

UNIVERSIDAD CATÓLICA DEL MAULE

Facultad de Ciencias de la Ingeniería

Escuela de Ingeniería Civil Informática

PROFESOR GUÍA

Dr. Ricardo Barrientos

Knn

Cristofher Andrés Rojas Rojas

Tesis para optar al Título Profesional de Ingeniero Civil

Informático

TALCA, JUNIO, 2017

UNIVERSIDAD CATÓLICA DEL MAULE
FACULTAD DE CIENCIAS DE LA INGENIERÍA
ESCUELA DE INGENIERÍA CIVIL INFORMÁTICA

TESIS PARA OPTAR AL
TÍTULO PROFESIONAL DE INGENIERO CIVIL INFORMÁTICO

KNN

Cristofher Andrés Rojas Rojas

COMISIÓN EXAMINADORA

FIRMA

Profesor Guía

Dr. Ricardo Barrientos

Universidad Católica del Maule

Profesor Evaluador

Universidad Católica del Maule

Profesor Evaluador

Universidad Católica del Maule

NOTA FINAL EXAMEN

TALCA, JUNIO, 2017

Agradecimientos

Cristofher Rojas

, 2017

Resumen

Hoy en día, es indudable la utilización de un computador para realizar análisis más profundos, dado que de otro modo se emplearía demasiado tiempo, por esto es fundamental contar con algoritmos y/o software que nos permitan mejorar las respuestas de tiempo al realizar dichos análisis.

En el presente trabajo se desarrolla un software el cual fue pensado para ser utilizado por usuarios que no cuenten con conocimientos en el área de programación, por tanto no puedan programar el algoritmo Knn ni secuencial ni paralelamente. Este software esta desarrollado de manera que pueda ser intuitiva su utilización, donde solo deben cargar los datos, tanto la base de datos como las consultas, estas deben estar en un archivo de texto plano, para estos posee botones para examinar y localizar en archivo en cuestión, no obstante puede ser que el usuario no sepa bien el nombre de archivo o no seleccione el archivo correcto (tanto base de datos como consulta), de manera que el software cuenta con una vista previa de dichos archivos de tal manera el usuario puede verificar los datos antes de realizar algún procedimiento.

No obstante el proceso anteriormente descrito es válido para todos los algoritmos K-nn que posee el software, mas no es similar para todos los algoritmos debido a sus entradas particulares, dado que el software cuenta con el algoritmo secuencial Knn, algoritmo paralelo multi-hilos knn (utilizando la librería OpenMP), algoritmo paralelo Intel Xeon Phi (Utilizando la librería OpenMP), algoritmo paralelo GPU (utilizando la librería de paralelismo en CUDA).

De manera que el algoritmo secuencial utiliza como campos de entrada la base de datos y la consultas, además el usuario debe indicar el tamaño del vector, como también la cantidad de elementos más cercanos deseados (K). A su vez los algoritmos paralelos (Multi-Hilos y Intel Xeon Phi) utilizan las bases de datos y las consultas, además el

usuario debe indicar el tamaño del vector, como también la cantidad de elementos más cercanos deseados(K) de la misma manera que lo hace el algoritmo anterior pero estos algoritmo cuentan con una diferencia debido a que se debe indicar el número de hilos con los que se desea que se realice el proceso. Por defecto el software reconoce la cantidad de procesadores que tiene el computador y los establece como la cantidad de hilos a utilizar, este dato es solo un dato sugerido de manera que el usuario puede poner el numero que el desee. A su vez para la GPU se estableció un valor por defecto debido a las características de la implementación del algoritmo en GPU.

Cabe mencionar que los resultados pueden ser exportados a los siguientes formatos XLS (Archivos de Excel), DOC (Archivos word), TXT (Archivos de texto plano), PDF (Formato de documento portátil), de forma que el usuario pueda elegir el formato que más le acomode. A su vez para aquellos que si poseen conocimientos de programación pueden añadir sus métodos utilizando las entradas establecidas para cada caso. Dado que el software cuenta con un un menú para añadir nuevos métodos del algoritmo, de modo que el usuario en cuestión debe seleccionar a que tipo de algoritmo corresponde (ya sea secuencial, paralelo multi-hilos, paralelo Xeon phi, paralelo GPU). con esto debe seleccionar su archivo fuente (respetando ciertas características establecidas previamente), de manera similar al caso de las bases de datos y consultas el software cuenta con una vista previa para el chequeo correcto de el archivo cargado, además el usuario debe indicar el nombre del menú que desee.

Índice general

1. Introducción	2
1.1. Motivación	3
1.2. Objetivos	4
2. Marco Teórico	6
2.1. Sistemas Multi-core y OpenMP	6
2.1.1. Threads	6
2.2. Coprocesadores	7
2.2.1. Intel Xeon Phi	8
2.2.2. NVIDIA GPU	9
2.3. Plataforma JAVA	18
2.3.1. Lenguaje JAVA	18
2.3.2. La maquina virtual	19
2.3.3. Bibliotecas	19
2.4. Metodología	20
2.5. Selección de algoritmos	22
2.6. Selección de lenguajes	25
3. Estado del Arte	34
4. Desarrollo	37
4.1. Introducción	37

ÍNDICE GENERAL

v

4.2. Desarrollo interfaz	37
4.3. Integración de algoritmo K-nn Secuencial	42
4.3.1. Funciones claves del algoritmo secuencial	44
4.4. Integración de algoritmo K-nn Paralelo Multi-hilos	48
4.4.1. Funciones claves del algoritmo multi-hilos	50
4.5. Integración de algoritmo K-nn Paralelo Xeon Phi	50
4.5.1. Funciones claves del algoritmo Xeon phi	54
4.6. Integración de algoritmo K-nn Paralelo GPU	55
4.7. Métodos de exportación de resultados	56
4.8. Nuevo modulo Añadir menú	58
5. Conclusiones y Trabajos Futuros	62
5.1. Conclusiones	62
5.2. Trabajos futuros	63

Índice de figuras

2.1. Plataforma multi-core.	7
2.2. Diagrama de una arquitectura MIC de un núcleo de un coprocesador Xeon Phi	10
2.3. Diagrama de una arquitectura MIC de un núcleo de un coprocesador Xeon Phi	11
2.4. Composición del <i>device</i>	12
2.5. Esquema de división de threads.	13
2.6. Composición de la GPU a) vs. CPU b).	14
2.7. Ejemplo de suma de vectores con CUDA.	17
2.8. Ejemplos para ilustrar la sincronización de los threads de un <i>warp</i> . . .	18
2.9. Modelo incremental	20
2.10. Ejemplo de Max Heap	23
2.11. Ejemplo de Min Heap	23
2.12. Ejemplo de Min Max Heap	24
2.13. Gráfica de tiempo de velocidad	27
2.14. Gráfica de tiempo de velocidad	28
4.1. Interfaz gráfica inicial del software	39
4.2. Interfaz gráfica item multihilos del software	41
4.3. Interfaz gráfica item secuencial del software	42
4.4. Mensajes al usuario	43
4.5. Interfaz gráfica de ventana de exportación de resultados	57

<i>ÍNDICE DE FIGURAS</i>	VII
4.6. Interfaz gráfica de ventana de guardar archivo	57
4.7. Interfaz gráfica de ventana Agregar menú	61

Índice de tablas

2.1. Tabla comparativa CPU vs GPU	15
---	----

Índice de Algoritmos

1.	Algoritmo Knn básico	3
2.	Estructura para realizar los <i>Heap</i>	43
3.	Utilización de <i>malloc</i>	44
4.	Proceso iterativo de una consulta K-nn.	45
5.	Cálculo de distancias entre vectores.	45
6.	Valor de la raíz del heap.	46
7.	Realiza la inserción de un elemento al heap.	46
8.	Realiza la extracción de un elemento al heap.	47
9.	Realiza la extracción y la inserción de un elemento al heap.	47
10.	Ejecutar un programa o ejecutable desde <i>Java</i>	48
11.	Proceso iterativo de una consulta K-nn multi-core.	49
12.	Utilización de <i>malloc</i>	52
13.	Proceso iterativo de una consulta K-nn Xeon phi.	53
14.	Función distancia <i>Xeon phi</i>	54
15.	Pasa una matriz a vector.	54
16.	Valor de la raíz del heap.	55
17.	Método de exportación de resultados a <i>Excel</i>	59
18.	Método de exportación de resultados a <i>PDF</i>	59
19.	Método de exportación de resultados a <i>TXT</i>	60

Capítulo 1

Introducción

En esta tesis se propone un software pensado en usuarios que poseen poco o nulo conocimientos de programación, de modo que puedan utilizar eventualmente este software para resolver consultas K-nn. Para esto se han recopilado distintas versiones del algoritmo K-nn propuestos en la literatura, de modo que se presentan al menos cuatro algoritmos diferentes, estos son: Algoritmo secuencial K-nn, Algoritmo paralelo K-nn utilizando la librería OpenMP (Multi-núcleo o Multi-hilos), Algoritmo paralelo K-nn utilizando librería OpenMP para Intel Xeon Phi, Algoritmo paralelo K-nn GPU. Se utilizó JAVA 8 [Oracle, 2017] para programar la interfaz gráfica del software debido a que este lenguaje es soportado en los distintos sistemas operativos:

Windows en sus versiones 10, 8.x (Escritorio), 7 SP1, Vista SP2, Windows Server 2008 R2 SP1 (64 bits), Windows Server 2012 y 2012 R2 (64 bits) y los recursos necesarios son 128 MB de RAM, un espacio mínimo de 124 MB, procesador pentium 2 a 266 MHz, a su vez en MAC OSx equipos Mac con Intel que ejecuta Mac OS X 10.8.3+, 10.9+, la instalación se debe realizar en modo administrador, En linux en cambio esta disponible para Oracle Linux desde la versión 5.5+ o en versiones superiores, en Red Hat Enterprise Linux desde la versión 5.5+ o en versiones superiores, Suse Linux Enterprise Server desde 10 SP2+ o versiones superiores y en Ubuntu Linux en desde versiones 12.04 LTS o en versiones superiores.

Algoritmo 1 Algoritmo Knn básico

Require: \mathbf{D} , es el conjunto de datos de entrenamiento, El objeto de prueba \mathbf{Z} , el cual es un vector descriptor y \mathbf{L} es el conjunto de clases con objetos etiquetados.

Ensure: $C_z \in \mathbf{L}$, la clase de \mathbf{z}

```
1: for objeto  $y$   $D$  do  
2:   Calcular  $d(\mathbf{z}, \mathbf{y})$ , la distancia entre  $\mathbf{z}$  e  $\mathbf{y}$ ;  
3: end for
```

Para la programación de los algoritmos se empleo el lenguaje de programación C [Kernighan and Ritchie, 2006] dado que este lenguaje puede ser utilizando en diversos sistemas operativos, además de poder utilizar librerías que permiten el uso del paralelismo en los distintos tipos de plataformas paralelas, como son Multi-Hilos, Intel Xeon Phi y GPU.

Las plataformas paralelas que se emplean en esta tesis son las mencionadas anteriormente, tanto para Multi-hilos y Intel Xeon Phi se utiliza la librería OpenMP [Chapman et al., 2007][Nichols et al., 1996], esta librería esta pensada para lenguajes como C, C++ y Fortran, disponible además para diversos tipos de arquitecturas, incluidas algunas plataformas como Windows y Unix. A su vez para GPU se cuenta con librerías propuestas por NVIDIA y CUDA [cud,] la cual permite trabajar las concurrencias en GPU y poder utilizar los hilos y bloques que propone GPU.

1.1. Motivación

K-nn (*K nearest neighbors*) es uno de los métodos de minería de datos más utilizado, este es un clasificador basado en instancias (aprendizaje por analogía). De manera que se buscan los casos más parecidos y la clasificación se realiza en función de la clase analizada, este es conocido por ser un paradigma perezoso debido a ser un análisis exhaustivo con la consulta y todos lo elementos del conjunto (Vease algoritmo 1.1).

Si bien el algoritmo Knn es bastante lento, al ser exhaustivo (evaluar cada

opción del conjunto) es sumamente exacto con una complejidad del tipo $O[D \times N]$ donde N es el número de tuplas y D su dimensión, suele ser utilizado en muchos campos, como lo son minería de datos usado para estimar una función de densidad $\mathbf{F}(\mathbf{x}/C_j)$ donde predice el valor x para la clase C_j . Este algoritmo además es muy utilizado en el reconocimiento de patrones, imaginemos dado el caso de una base de datos con rostros y se desea identificar a quien de ellos corresponde un rostro en consulta, en este caso con las imágenes de los rostros se realizan descriptores de la imágenes tanto para la base de datos como para la consulta en cuestión; Este descriptor esta compuesto por un vector lineal el cual con el cálculo de la distancia entre dos puntos se obtiene la similitud entre ambos. Se debe considerar ciertos aspectos para establecer una variable K , dado que este depende fundamentalmente de los datos, de modo que si es se selecciona un valor muy pequeño estos pueden ser ruidos (ser elementos de otra clase) y si se selecciona un K muy grande puede pasar exactamente lo mismo (abarcar elementos de otra clase). De tal manera que la elección del K es recomendada a un valor pequeño de acuerdo a la cantidad de datos.

No obstante, este algoritmo puede llevarse a un sin fin de otros usos, asociados a la clasificación y reconocimiento de patrones. En este punto es importante añadir que existen diferentes formas de emplear y aplicar el algoritmo de modo que en situaciones es utilizado como un clasificador previo a análisis más potentes.

1.2. Objetivos

El objetivo general de esta tesis se centra en la recopilación de algoritmos de vecinos más cercanos (k-nearest neighbor) encontrados en la literatura que proponen una solución más óptima en tiempos de ejecución, para esto se estableció que los objetivos que cumplen con esta condición son aquellos que han sido propuestos en las líneas de paralelismo, donde a su vez se emplean heap para la resolución de este algoritmo.

Para poder realizar la unión de los diferentes algoritmos paralelos, se propone

un software el cual permite la agrupación de algoritmos de diferentes plataformas paralelas, además de añadir el algoritmo secuencial base como un agregado. En base a lo anteriormente mencionado se pretende que el software sea diseñado en consideración a usuarios los cuales no estén familiarizados con la programación de estos metodos, de modo que la utilización del software no se vea ligada a sus conocimientos en áreas de programación.

Los objetivos específicos son los siguientes:

- Realizar la interfaz gráfica del software

Esta se realiza en JAVA debido a su capacidad de ser interpretada tanto en Windows, Linux y OSX

- Añadir los algoritmos K-nn al software

K-nn multi-núcleo

K-nn Xeon Phi

K-nn GPU

- Añadir menú para añadir nuevos métodos

- Permitir al usuario exportación de los resultados en los siguientes formatos

Excel

Word

Archivo de texto plano (TXT)

Documento portátil PDF

Capítulo 2

Marco Teórico

2.1. Sistemas Multi-core y OpenMP

En este capítulo se describen los conocimientos básicos sobre programación multi-núcleos, programación en coprocesadores tales como programación en Intel Xeon Phi, programación en GPU y trabajos relacionados que dan base a esta tesis.

Actualmente las arquitecturas de los dispositivos personales están diseñadas con multiprocesadores o coprocesadores, desde un equipo de bolsillo como un celular (Smartphone) hasta un computador personal (notebook), esto quiere decir que los dispositivos cuentan con varios núcleos en una sola CPU (Unidad central de procesamiento) en el caso de multiprocesadores y con más de una CPU en el caso de un Coprocesador.

