Universidad Carlos III de Madrid
Escuela Politécnica Superior
Máster en Ciencia y Tecnología Informática

Trabajo Fin de Máster

Patrones de programación paralelos de alto nivel en arquitecturas de memoria distribuida

Autor: Javier Prieto Cepeda
Director: Javier Fernández Muñoz

Leganés, Madrid, España
27 de Septiembre 2018

"No hay nada más bonito, que poder dedicar tu tiempo a tu vocación. Lucha por tus sueños, y trabaja en lo que te haga vibrar."

Florin Isaila, UC3M.

Agradecimientos

Con este trabajo pongo fin a una etapa muy bonita de mi vida, en la cual no solamente me llevo todo lo que he aprendido en la universidad, sino también grandes experiencias y el placer de haber conocido a grandísimas personas.

En primer lugar quiero agradecer a mis padres, Tomás y Raquel, y a mi hermano, David, la paciencia que han tenido conmigo y los ánimos que me han dado. ¡Que sería de mi sin vosotros! También quiero agradecer a mis abuelos su preocupación y ánimos en estos años, ¡Por fin dejo de daros la brasa con mis agobios! A mis tías, mis tíos, y mis pequeñajos, que siempre me sacan una sonrisa.

Por otro lado, también quiero agradecer su paciencia, ánimos y los grandes momentos que me ha brindado a Miriam. Cariño, contigo empecé mi aventura en la universidad, y contigo la he finalizado. Gracias por todo, te quiero.

También me gustaría agradecer a mis tutores de este trabajo, Javier y Manuel su esfuerzo y dedicación en estos meses. Pese a que la universidad no me ha permitido añadir a Manuel (bendita burocracia...), unas palabras en un documento administrativo no quitan su trabajo y ayuda durante meses. Nuevamente, os doy las gracias Manu y Javi.

Por último, pero no por ello menos importante, me gustaría destacar a esas personas que han hecho que mi día a día en la universidad haya sido más llevadero y, por qué no, una alegría en muchas ocasiones. Mis compañeros en ARCOS, Carlos, Fran, Estefanía, Silvina, David, Javi y Alberto, gracias por tantas risas en el laboratorio (y fuera de él). También a Álex, ¡Cuanto me has enseñado, amigo!. A los chicos del lab (Óscar, Jaime y Roberto) que siempre te arrancan una sonrisa. Y a Saúl, Guille, Álvaro, Rubén y Mario; con quienes he compartido grandísimos momentos y desahogos. A todos vosotros, Gracias.

Patrones de programación paralelos de alto nivel en arquitecturas de memoria distribuida

Trabajo Fin de Máster

Javier Prieto Cepeda

Resumen

En los últimos años, los grandes volúmenes de flujo datos y los requisitos casi en tiempo real de las aplicaciones de transmisión de datos han incrementado la necesidad de nuevos algoritmos escalables e interfaces de programación para plataformas de memoria compartida y distribuida. Para contribuir en esta dirección, este trabajo presenta un nuevo back-end MPI distribuido para GrPPI, una interfaz genérica de alto nivel de C ++ de patrones paralelos de procesamiento intensivo de datos y streaming. Este back-end, como una nueva política de ejecución, admite la ejecución paralela distribuida e híbrida (distribución y memoria compartida) de los patrones de pipeline y farm, donde el modo híbrido combina la política MPI con una memoria compartida GrPPI. Un análisis detallado de la política de ejecución de GrPPI MPI muestra considerables beneficios desde los puntos de vista de programación, flexibilidad y legibilidad. La evaluación experimental en una aplicación de streaming con diferentes escenarios de memoria compartida y distribuida indica ganancias de rendimiento considerables con respecto a las versiones secuenciales a expensas de gastos indirectos insignificantes de GrPPI.

Palabras clave: Patrones de programación paralelos, Procesamiento de streaming, Patrones de programación distribuidos, Programación C++, Programación genérica

High-Level parallel programming patterns for distributed memory platforms

Master's Thesis

Javier Prieto Cepeda

Abstract

In the recent years, the large volumes of stream data and the near real-time requirements of data streaming applications have exacerbated the need for new scalable algorithms and programming interfaces for distributed and shared-memory platforms. To contribute in this direction, this work presents a new distributed MPI back end for GrPPI, a C ++ high-level generic interface of data-intensive and stream processing parallel patterns. This back end, as a new execution policy, supports the distributed and hybrid (distributed and shared-memory) parallel execution of the pipeline and farm patterns, where the hybrid mode combines the MPI policy with a GrPPI shared-memory one. A detailed analysis of the GrPPI MPI execution policy reports considerable benefits from the programmability, flexibility and readability points of view. The experimental evaluation on a streaming application with different distributed and shared-memory scenarios reports considerable performance gains with respect to the sequential versions at the expense of negligible GrPPI overheads.

Keywords: Parallel Patterns, Stream Processing, Distributed Patterns, C++ Programming, Generic Programming

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este primer capítulo presenta brevemente el proyecto, incluyendo sus características principales y motivación (Sección 1.1), los objetivos del proyecto (Sección 1.1), y toda la estructura del documento (Sección 1.3).

1.1 Motivación

Numerosos experimentos científicos, que van desde aceleradores de partículas hasta sensores ambientales, están generando en la actualidad grandes volúmenes de transmisión de datos que deben procesarse casi en tiempo real. Para habilitar la próxima generación de descubrimientos científicos, se han identificado varios desafíos en el área de la computación de alto rendimiento (HPC) para manejar las altas tasas de rendimiento y las demandas de baja latencia de las aplicaciones de procesamiento de flujo de datos (DaSP) [1]. Estos desafíos, previstos para llenar el vacío en la gestión y el procesamiento de la transmisión de datos, se centran en la búsqueda de algoritmos de transmisión escalables y eficientes, modelos de programación, lenguajes y sistemas en tiempo de ejecución para este tipo de aplicaciones. Para lograr estos objetivos, se requieren esfuerzos considerables en el ámbito del software de HPC para DaSP que permitan afrontar el crecimiento incesante de los datos de streaming [2].

