Trabajo de Fin de Grado

LUCID. Unified Computer Image and Data-Processing

Lucas Hernández Abreu

La Laguna, 14 de julio de 2023

D. **Francisco de Sande González**, con DNI nº 42067050G profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor, y

D. **Pablo López Ramos**, con DNI nº 32752269Q, máster en *Computer Science* por *Queen's college, CUNY*, ingeniero informático senior del Instituto de Astrofísica de Canarias, como co-tutor

C E R T I F I C A N

Que el presente trabajo de Fin de Grado titulado:

"`LUCID. Unified Computer Image and Data-Processing`"

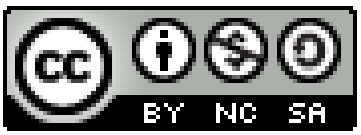
ha sido realizado bajo su dirección por D. **Lucas Hernández Abreu**, con DNI nº 45938122S

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente memoria del Trabajo en La Laguna a 14 de julio de 2023

Agradecimientos

Pablo López Ramos,
Francisco de Sande González,
Violeta Carrillo García-Ramos,
Lorenzo Gabriel Pérez González,
Juan Guillermo Zafra Fernández,
Familia,
Amigos

Licencia



Resumen

Este documento refleja el trabajo de investigación del alumno durante el proceso de desarrollo de una aplicación que implementa un prototipo de un framework para procesamiento concurrente de datos. El procesamiento que se lleva a cabo es genérico y es susceptible de ser aplicado a diversos tipos de datos. En este trabajo se muestra su funcionamiento a la hora de procesar imágenes. La aplicación surge como una necesidad propuesta por el departamento de ingeniería informática del Instituto de Astrofíscia de Canarias (IAC).

Se trata de una aplicación orientada a objetos en la que se han desarrollado diferentes clases que permiten un acceso seguro y rápido a la memoria del sistema. Este proyecto ha sido programado de tal forma que el usuario final no tenga que revisar el código fuente de la aplicación, ya que dispone de una interfaz de programación que podrá utilizar para desarrollar funciones de procesamiento de datos (unidades de procesado) y hacerlas funcionar de forma concurrente en el sistema.

Palabras clave: *Aplicaciones de Consola, C++, Gestión de Memoria, Concurrencia, Hilos, Semáforos.*

Abstract

This document reflects the student's research work during the process of development of an application that implements a concurrent processing data prototype. The processing that is carried out is generic and is caplable of being applied to various types of data. In this work it is shown its operation when processing images. The application comes as a need proposed by the computer engineering department of the Institute of Canary Islands Astronomy (IAC).

The application has been developed following the indications of the Google's C++ style guide and has involved the implementation of different classes that enable safe and fast access to system memory and concurrency. This project has been programmed in such a way that the end user does not have to review the source code to perfom data processing functions (processing units) and make them work concurrently in the system.

Keywords: *Console Applications, C++, Pointers, Memory Management, Mutexes, Concurrency, Threads, Semaphores.*

Índice general

Índice de figuras

Capítulo 1

Introducción

Este documento comprende el trabajo de investigación y desarrollo realizado por el alumno en la consecución de su Trabajo de Fin de Grado (TFG), con el que culminará sus estudios del Grado en Ingeniería Informática cursados en la Escuela Superior de Ingeniería y Tecnología (ESIT) de la Universidad de la Laguna (ULL).

Los sistemas de adquisición de imagen desarrollados para algunos de los instrumentos del Gran Telescopio de Canarias (GTC), en cuya gestión el IAC está involucrado, requieren de un procesado básico de los datos leídos por un sensor, ordenamiento y adecuación a la arquitectura hardware de los datos, para ser enviados a otros componentes del sistema de control. Es necesario realizar este proceso de forma ininterrumpida dentro del tiempo de lectura del detector por largos periodos de tiempo.

LUCID (Unified Computer Image and Data-processing) ha sido desarrollado para solucionar este problema. LUCID es un framework de procesamiento concurrente compuesto por unidades de procesado que permiten, por un lado, y gracias a su concurrencia, mantener el flujo continuo de lectura en el sensor y, por otro, su uso en diferentes instrumentos, dado que el nivel de generalización con que se ha desarrollado permite adecuarlo a diferentes ámbitos de trabajo. El procesamiento de datos que realiza LUCID sigue el paradigma de procesado utilizando un *pipeline* (tubería) en el que los datos a procesar (*unidades de procesado* en la terminología que utilizaremos) pasan por diferentes etapas. Aunque estas unidades de procesado son genéricas, este desarrollo se ha realizado teniendo en mente el procesado de imágenes.

El desarrollo de este TFG resuelve algunos problemas presentes en una versión preliminar del sistema desarrollada por P. López, uno de los tutores de este trabajo. Partiendo de esa versión básica se ha desarrollado este prototipo, teniendo en mente posibilitar su uso en ámbitos genéricos.

Capítulo 2

Objetivos

Este TFG tiene por objetivo el desarrollo de una infrastructura genérica para el procesado de datos procedentes de uno de los sensores del Gran Telescopio de Canarias.

Previo a este desarrollo, el telescopio ya contaba con un software de procesamiento de las imágenes que se reciben desde el sensor. En este software, los procesos a los que son sometidas las imágenes suceden de forma serializada, mientras que en LUCID se aporta concurrencia a esos procesos.

En base a esto, se han definido los siguientes objetivos:

1. Desarrollo de una infrastructura genérica para el procesado de datos, particularizado para el caso de imágenes.

2. Adquisición de conocimientos sobre concurrencia.

3. Desarrollar el código de acuerdo a los estándares definidos en la guía de estilo de Google [1] y generar la documentación del proyecto utilizando Doxygen.

4. Profundizar los conocimientos en diversas tecnologías accesorias que han sido utilizadas en el desarrollo del trabajo.

   Entre estas, cabe destacar el uso de Github [2] como herramienta de control de versiones, LaTeX [3] para la edición de esta memoria, Doxygen [4] para la generación de documentación del proyecto o GTest [5], el framework de pruebas de Google.

5. Por último, tras las labores de investigación y recopilación de información correspondientes, aplicar los conocimientos adquiridos para desarrollar un prototipo funcional que permita el procesamiento de datos de forma concurrente, con el consiguiente aumento en el rendimiento.

Capítulo 3

Herramientas y Tecnologías utilizadas.

En este capítulo se revisarán las tecnologías utilizadas a lo largo del desarrollo de este Trabajo de Fin de Grado.

El proyecto ha sido desarrollado en C++ en un entorno Linux, usando las herramientas más habituales de ese ecosistema de desarrollo. Cabe destacar que todo el código del proyecto se encuentra alojado en un repositorio público de Github [2]. Asimismo, todo el código desarrollado se adhiere a las reglas de estilo de la *Guía de estilo de Google para C++* [1].

3.1. Herramientas de desarrollo y documentación

En esta sección se enumeran las tecnologías de propósito general que se han utilizado para el desarrollo de este TFG. Herramientas como: LaTeX, GTest, Doxygen o CMake.

3.1.1. LaTeX

LaTeX es un sistema de preparación de documentos para composición tipográfica de alta calidad. Se utiliza con mayor frecuencia para documentos técnicos o científicos de tamaño mediano a grande, pero también es útil para otros tipos de publicación.

LaTeX no es un procesador de textos. En cambio, alienta a los autores a no preocuparse demasiado por la apariencia de sus documentos, sino a concentrarse en obtener el contenido correcto [3].

Se usa en:

- Composición tipográfica de artículos de revistas, informes técnicos, libros y

presentaciones de diapositivas.

- Control sobre documentos de gran tamaño que contienen secciones, referencias cruzadas, tablas y figuras.
- Composición tipográfica de fórmulas matemáticas complejas.
- Composición tipográfica avanzada de matemáticas con *AMS-LaTeX*.
- Generación automática de bibliografías e índices.
- Composición tipográfica multilingüe.
- Inclusión de material gráfico y color de proceso o directo.
- Uso de fuentes `PostScript` o `Metafont`.

3.1.2. CMake

CMake es un sistema extensible de código abierto que administra el proceso de compilación en un sistema operativo de manera independiente del compilador. A diferencia de muchos sistemas multiplataforma, está diseñado para usarse junto con el entorno de compilación nativo. Los archivos de configuración simples ubicados en cada directorio de origen (llamados archivos *CMakeLists.txt*) se usan para generar archivos de compilación estándar (por ejemplo, archivos *Makefile* en Unix y proyectos/áreas de trabajo en Windows MSVC) que se usan de la manera habitual. Puede generar un entorno de compilación nativo que compilará código fuente, creará bibliotecas, generará contenedores y compilará ejecutables en combinaciones arbitrarias. Admite compilaciones en el lugar y fuera del lugar y, por lo tanto, puede admitir múltiples compilaciones desde un solo árbol de origen. CMake también admite compilaciones de bibliotecas estáticas y dinámicas. Otra buena característica es que genera un archivo de caché que está diseñado para usarse con un editor gráfico. Por ejemplo, cuando se ejecuta `cmake`, localiza archivos, bibliotecas y ejecutables, y puede encontrar directivas de compilación opcionales. Esta información se recopila en la memoria caché, que el usuario puede modificar antes de generar los archivos de compilación nativos.