Imagen de referencia 2.1

2.1.1. Threads

Existen varias bibliotecas que permiten la programación en multi-hilos, con esto somos capaces de ejecutar hilos en paralelo, ejecutándose en un núcleo diferente.

Existen diferentes modelos y estándares que permiten la programación de múltiples hilos, tales como Pthreads [Nichols et al., 1996], TBB [Reinders, 2007] u OpenMP

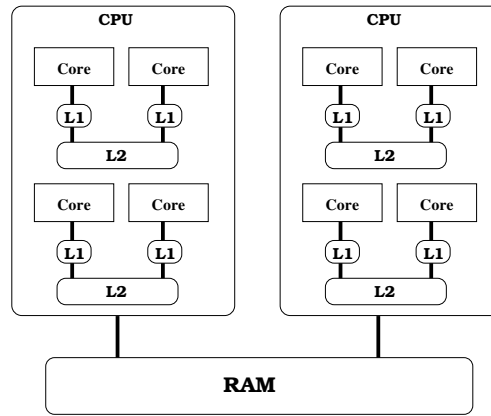


Figura 2.1: Plataforma multi-core.

[Chapman et al., 2007], en particular en esta tesis se selecciono el uso de la biblioteca OpenMP que fue desarrollada en 1997. Además es importante mencionar el hecho de que actualmente los compiladores en esta linea son bastante robustos, como tambien el hecho de que el equipo que gestiona OpenMP esta conformado por diferentes compañías (AMD, Intel, Sun Mycrosystem y otros) y no siendo gestionada por una sola, esto nos proporciona confianza.

2.2. Coprocesadores

Un Coprocesador es utilizado para desligar funciones de la CPU, en esta linea existen desde coprocesadores matemáticos hasta coprocesadores gráficos, estos poseen características y especificaciones a ciertos tipos de operaciones, por ejemplo un coprocesador matemático solo esta desarrollado para revolver problemas aritméticos o afines al área esto quiere decir que pueden realizar operaciones matemáticas a una velocidad mayor que el procesador principal (CPU), a su vez un coprocesador gráfico esta diseñado en otra área. Estos proporcionan al hardware una aceleración en la operación solo cuando se ejecuta una tarea o software que se haya diseñado para aprovechar el coprocesador, las instrucciones para un coprocesador son diferentes que

las instrucciones de una CPU de modo que existe un programa que detecta la existencia de un coprocesador y entonces se ejecutan las instrucciones escritas explícitamente para aquel coprocesador, por tanto existen diferentes líneas de coprocesadores en el mercado pero en esta tesis solo consideramos Intel Xeon Phi y GPU NVIDIA.

2.2.1. Intel Xeon Phi

El coprocesador Intel Xeon Phi [xco,] [Wang, 2014] [LA, 2017] consta de hasta 72 conectados en un anillo bidireccional de alto rendimiento en el chip. El coprocesador ejecuta un sistema operativo linux y es compatible con todas las principales herramientas de desarrollo de Intel como C / C ++, Fortran, MPI y OpenMP. El coprocesador se conecta a un procesador Intel Xeon (el host) a través del bus PCI Express (PCIe). Las propiedades más importantes de la arquitectura MIC se muestran en figura 2.2.1.

Core: El núcleo de la intel Xeon phi (Scalar Unit en la figura 2.2.1) es una arquitectura de orden (basada en la familia de procesadores Intel Pentium), Implementa instrucciones de recuperación y decodificación para 4 hilos (hardware) por núcleo. Las instrucciones vectoriales que posee el coprocesador Intel Xeon Phi, utiliza una unidad de punto flotante (VPU) dedicada, con un ancho de 512-bit, la que está disponible en cada núcleo. Un núcleo puede ejecutar dos instrucciones por ciclo de reloj, una en U-pipe y otra en V-pipe (no todos los tipos de instrucciones pueden ser ejecutadas por el V-pipe). Cada núcleo compone una interconexión en anillo mediante la CRI (Core Ring Interface).

Vector Processing Unit (VPU): La VPU incluye la EMU (Extended Math Unit), y es capaz de realizar 16 operaciones en punto flotante con precisión simple por ciclo, 16 operaciones de enteros de 32-bit por ciclo u 8 operaciones en punto flotante con precisión doble por ciclo.

L1 Cache: Tiene una caché L1 de 32KB para instrucciones y de 32KB para datos, asociativa de 8-vías, con tamaño de línea de caché de 64 bytes. Tiene una latencia de cargado de 1 ciclo, que significa que un valor entero cargado desde L1 puede ser usado en el siguiente ciclo por una instrucción entera (instrucciones vectoriales tienen diferente latencia que las enteras).

L2 Cache: Cada núcleo posee 512KB de caché L2. Si ningún núcleo comparte algún dato o código, entonces el tamaño total de la caché L2 es de 31 MB. Por otro lado, si todos los núcleos comparten exactamente el mismo código y datos en perfecta sincronía, el tamaño total de la caché L2 sería solo de 512KB. Su latencia es de 11 ciclos de reloj.

Ring: Incluye interfaces y componentes, ring stops, ring turns, direccionamiento y control de flujo. Un coprocesador Xeon Phi tiene 2 de estos anillos, uno viajando en cada dirección (Figure 2.2.1).

La Figura 2.2.1 ilustra una visión general de la arquitectura, donde se muestra como los núcleos están conectados con cachés coherentes mediante un anillo bidireccional de 115GB/sec. La figura también muestra una memoria GDDR5 con 16 canales de memoria, que alcanzan una velocidad de hasta 5.5GB/sec.

2.2.2. NVIDIA GPU

Todas las implementaciones sobre GPU se realizaron usando el modelo de programación CUDA [cud,] de NVIDIA. Este modelo hace una distinción entre el código ejecutado en la CPU (host) con su propia DRAM (host memory) y el ejecutado en GPU (device) sobre su DRAM (device memory). La GPU se representa como un coprocesador capaz de ejecutar funciones denominadas kernels, y provee extensiones

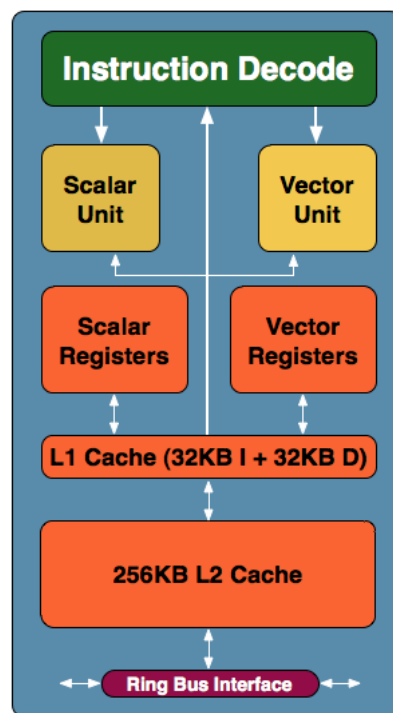


Figura 2.2: Diagrama de una arquitectura MIC de un núcleo de un coprocesador Xeon Phi

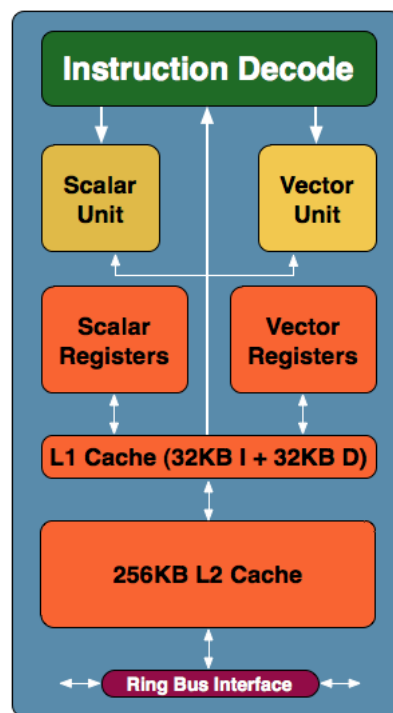
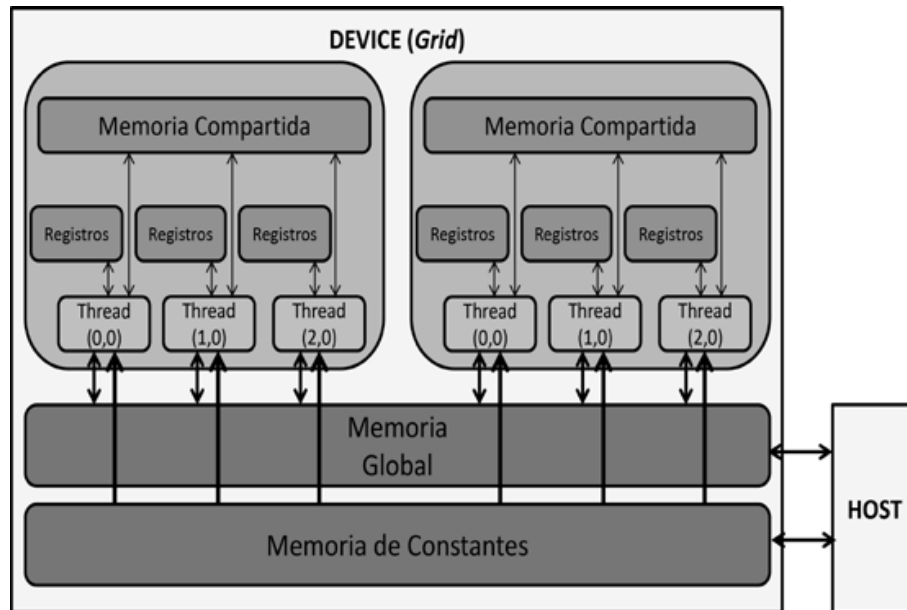


Figura 2.3: Diagrama de una arquitectura MIC de un núcleo de un coprocesador Xeon Phi

Figura 2.4: Composición del *device*

para el lenguaje C que permiten alojar datos en la memoria de la GPU y transferir datos entre GPU y CPU. Por lo tanto, la GPU sólo puede ejecutar las funciones declaradas como kernels dentro del programa, y sólo puede usar variables alojadas en device memory, lo cual significa que todos los datos necesarios deberán ser movidos a esta memoria de forma previa a la ejecución la figura 2.6 muestra un esquema de la diferencia entre una CPU y una GPU. Esta última está especializada para cómputo intensivo, y por lo tanto más transistores están dedicados para el procesamiento de datos en vez de caching de datos y control de flujo.

Esto implica que la GPU es capaz de realizar un mayor número de operaciones aritméticas en punto flotante que una CPU. El correcto aprovechamiento del sistema de memoria en la GPU es primordial. La figura 2.4 muestra un esquema del hardware de una GPU, la que consta de las siguientes características:

- Una GPU está constituida por una serie de *multiprocesadores*, donde cada uno de éstos cuenta con varios núcleos.
- Cada núcleo cuenta con sus propios registros y todos los núcleos del mismo

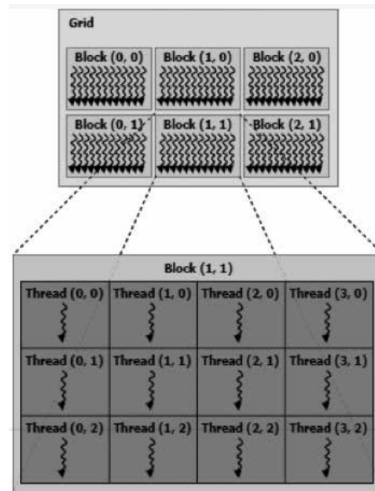


Figura 2.5: Esquema de división de threads.

multiprocesador comparten una *shared memory*.

- **Shared memory** es una memoria que se encuentra próxima a los núcleos del multiprocesador, y por lo tanto es de muy baja latencia.
- Cada multiprocesador cuenta con un par de memorias caché compartidas por todos los núcleos, *constant cache* y *texture cache*.
- **Constant cache** es una caché de datos de la constant memory, que es una memoria de sólo lectura de reducido tamaño alojada en *device memory*.
- **Texture cache** es una caché de la texture memory, que es una memoria optimizada para localidad espacial 2D, y está alojada en *device memory*.
- **Device memory** es una memoria compartida por todos los multiprocesadores, y es de gran tamaño.

Organización y ejecución de threads

Los threads son organizados en *CUDA Blocks*, y cada uno de estos se ejecuta completamente en un único multiprocesador. Esto permite que todos los threads de

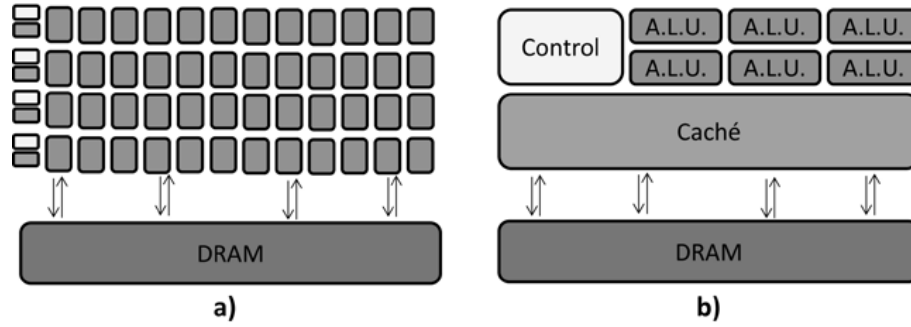


Figura 2.6: Composición de la GPU a) vs. CPU b).

un mismo CUDA Block compartan la misma *shared memory*. Un *kernel* puede ser ejecutado por un número limitado de CUDA Blocks, y cada CUDA Block por un número acotado de threads. Sólo los threads que pertenecen al mismo CUDA Block pueden ser sincronizados. Es decir, hilos que pertenezcan a CUDA Blocks distintos sólo pueden ser sincronizados realizando un nuevo lanzamiento de kernel.

Para soportar la gran cantidad de threads en ejecución se emplea una arquitectura SIMT (Single-Instruction, Multiple-Thread). La unidad de ejecución no es un thread, sino un *warp*, que es un conjunto de threads, y un *half-warp* corresponde a la primera o segunda mitad del conjunto de threads de un *warp*. La forma en que un CUDA Block es dividido en *warps* es siempre la misma, el primer conjunto de threads representan el primer *warp*, el segundo conjunto consecutivo de threads el segundo *warp* y así sucesivamente.

En la figura 2.5 se aprecia como jerarquía donde los threads se agrupan en estructuras de 1, 2 ó 3 dimensiones denominados bloques, los bloques a su vez, se agrupan en un grid de bloques que también puede ser de 1, 2 ó 3 dimensiones.

En la figura 2.6 se aprecia la comparación de una CPU (a) y GPU (b), estas poseen diferencias técnicas, como se aprecia en la siguiente tabla 2.1.

Tabla 2.1: Tabla comparativa CPU vs GPU

CPU	GPU
Procesador escalar	Procesador vectorial
Acceso aleatorio a memoria	Acceso secuencial a memoria
Modelo de programación muy conocido	Modelo de programación poco conocido
20 GB/s Ancho de banda	100 -150 GB/s Ancho de banda
Consumo de 1 Gflop/watt	Consumo de 10 Gflop/watt

Reducción de lecturas a memoria

Los accesos a memoria de un *half-warp* a device memory son fusionados en una sola transacción de memoria si las direcciones accedidas caben en el mismo segmento de memoria. Por lo tanto, si los threads de un *half-warp* acceden a n diferentes segmentos, entonces se ejecutarán n transacciones de memoria. Si es posible reducir el tamaño de la transacción, entonces se lee sólo la mitad de ésta.