Para lograr el objetivo de la escalabilidad, se requieren múltiples plataformas multi-core de memoria compartida para aumentar el rendimiento en este tipo de aplicaciones. En este sentido, los modelos de programación de facto para plataformas de memoria compartida y distribuida son, respectivamente, las interfaces MPI [3] y OpenMP [4], que se pueden usar en conjunto para permitir el paralelismo híbrido. Independientemente de su eficiencia, ambas interfaces ofrecen abstracciones de bajo nivel y exigen una experiencia considerable en los dominios de aplicación y sistema para ajustar la aplicación objetivo [5].

Una opción que permite reducir esta carga es la utilización de modelos de programación basados en patrones, que encapsulan aspectos algorítmicos siguiendo un enfoque de bloques de construcción. En

general, los patrones paralelos ofrecen una alternativa para implementar soluciones robustas, legibles y portátiles, ocultando las complejidades relacionadas con los mecanismos de concurrencia, sincronizaciones o intercambio de datos [6]. Es por ello, que algunas interfaces y bibliotecas basadas en patrones del estado de la cuestión, como por ejemplo FastFlow [7], Muesli [8] o SkePU [9] ya admiten clústeres de máquinas multi-core y utilizan algoritmos de programación para mejorar el balanceo de carga entre nodos.

1.2 Objetivos

Para facilitar el camino hacia los modelos de programación escalables de HPC para DaSP, en este trabajo ampliamos la interfaz genérica y reutilizable de patrones paralelos (GrPPI [10]) de C ++ con un nuevo back-end MPI, que permite la ejecución de algunos patrones de *streaming* en plataformas distribuidas.

Básicamente, GrPPI acomoda una capa unificada de patrones paralelos genéricos y reutilizables sobre los entornos de ejecución existentes y los frameworks basados en patrones. Esta capa permite a los usuarios hacer que sus aplicaciones sean independientes del framework de programación paralelo utilizado debajo, proporcionando así códigos portátiles y legibles.

Con esta primera versión del back-end de MPI, se admite la ejecución distribuida e híbrida de los patrones de *streaming Pipeline* y *Farm*. Para admitir escenarios híbridos, el back-end combina una política de ejecución de memoria compartida intra-nodo que, si es necesario, se utiliza para ejecutar múltiples operadores de *Pipeline* o *Farm* dentro de un proceso MPI usando OpenMP, C++ Threads o Intel TBB, teniendo como objetivos en este trabajo:

- **O1**: Presentar una nueva política de ejecución de GrPPI-MPI para entornos distribuidos e híbridos para los patrones paralelos *Pipeline* y *Farm*.
- **O2**: Describir el diseño de la interfaz GrPPI y las políticas internas de MPI para permitir la ejecución distribuida e híbrida de aplicaciones DaSP.
- **O3**: Presentar un nuevo operador para patrones de *streaming* que, como un contenedor, permite a los usuarios reemplazar la política de ejecución predeterminada de un patrón.
- **O4**: Analizar la usabilidad del patrón en términos de líneas de código y complejidad ciclomática, y realizando una comparación *side-by-side* de ambas interfaces de programación GrPPI y MPI.
- **O5**: Evaluar los patrones distribuidos *Pipeline* y *Farm* de *streaming* desde los puntos de vista de usabilidad y rendimiento usando una aplicación que renderiza frames de Mandelbrot. Esta evaluación se llevará a cabo bajo diferentes configuraciones híbridas.

1.3 Estructura del documento

El documento está dividido en los siguientes capítulos:

- Capítulo 1, *Introducción*, presenta una breve descripción del contenido del documento. También incluye la motivación y los objetivos del proyecto.
- Capítulo 2, *Estado de la cuestión*, presenta el trabajo relacionado, incluyendo una descripción de los diferentes paradigmas de programación paralela y sus frameworks.
- Capítulo 3, *Patrones de paralelismo*, introduce los patrones de paralelismo, describiéndolos y realizando su formalización matemática; y presenta la interfaz genérica y reutilizable de patrones paralelos (GrPPI).
- Capítulo 4, *Interfaz de patrones de programación paralela para memoria distribuida*, presenta la nueva GrPPI-MPI para entornos distribuidos e híbridos para los patrones paralelos *Pipeline* y *Farm*, explicando la nueva política de ejecución.
- Capítulo 5, *Evaluación experimental*, presenta el análisis y evaluación de la usabilidad y rendimiento de los patrones en GrPPI-MPI.
- Capítulo 6, *Planificación del proyecto*, presenta los conceptos relacionados con la planificación del proyecto.
- Capítulo 7, *Conclusiones y trabajo futuro*, incluye las contribuciones y los resultados del proyecto, explicando los principales hallazgos del proyecto y presentando el trabajo futuro.

Capítulo 2

Estado de la cuestión

Este capítulo presenta el estado de la cuestión, la última y más avanzada etapa de las tecnologías relacionadas con nuestra solución. Primero, discutimos los principales paradigmas de programación paralela (Sección 2.1). Después, presentamos los principales frameworks de programación paralela disponibles (Sección 2.2). Finalmente, se exponen los frameworks basados en patrones paralelos (Sección 2.3).