CMake está diseñado para admitir jerarquías de directorios complejas y aplicaciones que dependen de varias bibliotecas. Por ejemplo, admite proyectos que constan de varios juegos de herramientas (es decir, bibliotecas), donde cada juego de herramientas puede contener varios directorios y la aplicación depende de los juegos de herramientas más el código adicional. También puede manejar situaciones en las que se deben crear ejecutables para generar código que luego se compila y vincula a una aplicación final. Debido a que CMake es una aplicación de código

abierto y tiene un diseño simple y extensible, se puede ampliar según sea necesario para admitir nuevas funciones. El proceso de compilación se controla mediante la creación de uno o más archivos *CMakeLists.txt* en cada directorio (incluidos los subdirectorios) que conforman un proyecto. Cada *CMakeLists.txt* consta de uno o más comandos. Cada comando tiene la forma `COMMAND (args...)` donde `COMMAND` es el nombre del comando, y `args` es una lista de argumentos separados por espacios en blanco. *CMake* proporciona muchos comandos predefinidos, pero si lo necesita, puede agregar sus propios comandos. Además, el usuario avanzado puede agregar otros generadores de archivos *Makefile* para una combinación particular de compilador/SO [6].

```
1  cmake_minimum_required(VERSION 3.14)
2  project(LUCID)
3
4  set(CMAKE_CXX_STANDARD 11)
5  if (POLICY CMP0135)
6    cmake_policy(SET CMP0135 NEW)
7    set(CMAKE_POLICY_DEFAULT_CMP0135 NEW)
8  endif()
9
10 set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
11 FetchContent_MakeAvailable(googletest)
12
13 add_subdirectory(include)
14
15 add_executable(lucid src/main.cc)
16
17 target_link_libraries(lucid libraries)
```

Listado 3.1: Ejemplo de CmakeLists.txt

En el Listado 3.1 se puede observar un ejemplo de fichero de configuración de *CMake* utilizado en la compilación de LUCID. Algunos de los parámetros en este fichero son:

- Exigir que la versión del compilador a utilizar sea C++11.
- Exigir que los contenidos de *GTest* estén disponibles para su uso.
- Construir el ejecutable y enlazarlo a las librerías.

3.1.3. Google Test

Como parte del desarrollo de este proyecto se han programado un conjunto de tests de código para garantizar, en la medida de lo posible, la corrección de las diferentes funcionalidades que se han ido incorporando al proyecto, y para esta finalidad se ha elegido Gtest.

GTest (GoogleTest) es el framework de Google para escribir pruebas de C++ en una variedad de plataformas (Linux, Mac OS X, Windows, ...). Basado en la

arquitectura xUnit, admite el descubrimiento automático de pruebas, un amplio conjunto de aserciones, aserciones definidas por el usuario, pruebas de muerte, fallas fatales y no fatales, pruebas parametrizadas por valor y tipo, varias opciones para ejecutar las pruebas y generar informes XML de las pruebas [7].

En primer lugar, para poder utilizar GTest, es necesario haber instalado el paquete **gtest** y haber generado los ficheros de testing a través de CMake.

```
enable_testing()
add_executable(all_tests
  tests/semaphore.test.cc
  tests/memory_manager.test.cc
  tests/pipeline.test.cc
)

target_link_libraries(all_tests
  GTest::gtest_main
  include
)

include(GoogleTest)
gtest_discover_tests(all_tests)
```

Listado 3.2: Ejemplo de CmakeLists.txt para testing con GTest

Para poder habilitar los test en la compilación del programa es necesario añadir la opción *enable_testing()* en el fichero *CMakeLists.txt* como se observa en el Listado 3.2. Después de añadir esta opción se creará un ejecutable que iniciará los tests del programa tras su ejecución.

Entonces se podrá utilizar el comando `ctest` en el directorio de construcción de la aplicación para ejecutar los tests especificados en la construcción del ejecutable de tests, en el caso de LUCID: `all_tests`.

```
TEST(SemaphoreTest, MultipleWaitSignal) {
  Semaphore semaphore(0);
  std::vector<std::thread> thread_vector;
  for (int it = 0; it < 10; it++) {
    thread_vector.emplace_back([&semaphore]() {
      semaphore.Wait();
      semaphore.Signal();
    });
  }
  for (int it = 0; it < 10; it++) {
    semaphore.Signal();
  }
  for (auto &thread : thread_vector) {
    thread.join();
  }
  semaphore.Wait();
  semaphore.Signal();
  EXPECT_EQ(semaphore.count(), 10);
}
```

Listado 3.3: Ejemplo de test para la clase *Semaphore*

Para desarrollar una prueba unitaria en la aplicación es necesario que se utilize la macro *TEST*, como se observa en la línea 1 del Listado 3.3. A continuación se debe insertar código entre las llaves de la misma forma en la que se escribe una función de C++. En el test que se encuentra en el Listado se comprueba el funcionamiento del semáforo para múltiples señales provenientes de diferentes hilos. Para ello instancia un semáforo y 10 hilos, pone los hilos en espera y envía señales desde el hilo principal. Luego espera que los hilos terminen su ejecución para terminar. Finalmente comprueba que el número de señales sea 10.

3.1.4. Doxygen

Todo el código fuente de LUCID ha sido anotado con etiquetas de Doxygen para generar la documentación del proyecto. Doxygen [4] es la herramienta estándar de facto para generar documentación a partir código fuente C++ anotado, pero también es compatible con otros lenguajes de programación populares como C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft y versiones UNO/OpenOffice), Fortran y, en cierta medida, D. Doxygen también es compatible con el lenguaje de descripción de hardware VHDL.

Permite tres formas de complementar el desarrollo de código:

1. Puede generar un documento HTML navegable en línea y/o un manual de referencia fuera de línea (en LaTeX) a partir de un conjunto de ficheros de código anotados. También hay soporte para generar resultados en formatos RTF, PostScript, PDF con hipervínculos, HTML comprimido y páginas man de Unix. La documentación se extrae directamente del cópdigo fuente, lo que hace que sea mucho más fácil mantener la coherencia de la documentación con el código.

2. Puede configurar Doxygen para extraer la estructura del código de archivos fuente no documentados. Esto es muy útil para orientarse rápidamente en distribuciones de fuentes grandes. Doxygen también puede visualizar las relaciones entre los diversos elementos mediante la inclusión de gráficos de dependencia, diagramas de herencia y diagramas de colaboración, que se generan automáticamente.

3. También puede usar Doxygen para crear documentación normal, como puede ver en la propia documentación del sitio [4].

Para ejecutar Doxygen debe colocarse en el directorio donde se encuentra el archivo *doxygen.config* y ejecutar el comando `doxygen doxygen.config`

3.2. Soporte en C++ para programación concurrente

3.2.1. Programación multihilo

El procesamiento multihilo (multithreading) es la capacidad de una CPU (o un solo núcleo en un procesador multi-core) de proporcionar múltiples hilos de ejecución de manera concurrente, con el soporte del sistema operativo. En una aplicación multihilo, los hilos comparten los recursos de uno o varios núcleos, que incluyen las unidades de cómputo, las cachés de la CPU y el búfer de búsqueda de traducciones (TLB).

Mientras que los sistemas de multiprocesamiento incluyen múltiples unidades de procesamiento completas en uno o varios núcleos, el multithreading tiene como objetivo aumentar la utilización de un solo núcleo mediante el uso del paralelismo a nivel de hilo, así como el paralelismo a nivel de instrucción. Dado que las dos técnicas son complementarias, se combinan en casi todas las arquitecturas de sistemas modernos con múltiples CPUs de multithreading y con CPUs con múltiples núcleos de multithreading [8].

En C++ el soporte multihilo se añadió con la versión 11 del estándar. Antes de C++11, se debían usar hilos POSIX o la biblioteca `<pthread.h>`. Si bien esta biblioteca cumplía su función, la falta de un conjunto de características estándar proporcionadas por el lenguaje causaba problemas serios de portabilidad. C++11 implementó std::thread. Las clases de hilo y las funciones relacionadas se definen en el archivo de encabezado `<thread>` [9].

std::thread es la clase de hilo que representa un solo hilo en C++. Para iniciar un hilo, se debe crear un nuevo objeto de hilo y pasar al constructor un tipo de objeto que pueda ser llamado, dentro de C++, un objeto de tipo *callable* . Una vez se crea el objeto, se lanza un nuevo hilo que ejecutará el código especificado en el objeto de tipo *callable*. Un objeto de tipo *callable* puede ser cualquiera de los siguientes:

- Un puntero a una función.
- Un objeto de función.
- Una expresión lambda.

LUCID usa hilos para conseguir las mejoras en el rendimiento del procesado de datos permitiendo así la utilización de todos los núcleos disponibles.

3.2.2. Mutex

Mutex (abreviatura de “mutual exclusion”: exclusión mutua; también llamado semáforo binario o cerrojo) es una primitiva de sincronización que se utiliza para

proteger datos compartidos de ser accedidos simultáneamente por varios threads de ejecución de un programa. La primitiva permite realizar dos operaciones: *activar el cerrojo* y *eliminar el cerrojo*. Cuando un hilo adquiere un *mutex* lo bloquea para echar el cerrojo a una sección crítica, de forma que ningún otro hilo pueda acceder a ella, y cuando lo libera, desbloquea el *mutex*, es decir, quita el cerrojo.

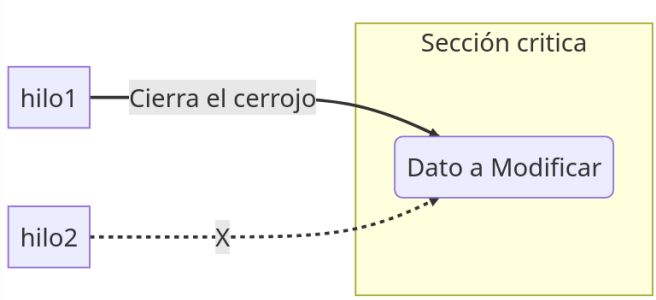


Figura 3.1: Diagrama sobre el funcionamiento de un *mutex*

Según lo expuesto, cabe suponer que un *mutex* tiene la propiedad de que, cuando es adquirido por un hilo, ningún otro hilo puede acceder a la sección crítica, como se representa en el diagrama de la Figura 3.1, hasta que el primero lo haya liberado [10].

3.2.3. Atomic

El tipo de variable atómica funciona de manera similar a la que se aprecia en el diagrama de la Figura 3.1. El principal objetivo de los datos atómicos es conseguir que los accesos a los datos no produzcan comportamientos indefinidos, es decir, condiciones de carrera que terminen por retornar datos inesperados durante la ejecución de la aplicación. La descripción del funcionamiento de los tipos atómicos se encuentra en la documentación del modelo de memoria de C++ [11].

Capítulo 4

Desarrollo de la aplicación

En este capítulo se revisarán en detalle las características de las clases de LUCID y un ejemplo de unidad de procesado.