Funciones atómicas

Existe un grupo de funciones atómicas que realizan operaciones de tipo *read-modify-write* sobre palabras de residentes en *global memory* o *shared memory*. Estas funciones garantizan que serán realizadas completamente sin interferencia de los demás threads. Cuando un thread ejecuta una función atómica, ésta bloquea el acceso a la dirección de memoria donde reside la palabra involucrada hasta que se complete la operación. Las funciones atómicas se dividen en funciones aritméticas y funciones a nivel de bits, a continuación se muestran algunas de las funciones aritméticas relevantes para el presente trabajo:

- `atomicAdd int atomicAdd(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor $(old+val)$. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- `atomicSub int atomicSub(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor (*old-val*). El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- `atomicMin int atomicMin(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor mínimo entre (*old-val*). El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

- `atomicMax int atomicMax(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor máximo entre (*old-val*). El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

Ejemplos

La figura 2.7 muestra un código que suma 2 vectores utilizando la GPU. Para esto primero se asigna memoria a los arreglos en device memory, luego se copian los valores necesarios a estas variables, y posteriormente se invoca al kernel con 1 CUDA Block (dado por la variable `N_BLOQUES`) y 200 threads por cada CUDA Block (dado por la variable `N`).

Cada thread obtiene su identificador (variable `tid`) usando variables predefinidas por la GPU, ID_{Thread} (que retorna el ID del thread en el CUDA Block), T_{Block} (que retorna la cantidad de threads de un CUDA Block) y ID_{Block} (que retorna el identificador del CUDA Block). Cada thread realiza la suma del elemento `tid`-ésimo de ambos arreglos (`dev_A` y `dev_B`). Y ya una vez terminado el kernel se copia el arreglo resultado a la memoria de CPU para poder imprimir los resultados, pues no se puede invocar una función para imprimir dentro del kernel en la GPU utilizada.

Los threads de un warp siempre están sincronizados, es decir, ningún thread se puede adelantar a otro que pertenezca al mismo warp en la ejecución de instrucciones.

```
#include <stdio.h>
#include <cuda.h>
#define N 200
#define N_BLOQUES 1

/* Se define un kernel llamado 'function' */
__global__ void function(float *A, float *B, float *result)
{
    /* Se obtiene el identificador del thread */
    int tid = ID_Thread + (T_Block * ID_Block);
    /* Cada thread realiza la suma de un elemento distinto */
    result[tid] = A[tid] + B[tid];
    return;
}

main()
{
    float host_A[N], host_B[N], host_result[N];
    float *dev_A, *dev_B, *dev_result;

    /* Asignamos memoria en device memory */
    cudaMalloc((void **)&dev_A, sizeof(float)*N);
    cudaMalloc((void **)&dev_B, sizeof(float)*N);
    cudaMalloc((void **)&dev_result, sizeof(float)*N);

    inicializar_valores(host_A, host_B, host_result);
    /* Se copian los valores a las variables alojadas en device memory */
    cudaMemcpy(dev_A, host_A, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, host_B, sizeof(float)*N, cudaMemcpyHostToDevice);

    /*Invocamos al kernel con 'N_BLOQUES' bloques y 'N' threads por bloque
       con las variables alojadas en device memory */
    function<<<N_BLOQUES, N>>>>(dev_A, dev_B, dev_result);

    /* Se copian los resultados a memoria de la CPU */
    cudaMemcpy(host_result, dev_result, sizeof(float)*N, cudaMemcpyDeviceToHost);

    imprimir_resultados(host_result);
    cudaThreadExit();
    return 0;
}
```

Figura 2.7: Ejemplo de suma de vectores con CUDA.

<pre> int tid = ID_{Thread} ... int ref1 = myArray[tid] * 1; __syncthreads(); myArray[tid + 1] = 2; __syncthreads(); int ref2 = myArray[tid] * 1; result[tid] = ref1 * ref2; ... </pre>	<pre> int tid = ID_{Thread} ... if (tid < warpSize) { int ref1 = myArray[tid] * 1; myArray[tid + 1] = 2; int ref2 = myArray[tid] * 1; result[tid] = ref1 * ref2; } ... </pre>
(a) Caption Subfigura 1	(b) Caption Subfigura 2

Figura 2.8: Ejemplos para ilustrar la sincronización de los threads de un *warp*.

Un ejemplo de esto lo muestra la figura 2.8, donde en el código 2.8(a) se deben realizar instrucciones de sincronización para garantizar el correcto valor del arreglo `result`, mientras que el código 2.8(b) es realizado solamente por los threads de un warp, y debido a esto no es necesario utilizar sincronización.

2.3. Plataforma JAVA

Una plataforma es el entorno de hardware o software en el cual se ejecuta un programa. A diferencia de las plataformas tradicionales como Linux, Windows y Solaris, la plataforma Java está basada sólo en software que se ejecuta sobre otras plataformas basadas en hardware. Por esto la plataforma Java y su software puede ser utilizado sobre variados sistemas operativos y hardware.

La plataforma Java está constituida de tres componentes: el lenguaje, la máquina virtual y las bibliotecas. [Nuñez, 2003, pág. 5]

2.3.1. Lenguaje JAVA

El lenguaje Java es un lenguaje de propósito general, de alto nivel, que utiliza el paradigma de orientación a objetos. Su sintaxis y tipos están basados principalmente en C++, sin embargo, las diferencias principales con éste son la administración de

memoria, siendo ejecutada por la máquina virtual automáticamente y no por el código de cada programa, y el soporte de procesos livianos o threads a nivel del lenguaje, que ayuda a controlar la sincronización de procesos paralelos. Estas características dan al lenguaje Java las propiedades de robustez y seguridad, evitando por ejemplo problemas de buffer overflow utilizados en ataques a sistemas. [Nuñez, 2003, pág. 5]

2.3.2. La maquina virtual

Los programas escritos en Java son compilados como archivos ejecutables de una máquina virtual llamada Java Virtual Machine (JVM). Existen implementaciones de esta máquina para múltiples plataformas, permitiendo ejecutar en diferentes arquitecturas el mismo programa ya compilado. La característica de independencia de la plataforma hace posible el libre intercambio de software desarrollado en Java sin necesidad de modificaciones, lo que ha sido llamado “Write once, Run anywhere”. [Deepak et al., 2001] Java es un lenguaje compilado e interpretado a la vez. Compilado ya que previo a su ejecución un programa debe ser transformado a un lenguaje intermedio, llamado Java bytecodes. Interpretado porque cada programa luego debe ser procesado y ejecutado por alguna implementación de la JVM específica a la plataforma. [Nuñez, 2003, pág. 6]

2.3.3. Bibliotecas

El conjunto de bibliotecas del lenguaje es conocido como la Java Application Programming Interface (Java API) que es un gran conjunto de componentes que proporcionan diferentes herramientas para el desarrollo de programas Java. La API de Java está agrupada en conjuntos de bibliotecas relacionadas conocidas como paquetes, que contienen grupos de elementos básicos de Java, llamados clases e interfaces. [Nuñez, 2003, pág. 6]

2.4. Metodología

La metodología utilizada es el desarrollo incremental (iterativo y creciente)

Defino por José Joskowicz de la siguiente manera:

”El modelo incremental consiste en un desarrollo inicial de la arquitectura completa del sistema, seguido de sucesivos incrementos funcionales. Cada incremento tiene su propio ciclo de vida y se basa en el anterior, sin cambiar su funcionalidad ni sus interfaces. Una vez entregado un incremento, no se realizan cambios sobre el mismo, sino únicamente corrección de errores. Dado que la arquitectura completa se desarrolla en la etapa inicial, es necesario, al igual que en el modelo en cascada, conocer los requerimientos completos al comienzo del desarrollo. Respecto al modelo en cascada, el incremental tiene la ventaja de entregar una funcionalidad inicial en menor tiempo.”

[Joskowicz, 2008, pág. 6 - 7]

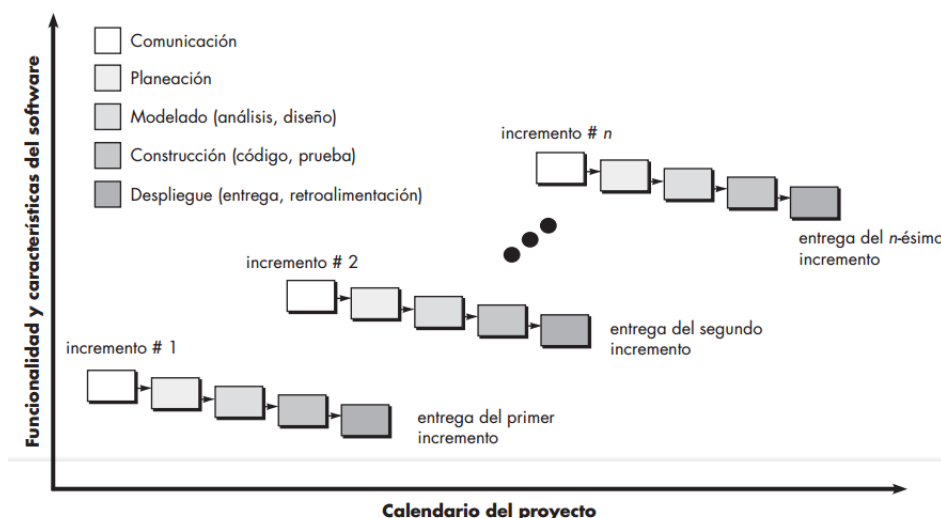


Figura 2.9: Modelo incremental

En relación con la figura 2.9, el modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce “incrementos” de software susceptibles de entregarse [McDermid, 1993] de manera parecida a los incrementos producidos en un flujo de proceso evolutivo.

[Pressman and Troya, 1988, pág. 35]

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande tal vez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante. [Pressman and Troya, 1988, pág. 36]

Se implemento esta metodología debido a la característica propia del software, donde una etapa del software no estaba ligada a otra, de manera que se agrego algoritmo por algoritmo a medida que su desarrollo se finalizaba correctamente y luego se añadían al software.

Ventajas del modelo

El modelo iterativo y creciente presenta excelentes ventajas para ser aprovechadas en el desarrollo del software propuesto, tales como:

- No se debe esperar a que el desarrollo del sistema o software este completo para poder ser utilizado por usuarios. Donde el primer incremento es el más importante y da paso a que los usuarios interesados ya puedan utilizar el sistema o software.
- Los usuarios pueden utilizar incrementos iniciales como el prototipo, esto genera una experiencia real sobre el cumplimiento de los requisitos del sistema o software.
- Como el sistema se va probando gradualmente existe menos probabilidad de

errores en la etapa final

- Se entregan primero los requisitos principales de modo que al integrar nuevas características
- Al ser un proceso iterativo da una mejor retroalimentación del sistema o software final

En base a las ventajas mencionadas anteriormente y en base a la característica del software, se opto por emplear esta metodología.

- Su primer incremento correspondió a la creación de la interfaz gráfica, ya que esta es la base de todo el software que se desarrollo y luego correspondió la comunicación entre la interfaz gráfica en JAVA y el algoritmo k-nn secuencial en el lenguaje de programación C.

En base a las ventajas del modelo, el primer incremento cumple con el punto anterior de entregar los requisitos principales y luego poder integrar nuevas características.

2.5. Selección de algoritmos

La selección de algoritmos se realizo bajo la búsqueda de los algoritmos propuestos en la literatura que resuelven el algoritmo K-nn, con esto se realizo el filtro en base a los algoritmos que utilizan Heap.

Estructura Heap

En programación un heap es una estructura de datos del tipo "árbol" con información de un conjunto ordenado de datos. De manera que es utilizada para implementar colas con prioridad. En este tipo de colas, el elemento a ser eliminado o borrado es el cual tiene la mayor o menor prioridad. A su vez en cualquier momento se pueden insertar elementos con una prioridad arbitraria.

Existen tres tipos de heap, de acuerdo a sus características de orden

- Max heap: Es un árbol en el cual el valor del nodo padre es mayor a los valores de los nodos de sus hijos (en caso de poseer algún hijo), además de ser un árbol binario completo. Imagen de referencia 2.10

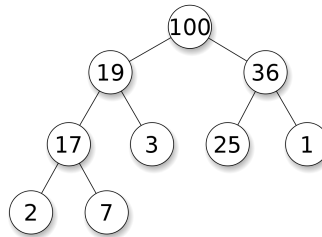


Figura 2.10: Ejemplo de Max Heap

- Min heap: Es un árbol en el cual el valor del nodo padre es menor a los valores de los nodos de sus hijos (en caso de poseer algún hijo), además de ser un árbol binario completo. Imagen de referencia 2.11

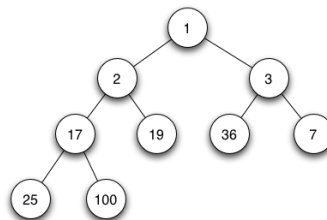


Figura 2.11: Ejemplo de Min Heap

- Min-Max heap: Es un árbol binario, que se construye a partir de un orden min o max de acuerdo a los valores de los nodos, el valor del nodo padre debe ser menor que el valor del nodo hijo (En caso de poseer algún hijo), si posee uno o dos hijos estos tiene un valor superior tanto al nodo padre como también a los sus correspondientes nodos hijos, este es el procedimiento secuencialmente hasta el ultimo nodo del árbol. Imagen de referencia 2.12

Las operaciones básicas de un heap son las siguientes:

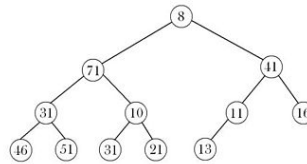


Figura 2.12: Ejemplo de Min Max Heap

1 Creación de un heap vacío.

Si no existe un heap se crea la estructura.

2 Inserción de un nuevo elemento en la estructura heap.

Si existe un heap se inserta el elemento aplicando criterios de ordenación, de modo de respetar el orden de los tipos de heap vistos anteriormente, el árbol se ordena de manera tal que siempre cumpla con el orden binario.

Si no existe un heap se crea uno y se añade como el nodo padre.

3 Eliminación del elemento más grande del heap.

Si existe un heap se puede eliminar el elemento raíz (nodo padre) del heap y el heap debe respetar el orden binario de modo que se reordena el árbol y uno de sus nodos hijos debe ocupar la raíz vacía, luego de haber ocupado ese nodo se debe comparar con los hijos si otro puede ocupar el lugar de nodo padre de acuerdo a las características del tipo de heap.

En esta tesis se han utilizado para los algoritmos el heap de tipo Max Heap, esto debido a que el algoritmo busca los K vecinos mas cercanos donde realiza el calculo exhaustivo de la distancia entre dos elementos vectorizados dentro de una base de datos con un numero variado de elementos para comparar, de modo que en un comienzo se debe ir llenado el heap de tamaño K , con los primeros K elementos analizados, el heap realiza su organización de árbol binaria y el nodo padre siempre sera el elemento de mas distancia al elemento comparado. De esta manera si se descubre una nueva distancia mas cercana al valor del nodo padre (raíz del árbol), se procese a eliminar el valor del nodo actual y se realiza la inserción del nuevo valor y se reordena el árbol nuevamente.

Con esto solo se debe comparar con el valor del nodo padre (raíz) con cada una de las distancias que se van obteniendo y no con todos los valores dentro del heap, siendo muy útil esta estructura para realizar las consultas K-nn y además de poder reducir tiempos en comparación debido a lo mencionado anteriormente de solo comparar con el valor del nodo padre (raíz del árbol).