2.1 Paradigmas de programación paralela

En los últimos 50 años, se han realizado grandes avances en el rendimiento y la capacidad de un sistema informático. Estos avances han sido posibles gracias a la ayuda de la tecnología VLSI. La tecnología VLSI permite que una gran cantidad de componentes se alojen en un solo chip y aumenten las velocidades de reloj. Por lo tanto, se pueden realizar más operaciones a la vez, en paralelo.

El procesamiento en paralelo también está asociado con la ubicación de datos y la comunicación de datos. Las arquiteturas paralelelas son el método de organizar todos los recursos para maximizar el rendimiento y la programabilidad dentro de los límites dados por la tecnología y el coste en cualquier instancia de tiempo. Estas arquitecturas agregan una nueva dimensión en el desarrollo del sistema informático al usar cada vez más procesadores. En principio, el rendimiento logrado al utilizar una gran cantidad de procesadores es más alto que el rendimiento de un solo procesador en un momento determinado. Con el avance de la capacidad del hardware, también aumentó la demanda de prestaciones, siendo necesario el desarrollo de la arquitectura de la computadora.

Antes de la era de los microprocesadores, se obtenía un sistema informático de alto rendimiento mediante tecnología de circuitos específicos y organización de máquinas, lo que resultaba muy costoso. Actualmente, el sistema informático de alto rendimiento se obtiene mediante el uso de múltiples procesadores, y las aplicaciones más importantes y exigentes se escriben como programas paralelos.

Por lo tanto, para un mayor rendimiento, se necesitan desarrollar arquitecturas paralelas y aplicaciones paralelas.

Las máquinas paralelas se han desarrollado con varias arquitecturas distintas. La arquitectura paralela mejora los conceptos convencionales de la arquitectura de la computadora con la arquitectura de comunicación. La arquitectura de la computadora define las abstracciones críticas (como el límite del sistema del usuario y el límite del software/hardware) y la estructura de la organización, mientras que la arquitectura de comunicación define las operaciones básicas de comunicación y sincronización. También aborda la estructura organizacional. El modelo de programación es la capa superior. Las aplicaciones están escritas en el modelo de programación. Los modelos de programación paralela [11], [12] incluyen memoria compartida, paso de mensajes y paralelismo de datos, los cuales son descritos a continuación.

2.1.1 Memoria compartida

Los multiprocesadores de memoria compartida son una de las clases más importantes de máquinas paralelas. Proporciona un mejor rendimiento en cargas de trabajo de multiprogramación y admite programas paralelos. En este caso, todos los sistemas informáticos permiten que un procesador y un conjunto de controladores de E/S accedan a una colección de módulos de memoria mediante alguna interconexión de hardware. La capacidad de la memoria aumenta al agregar módulos de memoria y la capacidad de E/S aumenta al agregar dispositivos al controlador de E/S o al agregar un controlador de E/S adicional. La capacidad de procesamiento se puede aumentar esperando a que esté disponible un procesador más rápido o agregando más procesadores.

Todos los recursos están organizados alrededor de un bus de memoria central. A través del mecanismo de acceso al bus, cualquier procesador puede acceder a cualquier dirección física en el sistema. Como todos los procesadores son equidistantes de todas las ubicaciones de memoria, el tiempo de acceso o la latencia de todos los procesadores es el mismo en una ubicación de memoria. Esto se llama multiprocesador simétrico [13].

Los dos modelos de multiprocesadores de memoria compartida más relevantes son:

- ***Acceso a memoria uniforme (UMA)***: En este modelo, todos los procesadores comparten la memoria física de manera uniforme [14]. Todos los procesadores tienen igual tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener una memoria caché privada. La misma regla se sigue para los dispositivos periféricos. Cuando todos los procesadores tienen el mismo acceso a todos los dispositivos periféricos, el sistema se denomina multiprocesador simétrico. Cuando solo uno o unos pocos procesadores pueden acceder a los dispositivos periféricos, el sistema se denomina multiprocesador asimétrico.

- ***Acceso a memoria no uniforme (NUMA)***: En el modelo de multiprocesador NUMA [15], el tiempo

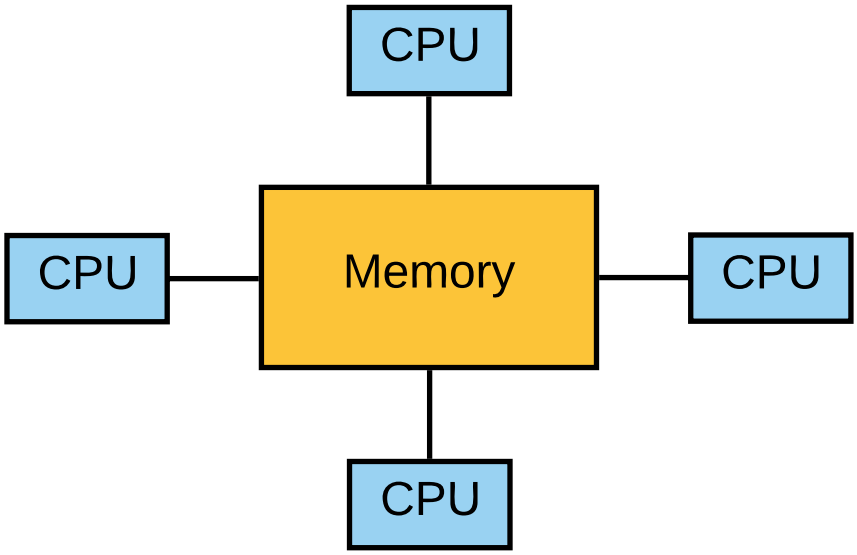


Figura 2-1: *Esquema de arquitectura UMA.*

de acceso varía según la ubicación de la palabra de memoria. Aquí, la memoria compartida se distribuye físicamente entre todos los procesadores, llamadas memorias locales. La colección de todas las memorias locales forma un espacio de direcciones global al que pueden acceder todos los procesadores.