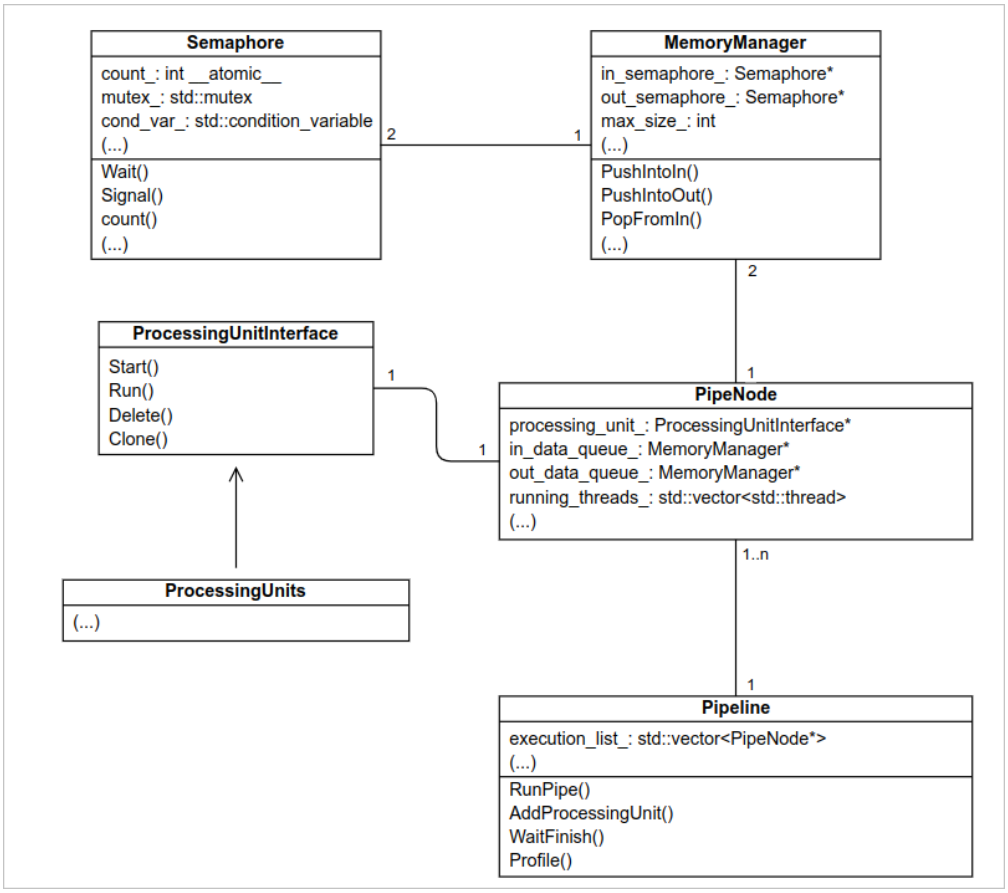


Figura 4.1: Diagrama de clases de LUCID

4.1. Diagrama de clases

En la Figura 4.1 se puede observar un diagrama UML con las relaciones entre las clases de LUCID.

Este diagrama explica la estructura interna de las clases de LUCID y el flujo del programa desde más alto nivel (abajo) hasta más bajo nivel (arriba).

De forma breve se explicarán las relaciones entre las clases que se observan en el diagrama.

- *Pipeline*: es el controlador principal de LUCID, controla los hilos y los accesos a los datos así como la interacción con el usuario. Cada instancia de *Pipeline* contiene n instancias de *PipeNode* que permiten la manipulación de los datos de forma implícita.
- *PipeNode*: cada nodo del *pipeline* está representado por esta clase. Posee una única unidad de procesado que puede ser clonada por múltiples hilos para realizar procesos de forma concurrente sobre los datos de entrada al nodo. Además posee dos instancias de la clase *MemoryManager* que funcionan como colas de entrada y salida para el nodo del *pipeline.*
- *ProcessingUnitInterface*: se trata de la clase padre de todas las unidades de procesado que se vayan a utilizar en el *pipeline*.
- *MemoryManager*: es la clase que gestiona la memoria del programa. Realiza las peticiones y devoluciones de memoria para los datos a procesar por LUCID. Se encuentra en el nivel más bajo del programa junto a la clase *Semaphore*. Posee dos instancias de *Semaphore* que le permiten sincronizar los accesos a memoria.
- *Semaphore*: permite poner en espera a los hilos que solicitan acceso a una sección crítica para que no sucedan condiciones de carrera.

4.2. Semaphore

La clase *Semaphore*[1] representa el semáforo que será utilizado durante la ejecución de LUCID para mantener la sincronización en el acceso a la memoria del programa.

El estándar de C++ incluyó soporte para semáforos en su versión estable de 2020 [12]. Dado que el IAC utiliza la versión del compilador C++11 no es posible hacer uso de esos semáforos y es por ello que se ha optado por implementar una clase semáforo *ad-hoc* para este proyecto.

[1] Clase Semaphore: LUCID/include/definitions/semaphore.cc

Para desarrollar una clase semáforo es necesario entender el orden de los accesos a memoria dentro del programa y especificar, de forma concreta, cómo deben los hilos acceder a los datos.

```
class Semaphore {
 public:
  enum SemaphoreType {
    kNone,
    kIn,
    kOut,
  };
  Semaphore(int = 0, SemaphoreType = SemaphoreType::kNone, bool = false);
  ~Semaphore();
  void Wait();
  void Signal();
  int count();

 private:
  std::atomic<int> count_;
  std::mutex mutex_;
  std::condition_variable condition_variable_;
  std::string type_;
  bool debug_;
};
```

Listado 4.1: Declaración de la clase *Semaphore*.

Los atributos de la clase permiten el control de los accesos de los hilos a la memoria del programa, evitando que las instrucciones de extracción o inserción generen condiciones de carrera.

A continuación se describe brevemente cada uno de los atributos que se pueden ver en el Listado 4.1:

- *count_*: se trata de un atributo de tipo atómico que permite el aumento o decremento de un contador de hilos a la espera sin incurrir en condiciones de carrera.
- *mutex_*: permite bloquear la sección crítica para incrementar o decrementar el atributo *count_*.
- *cond_var_*: permite notificar a los hilos en espera para que continúen el proceso.
- *type_*: este atributo se utiliza solamente si la opción de depurado está activada. Una vez activada, el tipo de semáforo se utilizará para mostrarlo en el mensaje de depurado indicando si el semáforo que realiza una operación es de salida, de entrada o no tiene un tipo asociado.
- *debug_*: si está activado emite los mensajes de depuración de la clase *Semaphore*

En el Listado 4.1 se muestra como la clase consta de tres métodos principales y un tipo enumerado que define el tipo de semáforo.

- *Wait*: este método pone en espera al hilo hasta que reciba la señal de que existe un elemento que pueda extraer, luego decrementa el atributo de tipo atómico *count_* para mostrar que ese ítem que estaba esperando ha sido retirado y permita a otros hilos esperar por los siguientes elementos. De esta forma se evita que dos hilos puedan acceder a un dato, uno de ellos extraiga el dato y el otro extraiga un puntero nulo y produzca un fallo en el programa.
- *Signal*: este método envía una señal mediante el atributo de condición *cond_var_* a alguno de los hilos que esté esperando en *Wait*, aumentando la atributo atómico *count_* en una unidad para que el hilo pueda salir de su espera y continuar con la extracción del dato.
- *count*: este método retorna el contenido actual del atributo *count_*.

Los tres tipos de semáforo que se encuentran en el tipo enumerado del Listado 4.1 indican el tipo de semáforo que está siendo utilizado. Esta información sólo se utilizará si la opción *debug_* está activa. Si esta opción de depurado está activa se podrán comprobar los accesos y las salidas de hilos en el semáforo.

```
(SEMAPHORE Out) BLOCKED, count: -1
(SEMAPHORE Out) PASSED, count: 2
(SEMAPHORE In) SIGNAL count: 1
(SEMAPHORE In) PASSED, count: 0
(SEMAPHORE In) BLOCKED, count: -1
```

Listado 4.2: Ejemplo de mensaje de depuración de *Semaphore*

En el Listado 4.2 se muestra un mensaje de depuración de una ejecución real de `LUCID`. El mensaje se mostrará con diferentes colores dependiendo de la terminal que se utilice para ejecutar el programa. Utilizando *kitty* (un emulador de terminal para Linux y Mac) se podrán ver las siguientes acciones en el mensaje:

- PASSED (verde): indica que el semáforo ha permitido pasar a un hilo.
- BLOCKED (rojo): indica que el semáforo ha bloqueado a un hilo a la espera de la modificación de *count_* y la recepción de una señal.
- SIGNAL (gris): indica que se ha enviado una señal para que un hilo bloqueado continúe su ejecución.
- *count* (sin código de color): se indica el valor del atributo *count_* en el momento de la acción.

4.3. MemoryManager

La clase *MemoryManager*[2] es un gestor de cola genérico que permite el acceso concurrente de múltiples hilos productores y consumidores. El *pipeline* de LUCID depende de esta clase para su funcionamiento, pero *MemoryManager* puede ser utilizada en cualquier ámbito que requiera este tipo de acceso a memoria.

4.3.1. Colas de datos

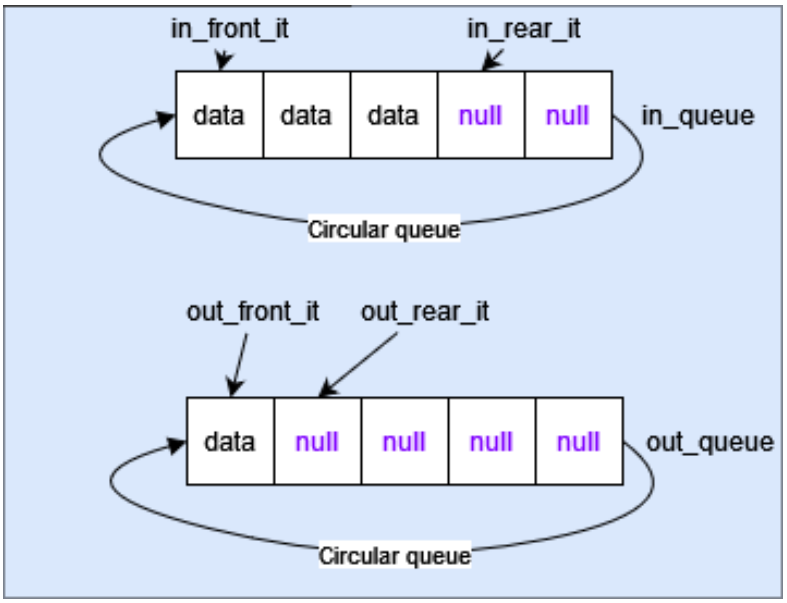


Figura 4.2: Diagrama de las colas internas del *MemoryManager*

Esta clase contiene dos colas de datos cuyo tamaño es fijo. Ambas colas son independientes entre sí, pero pertenecen al mismo objeto.