2.6. Selección de lenguajes

Para la selección de un lenguaje de programación óptimo para el algoritmo K-nn, se debió considerar ciertas condiciones

- Tipo de lenguaje
- Tiempos de ejecución de instrucciones
- Tiempos de ejecución de estructura heap
- Posibilidad de utilizar librerías para trabajos en paralelismo

Tipo de lenguaje

En esta área se establecen muchas diferencias en los lenguajes de programación, como se sabe un computador solo reconoce el código binario y sus instrucciones están en ordenes de 0's y 1's.

En base a esto existen lenguajes de programación de acuerdo a la proximidad a la arquitectura de un hardware, existiendo las siguientes categorías.

- Lenguajes de bajo nivel:

Estos lenguajes son totalmente dependientes de la maquina para aprovechar al máximo las características de esta, dentro de esta categoría se consideran lenguajes como:

- Lenguaje de maquina
- Lenguaje ensamblador

■ Lenguajes de alto nivel:

Estos lenguajes son los mas lejanos a la arquitectura de un computador, de modo que son totalmente independientes de la maquina y muy cercanos al lenguaje humano, dentro de esta categoría se consideran lenguajes como:

- Python
- Fortran
- Etc

■ Lenguajes de medio nivel:

Estos lenguajes se encuentran en un punto medio de entre los dos anteriores, debido a que su sintaxis se asimila al lenguaje humano pero se puede acceder a registros del sistema o direcciones de memoria, estas características son de lenguajes de bajo nivel, dentro de esta categoría se consideran lenguajes como:

- C

Además un lenguaje se puede agrupar en dos categorías de acuerdo a la naturaleza del lenguaje, estas categorías son:

■ Lenguaje interpretado :

Estos lenguajes se caracterizan por que sus instrucciones o su código fuente, escritos mayoritariamente en lenguajes de altos nivel, estos son traducidos por un interprete (o maquina virtual según sea el caso) a un lenguaje que sea comprensible por la maquina, instrucción por instrucción. Este proceso se repite cada vez que se ejecuta un código de este tipo.

La principal ventaja es una gran independencia de la plataforma donde se ejecuta los códigos o instrucciones anteriormente mencionados.

Su principal desventaja es el tiempo que necesitan para ser interpretados, generalmente son más lentos en tiempo de ejecución de un lenguaje compilado.

- Lenguaje compilado:

Estos lenguajes se caracterizan por que su código fuente escrito en alto nivel o medio nivel (C), es traducido por un compilador a un archivo ejecutable para una maquina en específico, en otras palabras cada maquina debe compilar sus códigos fuentes. De manera que se pueden ejecutar las veces que sea necesario sin la necesidad de repetir el proceso de compilación.

Tiempos de ejecución de instrucciones

Según [spe, 2011] realizo la comparación en términos de velocidad de los lenguajes de programación, obteniendo los resultados como se aprecian en la siguiente figura 2.13

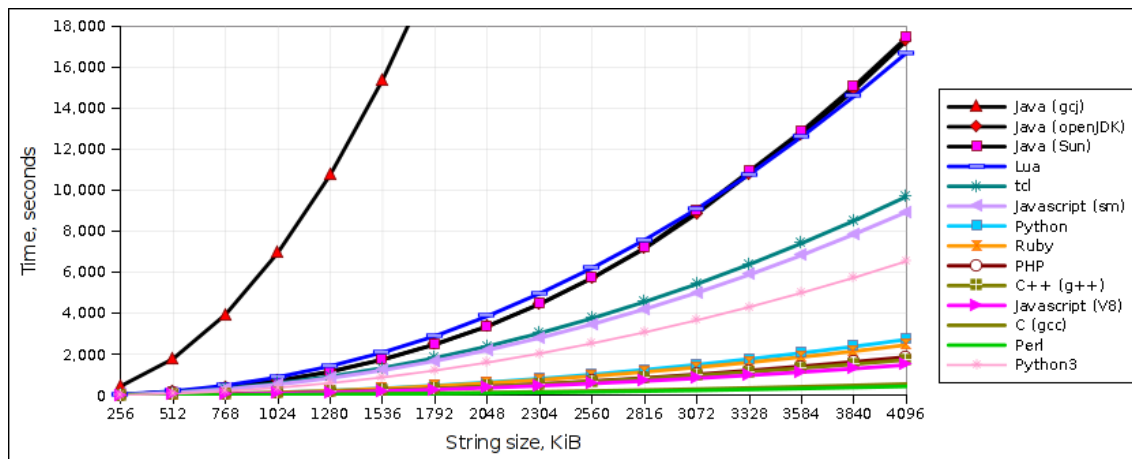


Figura 2.13: Gráfica de tiempo de velocidad

Tiempos de ejecución de estructuras Heap

Una métrica importante para seleccionar el lenguaje más acorde a las necesidades que requiere el algoritmo K-mn, recordando que el tiempo de ejecución de este es importante, según [Bernabeu, 2013] realizo una comparación de tiempos en diversos lenguajes de programación, como C, C++ y Python, los resultados obtenidos se aprecian en la siguiente figura 2.14

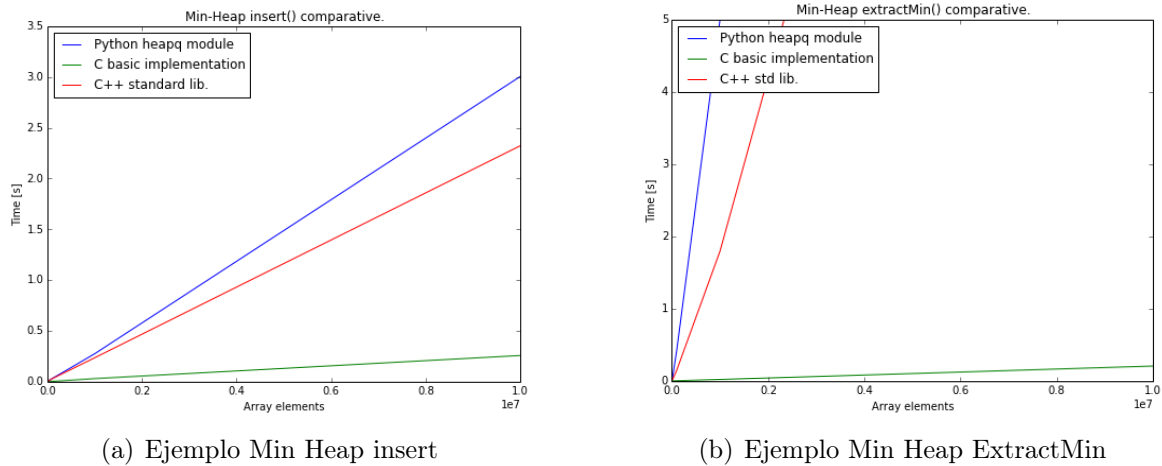


Figura 2.14: Gráfica de tiempo de velocidad

Como se aprecia en la figura 2.14 el tiempo de ejecución de un heap es menor en C vs C++ / Python, de manera que considerando los tiempos de ejecución el lenguaje de programación C es superior en base a esta características.

Posibilidad de utilizar librerías para trabajos en paralelismo

Basado en la guía de programación en paralelo [Acosta et al., 2012], se establecen los conceptos básicos de la programación en paralelo basado tres en diferentes modelos como son:

- **MPM** - *Message Passing Model* mediante la librería **MPI**

Este modelo aprovecha los múltiples *cores* de trabajo que están instalados en los equipos de nueva tecnología. Se asume el *hardware* como reunión de diferentes procesos a través de una red de interconexión donde cada uno tiene acceso directo y exclusivo a la información almacenada en su propia memoria local.

MPM maneja un tipo de acceso a memoria denominado **NUMA** (*Nom-Uniform*

Memory Access). En el que se describe que los procesos que se comunican por la red escalable de interconexión.

El hardware a su vez, cada memoria esta conectada directamente a una CPU, en vez de estar controlado a una memoria (modelo UMA). Las CPUs están conectadas con un I/O hub que permite el ordenamiento de los datos y reduciendo los problemas de tráfico.

Para el uso de un programa en MPM se debe especificar el número de procesos que cuenta la aplicación, cada proceso posee un ID o Rank.

Los mensajes de MPM generalmente contienen:

- La variable en la que reposan los datos que se envían.
- La cantidad de datos que se envían.
- El proceso receptor, el que recibe el mensaje.
- Los datos que se esperan recibir por parte del receptor.
- El tipo de dato que se envía.
- El proceso emisor del mensaje.
- La ubicación o espacio en memoria (puntero en C) donde se almacenarán los datos.

■ Librería MPI

La primera versión de esta librería en 1993 utilizo PVM V3 (*Parallel Virtual machine*), luego en 1994 se ajustaron complementos a la librería de PVM y nace la primera versión de MPI 1.0 (*Message Passing Interface*)

Una característica llamativa de MPI es que permite trabajar con grupos de procesadores definidos según el programador lo disponga mediante objetos virtuales denominados comunicadores, que permiten distribuir los recursos según

la tarea a realizar. Con la librería MPI se debe tener claro que sólo se puede declarar una única vez el ambiente en paralelo (sección comprendida entre las funciones `MPI_Init` y `MPI_Finalize`) y que todo el código que este dentro de la zona se ejecutará en simultáneo por todos los procesos.

Dentro de los grupos de funciones para MPI se destacan algunas como:

- Funciones de control de flujo: permiten crear y establecer parámetros de la sección en paralelo como número de procesos a usar, el comunicador, los ID de los procesos de la aplicación, etc.
- Funciones para el manejo de grupos y comunicadores: facilitan la conformación de los grupos de procesadores.
- Funciones de administración de recurso.
- Funciones de comunicación: permiten la interacción (enviar y recibir información) entre diferentes procesos. Según el número de procesos presentes en la comunicación ésta se clasifica en punto a punto y multipunto.
- Funciones para comunicación punto a punto: implican la interacción de dos procesos exclusivamente (maestro y esclavo), que según el tipo de petición para establecer la conexión se dividen en método bloqueante y no bloqueante.
- Funciones para comunicación multipunto: interactúan con múltiples procesos simultáneamente, el uso de ellas requiere que el desarrollador tenga claro el recurso con el que cuenta.

■ **SMM - *Shared Memory Model* mediante la librería **OpenMP****

El modelo SMM es una abstracción del modelo de multiprocesamiento centralizado. El método de acceso a memoria es llamado UMA (*Uniform Memory Access*), también conocido como SMP (*Symmetric Multi- Processing*).

El hardware en SMM está basado en FSB (front-side bus) que a su vez es el modelo acceso a memoria usado en UMA (Uniform Memory Access). Consta de un controlador de memoria (MCH) al que se conecta a la memoria general, las CPU interactúan con el MCH cada vez que necesitan acceder a memoria. También, cuenta con un controlador de I/O que se conecta al MCH.

SMM está fundamentado en un modelo de ejecución denominado *fork/join* que básicamente describe la posibilidad de pasar de una zona secuencial ejecutada por un único hilo maestro (*master thread*) a una zona paralela ejecutada por varios hilos esclavos (*fork*), posteriormente, cuando finalice la ejecución en paralelo, la información y resultados se agrupan de nuevo mediante un proceso de escritura en memoria en un proceso llamado *join*.

- Librería OpenMP

La gran portabilidad es una característica de MPI debido a que está soportado para C, C++ y Fortran, disponible para sistemas operativos como Solaris, AIX, HP-UX, GNU/Linux, MAC OS, y Windows.

Además, OpenMP soporta la interacción con el modelo de paso por mensajes (MPM) permitiendo la integración de la librería MPI en las aplicaciones, lo que amplía aún más las alternativas de programación.

Una particularidad de OpenMP es que permite administrar el recurso en rutinas que contienen cálculos iterativos, mediante la distribución de iteraciones de los ciclos en los diferentes threads (hilos) que maneja la aplicación mediante funciones especiales denominadas constructores.

- **CUDA** - (*Compute Unified Device Architecture*)

La Gpu (*Graphic Processing Unit*) se caracteriza por tener funciones de punto flotante, en sus inicios nació como alternativas para grandes volúmenes de

información, aplicables en el desarrollo de videojuegos, reproductores de video y simulaciones complejas. Posteriormente se el modelo de programación extendida a C genero lo que hoy se conoce como CUDA.

El hardware de CUDA se compone por la CPU de un equipo principal o host y una GPU dispuesta en un dispositivo externo o device.

Las ventajas de programación en GPU son:

- Código compacto: una instrucción define N operaciones.
- Reduce la frecuencia de los saltos en el código.
- No se requiere de hardware adicional para detectar el paralelismo.
- Permite la ejecución en paralelo asumiendo N flujos de datos paralelos.
- No maneja dependencias.
- Usa patrones de acceso a memoria continua.

Ademas cuenta con varias memorias tales como: *Memoria compartida de lecto/escritura*.- *Memoria de constantes* - *Memoria de texturas*

En base a todos los puntos mencionados y dadas cada una de las características se opto por el *Lenguaje de programación C* debido a ser un lenguaje de nivel medio, el que permite acceder a direcciones de memoria como un lenguaje de bajo nivel y además de su sintaxis estar considerada como alto nivel, esto lo hace un lenguaje más potente en esta linea, también cabe destacar que el lenguaje de programación C es un lenguaje compilado, esto permite que no se necesita utilizar de una maquina virtual o un interprete para procesar sus ejecuciones o el código fuente como lo realizan otros lenguajes, con esto solo se debe compilar una vez y ejecutar el código iterativamente. En términos de tiempos de ejecución de instrucciones como se aprecia en la gráfica de la figura 2.13, se aprecia que el lenguaje de programación C es más veloz que muchos otros lenguajes, debido a las características ya mencionadas.

Ya acotando el rango de lenguajes entre C, C++, Python como se aprecia en la gráfica

2.14, el lenguaje de programación C es considerablemente mas veloz para procesar la estructura Heap y a esto se le añade la característica de ser un lenguaje que soporta paralelismo en todas las plataformas y cuenta con diversas librerías como se menciono anteriormente, siendo utilizadas en esta tesis OpenMP y CUDA en la cual ambas son soportadas en el lenguaje de programación C.

Capítulo 3

Estado del Arte

La regla del vecino más cercano fue originalmente propuesta por Cover y Hart [Hart, 1966], [Cover and Hart, 1967a] y a lo largo de las investigaciones está siendo utilizada por varios investigadores. Una razón para el uso de esta regla es su simplicidad conceptual, que conduce a la programación directa, si no necesariamente la más eficiente. Este algoritmo utiliza la idea central de la distancia euclidiana entre dos puntos en el espacio, en base a este cálculo se analiza la similitud entre todos los puntos existentes. En un artículo posterior, Hart [HILBORN, 1968] sugirió un medio de disminución de la memoria y los requisitos de computación. Este artículo introduce una técnica, la regla del vecino más cercano reducido, que puede conducir a más ahorros. Los resultados de esta regla se demuestran aplicándola a los datos de "*Iris*" [Freeman, 1969].

El algoritmo k-NN [Aha et al., 1991] [Cover and Hart, 1967b] tiene grandes requisitos de almacenamiento, pero pueden reducir significativamente el trabajo de aprendizaje de otro tipo de técnicas y aumentar la precisión en la clasificación. Uno de los problemas más grandes que presenta este algoritmo es el alto costo de procesamiento y memoria para resolver las consultas.