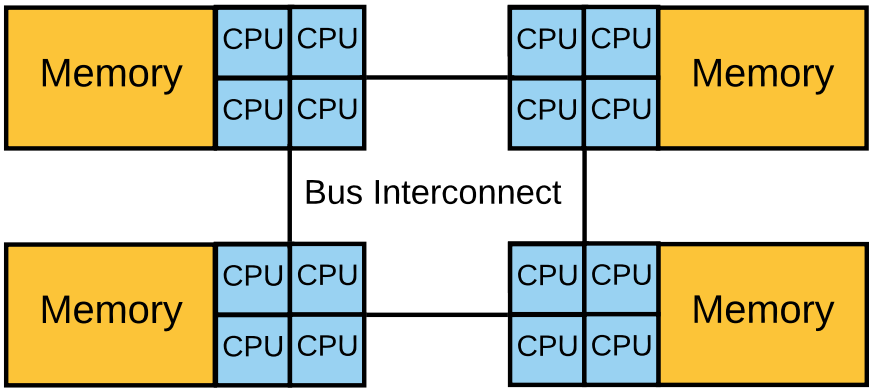


Figura 2-2: *Esquema de arquitectura NUMA.*

El modelo de programación de memoria compartida depende de los procesadores multi-core de memoria compartida. Estos modelos de programación se comunican al compartir los datos en el espacio de direcciones global. Suponen que todos los procesos paralelos pueden acceder a todo el conjunto de memoria. Por ello, es importante lograr consistencia en los datos [16], ya que diferentes procesadores pueden comunicarse compartiendo el mismo el mismo dato. Para resolver este problema, se utilizan los protocolos de coherencia de caché [17]. La comunicación entre procesos paralelos debe realizarse

haciendo uso de diferentes primitivas de sincronización de procesos como los bloqueos o la memoria transaccional [18]. Estos modelos de programación, proporcionan una serie de ventajas y desventajas que son expuestas a continuación:

Ventajas
✓Facilitan el desarrollo sencillo de la aplicación en comparación con los sistemas basados en memoria distribuida. ✓Evitan la multiplicidad de los datos, adoptando esa responsabilidad el modelo y quitándosela al programador. ✓Ofrecen un mejor rendimiento que los modelos de programación paralela basados en memoria distribuida.
Desventajas
✗Los requisitos de hardware para estos modelos son muy altos y complejos, de forma que incrementa los costes. ✗En ocasiones se encuentran carreras de datos y *deadlocks* durante la ejecución de las aplicaciones.

Tabla 2.1: *Ventajas y desventajas de las arquitecturas basadas en memoria compartida.*

En la actualidad, existen diferentes modelos de programación paralelos basados en memoria compartida. El modelo de hilos [19] está basado en la biblioteca de hilos que proporciona rutinas de bajo nivel para la paralelización de las aplicaciones. Estos modelos utilizan bloqueos de exclusión mutua y variables condicionales para establecer las comunicaciones y sincronizaciones entre los diferentes hilos. Las ventajas y desventajas que proporciona este modelo son las siguientes:

Ventajas
✓Es el más adecuado para aplicaciones basadas en la multiplicidad de datos. ✓Proporciona una alta flexibilidad al programador. ✓Las bibliotecas de hilos son las más utilizadas, por lo que resulta muy fácil encontrar herramientas para este tipo de desarrollo. ✓El rendimiento de la aplicación puede ser mejorado mediante el uso de esperas condicionales y esperas activas. ✓Resulta fácil desarrollar rutinas paralelas para modelos de threads.
Desventajas
✗Resulta complicado escribir aplicaciones que utilicen modelos de threads, ya que establecer una comunicación o sincronización implica una sobrecarga de código que es difícil de gestionar y, por lo tanto, tiene tendencia a causar más errores. ✗El programador debe ser más cuidadoso al usar datos globales, debido a que esto conduce a carreras de datos, interbloqueos y uso compartido falso. ✗Los modelos de hilos tienen un bajo nivel de abstracción, que resulta necesario para un mejor modelo de programación.

Tabla 2.2: *Ventajas y desventajas del modelo de programación basado en threads.*

Otro de estos modelos, es el modelo basado en directivas [20]. Este modelo utiliza las directivas de compilación de alto nivel para paralelizar las aplicaciones, siendo una extensión de los modelos basados en hilos. Los modelos basados en directivas se ocupan de las características de bajo nivel como la partición, la gestión de los hilos trabajadores, la sincronización y la comunicación entre los hilos. Las ventajas y desventajas que proporciona este modelo son las siguiente:

Por último, el modelo de tareas está basado en el concepto de especificar tareas [21] en lugar de hilos como lo hacen otros modelos. Esto se debe a que las tareas son de corto alcance y más ligeras que los hilos. En general, las tareas son 18 veces más rápidas que los hilos en las implementaciones

Ventajas
✓Estos modelos surgieron como un estándar.
✓Es fácil escribir aplicaciones paralelas haciendo uso de ellos.
✓Se incurre en menos gastos generales de código y es fácil gestionar el código desarrollado mediante directivas.
✓Este modelo se encuentra en un bajo nivel de abstracción.
✓El programador no necesita considerar problemas como condiciones de carrera e interbloqueos.
Desventajas
✗Estos modelos son menos utilizados.
✗Proporciona una baja flexibilidad al programador.
✗El apoyo de las herramientas al desarrollo es bastante bajo.