El proceso de inserción y extracción de datos de la clase *MemoryManager*, reflejado en la Figura 4.2, consta de dos partes:

- Inserción (Push): en la etapa de inserción, el sistema detecta que se ha solicitado a la instancia de *MemoryManager* que inserte un dato en una de las colas. Antes de insertar el dato, se bloquea la sección con un cerrojo. Posteriormente el dato se inserta en la posición señalada por el iterador de cola, llamado *in_rear_it* para la cola de entrada y *out_rear_it* para la cola de salida. Una vez se coloca el dato en esta posición, se incrementa la posición

[2]Clase MemoryManager: `LUCID/include/definitions/memory_manager.cc`

del iterador de cola en una unidad, se desbloquea la sección bloqueada previamente y se envía una señal al semáforo de la cola correspondiente.

- Extracción (Pop): en la etapa de extracción, el sistema detecta que se ha solicitado a la instancia de *MemoryManager* extraer un dato de una de las colas. Para ello se solicita al semáforo de una de las colas que se le permita la posesión de la sección crítica. Una vez obtenida, bloquea la sección con un cerrojo y extrae el dato de la posición apuntada por el iterador frontal, *in_front_it* para la cola de entrada y *out_front_it* para la cola de salida. Si la operación falla, devuelve un puntero nulo y lanza una excepción, mientras que, si la operación es exitosa, devuelve el dato, libera el cerrojo e incrementa el iterador en una unidad.

```
1  class MemoryManager {
2      public:
3      MemoryManager(int, bool = false);
4      ~MemoryManager();
5      bool PushIntoIn(void *);
6      bool PushIntoOut(void *);
7      void *PopFromIn();
8      void *PopFromOut();
9      void LoadMemoryManager(void *);
10     int in_queue_count() const;
11     int out_queue_count() const;
12     int max_size() const;
13     void wait_finish();
14     enum MemoryManagerError {
15         kBadSizing,
16         kNullPtr,
17     };
18     private:
19     void **in_queue_;
20     void **out_queue_;
21     int max_size_;
22     int rear_in_iterator_;
23     int front_in_iterator_;
24     int rear_out_iterator_;
25     int front_out_iterator_;
26     std::atomic<int>
27         in_queue_count_;
28     std::atomic<int>
29         out_queue_count_;
30     Semaphore *in_semaphore_;
31     Semaphore *out_semaphore_;
32     std::mutex push_in_mutex_;
33     std::mutex push_out_mutex_;
34     std::mutex pop_in_mutex_;
35     std::mutex pop_out_mutex_;
36     bool debug_;
37 };
```

Listado 4.3: Declaración de la clase *MemoryManager*.

Los atributos y métodos de la clase *MemoryManager* permiten un acceso a memoria que no resultará en condiciones de carrera.

Cada atributo debe ser explicado de forma breve para que se pueda comprender la obtención de las ventajas descritas con anterioridad. Los atributos de la clase están contenidos en la declaración de la clase en el Listado 4.3.

- *in_queue_*: representa la cola de entrada de la clase. En ella se colocarán los datos que serán leídos por el *pipeline* de LUCID para su procesado.
- *out_queue_*: representa la cola de salida de la clase. En ella se colocarán los resultados de los procesos que se hayan realizado en LUCID.
- *max_size_*: el tamaño máximo de las colas.
- *rear_in_iterator_*: controla la posición de inserción de la cola de entrada. Si la cola está llena no permite insertar más elementos.
- *front_in_iterator_*: controla la posición de extracción de la cola de entrada. Si la cola está vacía hace que el hilo que ha llamado al método de extracción se quede a la espera de que haya elementos.
- *rear_out_iterator_*: controla la posición de inserción de la cola de salida. Si la cola está llena no permite insertar más elementos.
- *front_out_iterator_*: controla la posición de extracción de la cola de salida. Si la cola está vacía hace que el hilo que ha llamado al método de extracción se quede a la espera de que haya elementos.
- *in_queue_count_*: este atributo lleva el conteo de datos que han sido insertados en la cola de entrada.
- *out_queue_count_*: este atributo lleva el conteo de datos que han sido insertados en la cola de salida.
- *in_semaphore_*: este semáforo deja los hilos que ejecuten el método de extracción a la espera de que haya elementos en la cola de entrada.
- *out_semaphore_*: este semáforo deja los hilos que ejecuten el método de extracción a la espera de que haya elementos en la cola de salida.
- *push_in_mutex_*: permite el cierre de la sección crítica de la operación de inserción en la cola de entrada.
- *push_out_mutex_*: permite el cierre de la sección crítica de la operación de inserción en la cola de salida.

- *pop_in_mutex_*: permite el cierre de la sección crítica de la operación de extración de la cola de entrada.
- *pop_out_mutex_*: permite el cierre de la sección crítia de la operación de extracción de la cola de salida.

Para mantener un seguimiento del estado de las colas internas de *MemoryManager* se han creado cuatro atributos de indexación que controlan la posición de insertado y extracción en las mismas. Los atributos que contienen *front* en su identificador son aquellos de los que se extraerá un dato cuando se ejecute una de las operaciones de *PopFromIn* o *PopFromOut*. Los atributos nominados *rear* son aquellos que mantienen un seguimiento de los datos que se insertan en cada cola, y apuntan siempre a la siguiente posición de insertado cuando se realiza una operación *PushIntoIn* o *PushIntoOut*.

Cada vez que se realice la acción de extracción de datos mediante *PopFromIn* y *PopFromOut*, si no hay datos que recoger, los hilos se pondrán a la espera de que lleguen datos a la cola de entrada, o salida, para recogerlos.

Además de lo descrito anteriormente, *MemoryManager* contiene un tipo enumerado (*MemoryManagerError*) cuyas constantes se utilizan en las excepciones lanzadas para que el hilo principal las recoja y gestione adecuadamente.

Las método *PushIntoIn* y *PushIntoOut*, en las líneas 5 y 6 del Listado 4.3, insertan datos en la cola a la que hace referencia el sufijo de la función: *In/Out*.

Se espera que después de instanciar el objeto *MemoryManager* se utilice el método *LoadMemoryManager*. Esta función recoge un puntero como argumento y lo hace pasar por ambas colas, primero la de salida y luego la de entrada, cargando así por primera vez la cola de entrada para poder extraer datos de ella y enviarlos a la cola de salida para su procesado .

4.4. PipeNode

La clase *PipeNode*[3] representa un nodo dentro de la clase *PipeLine*. Cada objeto nodo alojará una unidad de procesado que podrá ser clonada para su ejecución de forma concurrente.

[3]Clase Pipenode: `LUCID/include/definitions/pipe_node.cc`

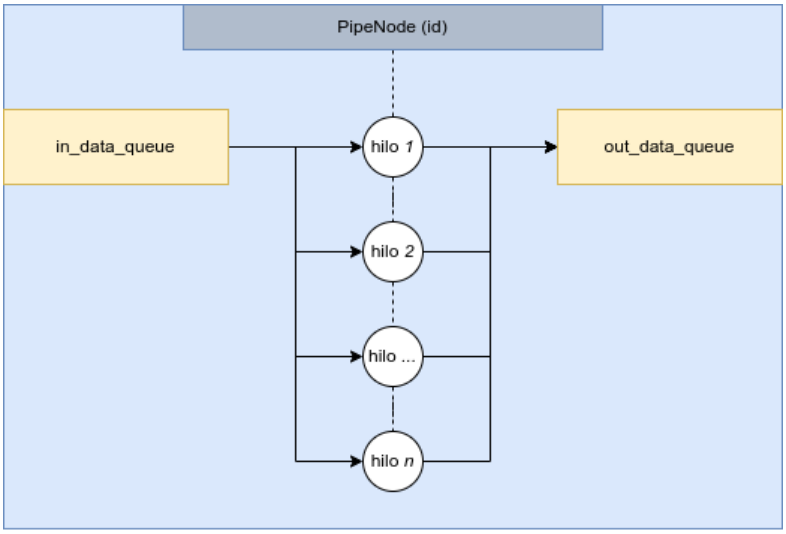


Figura 4.3: Diagrama de funcionamiento básico de PipeNode

En la Figura 4.3 se puede observar el funcionamiento de la clase *PipeNode* y su interacción con las colas de entrada y salida, que son instancias de la clase *MemoryManager* 4.3. El proceso comienza por la creación de los diferentes hilos que se ejecutarán de forma concurrente para ejecutar los procesos a los que se deben someter los datos de entrada. A posteriori de instanciar los hilos para cada uno de ellos se extraerá un dato de la cola de entrada, o quedará bloqueado a la espera de que llegue uno, y lo procesará en la unidad de procesado asignada al nodo. Al finalizar el procesado del dato, lo insertará en la cola de salida para que el siguiente nodo pueda recogerlo.

Los atributos de la clase *PipeNode* permiten mantener control sobre las operaciones que realiza el nodo:

- *node_id_*: identificador del nodo.
- *number_of_instances_*: numero de instancias que mantendrá de forma concurrente el *pipeline* cuando instancie el nodo.
- *in_data_queue_*: permite el acceso a la cola de entrada del nodo para extraer datos a procesar por LUCID.
- *out_data_queue_*: permite el acceso a la cola de salida del nodo para insertar los datos ya procesados y que sean recogidos por el siguiente nodo.
- *processing_unit_*: permite el acceso a la unidad de procesado asignada al nodo, de esta forma será posible clonar dicha unidad para realizar el procesado concurrente.