Se aplica ampliamente en el reconocimiento de patrones y en clasificación para minería de datos, debido a su simplicidad y baja tasa de error. Antes

de la aparición masiva de plataformas paralelas, la realización por fuerza bruta (exhaustivamente) no se consideraba como una opción válida, especialmente para grandes bases de datos de entrenamiento y espacios de alta densidad. Para reducir el espacio de búsqueda y evitar tantos cálculos de distancia como sea posible, se han propuesto muchos enfoques de indexación. La mayoría de los métodos de recuperación se basan en kd-trees [Bentley and Friedman, 1979]. Hay una gran cantidad de trabajo sobre las adaptaciones de la estructura básica kd-tree para el problema k-NN [Brisaboa et al., 2008] [Paredes and Navarro, 2009]. También se han propuesto estructuras no-tree que dividen eficientemente el espacio de búsqueda [Chávez and Navarro, 2000] [Brisaboa et al., 2006]. Las implementaciones basadas en MPI que utilizan estas estructuras también aparecen con mucha frecuencia en la literatura [Plaku and Kavraki, 2007].

La principal desventaja de estos métodos es que necesitan construir y mantener estructuras de datos complejas para el conjunto de datos de referencia. De manera que, k-NN se implementa típicamente utilizando métodos de fuerza bruta.

Por lo tanto, las soluciones actuales de GPU han sido de alguna manera más simples y menos eficientes desde el punto de vista del algoritmo. Por ejemplo, Benjamin Bustos et al. [Bustos et al., 2006] propuso una implementación no-CUDA explotar memorias de textura GPU. Su implementación sólo busca el elemento mínimo, lo que simplifica significativamente el problema.

El cálculo de la distancia exhaustiva en conjunción con la clasificación en paralelo se propuso en [Garcia et al., 2008], [Kuang and Zhao, 2009]. Vecente Garcia et al. [Garcia et al., 2008] propuso un orden de inserción paralelo modificado con el fin de obtener sólo los k elementos más cercanos, mientras que Kuang et al. [Kuang and Zhao, 2009] propuso un mejor Radix-sort para realizar la clasificación final.

Además existe una versión de GPU de Quicksort que fue propuesta por Daniel Cederman y Philippas Tsigas [Cederman and Tsigas, 2009], que mostró ser más rápido que los algoritmos de clasificación anteriores. Este trabajo se basa en la propuesta que también calcula exhaustivamente todas las distancias, pero utiliza una metodología

específica basada en heap para encontrar los k elementos más cercanos, propuesta por Ricardo Barrientos y José Gómez [Barrientos et al., 2010]

Capítulo 4

Desarrollo

4.1. Introducción

En este capítulo se describirá el proceso de creación del Software mencionado ya anteriormente en capítulos previos, en los cuales se indica que el software contiene una recopilación de diversos algoritmos K-nn en diferentes plataformas paralelas y además del K-nn secuencial para equipos que no posea las características necesarias para utilizar este software, de este modo se presentara desde la metodología utilizada hasta su desarrollo final con el software en su primera versión terminada. Para ello sera necesario describir procesos y etapas que se dividirían en este capítulo.

4.2. Desarrollo interfaz

En esta sección se aborda como se desarrolló la interfaz gráfica, en un principio se estableció *JAVA* como el lenguaje de programación y se utilizo el IDE *netbeans* [and/or its affiliates, 2017], a continuación se explican cada uno de los pasos tomados en consideración al realizar la interfaz gráfica.

En un principio se estableció *JAVA* debido a su capacidad de ser soportado por diversos sistemas operativos, como son Windows, OSX, Linux. Dado que *JAVA* es un lenguaje interpretado y no compilado, el interpretador (Maquina virtual) de Java

es soportado por todos los sistemas antes mencionados, además el uso de *netbeans* permite la fácil creación de una interfaz gráfica, esto debido a poseer *drag and drop* para sus componentes gráficos, en otras palabras solo se necesita arrastrar una componente y situarla en la posición necesaria, esta ventaja dada por *netbeans* es útil, debido al menor tiempo que se ejecuta para crear la interfaz y poder otorgar el tiempo a las funciones correspondientes.

Otra característica importante de *JAVA* se debe a que se puede realizar la ejecución de los ejecutables creados luego de la compilación de los algoritmos K-nn y retornar los valores del ejecutable a la interfaz, de esto se logra poder potencial de mejor manera el software. De modo que la interfaz gráfica solo realiza funciones de selección de bases de datos y campos de formularios, en otras palabras la interfaz gráfica solo es una fachada con la finalidad de no realizar la ejecución de los algoritmos desde la terminal.

La figura 4.1 muestra la interfaz de usuario cuando el software inicia, la cual se compone de la barra superior la cual posee dos menús "File" y "Menus", una sección de *input* que corresponde a los datos de entrada para el algoritmo K-nn, una sección de *Result* donde en el primer recuadro de la izquierda se muestra la vista previa de las primeras 1000 líneas del archivo de base de datos, en el recuadro central se muestra la vista previa de las primeras 1000 líneas del archivo de consultas, en el cuadro de la derecha muestra los resultados del proceso, muestra si la carga de los datos fue exitosa, el tiempo de ejecución o algún error que pudiese haber ocurrido, como el fallo en la carga de los datos. En la parte inferior del software indica la cantidad de núcleos que posee la maquina donde el software esta instalado.

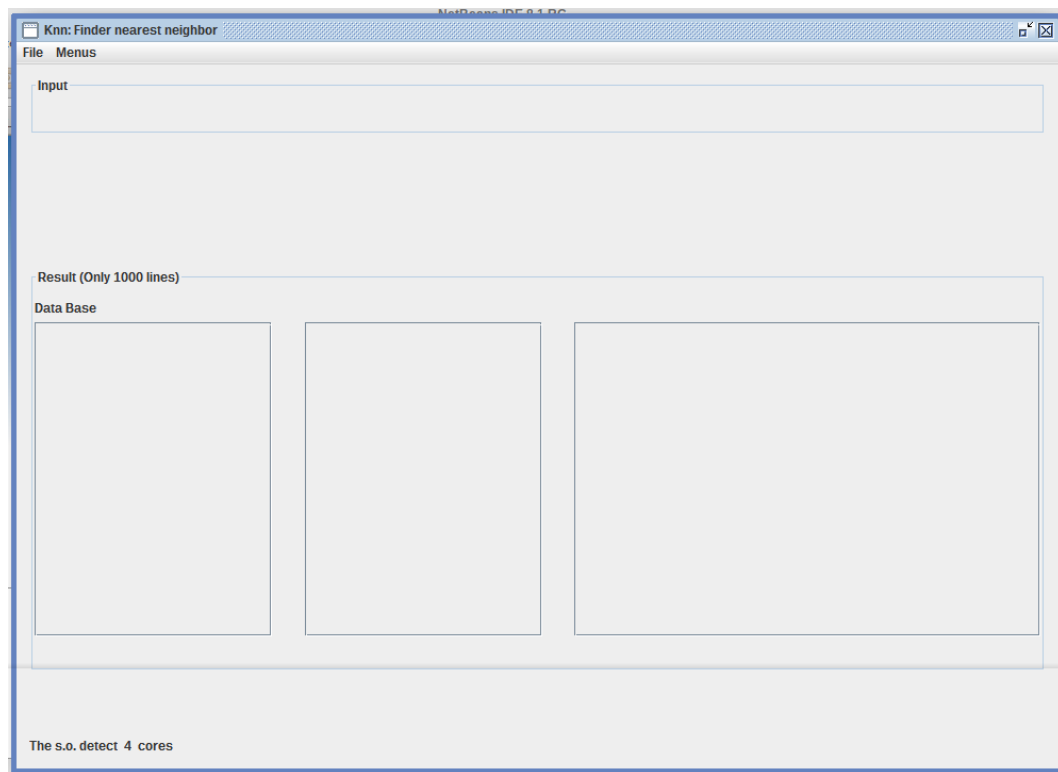


Figura 4.1: Interfaz gráfica inicial del software

Menús del software

El software posee dos menús como se menciono anteriormente, estos corresponden a "File" y "Menus", en el primer menú *File* posee dos sub-menús *New* y *Exit*, El primero posee sub-menús que corresponde a los algoritmos que posee el software, estos son:

- *Secuencial*
- *Multihilos*
- *Xeon Phi*
- *GPU*

Cada uno de estos sub-menús, posee al menos un item que corresponde al algoritmo K-nn programado de acuerdo a la lógica que su nombre indica, esto quiere decir que el sub-menú Multihilos cuenta con a lo menos un item que permite ejecutar el algoritmo K-nn Multihilos, donde este utiliza *OpenMP* para utilizar de mejor manera los recursos de la CPU.

Al seleccionar un item del sub-menú *Multihilos* la interfaz gráfica tiene las siguientes características como se aprecia en la figura 4.2, en la sección *input* se visualizan los campos de entrada que posee el software, en primer caso es la base de datos, esta se carga mediante el botón examinar, para cargar los datos de consulta se utiliza el botón examinar, *sizeobj* es un valor numérico que corresponde a la magnitud del vector ingresado en la base de datos y base de consultas, *K* es la cantidad de vecinos más cercanos que se desea obtener, *Threads* por defecto se utiliza la cantidad cantidad de núcleos que posee la maquina donde esta instalado el software, este valor puede ser modificado por la cantidad de hilos que desea el usuario.

Además se cuenta con el botón *Start* para comenzar a ejecutar el las consultas K-nn, se añadió bajo el botón *Start* un *check box* que permite ejecutar un *Profiler* que

utiliza la librería PapiC [IT, 2017] que indica estadísticas sobre el uso de los hilos. En la sección *Result* no varía en relación al ítem seleccionado.

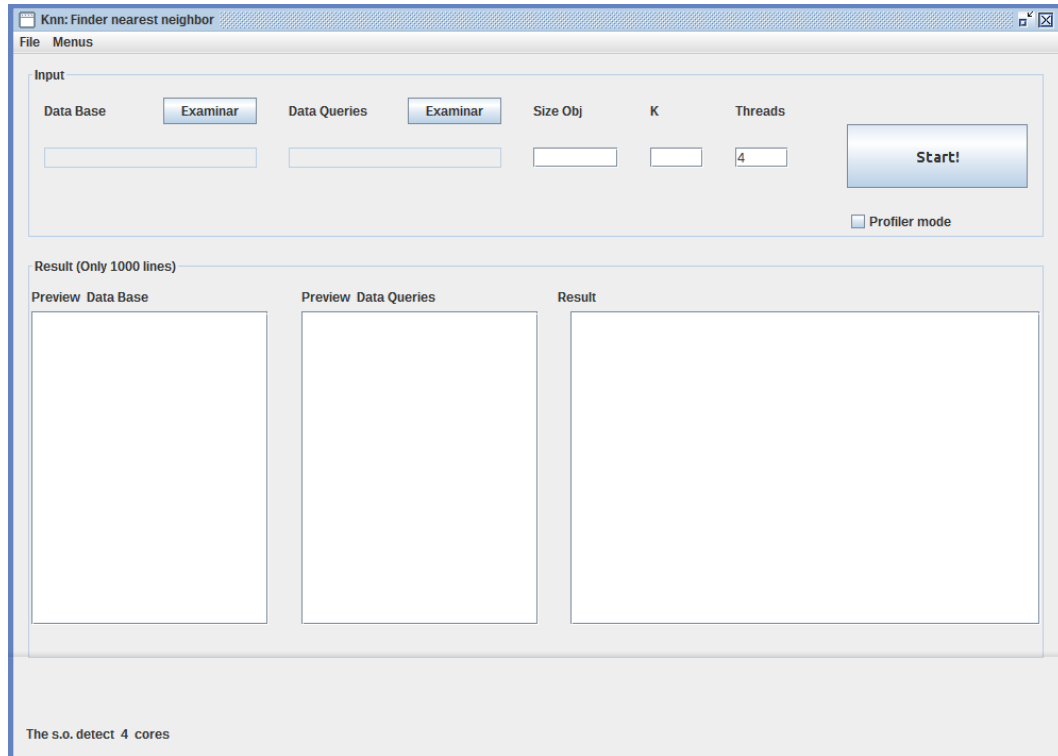


Figura 4.2: Interfaz gráfica ítem multihilos del software

Al seleccionar un ítem del sub-menú *Secuencial* la interfaz gráfica tiene las siguientes características como se aprecia en la figura 4.3, en la sección *input* se visualizan los campos de entrada que posee el software, en primer caso es la base de datos, esta se carga mediante el botón *examinar*, para cargar los datos de consulta se utiliza el botón *examinar*, *sizeobj* es un valor numérico que corresponde a la magnitud del vector ingresado en la base de datos y base de consultas, *K* es la cantidad de vecinos más cercanos que se desea obtener.

La selección de los otros ítem en los sub-menús son similares a los antes mencionados, de modo que solo varía la cantidad de datos de entrada que se ingresan dado las condiciones de programación de cada algoritmo.

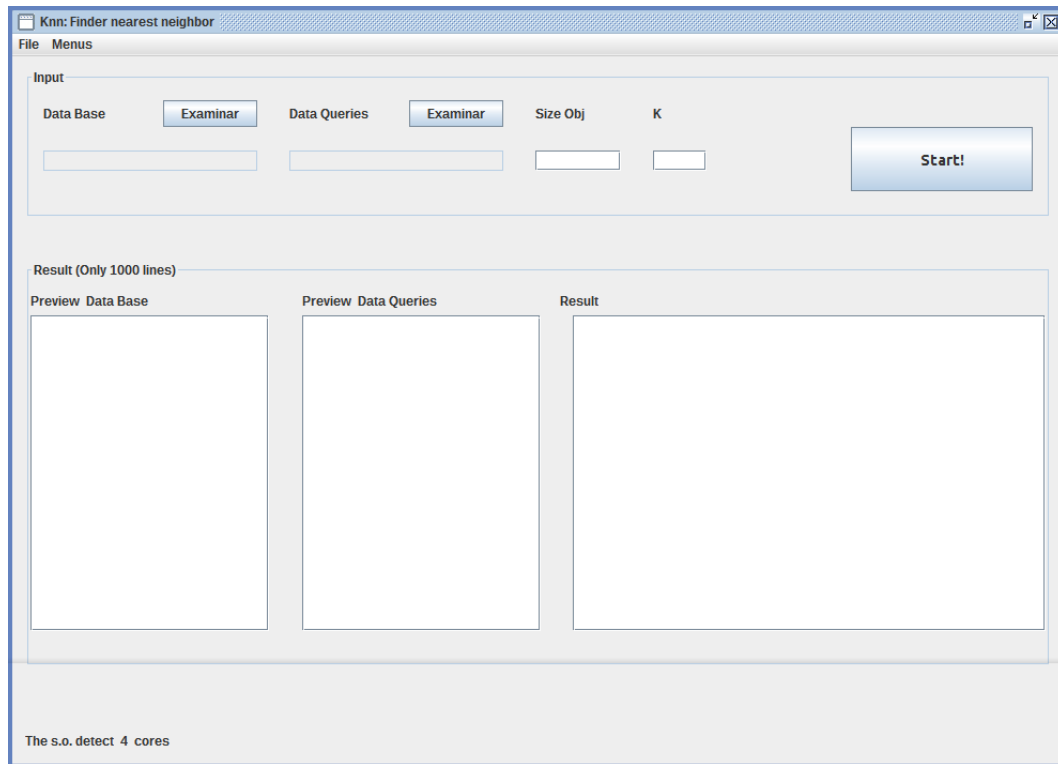


Figura 4.3: Interfaz gráfica item secuencial del software

Además el software cuenta con mensajes dirigidos al usuario, estos son divididos en mensajes de alerta y mensajes de éxito. El primero se aprecia en la figura 4.4(a), este mensaje aparece una vez que se presiona el botón *Start* y si algún campo está vacío, indica que los campos deben completarse para poder ejecutar correctamente el proceso. Si no presenta errores el proceso parte correctamente, si los archivos de bases de datos y consultas son archivos correspondientes, el proceso muestra un mensaje de éxito, como se aprecia en la figura 4.4(b), al presionar el botón aceptar se muestra la ventana de los resultados de la consulta K-nn, este paso corresponde a la sección 4.7 *Métodos de exportación de resultados*.