Tabla 2.3: *Ventajas y desventajas del modelo de programación basado en directivas.*

de UNIX y 100 veces más rápidas que los hilos en las implementaciones basadas en Windows [22], [23]. Una diferencia entre tareas e hilos es que las tareas siempre se implementan en el espacio de usuario [24]. Además, las tareas son paralelas pero no concurrentes, por lo que no tienen prioridad y pueden ejecutarse de forma secuencial. Las ventajas y desventajas que proporcionan estos modelos son las siguientes:

Ventajas
✓Las tareas disminuyen la sobrecarga asociada con las comunicaciones tal como se observa en otros modelos.
✓Facilita el desarrollo de las aplicaciones paralelas.
✓Existe una gran variedad de herramientas relacionadas con los modelos basados en tareas, facilitando la depuración de errores.
Desventajas
✗Estos modelos no figuran como estándar.
✗Tienen una curva de aprendizaje grande.
✗La flexibilidad que proporcionan estos modelos es baja.

Tabla 2.4: *Ventajas y desventajas del modelo de programación basado en tareas.*

2.1.2 Memoria distribuida

La arquitectura de memoria distribuida [25] también es una clase importante de máquinas paralelas. Proporciona comunicación entre procesadores como operaciones explícitas de E/S. En este caso, la comunicación se combina en el nivel de E/S, en lugar del sistema de memoria.

En esta arquitectura, la comunicación del usuario se ejecuta mediante el uso de llamadas al sistema operativo o a la biblioteca que realizan muchas acciones de nivel inferior, que incluyen la operación de comunicación real. Como resultado, hay una distancia entre el modelo de programación y las operaciones de comunicación en el nivel de hardware físico.

Enviar y *Recibir* son las operaciones de comunicación de nivel de usuario más comunes en este sistema. Enviar especifica un búfer de datos local (que se va a transmitir) y un procesador receptor remoto.

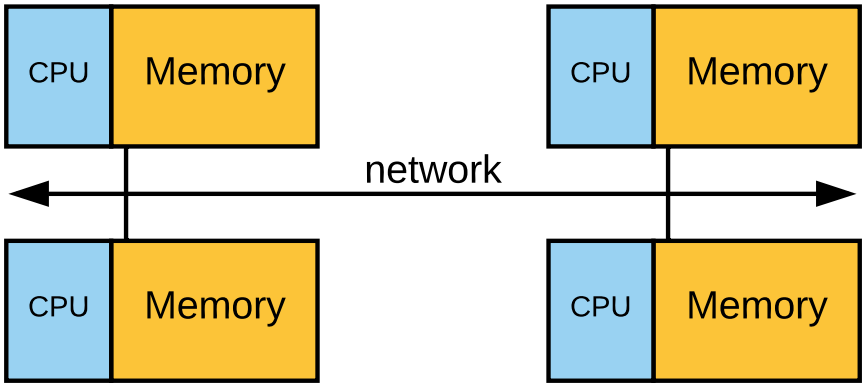


Figura 2-3: *Esquema de arquitectura de memoria distribuida.*

Recibir especifica un proceso de envío y un búfer de datos local en el que se colocarán los datos transmitidos. En la operación de envío, se adjunta un identificador o una etiqueta al mensaje y la operación de recepción especifica la regla de coincidencia, como una etiqueta específica de un procesador específico o cualquier etiqueta de cualquier procesador.

La combinación de un envío y una recepción coincidente completa una copia de memoria a memoria. Cada extremo especifica su dirección de datos local y un evento de sincronización por pares.

El modelo más conocido en este tipo de arquitecturas es la arquitectura de memoria solo caché (COMA) [26]. El modelo COMA es un caso especial del modelo NUMA. Aquí, todas las memorias principales distribuidas se convierten en memorias de caché. Un sistema de memoria distribuida consta de varias computadoras, conocidas como nodos, interconectadas por la red que pasa mensajes. Cada nodo actúa como una computadora autónoma que tiene un procesador, una memoria local y, a veces, dispositivos de E/S. En este caso, todas las memorias locales son privadas y solo son accesibles para los procesadores locales. Esta es la razón por la cual las máquinas tradicionales se llaman máquinas sin acceso a memoria remota (NORMA).

Este tipo de modelos de programación paralela a menudo se conocen como modelos de paso de mensajes, que permiten la comunicación entre nodos mediante el uso de las rutinas de comunicación de envío/recepción. Los modelos de paso de mensajes evitan las comunicaciones entre procesadores basadas en datos compartidos/globales [24]. Normalmente, este modelo se utiliza para programar aplicaciones para clusters, donde cada nodo en la arquitectura obtiene su propia instancia de datos e instrucciones. Las ventajas y desventajas de los modelos de programación basados en memoria distribuida son:

2.1.3 Esquemas híbridos

En la actualidad y con motivo de la cada vez mayor necesidad de aumentar el rendimiento de las computadoras, las computadoras más grandes y rápidas en el mundo emplean arquitecturas de memo-

