
        It time:    4.53663

    PCG it: 1
        Sxp time:   1.61929
        Error:      0.0133303
        Minvr time: 2.91343
        It time:    4.53287

    PCG it: 2
        Sxp time:   1.55563
        Error:      0.00218644
        Minvr time: 2.9094
        It time:    4.46515

    PCG it: 3
        Sxp time:   1.40901
        Error:      0.00115025
        Minvr time: 2.91006
        It time:    4.3192

    PCG it: 4
        Sxp time:   1.35482
        Error:      0.000113899
        Minvr time: 2.92652
```

```
        It time:    4.28149

    PCG it: 5
        Sxp time:   1.24077
        Error:      2.11011e-05
        Minvr time: 2.9296
        It time:    4.17052

    PCG it: 6
        Sxp time:   1.10877
        Error:      5.09121e-06
        Minvr time: 2.90084
        It time:    4.00973

    PCG it: 7
        Sxp time:   0.952878
        Error:      2.42562e-07
        Minvr time: 2.87756
        It time:    3.83055

    PCG it: 8
        Sxp time:   0.844861
        Error:      3.10389e-08
        Minvr time: 2.87912
        It time:    3.72411

    PCG it: 9
        Sxp time:   0.629162
        Error:      1.22616e-09
PCG results:
     It count: 9, error: 1.22616e-09
Total memory: 32480, free memory: 30739
Interface time: 41.4334
**************************************************************************
**************************************************************************
Beginning Subdomains:
Results[4]:
     It count: 1253, error: 9.92208e-13
Results[0]:
```

```
     It count: 1216, error: 9.35958e-13
Results[2]:
     It count: 1278, error: 9.89931e-13
Results[12]:
     It count: 1294, error: 9.75607e-13
Results[6]:
     It count: 1307, error: 9.93249e-13
Results[10]:
     It count: 1321, error: 9.58978e-13
Results[14]:
     It count: 1329, error: 9.91439e-13
Results[8]:
     It count: 1278, error: 9.8993e-13
Results[5]:
     It count: 1287, error: 9.4652e-13
Results[1]:
     It count: 1253, error: 9.92208e-13
Results[13]:
     It count: 1321, error: 9.13823e-13
Results[7]:
     It count: 1321, error: 9.13823e-13
Results[3]:
     It count: 1294, error: 9.75607e-13
Results[11]:
     It count: 1329, error: 9.91439e-13
Results[9]:
     It count: 1307, error: 9.93248e-13
Results[15]:
     It count: 1294, error: 9.81544e-13
Total memory: 32480, free memory: 30699
Subdomains time: 1.61479
************************************************************************
************************************************************************
Beginning verification:
verification time: 0.0371686        error: 1.61214e-05      Interface error: 1.61214e-05
************************************************************************
Total memory: 32480, free memory: 30699
Execution ended successfully, etime: 64.25
```