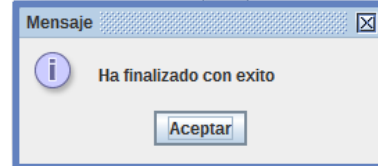
4.3. Integración de algoritmo K-nn Secuencial

Si bien este algoritmo, no estaba considerado en un comienzo este se añadió para la posibilidad de poder ejecutar el software en equipos que no cuenten con procesadores

Figura 4.4: Mensajes al usuario



(a) Mensaje 1 - Error



(b) Mensaje 2 - Éxito

Algoritmo 2 Estructura para realizar los *Heap*.

```

1  struct _Elem {
2      double dist;
3      int ind;
4  };
5  typedef struct _Elem Elem;

```

con multi-núcleos y necesiten realizar consultas K-nn. Este algoritmo se realizó con la estructura heap 2.5.

Para implementar el algoritmo K-nn secuencial, se definió la estructura heap, la cual se aprecia en el segmento de código en el algoritmo 2, este almacena en *double dist*, la distancia del elemento consultado esta es de tipo *double* debido a que las distancias son reales y puede ser el caso que se necesite almacenar un valor muy grande, a su vez *intind* indica la posición del vector dentro de la base de consultas. Cabe recalcar que solo se almacenan K resultados en esta estructura, además de solo almacenar los valores cuya distancia *dist* sean las k menores.

Con el lenguaje de programación C, es posible utilizar la cantidad de memoria justa gracias a la instrucción *malloc* que permite asignar la cantidad de memoria que corresponde al tamaño de la base de datos y la base de consulta, tal como se aprecia en el algoritmo 3. Donde *consultas* se crea una matriz de $N_queries \times DIM$ donde $N_queries$ es el número tuplas de vectores contenidas en la base de consulta y DIM es el tamaño del vector. De manera análoga se establece la matriz *DB* de $N_DB \times DIM$

Algoritmo 3 Utilización de *malloc*.

```

1   Consultas = (double **) malloc(sizeof (double *)*N_QUERIES);
2   for (i = 0; i < N_QUERIES; i++)
3       Consultas[i] = (double *) malloc(sizeof (double)*DIM);
4
5   DB = (double **) malloc(sizeof (double *)*N_DB);
6   for (i = 0; i < N_DB; i++)
7       DB[i] = (double *) malloc(sizeof (double)*DIM);
8
9   answer = (Elem *) malloc(sizeof (Elem)*N_QUERIES*TOPK);
10
11  heap = (Elem *) malloc(sizeof (Elem) * TOPK);

```

donde N_DB es el número tuplas de vectores contenidas en la base de datos y DIM es el tamaño del vector. A su vez *answer* corresponde a la asignación de memoria para las $N_queries$ con K vecinos más cercanos donde se almacenan todas las respuestas a las consultas para ser finalmente mostrados. *Heap* es la estructura a utilizar con cada consulta, esta corresponde a un heap del tipo heap max, donde el nodo padre es el mayor valor de la estructura.

El desarrollo central del algoritmo para obtener los k vecinos más cercanos esta dado por el algoritmo 4, este algoritmo indica las siguientes acciones, se establece un contador $n_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como comparador debido a que se deben ingresar k elementos al heap, mientras n_elem sea menor que el número k se ingresaran los datos al heap en caso contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor se extrae la raíz y se añade el elemento actual y se reordena el heap. Este proceso itera exhaustivamente hasta que no queden elementos en la base de datos, cuando el heap esta finalizado se almacenan sus resultados en la matriz *answer* y se debe continuar con la siguiente consulta hasta que se realicen todas las consultas del base de consultas.

4.3.1. Funciones claves del algoritmo secuencial

Distancia : Algoritmo 5 es la función de cálculo de la distancia entre vectores, tanto del vector de la base de datos y el vector de la base de consultas, este cálculo se realiza

Algoritmo 4 Proceso iterativo de una consulta K-nn.

```

1  for (i = 0; i < N_QUERIES; i++) {
2      n_elem = 0;
3      for (j = 0; j < N_DB; j++) {
4          d = distancia(Consultas[i], DB[j]);
5          if (n_elem < TOPK) {
6              e_temp.dist = d;
7              e_temp.ind = j;
8              inserta2(heap, &e_temp, &n_elem);
9          }
10         else {
11             if (d < topH(heap, &n_elem)) {
12                 e_temp.dist = d;
13                 e_temp.ind = j;
14                 popush2(heap, &n_elem, &e_temp);
15             }
16         }
17         for (int k = 0; k < TOPK; ++k) {
18             extrae2(heap, &n_elem, &e_temp);
19             answer[i*TOPK+k].ind = e_temp.ind;
20             answer[i*TOPK+k].dist = e_temp.dist;
21         }
22     }

```

Algoritmo 5 Cálculo de distancias entre vectores.

```

1  double distancia(double *p1, double *p2) {
2      int i = 0;
3      double suma = 0;
4
5      for (i = 0; i < DIM; i++)
6          suma += ((p1[i] - p2[i]) * (p1[i] - p2[i]));
7      return sqrt(suma);
8  }

```

con la formula (4.1).

Dados $A = (X_1, Y_1)$ y $B = (X_2, Y_2)$

$$distancia = \sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2} \quad (4.1)$$

Esta ecuación puede ser llevada a vectores de tamaño n .

Sumado a esta función existen las funciones para manipular un heap, estas corresponden a obtener el valor nodo padre (Algoritmo 6), realizar una inserción de un elemento, extraer un elemento y una añadida para facilitar el procedimiento del algoritmo como es el caso de una extracción e inserción que se utiliza cuando se debe cambiar un nodo dentro del heap y realizar el reordenamiento del heap.

Algoritmo 6 Valor de la raíz del heap.

```

1 | double topH(Elem *heap, int *n_elem) {
2 |     if (*n_elem == 0)
3 |         return DBL_MAX;
4 |     return heap[0].dist;
5 | }

```

Algoritmo 7 Realiza la inserción de un elemento al heap.

```

1 | void inserta2(Elem *heap, Elem *elem, int *n_elem) {
2 |     int i;
3 |     Elem temp;
4 |
5 |     heap[*n_elem].dist = elem->dist;
6 |     heap[*n_elem].ind = elem->ind;
7 |     (*n_elem)++;
8 |     for (i = *n_elem; i > 1 && heap[i - 1].dist > heap[(i / 2) - 1].dist; i = i / 2) {
9 |         temp = heap[i - 1];
10 |        heap[i - 1] = heap[(i / 2) - 1];
11 |        heap[(i / 2) - 1] = temp;
12 |     }
13 | }

```

El algoritmo 6 retorna el valor de la raíz (nodo padre) del heap si no esta vacío, en caso contrario retorna el valor maximo de un *double*.

El algoritmo 7 realiza la inserción de un elemento, guardando la distancia y la posición del elemento, el contador *n_elem* incrementa en 1 luego de la inserción, y se realiza el ordenamiento del heap. Esta función es la que se utiliza en el algoritmo 2 hasta llenar el heap.

El algoritmo 8 realiza la extracción de un elemento, guardando la distancia y la posición del elemento, el contador *n_elem* decrementando en 1 luego de la extracción, y se realiza el ordenamiento del heap. Esta función es la que se utiliza para completar la matriz *answer*.

El algoritmo 9 realiza la extracción del elemento raíz y la posterior inserción del elemento cuya distancia era menor que la distancia del elemento raíz, esta función guarda la distancia y la posición del elemento, con esto luego se realiza el ordenamiento del heap. Esta función es la que se utiliza en el algoritmo 2 cuando el heap esta lleno y se obtiene un elemento cuya distancia es menor a la distancia de la raíz.

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó

Algoritmo 8 Realiza la extracción de un elemento al heap.

```

1 void extrae2(Elem *heap, int *n_elem, Elem *elem_extraido) {
2     int i, k;
3     Elem temp;
4
5     (*elem_extraido).dist = heap[0].dist;
6     (*elem_extraido).ind = heap[0].ind;
7
8     heap[0] = heap[( *n_elem) - 1]; // Movemos el ultimo a la raiz y achicamos el heap
9     (*n_elem)--;
10    i = 1;
11    while (2 * i <= *n_elem) // mientras tenga algun hijo
12    {
13        k = 2 * i; //el hijo izquierdo
14        if (k + 1 <= *n_elem && heap[(k + 1) - 1].dist > heap[k - 1].dist)
15            k = k + 1; //el hijo derecho es el mayor
16        if (heap[i - 1].dist > heap[k - 1].dist)
17            break; //es mayor que ambos hijos
18
19        temp = heap[i - 1];
20        heap[i - 1] = heap[k - 1];
21        heap[k - 1] = temp;
22        i = k; //lo intercambiamos con el mayor hijo
23    }
24    return;
25 }
```

Algoritmo 9 Realiza la extracción y la inserción de un elemento al heap.

```

1 void popush2(Elem *heap, int *n_elem, Elem *elem) {
2     int i, k;
3     Elem temp;
4
5     heap[0].dist = elem->dist;
6     heap[0].ind = elem->ind;
7
8
9     i = 1;
10    while (2 * i <= *n_elem) // mientras tenga algun hijo
11    {
12        k = 2 * i; //el hijo izquierdo
13        if (k + 1 <= *n_elem && heap[(k + 1) - 1].dist > heap[k - 1].dist)
14            k = k + 1; //el hijo derecho es el mayor
15        if (heap[i - 1].dist > heap[k - 1].dist)
16            break; //es mayor que ambos hijos
17
18        temp = heap[i - 1];
19        heap[i - 1] = heap[k - 1];
20        heap[k - 1] = temp;
21        i = k; //lo intercambiamos con el mayor hijo
22    }
23    return;
24 }
```

Algoritmo 10 Ejecutar un programa o ejecutable desde *Java*.

```
1
2 import java.io.*;
3
4 public class llamarruntime
5 {
6     public static void main(String[] args)
7     {
8         try{
9             Process theProcess =
10                 Runtime.getRuntime().exec(Ejecutable);
11         }
12         catch(IOException e){
13             System.err.println("Error en el metodo exec()");
14             e.printStackTrace();
15         }
16     }
17 }
```

mediante una rutina de *Java* con *java.lang.Runtime.exec()*. La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 10 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.4. Integración de algoritmo K-nn Paralelo Multi-hilos

Este algoritmo a posee ciertas semejanzas y diferencias con respecto a la versión secuencial, este utiliza *malloc* como se indica en el algoritmo 3, estableciendo los mismos atributos y designando la memoria de la misma manera que la versión secuencial, léase 4.3.

A través de la librería *OpenMP* establece variables de tipo compartida *Shared* y variables del tipo privadas *Private*, donde con una variable del tipo compartida se puede obtener su valor o realizar alguna modificación sin restricciones desde cualquier hilo, a su vez una variable del tipo privada solo puede ser accesible por un hilo y se crea una copia de la variable para tantos hilos creados existan. De esta manera se han

Algoritmo 11 Proceso iterativo de una consulta K-nn multi-core.

```

1  #pragma omp master
2  for (i = tid; i < N_QUERIES; i += procs) {
3      n_elem = 0;
4      for (j = 0; j < N_DB; j++) {
5          d = distancia(Consultas[i], DB[j]);
6          if (n_elem < TOPK) {
7              e_temp.dist = d;
8              e_temp.ind = j;
9              inserta2(heap, &e_temp, &n_elem);
10         }
11         else {
12             if (d < topH(heap, &n_elem)) {
13                 e_temp.dist = d;
14                 e_temp.ind = j;
15             }
16         }
17         for (j = 0; j < TOPK; j++) {
18             extrae2(heap, &n_elem, &e_temp);
19             answer[i*TOPK+j].ind = e_temp.ind;
20             answer[i*TOPK+j].dist = e_temp.dist;
21         }
22     }
23 }
24 #pragma omp barrier

```

establecido variables solo variables del tipo *Shared* para este algoritmo.

El desarrollo central del algoritmo para obtener los k vecinos más cercanos esta dado por el algoritmo 11, este algoritmo indica las siguientes acciones, se establece un contador $n_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como comparador debido a que se deben ingresar k elementos al heap, mientras n_elem sea menor que el número k se ingresaran los datos al heap en caso contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor se extrae la raíz y se añade el elemento actual y se reordena el heap. Este proceso itera paralelamente hasta que no queden elementos en la base de datos, cuando el heap esta finalizado se almacenan sus resultados en la matriz *answer* y se debe continuar con la siguiente consulta hasta que se realicen todas las consultas del base de consultas.

A diferencia del método secuencial, el método multi-paralelo utiliza la librería OpenMP, por tanto como se muestra en el algoritmo 11 `#pragma omp master` indica el trabajo en la zona paralela, de manera que el ciclo *for* realiza un paso cíclico con una cantidad determinada de hilos dada por *procs*, donde *tid* es el identificador de

cada hilo, esto quiere que decir que si utilizo 2 hilos el hilo 0 con $tid = 0$ realiza la consulta 0 y el hilo 1 con $tid = 1$ realiza la consulta 1 simultáneamente finalizado el trabajo de al menos uno de estos, el hilo con su actividad finalizada toma el valor correspondiente, en caso del hilo 0 con $tid = 0$ le corresponde tomar la consulta 2 y al hilo 1 con $tid = 1$ le corresponde el hilo 3. Este proceso se realiza hasta llegar al final de la base de consultas $N_queries$.

4.4.1. Funciones claves del algoritmo multi-hilos

Estas funciones son las mismas que el método secuencial, donde se consideran *topH* para saber el valor del del nodo raíz, *inserta2* para insertar un nodo en un heap con el número de elementos menor al de k elementos, *extrae2* extrae el nodo raíz del heap, *popush2* realiza la extracción del nodo raíz e inserción de un nuevo nodo de menor valor que la raíz extraída, léase 4.3.1

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec()*. La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 10 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.5. Integración de algoritmo K-nn Paralelo Xeon Phi

A diferencia de los métodos anteriores acá se utiliza un coprocesador, de modo que se debe realizar la comunicación entre la CPU y el Coprocesador (*Intel Xeon Phi*). A continuación se describe en detalle las funciones de este método.