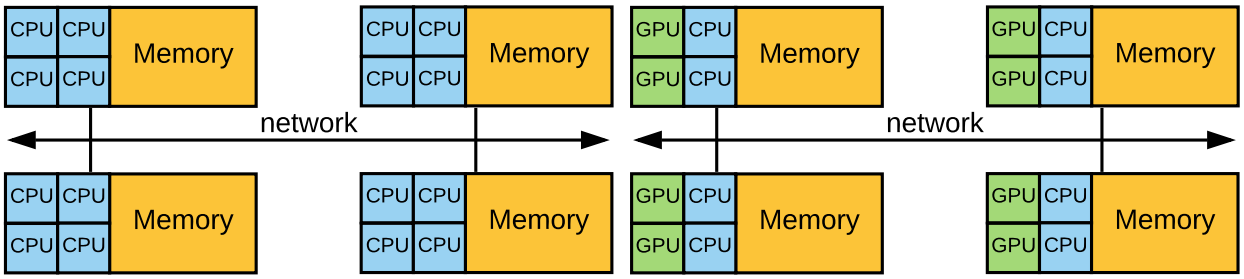
Ventajas
✓El requisito de hardware para estos modelos es bajo, menos complejo y tiene un coste pequeño. ✓Evitan las condiciones de carrera y, como consecuencia, el programador queda liberado de usar los bloqueos.
Desventajas
✗Encuentran interbloqueos durante el proceso de comunicaciones. ✗Su desarrollo es difícil y lleva más tiempo. ✗El desarrollador es responsable de establecer la comunicación entre los nodos. ✗Están menos orientados al rendimiento y tienen un alto coste de comunicación.

Tabla 2.5: *Ventajas y desventajas de las arquitecturas basadas en memoria distribuida.*

ria compartida y distribuida [27]. Estas máquinas se basan en la unión de los conceptos descritos en las secciones anteriores. El componente de memoria compartida puede ser una máquina de memoria compartida y/o unidades de procesamiento de gráficos (GPU). El componente de memoria distribuida es la conexión en red de varias máquinas de memoria compartida/GPU, que solo conocen su propia memoria, es decir, no conocen la memoria de otra máquina. Por lo tanto, se requieren comunicaciones de red para mover datos de una máquina a otra. Entre las ventajas y desventajas que presentan estos sistemas:

Ventajas
✓Tiene todas las ventajas propias tanto de un sistema de memoria compartida como de un sistema de memoria distribuida. ✓El aumento de la escalabilidad es una característica muy importante de estos sistemas.
Desventajas
✗El aumento de la complejidad del programador es una desventaja muy importante.

Tabla 2.6: *Ventajas y desventajas de las arquitecturas basadas en esquemas híbridos.*



(a) *Arquitectura de esquemas híbridos sin GPU.* (b) *Arquitectura de esquemas híbridos con GPU.*

Figura 2-4: *Arquitectura esquemas híbridos.*

Las tendencias actuales parecen indicar que este tipo de arquitectura de memoria continuará prevaleciendo e irá en aumento en los próximos años, puesto que gracias a la flexibilidad y escalabilidad que proporciona, permite modificar el sistema con relativa facilidad. A medida que los desarrolladores

continúan buscando rendimiento, se les alienta a explorar técnicas de programación híbrida [28], que ofrecen la oportunidad de mejorar el consumo de recursos, especialmente la memoria.

2.2 Frameworks de programación paralelos

Para poder explotar correctamente el funcionamiento de la computadora, los programadores hacen uso de interfaces de programación. Las interfaces de programación suponen que las órdenes de programa no tienen que mantenerse en absoluto entre las operaciones de sincronización. Se garantiza que todas las operaciones de sincronización estén explícitamente etiquetadas o identificadas como tales. La biblioteca de tiempo de ejecución o el compilador traduce estas operaciones de sincronización en las operaciones de preservación de orden adecuadas requeridas por la especificación del sistema.

Por lo tanto, el sistema asegura ejecuciones secuenciales consistentes aunque puede reordenar las operaciones entre las operaciones de sincronización de cualquier forma que desee sin interrumpir las dependencias a una ubicación dentro de un proceso. Esto le permite al compilador suficiente flexibilidad entre los puntos de sincronización para los reordenamientos que desea, y también le otorga al procesador realizar tantos reordenamientos como lo permita su modelo de memoria. En la interfaz del programador, el modelo de coherencia debe ser al menos tan débil como el de la interfaz de hardware, pero no tiene que ser el mismo. A continuación, se exponen los frameworks de programación paralela más relevantes para cada una de las tres arquitecturas descritas en la sección anterior.

2.2.1 Memoria compartida

En las arquitecturas multiprocesador de memoria compartida, los hilos se pueden usar para implementar el paralelismo. Históricamente, los proveedores de hardware han implementado sus propias versiones propietarias de hilos, lo que hace que la portabilidad sea una preocupación para los desarrolladores de software. Para los sistemas UNIX, el estándar IEEE POSIX 1003.1c ha especificado una interfaz de programación de hilos de lenguaje C estandarizada. Las implementaciones que se adhieren a este estándar se denominan POSIX threads o Pthreads. Pthreads [29] es un modelo de ejecución que existe independientemente del lenguaje, así como un modelo de ejecución paralelo. Permite que un programa controle múltiples flujos de trabajo diferentes superpuestos en el tiempo. Cada flujo de trabajo recibe el nombre de hilo o thread y la creación y el control de estos flujos se realiza mediante llamadas a la API de subprocesos POSIX. Las implementaciones de la API están disponibles en muchos sistemas operativos conformes a POSIX tipo Unix, como FreeBSD , NetBSD , OpenBSD , Linux , Mac OS X , Android y Solaris , generalmente empaquetados como biblioteca libpthread . También existen implementaciones de DR-DOS y Microsoft Windows: dentro del subsistema SFU/SUA que proporciona una implementación nativa de varias API POSIX, y también dentro de paquetes de terceros como pthreads-w32, que implementa Pthreads sobre la API de Windows existente.