- *is_last_node_*: opción que indica si el nodo es el último nodo del procesado.
- *running_threads_*: hilos que están en ejecución bajo este nodo y utilizando clones de su unidad de procesado.
- *extra_args_*: argumentos extra que pueden ser insertados en el método *Start* de la unidad de procesado.

```
1  class PipeNode {
2   public:
3    PipeNode();
4    ~PipeNode();
5    void EndNodeWork();
6    MemoryManager *in_data_queue();
7    MemoryManager *out_data_queue();
8    bool last_node();
9    ProcessingUnitInterface *processing_unit();
10   int number_of_instances();
11   int node_id();
12   std::vector<std::thread *> &running_threads();
13   void **extra_args();
14   void in_data_queue(MemoryManager *);
15   void out_data_queue(MemoryManager *);
16   void last_node(bool);
17   void processing_unit(ProcessingUnitInterface *);
18   void number_of_instances(int);
19   void node_id(int);
20   void PushThread(std::thread *);
21   void extra_args(void **);
22
23   private:
24   int node_id_;
25   int number_of_instances_;
26   MemoryManager *in_data_queue_;
27   MemoryManager *out_data_queue_;
28   ProcessingUnitInterface *processing_unit_;
29   bool is_last_node_;
30   std::vector<std::thread *> running_threads_;
31   void **extra_args_;
32 };
```

Listado 4.4: Declaración de la clase *PipeNode*.

La forma de insertar hilos al nodo será mediante el método *PushThread*, que se observa en la línea 20 del Listado 4.4, cada uno de estos hilos instanciará un clon de la unidad de procesado y se quedará en espera en la cola de entrada por la llegada de un dato. Después de terminar el procesado del dato insertará éste en la cola de salida. Una vez lo haya hecho, volverá a bloquearse en la cola de entrada para continuar el procesado si es necesario.

El funcionamiento estándar de un nodo dentro del *pipeline* seguirá los siguientes pasos:

1. Extrae un dato de la cola de entrada a través de la función de extración *PopFromOut* a la que se puede acceder utilizando el método que nos devuelve un puntero a la cola: *in_data_queue*

2. Ahora ese dato se tendrá que enviar través de la unidad de procesado utilizando su método *Run()*.

3. Una vez terminado de procesar el dato se insertará en la cola de salida atendiendo a las siguientes condiciones:

   a) Si el nodo tiene la opción *last_node* activada se insertará en la cola de entrada del gestor de memoria de salida.

   b) Si el nodo tiene desactivada la opción *last_node* se insertará el dato en la cola de salida del gestor de memoria de salida

Es necesario identificar si se trata del último nodo ya que el hilo principal está esperando datos en la cola de entrada mediante *PopFromIn*.

4.5. Data

```
1  class Data {
2   public:
3    struct DataKey {
4      std::string key;
5      void *data;
6    };
7    Data(void *, bool = false);
8    ~Data();
9    void PushExtraData(DataKey *);
10   void *GetExtraData(std::string);
11   void *data();
12
13  private:
14   void *data_;
15   unsigned int index_;
16   std::vector<DataKey *> extra_data_;
17   bool debug_;
18 };
```

Listado 4.5: Declaración de la clase Data

Como se ha podido deducir a partir de la visualización del código, las unidades de procesado reciben un puntero *void* y, por lo tanto, es responsabilidad de cada unidad realizar la conversión adecuada para acceder a los datos. Con el fin de permitir la existencia de unidades de procesado estándar como el indexado, la detección de bordes o la transmisión de datos derivados, se ha creado la clase *Data* que permite agregar información o datos adicionales a medida que avanza a través del *pipeline*.

La clase *Data*[4] representa un dato dentro de la línea de procesado. Esta clase se compone de 3 métodos y una estructura que permite enviar más datos con un único puntero.

Los atributos de la clase permiten la conservación de los datos a través del periodo de vida de LUCID.

- *data_*: el puntero al dato que ha llegado desde la función principal, este puntero no se eliminará del *pipeline*.
- *extra_data_*: se trata de un vector de objetos *DataKey* que permiten llevar múltiples datos a las unidades de procesado a través de un único puntero.
- *debug_*: la opción de depurado. Permite al usuario acceder a la misma para controlar la depuración de las unidades de procesado con este dato en concreto.

La estructura *DataKey* contiene una pareja de datos entre los que está una clave (*key*) y su valor como se puede identificar en el Listado 4.5:

- *key*: Una cadena de caracteres que estará emparejada con un dato específico.
- *data*: El dato al que está emparejado la *key*. Este dato podría llegar a ser otro par *DataKey*, pero no se recomienda realizar este proceso porque conlleva demasiadas solicitudes de memoria y ofuscaría el acceso a los datos. Anulando también el principio en el que se fundamenta LUCID, solicitar memoria una única vez.

Por otro lado, existen el constructor y los tres métodos que representan toda la funcionalidad de la clase:

- *Data*: El objeto de esta clase es necesario que mantenga un puntero que sea invariable durante toda la ejecución. De esta forma, conseguimos que los datos pasen por todo el procesado sin necesidad de llevar un seguimiento demasiado estricto de las direcciones de memoria que pasan por él. Este puntero lo recibe como parámetro en el constructor de la clase
- *PushExtraData*: Internamente, la clase posee un vector que almacena los pares *Datakey*. Este método empuja el nuevo par a dicho vector y lo mantiene almacenado hasta que el objeto sea destruido implícita o explícitamente (no recomendado).

[4]Clase Data: `LUCID/include/definitions/data.cc`

- *GetExtraData*: Este método devuelve el puntero al dato asociado a la *key* que es entregada como argumento. Si no hay datos asociados a esa *key* se devuelve un puntero nulo. Para evitar el almacenamiento innecesario de datos, podría ser útil utilizar la técnica de intercambio de punteros descrita en [13].

- *data*: Este método devuelve el puntero que se está utilizando para transportar esta información a través del *pipeline*. Se debe evitar modificar este puntero.

Cada objeto de esta clase puede contener múltiples datos que, en el caso particular del procesamiento de imágenes, pueden ser utilizados para transportar diferentes fases del procesado de una imagen y asociarlas con un nombre significativo. De esta forma las unidades de procesado posteriores podrán acceder al dato que necesitan para continuar el procesado y, si es necesario, insertar otra fase del procesado de la imagen al vector interno de *Data* o modificar el dato actual.

4.6. ProcessingUnitInterface

La clase *ProcessingUnitInterface*[5] es la interfaz de la que heredarán todas las unidades de procesado que vayan a ser utilizadas en LUCID.

```
1 class ProcessingUnitInterface {
2  public:
3   virtual void Start(void ** = nullptr) = 0;
4   virtual void Run(void *data) = 0;
5   virtual void Delete() = 0;
6   virtual ProcessingUnitInterface *Clone() { return nullptr; };
7 };
```

Listado 4.6: Declaración de la interfaz *ProcessingUnitInterface*

Para implementar nuevas unidades de procesado, estas, deben heredar de la clase *ProcessingUnitInterface* (ver Listado 4.6) y sobrescribir todos los métodos virtuales de la interfaz. Cada método tiene una tarea única dentro de la unidad de procesado:

- *Start*: permite solicitar memoria e inicializar datos antes de recoger los datos a procesar. Este método es invocado al instanciar un nuevo nodo en el *pipeline*.

- *Run*: este método es la "función principal"de la unidad de procesado. El método recibirá un puntero que contiene los datos que han salido de la cola de entrada, tras extraerlos es conveniente que se realice una conversión de

[5]Clase ProcessingUnitInterface: `LUCID/include/headers/processing_unit_interface.h`

la interpretación de los datos realizando una conversión de tipos de C++ `static_cast<Data*>(data)` o la conversión tradicional de C: `(Data*)data`. De esta forma, se obtiene acceso a los métodos de la clase *Data* para obtener el dato en sí o la información extra que ha sido depositada en el vector de datos extra.

- *Delete* Este método se puede utilizar dentro del metodo *Run* para eliminar memoria que haya sido solicitada y no sea necesario mantener. Dicha acción nos permite mantener un programa que consuma poca memoria, siguiendo así la filosofía principal del programa de capturar memoria sólo una vez a través del gestor de memoria.

- *Clone* Esta función debe retornar un puntero a una nueva instancia de la unidad de procesado que se está definiendo. Este método solo puede ser utilizado en unidades de procesado que sean reentrantes, es decir, que puedan ser invocadas de forma segura en entornos concurrentes. Por ejemplo: `{return new ProcessingUnitInterface}`

4.6.1. Ejemplo de Unidad de procesado: EdgeDetector

Para ilustrar el modo en que el usuario debe crear las unidades de procesado se ha desarrollado la unidad de procesado *EdgeDetector*. Esta unidad de procesado implementa un detector de bordes utilizando el filtro de Sobel.

El Listado 4.7 muestra la declaración de la clase.

```
1  class EdgeDetector : public ProcessingUnitInterface {
2   public:
3    EdgeDetector();
4    ~EdgeDetector();
5    void Start(void ** = nullptr) override;
6    void Run(void *) override;
7    void Delete() override;
8    ProcessingUnitInterface *Clone() override;
9    int **Convolution(int **, int **, int, int);
10   void Magnitude(pixel_value **, int **, int **, int, int, int);
11
12   private:
13    pixel_value IntToPixel(int);
14    int Bound(int);
15    int Hypotenuse(int, int);
16    int **sobelx_;
17    int **sobely_;
18    bool debug_;
19 };
```

Listado 4.7: Declaración de la clase *EdgeDetector*

Como se observa en el Listado 4.7, todos los métodos que heredan de *ProcessingUnitInterface* están marcados con la palabra clave *override*. De esta

forma cuando se quiera acceder a los métodos de la clase *ProcessingUnitInterface*, estos habrán sido sobreescritos por el funcionamiendo que se describe en la clase *EdgeDetector*.

```
//(...)
void EdgeDetector::Start(void **pre_process_args) {
  sobelx_ = (int **)malloc(3 * sizeof(int *));
  sobely_ = (int **)malloc(3 * sizeof(int *));
  for (int it = 0; it < 3; ++it) {
    sobelx_[it] = (int *)malloc(sizeof(int));
    sobely_[it] = (int *)malloc(sizeof(int));
  }
  sobelx_[0][0] = -1;
  // (...)
  sobely_[0][0] = 1;
  // (...)
}

void EdgeDetector::Run(void *data) {
  Data *handler = (Data *)data;
  int **img = (int **)handler->GetExtraData("truncated");
  // (...)
}

void EdgeDetector::Delete() {
  for (int it = 0; it < 3; ++it) {
    free(sobelx_[it]);
    free(sobely_[it]);
  }
  free(sobelx_);
  free(sobely_);
}

ProcessingUnitInterface *EdgeDetector::Clone() { return new EdgeDetector; }

// (...)
```

Listado 4.8: Definición de la clase *EdgeDetector*.