Con el lenguaje de programación C, es posible utilizar la cantidad de memoria justa gracias a la instrucción *malloc* que permite asignar la cantidad de memoria que corresponde al tamaño de la base de datos y la base de consulta, tal como se aprecia

en el algoritmo 12. Donde *queries* se crea una matriz de $num_queries \times dimaux$ donde $num_queries$ es el número tuplas de vectores contenidas en la base de consulta y $dimaux$ es el tamaño del vector. De manera análoga se establece la matriz *db* de $num_db \times dimaux$ donde num_db es el número tuplas de vectores contenidas en la base de datos y $dimaux$ es el tamaño del vector. A su vez *answer* corresponde a la asignación de memoria para las $num_queries$ con K vecinos más cercanos donde se almacenan todas las respuestas a las consultas para ser finalmente mostrados. *Heap* es la estructura a utilizar con cada consulta, esta corresponde a un heap del tipo heap max, donde el nodo padre es el mayor valor de la estructura. La Xeon phi permite solo permite realizar el trabajo con vectores de modo que las matrices deben ser trabajadas como vectores y además los vectores deben ser múltiplos de 16, de modo que para solucionar este problema y poder trabajar con la medida de cualquier vector de entrada se parametriza a su vector múltiplo de 16 superior, en otras palabras si ingresa un vector de tamaño 8 se parametriza a 16, si ingresa un vector 17 se parametriza a 32 y los valores restantes son establecidos como 0, para no interferir en el cálculo de la distancia.

El desarrollo central del algoritmo para obtener los k vecinos más cercanos esta dado por el algoritmo 13, este algoritmo indica las siguientes acciones, las operaciones de este algoritmo están dentro de la región paralela de la Xeon phi (*#pragma*) de manera que se deben enviar desde la CPU a la Xeon phi todas las variables o constantes a utilizar, donde se deben cumplir ciertas restricciones como vectorizar la matriz (explicado anteriormente), además se especifican claramente cuales son datos exclusivamente de entrada *In* y exclusivamente datos de salida *Out* o datos que pueden ser de entrada y salida (*Inout*). La Xeon phi a través de la librería *OpenMP* utiliza además variables de tipo compartida *Shared* y variables del tipo privadas *Private*, donde una variable del tipo compartida puede obtener su valor o realizar alguna modificación sin restricciones, a su vez una variable del tipo privada solo puede ser accesible por un hilo y se crea una copia de la variable para tantos hilos creados existan. A diferencia del método anterior que solo utilizaba variables tipo *Shared*, en este método las variables del tipo *Private*

Algoritmo 12 Utilización de *malloc*.

```

1  db= (double **)malloc(sizeof(double *)*num_db);
2  for (i=0; i<num_db; i++)
3      db[i] = (double *)malloc(sizeof(double)*dimaux);
4  queries = (double **)malloc(sizeof(double *)*num_queries);
5  for (i=0; i<num_queries; i++)
6      queries[i] = (double *)malloc(sizeof(double)*dimaux);
7  answer = (Elem *)malloc(sizeof(Elem)*num_queries*k);
8
9  //Se transfieren datos de una matriz a un vector
10 db_vector = (double *)_mm_malloc(sizeof(double)*dimaux*num_db, 64);
11 for (i=0; i < dimaux*num_db; i++)
12     db_vector[i] = 0.0;
13 if (sizeof(double)*dimaux*num_queries < 64)
14 {
15     queries_vector = (double *)_mm_malloc(sizeof(double)*16, 64);
16     for (i=0; i < 16; i++)
17         queries_vector[i] = 0.0;
18 }
19 else
20 {
21     queries_vector = (double *)_mm_malloc(sizeof(double)*dimaux*num_queries, 64);
22     for (i=0; i < dimaux*num_queries; i++)
23         queries_vector[i] = 0.0;
24 }

```

se han establecido las siguientes variables i , j , $thread_num$ de modo que cada hilo puede realizar independientemente sus iteraciones en los ciclos sin ser interrumpido por otro, esto debido a que se utilizan vectores y se debe emular la lectura de la matriz a través de los vectores. Dentro de la región paralela `#pragma omp parallel` se crea un heap de tamaño K . luego se establece un contador $n_elem = 0$ que obtiene el número de elementos que han ingresado al heap, este se utiliza como comparador debido a que se deben ingresar k elementos al heap, mientras n_elem sea menor que el número k se ingresaran los datos al heap en caso contrario se compara si la distancia del elemento actual es menor a la raíz del heap de manera que si el elemento es menor se extrae la raíz y se añade el elemento actual y se reordena el heap. Este proceso itera paralelamente hasta que no queden elementos en la base de datos, cuando el heap esta finalizado se almacenan sus resultados en la matriz *answer*. Este proceso dentro de la región paralela `#pragma omp parallel` realiza cada uno de los pasos mencionados por cada hilo utilizado en el proceso

Algoritmo 13 Proceso iterativo de una consulta K-nn Xeon phi.

```

1  #pragma offload target(mic:0) in(dim) in(db_vector:length(num_db*dimaux))
2    in(queries_vector:length(num_queries*dimaux)) out(answer:length(k*num_queries))
3  {
4      #pragma omp parallel private(i, j, thread_num) shared(db_vector, num_db,
5          queries_vector, num_queries, dimaux, k, answer, num_threads)
6      {
7          Elem *heap;
8          heap = (Elem *) malloc(sizeof(Elem)*k);
9          #pragma omp master
10         {
11             num_threads = omp_get_num_threads();
12             printf("run with %d threads\n", num_threads);
13         }
14         #pragma omp barrier
15         thread_num = omp_get_thread_num();
16         int n_elem;
17         Elem e_temp;
18         double d;
19
20         for(i=thread_num*dimaux; i<num_queries*dimaux; i+=num_threads*dimaux){
21             n_elem = 0;
22             for(j=0; j<k; j++){
23                 e_temp.dist = distancia(&(queries_vector[i]), &(db_vector[j*dimaux]),
24                     dimaux);
25                 e_temp.ind = j;
26                 inserta2(heap, &e_temp, &n_elem);
27             }
28             for(j=k; j<num_db; j++){
29                 d = distancia(&(queries_vector[i]), &(db_vector[j*dimaux]), dimaux);
30                 if(d < topH(heap, &n_elem))
31                 {
32                     e_temp.dist = d;
33                     e_temp.ind = j;
34                     popush2(heap, &n_elem, &e_temp);
35                 }
36             }
37
38             for(j=0; j<k; j++){
39                 extrae2(heap, &n_elem, &e_temp);
40                 printf("%d ind = %d :: dist = %f posicion:: %d \n", j, e_temp.ind,
41                     e_temp.dist, (i/dimaux)*k+j);
42                 answer[(i/dimaux)*k+j].ind = e_temp.ind;
43                 answer[(i/dimaux)*k+j].dist = e_temp.dist;
44             }
45
46         }
47         free(heap);
48     }
49 }

```


Algoritmo 14 Función distancia *Xeon phi*.

```

1  __attribute__((target(mic))) double distancia(double *p1, double *p2, int DIM){
2      int i=0;
3      double suma=0.0;
4      __assume_aligned(p1, 64);
5      __assume_aligned(p2, 64);
6      #pragma vector aligned
7      #pragma ivdep
8      #pragma simd
9      for (i=0; i < DIM; i++){
10         suma += (p1[i]-p2[i])*(p1[i]-p2[i]);
11     }
12     return sqrt(suma);
13 }
```

Algoritmo 15 Pasa una matriz a vector.

```

1  void matrixToVector(double **matrix, int num_cols, int num_rows, double *vector){
2      int i,j;
3      for(i=0; i<num_rows; i++)
4          for(j=0; j<num_cols; j++)
5              vector[(i*num_cols)+j] = matrix[i][j];
6  }
```

4.5.1. Funciones claves del algoritmo Xeon phi

A diferencia de los métodos anteriores donde solo se utiliza una *CPU*, acá se emplea tanto la *CPU* como un *Coprocesador* el cual solo realiza tareas específicas (léase 2.2). A continuación se detallan las principales funciones que se implementaron en *Xeon phi*.

Distancia : Algoritmo 14 es la función de cálculo de la distancia entre vectores, tanto del vector de la base de datos y el vector de la base de consultas, este cálculo se realiza con la formula (4.1). Esta función se realiza en la *Xeon phi* para indicar que se realizara en el coprocesador se utiliza la sintaxis `__attribute__((target(mic)))`. Las sintaxis `__assume_aligned()`, `#pragma vector aligned`, `#pragma ivdep`, `#pragma simd` alinea los vectores y permite que el cálculo de la distancia se realice correctamente.

Debido a que la *Xeon phi* no permite la utilización de matrices se implementa la función *matrixToVector* (Algoritmo 15) para traspasar la matriz a vector, esta función corresponde a la CPU no a la *Xeon phi* de manera que no utiliza sintaxis especiales.

Algoritmo 16 Valor de la raíz del heap.

```

1 | __attribute__((target(mic))) double topH(Elem *heap, int *n_elem)
2 | {
3 |     if ((*n_elem) == 0)
4 |         return MAXDOUBLE;
5 |     return heap[0].dist;
6 | }

```

De manera similar a los métodos anteriores (Secuencial y Multi-núcleo) se emplean las mismas funciones, pero estas no se ejecutan en la CPU, estas se ejecutan en el *Coprocesador* para esto se añade al principio de cada función la sintaxis `__attribute__((target(mic)))` por ejemplo el algoritmo 16 que devuelve la raíz del heap, solo varia en la sintaxis indicada anteriormente. A su vez tanto *inserta2*, *extrae2*, *popush2*, se deben ejecutar en la *Xeon phi* de modo que se debe añadir al principio de cada función la sintaxis `__attribute__((target(mic)))`.

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec()*. La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 10 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.6. Integración de algoritmo K-nn Paralelo GPU

Para GPU es necesario utilizar *CUDA* que es una arquitectura de cálculo paralelo de *NVIDIA* que aprovecha la potencia de la *GPU*, si bien la lógica central del algoritmo *K - nn* no varia a los métodos presentados anteriormente, solo difiere en estructuras y/o sintaxis específicas de *CUDA*. De esta manera nos enfocaremos solo en las funciones específicas utilizadas en este método. La arquitectura *CUDA* como se menciona en el capítulo 2.2.2

Para realizar la integración de este algoritmo en la interfaz gráfica se realizó mediante una rutina de *Java* con *java.lang.Runtime.exec()*. La implementación de esta rutina permite ejecutar otro programa o ejecutable y obtener los resultados para ser visualizados a través de la interfaz gráfica creada en *Java*. El algoritmo 10 muestra un ejemplo simple de ejecutar un programa o ejecutable desde *Java*.

4.7. Métodos de exportación de resultados

La exportación de los resultados obtenidos se realiza en consideración de los formatos más utilizados comúnmente como lo son archivos de texto plano (.txt), Formato de documento portable (.pdf), Archivos word (.doc), Archivos de hojas de cálculo (.xls), la figura 4.7 muestra como es la interfaz gráfica de esta parte del software.

Como se menciona en la sección anterior 4.2, cuando se ejecuta exitosamente un algoritmo K-nn se muestra la ventana gráfica de los resultados obtenidos, en la parte central de la figura 4.7 se aprecia el recuadro donde se muestra todos los resultados de los K vecinos más cercanos de cada uno de todas las consultas realizadas.

En la figura 4.6 muestra la gráfica de la opción de guardar, cuando se selecciona cualquiera de los tipos de exportación, la gráfica es la misma y es intuitiva de acuerdo al común de las gráficas de exportación de diversos software.

A continuación se muestra extracto de los códigos de exportación de acuerdo a sus formatos, el primer método (Algoritmo 17) corresponde al método de exportación de resultados a Excel en formato *.xls*. Para implementar tanto la exportación a Word y a Excel fue necesario incluir una API llamada *POI* en su versión 3.16 [foundation, 2017], esta API permite que desde una aplicación desarrollada en *Java* se pueda exportar

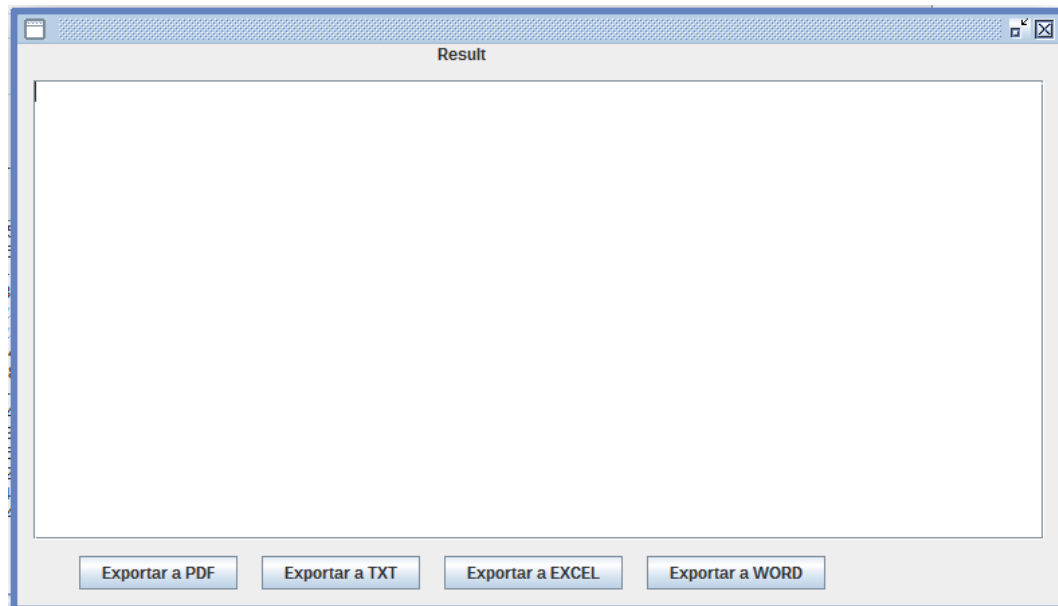


Figura 4.5: Interfaz gráfica de ventana de exportación de resultados

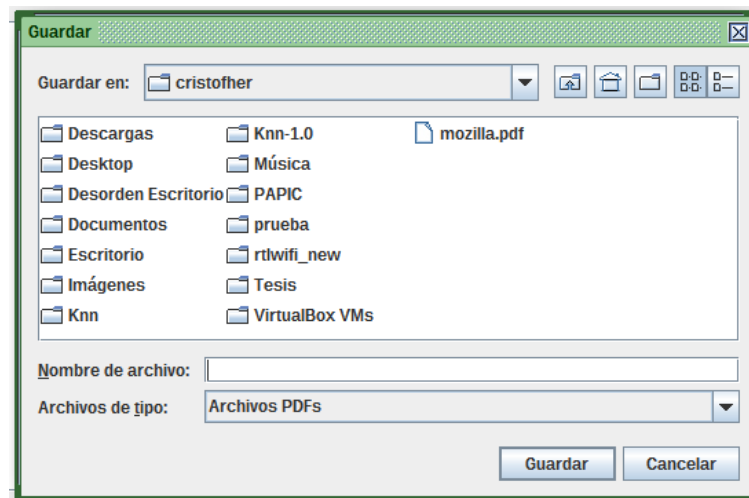


Figura 4.6: Interfaz gráfica de ventana de guardar archivo

a diversos formatos como *Word*, *Hojas de cálculo*, *Presentaciones*, etc siendo los dos primeros considerados en esta tesis.

El segundo método (Algoritmo 18) es el caso de la exportación a Formato de documento portátil (*PDF*) fue necesario la importación de *iText* en su *versión 5* [?], esta biblioteca es de código libre (Open Source) desarrollada por iText Group. Esta esta disponible para *Java* y *C#*. Esta API termine crear y manipular archivos *PDF* - *RTF* - *HTML* en java, siendo el primero considerado para esta tesis.

El tercer método (Algoritmo 19) es el caso de la exportación a archivo de texto plano (*TXT*), para este tipo de documentos no se utilizo librerías externas debido a que *Java* posee métodos para la creación y manipulación de estos archivos.