OpenMP [**?**], es un API para arquitecturas de memoria compartida que proporciona una capacidad multiproceso. Un bucle se puede paralelizar fácilmente invocando llamadas de subrutina desde librerías de hilos OpenMP e insertando las directivas del compilador OpenMP. De esta forma, los hilos pueden obtener nuevas tareas y las iteraciones de bucle no procesadas directamente desde la memoria compartida local. OpenMP es una especificación abierta para el paralelismo de memoria compartida. La idea básica detrás de OpenMP es la ejecución paralela de datos compartidos. Consiste en un conjunto de directivas de compilación, rutinas de biblioteca de tiempo de ejecución y variables de entorno que amplían los programas en los lenguajes de programación FORTRAN, C y C ++. OpenMP es portable en la arquitectura de memoria compartida. La unidad de trabajadores en OpenMP son los hilos. Cada hilo puede acceder a una variable en memoria caché o memoria RAM. OpenMP incluye suporte para las plataformas UNIX y Microsoft Windows.

Cilk [30], Cilk ++ y Cilk Plus [31] son lenguajes de programación de propósito general diseñados para computación paralela multiproceso . Están basados en los lenguajes de programación C y C++ , que se extienden con construcciones para expresar bucles paralelos y la expresión fork-join. Originalmente desarrollado en la década de 1990 en el Instituto de Tecnología de Massachusetts (MIT), Cilk fue posteriormente comercializado como Cilk++, aumentando su compatibilidad con el código C y C++ tras su compra por parte de Intel. El resultado de este aumento de compatibilidad dió lugar a su nuevo nombre, Cilk Plus.

Entre las abstracciones de hilos más recientes se incluyen Intel Threading Building Blocks (Intel TBB)[22]. Aunque OpenMP funciona tanto con C/C ++ como con Fortran, Intel TBB se basa en plantillas C ++, un modelo de programación genérico, por lo que está limitado a C ++. Debido a que Intel TBB contiene un amplio conjunto de características, a la vez que proporciona una abstracción clara que es fácil de seguir, se ha vuelto bastante popular entre los programadores de C ++.

El paralelismo anidado es natural tanto para Cilk Plus como para Intel TBB, en comparación con OpenMP, donde el desarrollador debe identificar y declarar el paralelismo anidado. Esto hace que Intel TBB y Cilk Plus sean ideales para incluir en las bibliotecas.

2.2.2 Memoria distribuida

Muchas aplicaciones buscaban una mayor potencia de cálculo de la que estaba disponible en un solo sistema multiprocesador. Esto condujo a la conexión de varios sistemas para formar grupos de computadoras para trabajar juntas y resolver una sola carga de trabajo computacional. Estos sistemas vincularon frecuentemente los sistemas junto con un frabricante propietario, con cada plataforma en el clúster que tiene su propia área de memoria privada dentro del clúster. El programa tenía que definir explícitamente los datos que se compartirían con otra plataforma y entregarlos a la otra plataforma en el clúster.

Con el enfoque de computación distribuida, se escribieron programas explícitos de paso de mensajes.

En este enfoque, un programa empaquetó datos de forma explícita y lo envió a otro sistema en el clúster. El otro sistema tuvo que solicitar explícitamente los datos y llevar los datos a su proceso. Se desarrolló un enfoque que vincula estaciones de trabajo llamadas máquinas virtuales paralelas, lo que permite que los programas funcionen en una red de estaciones de trabajo. Cada proveedor con una estructura patentada admite su propia biblioteca de transmisión de mensajes.

La comunidad se consolidó rápidamente alrededor de la Interfaz de paso de mensajes (MPI) [32]. MPI es una especificación para las operaciones de paso de mensajes. Define a cada trabajador como un proceso. MPI es actualmente el estándar de facto para desarrollar aplicaciones HPC en arquitecturas de memoria distribuida. Proporciona enlaces de lenguaje para C, C++ y FORTRAN. MPI ofrece portabilidad, estandarización, rendimiento y funcionalidad, e incluye el paso de mensajes de punto a punto y operaciones colectivas (globales), todo en un ámbito para grupos de procesos especificados por el usuario. MPI proporciona un conjunto sustancial de bibliotecas para la escritura, la depuración y los programas distribuidos de prueba de rendimiento.

La ventaja para el usuario es que MPI está estandarizado en muchos niveles. Por ejemplo, dado que la sintaxis está estandarizada, se asegura que el código MPI se ejecutará bajo cualquier implementación de MPI que se ejecute en la computadora. Dado que el comportamiento funcional de las llamadas MPI también está estandarizado, las llamadas MPI deben comportarse de la misma manera independientemente de la implementación, garantizando así la portabilidad de las aplicaciones.

La biblioteca MPI se usa a menudo para programación paralela en sistemas de clúster porque es un lenguaje de programación que de paso mensajes. Sin embargo, MPI no es el lenguaje de programación más apropiado para computadoras multicore porque aún cuando todavía hay muchas tareas asignadas a procesos esclavos sobrecargados que permanecen en la memoria compartida, otros procesos MPI esclavos en el mismo nodo de computación no pueden acceder a las tareas. En cambio, todos los procesos esclavos deben comunicarse directamente con el proceso maestro MPI para obtener nuevas tareas. En los grandes sistemas de clúster, el proceso maestro puede convertirse en un cuello de botella en el rendimiento del sistema debido a la excesiva cantidad de comunicación. Los cálculos del clúster explotan el paso de mensajes, porque las computadoras en el clúster tienen memoria distribuida. Cuando un proceso necesita datos de otro, debe administrar los datos que pasan a través de la red.

Con el incremento en los últimos años de la computación distribuída causado por el impulso de la computación Big-Data, han surgido nuevos frameworks para la computación distribuida. Apache Spark [33] es un framework de computación en clúster open-source, desarrollado inicialmente en la Universidad de California. Es un sistema de computación en clúster de propósito general y orientado a la velocidad que proporciona APIs para el desarrollo en Java, Scala, Python y R.