A continuación se presenta una explicación del orden de ejecución de los métodos de la unidad de procesado para procesar el dato entrante desde la cola de entrada:

1. *Start*: es el primer método en ejecutarse. En el Listado 4.7 se observa como se utiliza para solicitar la memoria necesaria para las variables *sobelx_* y *sobely_* para los cálculos que se realizarán posteriormente cuando llegue el dato del gestor de memoria.

2. *Run*: El siguiente método en ejecutarse tendrá como parámetro el dato de la cola de entrada, como se muestra en el Listado 4.8. Lo primero que se realiza es extraer toda la información necesaria del dato para el procesado. Posteriormente se aplica la convolución sobre la imagen y, una vez finalizado todo el proceso, se inserta el resultado en el vector de datos extra del dato que ha llegado como parámetro.

3. *Delete*: Es el último método en ejecutarse. En el Listado 4.8 se puede observar como este método se utiliza para liberar la memoria que se ha solicitado durante la ejecución del método *Start*.

4.7. Pipeline

La clase *Pipeline*[6] es el controlador principal de los objetos *PipeNode*. La instancia comenzará a ejecutarse antes de enviar los datos a través del gestor de memoria de la función principal (*main*).

Cada hilo de ejecución dentro de la *Pipeline* se bloquea esperando por la entrada de datos que, internamente, será la salida del anterior nodo a excepción del hilo principal, donde la entrada depende de la salida del objeto *Pipeline* y la salida depende del código que haya escrito el usuario en al programa principal.

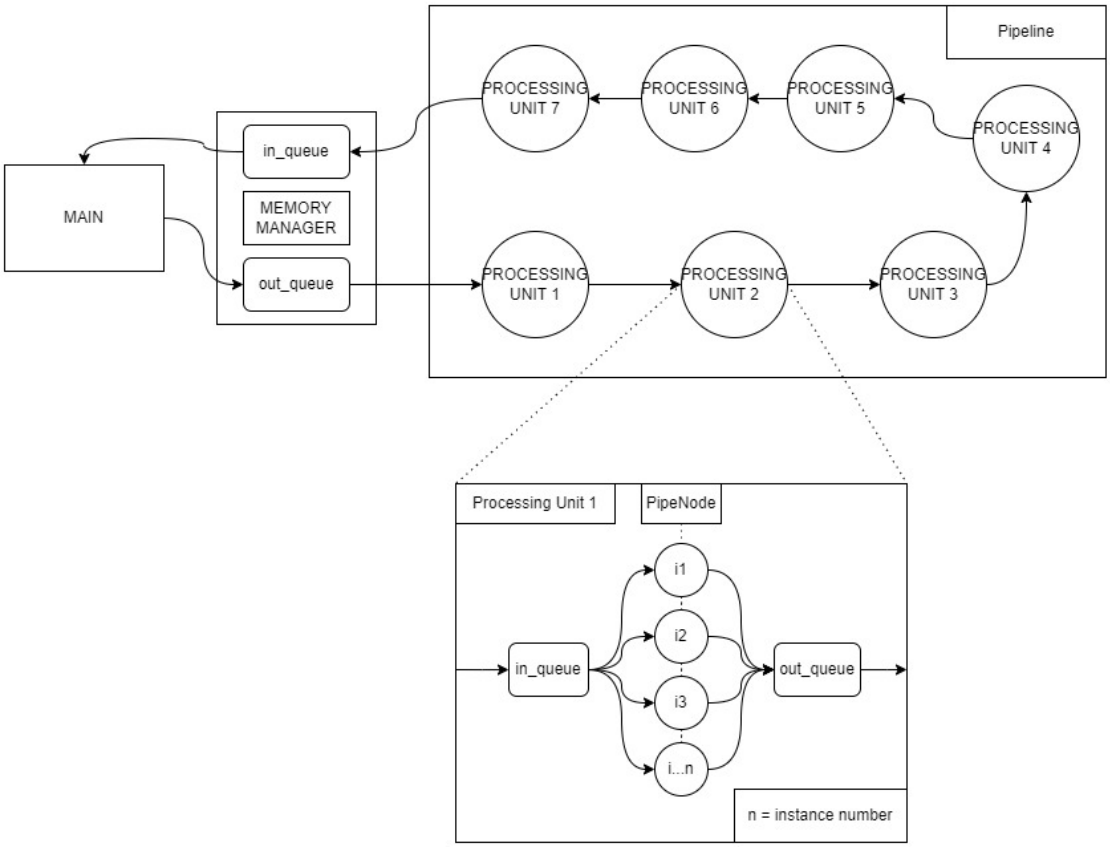


Figura 4.4: Diagrama de funcionamiento de la clase Pipeline

[6]Clase Pipeline: `LUCID/include/definitions/pipeline.cc`

Esta clase se encarga de ocultar la implementación real del *pipeline* a la función *main*. La función *main* solo posee una instancia de la clase *MemoryManager*, cuya definición se encuentra en la sección 4.3. Mientras, la implementación real del *pipeline* ha instanciado múltiples hilos y objetos *MemoryManager* (uno por unidad de procesado) como se puede observar en el diagrama de la Figura 4.4

```
class Pipeline {
public:
  enum PipelineError {
    kBadArgumentFormat,
    kBadArgumentType,
  };
  struct Profiling {
    i32 node_id;
    i32 thread_id;
    u64 cycles_start;
    u64 cycles_end;
    TIME_POINT time_start;
    TIME_POINT time_end;
    i64 sys_time_start;
    i64 sys_time_end;
  };
  Pipeline(ProcessingUnitInterface *, MemoryManager *, int, bool = false, bool = false);
  ~Pipeline();
  void AddProcessingUnit(ProcessingUnitInterface *, int, const char * = nullptr, ...);
  int RunPipe();
  void WaitFinish();
  void Profile();
private:
  enum ArgumentType {
    kInt,
    kUnsigned,
    kFloat,
    kExponential,
    kString,
    kChar
  };
  std::vector<PipeNode *> execution_list_;
  int count_arguments(const char *);
  ArgumentType extract_arg(const char *, u64);
  std::mutex execution_mutex_;
  std::mutex profiling_mutex_;
  int node_number_;
  bool debug_;
  bool show_profiling_;
  std::vector<Profiling> profiling_list_;
};
```

Listado 4.9: Declaración de la clase *Pipeline*.

Los atributos de la clase *Pipeline* mantienen el control de los nodos internos y de los datos de análisis de rendimiento (*profiling*) del programa.

- *ArgumentType*: se trata de un tipo enumerado que contiene los tipos de datos que pueden formar parte del método "variádico"[7] *AddProcessingUnit*.

[7]Funciones "variádicas": enlace.

- *execution_list_*: es el vector de nodos que deben ser ejecutados en el *pipeline*.
- *execution_mutex_*: el *mutex* de ejecución se encarga de cerrar la sección crítica al inicio de la ejecución de cada nodo.
- *profiling_mute_*: el *mutex* de *profiling* se encarga de cerrar la sección crítica a la hora de tomar medidas para, luego, imprimir por pantalla los tiempos de ejecución del programa.
- *node_number_*: el número de nodos del *pipeline*.
- *debug_*: esta opción permite mostrar los mensajes de depurado para el *pipeline*.
- *show_profiling_*: esta opción permite mostrar los mensajes de *profiling* de la aplicación.
- *profiling_list_*: se trata de la lista de datos de *profiling* que pueden ser impresos por pantalla.

El constructor de la clase *Pipeline* recibe cuatro argumentos. Estos argumentos permiten la manipulación de las instancias de la clase, así como permite la apertura de nuevas posibilidades como, por ejemplo, instanciar el *pipeline* dentro de unidades de procesado.

- Un puntero a una instancia de un hijo de la clase *ProcessingUnitInterface*: esta será la primera unidad de procesado en ejecución, los datos pasarán por la misma antes de circular a través de *pipeline*
- Un puntero a una instancia de la clase *MemoryManager*.

Esta clase contiene métodos que permiten manipular el comportamiento del *pipeline* sin modificar su implementación base. El método *AddProcessingUnit* que se encuentra en la línea 6 del Listado 4.9 permite añadir unidades de procesado, en el orden de inserción, al *pipeline*. Recibe por la línea de argumentos múltiples parámetros:

- Un puntero a un objeto hijo de la clase *ProcessingUnitInterface*: de esta forma se pueden añadir unidades de procesado para la modificación de datos que hayan sido desarrollados por el usuario de LUCID.
- Un entero: este número entero indica el número de instancias que tendrá la unidad de procesado para poder acelerar su ejecución, aprovechando el número máximo de núcleos para ejecutar las unidades.

- Una cadena de caracteres que indica el formato de los argumentos "variádicos".
- La lista de argumentos extra que se enviará al método Start de la unidad de procesado.

Otro de los métodos es *RunPipe*. Éste método pone en ejecución el *pipeline*, es decir, cada una de las unidades de procesado se pone a la espera de datos en la cola de entrada.