Los métodos mencionados anteriormente cuentan con una función *obtenerRutaArchivo*, esta permite guardar el archivo con el nombre que le da el usuario y la extensión correspondiente al documento que desea exportar. En caso que el archivo que desea crear ya existe pide la confirmación al usuario si se debe reemplazar el archivo.

4.8. Nuevo modulo Añadir menú

Para añadir un nuevo método al software además de los que ya cuenta por defecto, el usuario cuenta con un menú de opciones para añadir un método creado por el, pero debe cumplir con ciertos requisitos estipulados de acuerdo a cada uno de los tipos de algoritmo ya sea secuencial o paralelo según la arquitectura.

Cabe mencionar que la realización de este menú, fue realizado a través de una interfaz gráfica en JAVA y su lógica en el lenguaje C, de manera que se dividen las tareas en base al principio divide y vencerás. La interfaz gráfica como se aprecia en la figura 4.7, donde se aprecia en la parte izquierda de la interfaz los 4 tipos de algoritmos contenidos en el software de modo que si desea ingresar uno nuevo debe indicar a cual de ellos pertenece, de manera similar a la carga de bases de datos y consultas se debe cargar

Algoritmo 17 Método de exportación de resultados a *Excel*.

```

1 public void generarExcel() throws IOException {
2     String rutaArchivo = obtenerRutaArchivo("xls", "Archivos Excel");
3     if (rutaArchivo != null && jTextArea1.getText().length() != 0) {
4         File archivoXLS = new File(rutaArchivo);
5         if (archivoXLS.exists()) {
6             archivoXLS.delete();
7         }
8         archivoXLS.createNewFile();
9         Workbook libro = new HSSFWorkbook();
10        FileOutputStream archivo = new FileOutputStream(archivoXLS);
11        Sheet hoja = (Sheet) libro.createSheet("Resultados Knm");
12        String texto = jTextArea1.getText();
13        String[] lineas = texto.split("\n");
14        for (int i = 0; i < lineas.length; i++) {
15            Row fila = hoja.createRow(i);
16            String[] sublineas = lineas[i].split(" ");
17            for (int j = 0; j < sublineas.length; j++) {
18                Cell celda = fila.createCell(j);
19                celda.setCellValue(sublineas[j]);
20            }
21        }
22        libro.write(archivo);
23        archivo.close();
24    }
25 }

```

Algoritmo 18 Método de exportación de resultados a *PDF*.

```

1 public void generarPDF() throws IOException, DocumentException {
2     String rutaArchivo = obtenerRutaArchivo("pdf", "Archivos PDFs");
3     if (rutaArchivo != null) {
4         File archivoPDF = new File(rutaArchivo);
5         if (archivoPDF.exists()) {
6             archivoPDF.delete();
7         }
8         archivoPDF.createNewFile();
9         FileOutputStream archivo = new FileOutputStream(archivoPDF);
10        Document documento = new Document();
11        PdfWriter.getInstance(documento, archivo);
12        documento.open();
13        documento.add(new Paragraph("Resultados Knm \n"));
14        String texto = jTextArea1.getText();
15        String[] lineas = texto.split("\n");
16        for (String linea : lineas) {
17            documento.add(new Paragraph(linea));
18        }
19        documento.close();
20        JOptionPane.showMessageDialog(null,
21            "El archivo se a guardado Exitosamente",
22            "Informacion", JOptionPane.INFORMATION_MESSAGE);
23    }
24 }

```

Algoritmo 19 Método de exportación de resultados a *TXT*.

```
1      public void generarTXT() throws IOException {
2          String rutaArchivo = obtenerRutaArchivo("txt", "Archivo de texto plano TXT");
3          try {
4              if (rutaArchivo != null) {
5                  File archivoTXT = new File(rutaArchivo);
6                  if (archivoTXT.exists()) {
7                      archivoTXT.delete();
8                  }
9                  try (FileWriter save = new FileWriter(archivoTXT)) {
10                     save.write(jTextArea1.getText());
11                 }
12                 JOptionPane.showMessageDialog(null,
13                     "El archivo se a guardado Exitosamente",
14                     "Informacion", JOptionPane.INFORMATION_MESSAGE);
15                 Desktop.getDesktop().open(archivoTXT);
16             }
17         } catch (IOException ex) {
18             JOptionPane.showMessageDialog(null,
19                 "Su archivo no se ha guardado",
20                 "Advertencia", JOptionPane.WARNING_MESSAGE);
21         }
22     }
23 }
```

el archivo fuente, este sera visto previamente en el recuadro inferior izquierdo. Además del paso anterior debe especificar el nombre del menú el cual aparecerá en la lista de los menús, para finalizar y lograr añadir correctamente un nuevo método debe presionar el botón *Compile & add Menu* y el recuadro inferior derecho mostrara información detallada respecto del proceso de compilación, de manera que indica si el proceso fue exitoso o no.

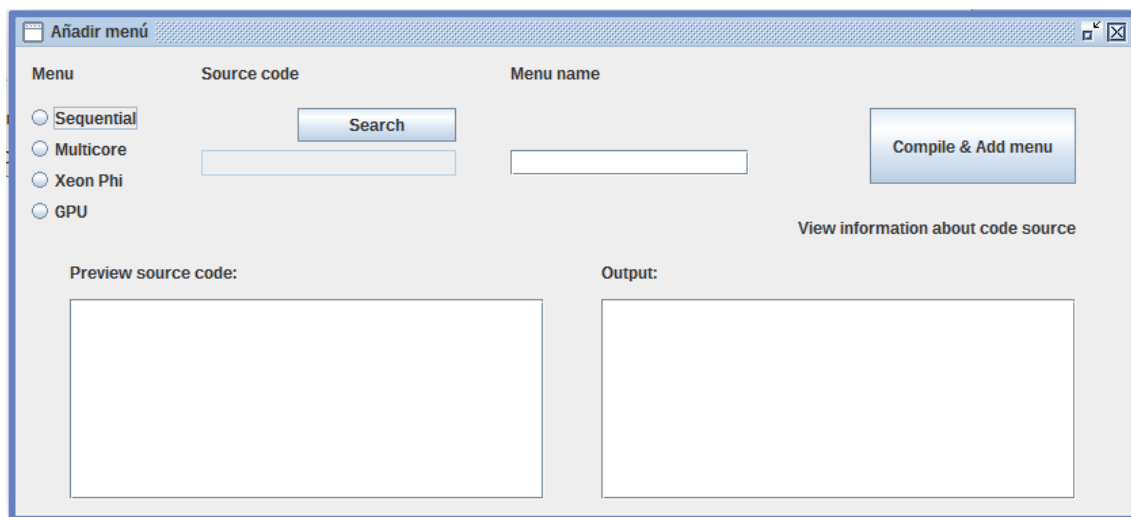


Figura 4.7: Interfaz gráfica de ventana Agregar menú

Capítulo 5

conclusion

5.1. Conclusiones

En este trabajo se han buscado e implementados algoritmos presentes en la literatura para la realización de consultas K-nn en un software diseñado pensando en usuarios los cuales no tengan conocimientos previos en programación de este algoritmo, este software cuenta con algoritmos en tres tipos diferentes de plataformas paralelas: Procesador multi-core, coprocesador Xeon Phi y coprocesador GPU.

Los algoritmos seleccionados se basan en la utilización de heap y búsquedas exhaustivas, de manera que estos presentan buenos resultados en relación al tiempo, de manera que estos fueron implementados exitosamente en un software que junta mundo completamente opuestos, tales como son Xeon Phi y GPU. Cabe destacar que el software no afecta el funcionamiento del algoritmo en términos de tiempo, debido a la capacidad de JAVA de ejecutar la shell (Consola) a través de una rutina simple, de este modo es posible ejecutar los algoritmos programados en C, los cuales son compilados previamente en la instalación y luego solo son llamados para ser ejecutados. Esta idea es la que permite la unión de estos mundos opuestos dentro de un solo software.

Este trabajo potencia la idea de juntar arquitecturas totalmente opuestas, si bien no es necesario realizar la interconectar entre ambas, es posible ampliar esta idea y realizar lo visto en esta tesis, donde un usuario puede disponer de diversas soluciones a un problema y no esta ligado a una arquitectura especifica. Además si no se cuenta con alguno de estos dos coprocesadores el usuario no esta forzado a adquirir uno de estos, puede realizar las consultas desde su propio equipo utilizando tanto el modo secuencial como paralelo pero utilizando las características propias del equipo.

5.2. Trabajos futuros

El presente trabajo es la primera versión de este software, el cual en un futuro puede contar con mejoras a este. Tales como:

- Llevar el software a otro sistema operativo como es windows, debido a ser el sistema operativo más utilizado. Además de intentar llevarlo a OSx, de manera que el software sea multi-plataforma y multi-copresadores. De esta manera se generarla un software más potente.
- Intercomunicación entre el paralelismo multi-núcleo con el paralelismo en Xeon phi
Este trabajo futuro puede ser implementado de variados modos, de manera que se debe investigar y realizar la implementación de esta mejora.
- Intercomunicación entre el paralelismo multi-núcleo con el paralelismo en GPU
Este trabajo futuro puede ser implementado de variados modos, de manera que se debe investigar y realizar la implementación de esta mejora.
- Añadir nuevos métodos de exportación a otros tipos de archivo, que sean requeridos según ciertos criterios.
- Añadir aparte de los 4 métodos principales, nuevos métodos los cuales utilicen otras propuestas para el algoritmo.

- Además no solo se podrían añadir algoritmos Knn sino también otros asociados a métodos de clasificación, o bien el software puede contener una recopilación de algoritmos en diversas áreas.

Bibliografía

- [cud,] Cuda: Compute unified device architecture. In *©2007 NVIDIA Corporation*.
- [xeo,] *PRACE (Partnership for Advanced Computing in Europe). Best Practice Guide - Intel Xeon Phi*.
- [spe, 2011] (2011). Perl, python, ruby, php, c, c++, lua, tcl, javascript and java comparison.
- [Acosta et al., 2012] Acosta, F. A., Segura, O. M., and Ospina, A. E. (2012). Guía y fundamentos de la programación en paralelo. *Revista en telecomunicaciones e informática*, 2(4):81–97.
- [Aha et al., 1991] Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6(1):37–66.
- [and/or its affiliates, 2017] and/or its affiliates, O. C. (2017). Netbeans ide fits the pieces together.
- [Barrientos et al., 2010] Barrientos, R., Gómez, J., Tenllado, C., and Prieto, M. (2010). Heap based k-nearest neighbor search on gpus. In *Congreso Espanol de Informática (CEDI)*, pages 559–566.
- [Bentley and Friedman, 1979] Bentley, J. L. and Friedman, J. H. (1979). Data structures for range searching. *ACM Computing Surveys (CSUR)*, 11(4):397–409.
- [Bernabeu, 2013] Bernabeu, S. R. (2013). Max/min heap, priority queues... ¿qué son y cómo lo hago? comparativa python, c/c++.

- [Brisaboa et al., 2008] Brisaboa, N., Pedreira, O., Seco, D., Solar, R., and Uribe, R. (2008). Clustering-based similarity search in metric spaces with sparse spatial centers. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 186–197. Springer.
- [Brisaboa et al., 2006] Brisaboa, N. R., Farina, A., Pedreira, O., and Reyes, N. (2006). Similarity search using sparse pivots for efficient multimedia information retrieval. In *Multimedia, 2006. ISM'06. Eighth IEEE International Symposium on*, pages 881–888. IEEE.
- [Bustos et al., 2006] Bustos, B., Deussen, O., Hiller, S., and Keim, D. (2006). A graphics hardware accelerated algorithm for nearest neighbor search. In *International Conference on Computational Science*, pages 196–199. Springer.
- [Cederman and Tsigas, 2009] Cederman, D. and Tsigas, P. (2009). Gpu-quicksort: A practical quicksort algorithm for graphics processors. *Journal of Experimental Algorithmics (JEA)*, 14:4.
- [Chapman et al., 2007] Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press.
- [Chávez and Navarro, 2000] Chávez, E. and Navarro, G. (2000). An effective clustering algorithm to index high dimensional metric spaces. In *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pages 75–86. IEEE.
- [Cover and Hart, 1967a] Cover, T. and Hart, P. (1967a). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.
- [Cover and Hart, 1967b] Cover, T. and Hart, P. (1967b). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.
- [Deepak et al., 2001] Deepak, A., Crupi, J., and Malks, D. (2001). Core j2ee patterns: Best practices and design strategies. *Sun Microsystems. Palo Alto*.

- [Freeman, 1969] Freeman, J. (1969). Experiments in discrimination and classification. *Pattern Recognition*, 1(3):207–218.
- [foundation, 2017] foundation, A. (2017). Apache poi - the java api for microsoft documents.
- [Garcia et al., 2008] Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW'08. IEEE Computer Society Conference on*, pages 1–6. IEEE.
- [Hart, 1966] Hart, P. E. (1966). An asymptotic analysis of the nearest-neighbor decision rule. Technical report, DTIC Document.
- [HILBORN, 1968] HILBORN, C. (1968). Dg lainiotis. *IEEE transactions on information theory*.
- [IT, 2017] IT, I. (2017). Papi - papic:profiling.
- [Joskowicz, 2008] Joskowicz, J. (2008). Reglas y prácticas en extreme programming. *Universidad de Vigo*, page 22.
- [Kernighan and Ritchie, 2006] Kernighan, B. W. and Ritchie, D. M. (2006). *The C programming language*.
- [Kuang and Zhao, 2009] Kuang, Q. and Zhao, L. (2009). A practical gpu based knn algorithm. In *International symposium on computer science and computational technology (ISCST)*, pages 151–155. Citeseer.
- [LA, 2017] LA, I. (2017). Procesadores intel® xeon phi™. [urlhttps://www.intel.la/content/www/xl/es/products/processors/xeon-phi/xeon-phi-processors.html](https://www.intel.la/content/www/xl/es/products/processors/xeon-phi/xeon-phi-processors.html),.
- [McDermid, 1993] McDermid, J. y. P. R. (1993). “Software Development Process Models”, en *Software Engineer’s Reference Book*. CRC Press. pp. 15/26-15/28.

- [Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly.
- [Nuñez, 2003] Nuñez, J. M. B. (2003). Investigación de la plataforma j2ee y su aplicación práctica. *Chile: Universidad de Chile, Facultad de Ciencias Físicas y Matemáticas, Departamento de Ciencias de la Computación*.
- [Oracle, 2017] Oracle (2017). ¿cuáles son los requisitos del sistema para java? url<https://www.java.com/es/download/help/sysreq.xml>.
- [Paredes and Navarro, 2009] Paredes, R. U. and Navarro, G. (2009). Egnat: A fully dynamic metric access method for secondary memory. In *Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 57–64. IEEE Computer Society.
- [Plaku and Kavraki, 2007] Plaku, E. and Kavraki, L. E. (2007). Distributed computation of the knn graph for large high-dimensional point sets. *Journal of parallel and distributed computing*, 67(3):346–359.
- [Pressman and Troya, 1988] Pressman, R. S. and Troya, J. M. (1988). Ingeniería del software.
- [Reinders, 2007] Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly.
- [Wang, 2014] Wang, E. Zhang, Q. S. B. Z. G. L. X. W. Q. W. Y. (2014). *High-Performance Computing on the Intel Xeon Phi(TM): How to Fully Exploit MIC Architectures*. Springer, New York.