El último método, *WaitFinish*, pone en espera al hilo principal hasta que el *pipeline* termine la totalidad de su ejecución.

```
int SleeperMain(bool debug_flag, bool pu_debug_flag, bool profiling) {
  Sleeper sleeper_unit;
  Sleeper sleeper_unit2;
  NullUnit void_unit;
  MemoryManager *data_in = new MemoryManager(2, debug_flag);
  for (int i = 0; i < 2; i += 1) {
    Data *holder = new Data(new int(i));
    holder->PushExtraData(new Data::DataKey{"profiling", &profiling});
    data_in->LoadMemoryManager(holder);
  }

  Pipeline *pipe = new Pipeline(&void_unit, data_in, 1, debug_flag, profiling);
  pipe->AddProcessingUnit(&sleeper_unit, 1, "d", 1);
  pipe->AddProcessingUnit(&sleeper_unit2, 10, "d", 1);
  pipe->RunPipe();

  for (int i = 0; i < 4; ++i) {
    if (debug_flag) {
      printf("%s(main) Popping from IN %s\n", LUCID_CYAN, LUCID_NORMAL);
    }
    int *data_handler = (int *)data_in->PopFromIn();
    if (debug_flag) {
      printf("%s(main) Pushing into OUT %s\n", LUCID_CYAN, LUCID_NORMAL);
    }
    data_in->PushIntoOut(data_handler);
  }
  pipe->WaitFinish();
  pipe->Profile();
  return 0;
}
```

Listado 4.10: Implementación de un *pipeline* de LUCID

En la Figura 4.10 se puede observar que la implementación de un *pipeline* de LUCID no tiene una complejidad destacable. El proceso de creación del *pipeline* sigue la estructura de la Figura:

1. Instanciar las unidades de procesado
2. Instanciar el *MemoryManager*: en esta instancia viajarán los datos entre la función *main* y el *pipeline* de LUCID.
3. Cargar el *MemoryManager*: los datos de entrada se cargan en el gestor de memoria de LUCID para introducirlos más tarde en el *pipeline*.

4. Instanciar el *Pipeline*: se inserta la primera unidad de procesado, la instancia del *MemoryManager*, el número de instancias de la unidad de procesado y las opciones de depurado y *profiling*.

5. Se añaden más unides de procesado (si es necesario): las unidades añadidas mediante el método *AddProcessingUnit* tienen la posibilidad de recibir argumentos ilimitados a través del Método variádico.

6. Comenzar la ejecución del *pipeline* mediante el método *RunPipe*.

7. Cargar los datos en el *pipeline* a través de la instancia del *MemoryManager*.

8. Esperar por la finalización de la ejecución

Capítulo 5

Resultados

En este capítulo se presentan los experimentos a los que se ha sometido LUCID y los resultados obtenidos con la ejecución de los mismos.

A continuación se van a exponer dos experimentos para comprobar las ventajas de LUCID sobre el software que no procesa los datos de forma concurrente.

Para ambos experimentos se ha utilizado la siguiente configuración de sistema:

- Arquitectura: x86_64.
- Procesador: Intel(R) Core(TM) i7-8750H CPU @2.20GHz.
- Hilos por núcleo: 2
- Núcleos: 6
- Compilador utilizado: c++ (GCC) 13.1.1 20230429
- Opciones del compilador: -g

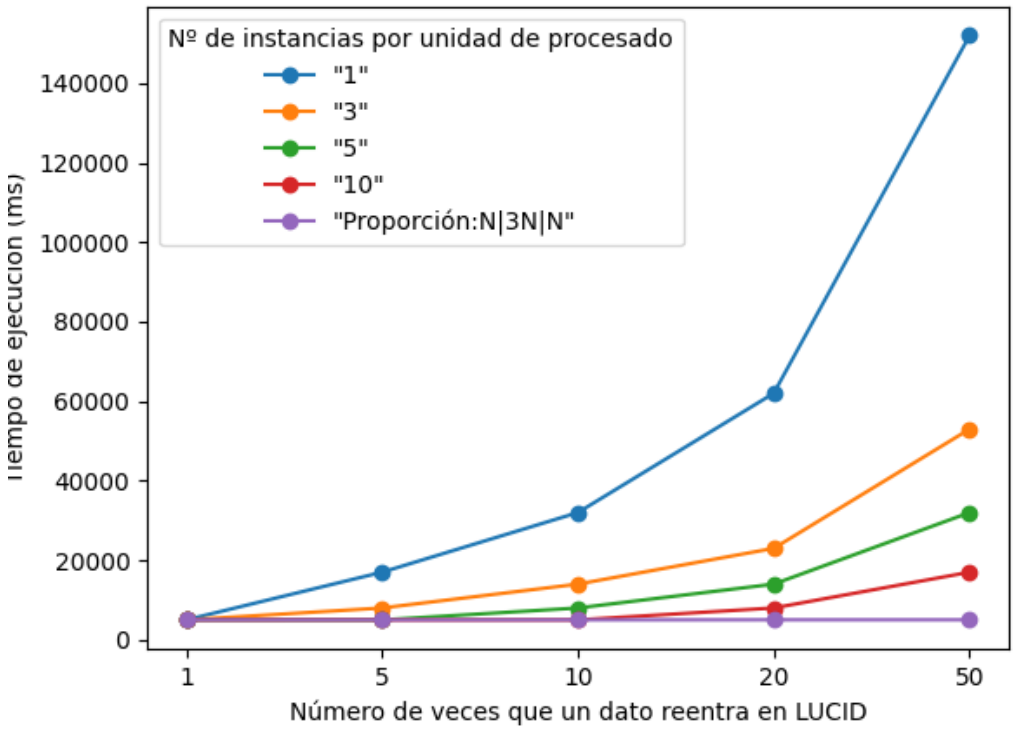


Figura 5.1: Experimento con Sleeper

El experimento con la unidad de procesado *Sleeper* 5.1 trata de medir la diferencia que conlleva utilizar múltiples instancias en unidades de procesado que tienen una gran carga de trabajo en vez de una única instancia.

Los *Sleeper* permiten medir con precisión las aceleraciones de procesado sin verse afectados por el número de procesadores disponibles ya que no realizan ningún tipo de cómputo. Sin embargo, sí interrumpen el flujo de datos a través del *pipeline* al simular diferentes retardos, lo cual es suficiente para medir el efecto en el flujo continuo de datos (*throughput*) que produce el incremento de instancias en diferentes unidades de procesado. Si bien la latencia siempre será la misma y, por lo tanto, tendrá un mayor impacto cuando se procesen menos datos, se puede apreciar una mejora sustancial en el *throughput*.

Para realizar el experimento se han instanciado tres objetos del tipo *Sleeper* y se han asignado los tiempos de bloqueo de los hilos:

1. Duerme el hilo por 1 segundo.

2. Duerme el hilo por 3 segundos.

3. Duerme el hilo por 1 segundo.

A continuación, se ha utilizado la función integrada de *nushell*, `timeit`, para medir los tiempos de ejecución con distinto número de datos de entrada y diferentes configuraciones de instancias. En la Figura 5.1:

- La línea azul representa a LUCID con 1 instancia de cada *Sleeper*.
- La línea naranja representa a LUCID con 3 instancias de cada *Sleeper*.
- La línea verde representa a LUCID con 5 instancias de cada *Sleeper*.
- La línea roja representa a LUCID con 10 instancias de cada *Sleeper*
- La línea morada representa a LUCID con un número de instancias igual a la proporción de $1N \times 3N \times 1N$ para $N =$ número de datos reentrantes.

Los peores datos registrados se obtuvieron con LUCID utilizando una única instancia para cada *Sleeper*.

Como se puede observar en la Figura 5.1, el número de datos de entrada afecta en mayor forma a las pruebas con menos instancias por unidad de procesado. A medida que aumenta el número de instancias disminuye el tiempo de procesado puesto que, si se tiene en cuenta el experimento realizado, se puede determinar que un dato tendrá que esperar 1 segundo para poder ser recogido por el *Sleeper* que se bloquea por 3 segundos y luego tendrá que esperar esos 3 segundos para ser recogido por el de 1 segundo final. Si se aumenta el número de instancias y se tiene en cuenta que los hilos siguen el patrón que hará esperar siempre a cada dato que llegue, se puede preveer que la mejor configuración de instancias para lucid es el número de segundos de bloqueo de cada unidad de procesado multiplicado por el número de datos reentrantes. De esa forma se ha obtenido el resultado que se aprecia en la línea morada de la figura. Dada la configuración óptima se puede esperar que el tiempo de procesado de LUCID sea siempre el mismo para cualquier número de datos de entrada.

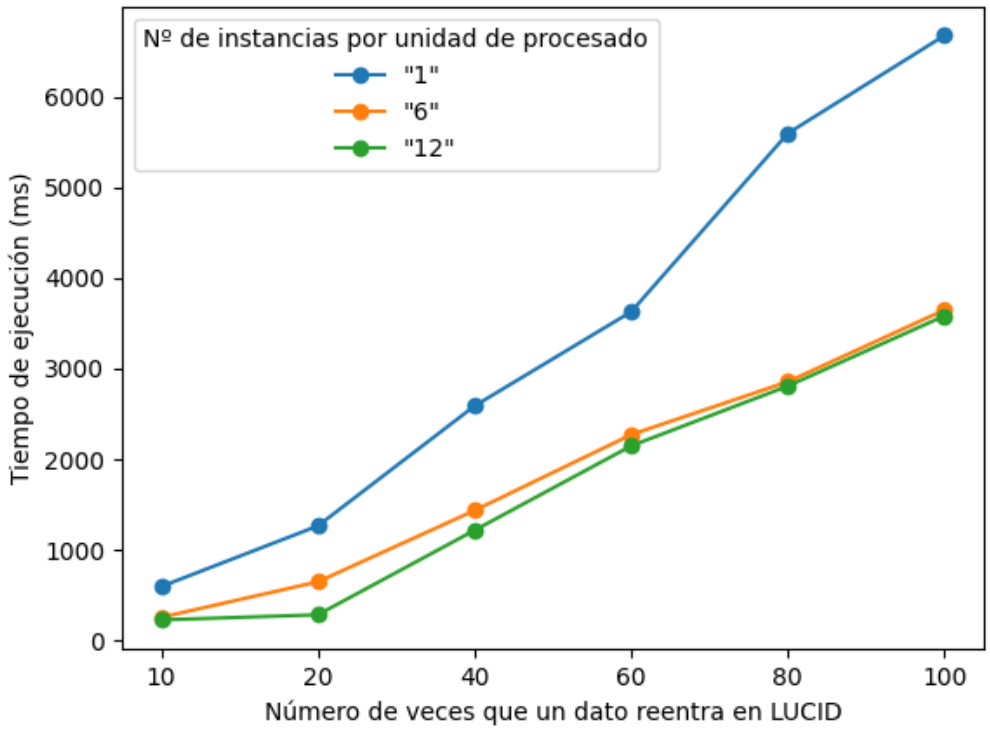


Figura 5.2: Experimento con EdgeDetector

El experimento para la unidad de procesado *Edgedetector* 5.2 obtiene datos sobre la capacidad de LUCID para optimizar un problema real añadiendo concurrencia a la solución. La línea de color azul representa el programa anterior (LUCID con una única instancia por unidad de procesado) y las líneas amarilla y roja representan a LUCID utilizando la concurrencia.

En los puntos de la gráfica se puede observar una mejora del 50 % en todos los puntos e, incluso, de más de un 50 % en la última medida.

La línea roja representa a LUCID con 6 instancias por unidad de procesado y la amarilla con 12 instancias. Las diferencias entre estas dos medidas casi no distan, esto se debe a que el procesador con el que se han ejecutado las pruebas dispone de 12 núcleos, pero varios de ellos están siendo utilizados por otros procesos como el control de temperatura, los gráficos de la pantalla, etc... Y, a diferencia del experimento con *Sleepers*, los procesos del *EdgeDetector* sí realizan operaciones pesadas en el procesador.

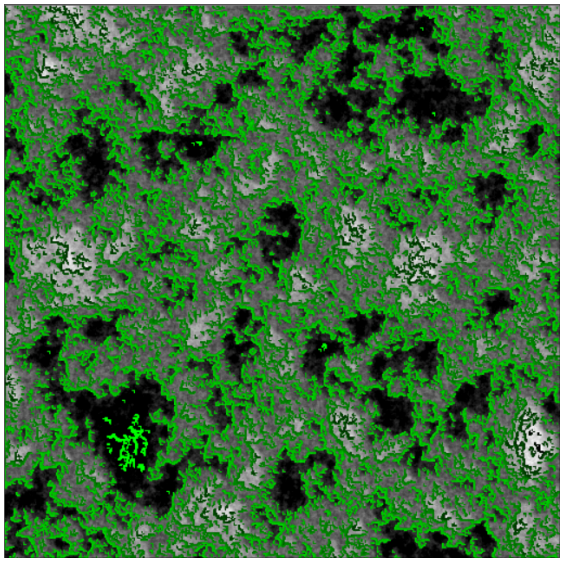


Figura 5.3: Ejemplo de resultado de *EdgeDetector* aplicado a un mapa de ruido de Perlin

En la Figura 5.3 se puede observar el resultado de las operaciones realizadas durante el experimento sobre la unidad de procesado *EdgeDetector*. Para que la unidad de procesado pudiese manipular los datos fue necesario someterlos a un proceso previo que influyó, también, en las medidas del experimento.

Este procesado previo consta de:

- Normalizar los datos de la imagen: de esta forma se evita que los datos sean menores a 0.
- Truncar la imagen: se debe realizar la operación de truncado que sigue la siguiente fórmula: $int(\frac{x}{100}) \times 100$ donde x será el dato de la matriz de la imagen. Al dividir los datos entre 100 y ser convertidos en datos enteros, estos pierden su parte decimal. De esta forma los datos de la matriz de la imagen ahora serán siempre múltiplos de 100.
- Detectar los bordes de la imagen: es el proceso que más carga aplica sobre el procesador.
- Exportar la imagen: finalmente se genera una imagen a partir de la matriz de datos y se colorean los bordes encima de los datos originales que han viajado a través del *pipeline*.

Los experimentos muestran que LUCID es capaz de realizar procesos, tanto ligeros como pesados, de forma concurrente y obtener resultados aceptables. Estos

resultados pueden mejorar siempre que se conozca la proporción de tiempo de espera. De esta forma se podría aplicar la configuración óptima de LUCID para obtener los resultados representados en la Figura 5.1.

Capítulo 6

Conclusiones y líneas de trabajo futuras

6.0.1. Conclusiones y líneas de trabajo futuras

La investigación realizada durante este Trabajo de Fin de Grado ha permitido desarrollar un *framework* de procesado concurrente de propósito general. LUCID, permite un desarrollo rápido y seguro de procesos concurrentes que deben aplicarse a datos en un *pipeline*,,especialmente sistemas de *streaming*, sistemas de adquisición/reducción de datos, sistemas de control de lazo cerrado (como la óptica adaptativa) o visualizadores de video, entre otros. La generalización de LUCID permite que el usuario final únicamente deba programar los procesos por los que quiere hacer pasar los datos, dejando así el proceso de coordinación concurrente de bajo nivel al framework. Además, LUCID ha sido desarrollado con documentación generada por Doxygen y utilizando el *framework* de tests de Google. Si el usuario necesitase acceder al código fuente de la aplicación, las explicaciones que se encuentran en la documentación del mismo evitarán que edite de forma innecesaria alguna de las clases base del proyecto.

El desarrollo de LUCID puede continuar partiendo del framework base que se ha creado, algunas de las propuestas son:

- Desarrollo de una aplicación gráfica para LUCID: en esta línea de trabajo se propone el desarrollo de una interfaz gráfica para LUCID que permita la creación dinámica de *pipelines* utilizando librerías de unidades de procesado. Esta interfaz también proporcionará la capacidad de visualizar el estado interno de las colas del *MemoryManager* y el estado general del *pipeline*.

- Permitir la modificación dinámica del *pipeline* durante la ejecución. Por ejemplo: poder insertar/eliminar unidades de procesado o incrementar/decrementar el número de instancias en tiempo real.

- Desarrollo de librería de unidades de procesado: en esta línea de trabajo se propone el desarrollo de una librería estándar que contenga suficientes unidades de procesado para que el usuario pueda realizar procesados básicos a sus datos combinándolas.

- Desarrollo de un sistema de *profiling* de la aplicación completo: a partir de la información que ya se recolecta en LUCID sobre los tiempos de ejecución, desarrollar un sistema inteligente que asigne automáticamente el número de instancias para optimizar las unidades de procesado.

6.0.2. Conclusions and future works

The research conducted during this Bachelor's Thesis has allowed the development of a general-purpose concurrent processing framework. LUCID enables fast and secure development of concurrent processes that need to be applied to any type of data in a pipeline, especially streaming systems, data acquisition/reduction systems, closed-loop control systems (such as adaptive optics), or video players, among others. The generalization of LUCID allows the end user to only program the processes through which they want to transform the data, leaving the low-level concurrent coordination process to the framework. Additionally, LUCID has been developed with documentation generated via Doxygen and using the Google testing framework. If the user needs to access the application's source code, the explanations found in its documentation will prevent unnecessary editing of any of the project's base classes.

The development of LUCID can continue based on the existing framework that has been created. Some of the proposals are:

- Development of a graphical application for LUCID: In this line of work, the development of a graphical interface for LUCID is proposed. This graphical interface will allow the dynamic creation of pipelines using processing unit libraries. It will also provide the capability to visualize the internal state of the *MemoryManager* queues and the overall *pipeline* status.

- Enable dynamic modification of the *pipeline* during execution: For example, being able to insert/delete processing units or increase/decrease the number of instances in real-time.

- Development of a processing unit library: In this line of work, the development of a standard library is proposed. This library would contain an adequate number of processing units, enabling users to perform basic data processing by combining them.

- Development of a complete profiling system: utilizing the runtime information already collected in LUCID regarding execution times, an intelligent system would be developed to automatically assign the optimal number of instances to optimize the processing units.

Capítulo 7

Presupuesto y despliegue

En este capítulo se expondrán las estimaciones de recursos necesarios para desplegar LUCID de forma efectiva.

El desglose del presupuesto de la aplicación se puede separar en dos partes: por un lado, el desarrollo de la aplicación, y por otro la integración y el mantenimiento de la misma. Se puede contabilizar de la siguiente manera:

Tarea	**Tiempo estimado (h)**	**Coste (€)**
Desarrollo de la aplicación	300	7200
Documentación y pruebas	150	3600
Integración	40	960
Mantenimiento	2080	14560
Total	**2570**	**26320**

Cuadro 7.1: Resumen de precios

Para la implementación de LUCID se propone la contratación de tres ingenieros que realicen jornadas de 40 horas semanales a razón de 3840€/mes durante 6 semanas para el desarrollo, la documentación, las pruebas y la integración en el lugar de despliegue. Finalmente se propone mantener un único ingeniero durante un año a razón de 1120€/mes para el mantenimiento de la aplicación.

Bibliografía

[1] Google, "Google C++ Style Guide." `styleguide/cppguide.html` [Disponible electrónicamente. Último acceso, julio de 2023]. 2, 3

[2] "Repositorio de LUCID." `TFG-pipeExec/tree/l-dev` [Disponible electrónicamente. Último acceso, julio de 2023]. 2, 3

[3] LaTeX3 Project, "LaTeX," 1985. `https://www.latex-project.org/` [Disponible electrónicamente. Último acceso, julio de 2023]. 2, 3

[4] "Doxygen: Generate documentation from source code." `https://www.doxygen.nl/`[Disponible electrónicamente. Último acceso, julio de 2023]. 2, 7

[5] Google, "GoogleTest." `google/googletest` [Disponible electrónicamente. Último Acceso, julio de 2023]. 2

[6] "CMake." `https://cmake.org/`[Disponible electrónicamente. Último acceso, julio de 2023]. 5

[7] Firefox, "GoogleTestFirefoxDocs." `gtest/index.html` [Disponible electrónicamente. Último acceso, julio de 2023]. 6

[8] "Multithreading." `https://en.wikipedia.org/wiki/Multithreading` [Disponible electrónicamente. Último acceso, julio de 2023]. 8

[9] "Multithreading in C++." `https://www.geeksforgeeks.org/multithreading-in-cpp/` [Disponible electrónicamente. Último acceso, julio de 2023]. 8

[10] F. J. Ceballos, *Programación orientada a objetos con C++*. Jarama, 3A, Polígono industrial Igarsa 28860 Paracuellos del Jarama, Madrid: RA-MA, 5 ed., 2018. 9

[11] "Modelo de Memoria de C++." `cpp/language/memory_model`[Disponible electrónicamente. Último Acceso, julio de 2023]. 9

[12] "Semáforo de la librería estándar de C++." `cpp/header/semaphore` [Disponible electrónicamente. Ultimo acceso, julio de 2023]. 11

[13] "Intercambio de punteros en C." `swapping-values-using-pointer-in-c` [Disponible electrónicamente. Último acceso, julio de 2023]. 22