Apache Storm[**?**] es un sistema computacional distribuido en tiempo real de código abierto para el procesamiento de flujos de datos. De forma similar a lo que Hadoop [34] solía hacer para el procesamiento por lotes, Apache Storm lo hace para flujos ilimitados de datos de manera confiable, siendo

capaz de procesar por cada nodo más de un millón de tareas por segundo. Con los datos masivos que se producen cada segundo, la necesidad de procesarlos en tiempo real ha crecido enormemente. Sin embargo, las compañías aún desean seguir el sistema tradicional de procesamiento por lotes y los flujos de trabajo interactivos. Apache Storm cumple estos requisitos junto con las herramientas y tecnologías adicionales que se usan comúnmente con Hadoop. Su escalabilidad, mayor velocidad y confiabilidad lo han convertido en una opción preferida entre los analistas de datos.

Otro ejemplo es StreamIt [35], un lenguaje que permite aplicaciones streaming de gran tamaño de alto rendimiento al mapear eficientemente los procesos a una amplia gama de entornos, incluidas las arquitecturas de memoria compartida y los clústeres HPC. Otro ejemplo es MPIStream, una biblioteca de prototipos implementada encima de MPI, que proporciona una interfaz a las aplicaciones MPI existentes para adoptar el modelo de transmisión [36, 37]. Básicamente, MPIStream proporciona un enfoque ligero para vincular los procesos de MPI con diferentes tareas mediante el uso de cuatro conceptos básicos de procesamiento de flujo: canales de comunicación, productor/consumidor de datos, flujos de datos y operaciones de flujo.

2.2.3 Esquemas híbridos

En la actualidad, la mayoría de los sistemas de computación de alto rendimiento (HPC) presentan un diseño de hardware jerárquico: los nodos de memoria compartida con varias CPU multinúcleo se conectan a través de una infraestructura de red. La programación en paralelo debe combinar la paralelización de la memoria distribuida en la interconexión del nodo con la paralelización de la memoria compartida dentro de cada nodo. En consecuencia, parece natural emplear un modelo de programación híbrido que utilice OpenMP para paralelización dentro del nodo y MPI para el paso de mensajes entre nodos.

En [38] proponen un framework de desarrollo de programas híbridos MPI+OpenMP en donde se genera de forma automática el código final. En primer lugar, mediante un analizador se inspecciona de forma automática el código desarrollado por el usuario, estimando el tiempo de computación de una función secuencial. Posteriormente, una herramienta se encarga de generar de forma automática el código híbrid basándose en los resultados obtenidos previamente por el analizador y en un modelo de predicción simple de paralelismo que estima el tiempo de ejecución del programa híbrido resultante.

Uintah [39] se desarrolló para proporcionar un entorno de resolución de problemas de interacción de fluídos. Este framework utiliza el paradigma basado en tareas y abstrae completamente al usuario del paralelismo. Debido a un problema de escalabilidad que surgió con el aumento en el número de cores de las computadoras, los desarrolladores de este framework dicidieron incorporar la ejecución híbrida haciendo de MPI+Pthreads.

KNEM [40] es un módulo desarrollado para el kernel de Linux que proporciona implementaciones de MPI con una interfaz flexible y escalable para realizar transferencias de datos de una sola copia asistidas por el kernel entre procesos locales. Permite la comunicación de alto rendimiento dentro de

la mayoría de implementaciones de MPI existentes y brinda importantes mejoras en el rendimiento de las aplicaciones gracias a operaciones más eficientes de punto a punto y colectivas, ya que gracias a su implementación híbrida permite la comunicación intranodo mediante threads y la creación automática de buffers internos.

Por otro lado, recientemente se están comenzando a utilizar esquemas híbridos donde se mezcla MPI con la ejecución de kernels GPU. FLAT [41] es framework de programación híbrida GPU-MPI que permite a los programadores utilizar funciones MPI dentro de los kernel de GPU, transformándolas en tiempo de ejecución en rutinas ejecutadas en la CPU.

ASUCA [42] es un framework de desarrollo utilizado en aplicaciones de predicción meteorológica. Estas aplicaciones, requieren de un gran rendimiento computacional para lograr simulaciones rápidas y de alta resolución. Gracias a este framework se permite la ejecución GPU distribuida ya que hace uso de MPI y kernels de GPU, abstrayendo al usuario de la implementación más costosa (paralelismo) y de las optimizaciones requeridas para GPU distribuidas. Los códigos escritos por el usuario son paralelizados por MPI con acceso directo punto a punto de la GPU intra-nodo.

2.3 Frameworks basados en patrones paralelos

GrPPI [10] es una interfaz de programación de patrones paralelos genérica y reutilizable de código abierto desarrollada en la Universidad Carlos III de Madrid. Básicamente, GrPPI acomoda una capa entre los desarrolladores y los marcos de programación paralelos existentes dirigidos a los procesadores multi-core, como ISO C ++ Threads, OpenMP, Intel TBB y FastFlow. Para lograr este objetivo, la interfaz aprovecha las características modernas de C ++, los conceptos de metaprogramación y la programación genérica para actuar como un cambio entre esos marcos.

Además, su diseño compacto facilita el desarrollo de aplicaciones paralelas, ocultando la complejidad detrás del uso de mecanismos de concurrencia. Los patrones paralelos admitidos por GrPPI están destinados tanto al procesamiento de flujo como a las aplicaciones de uso intensivo de datos, y se pueden componer entre ellos para que coincidan con construcciones más complejas. En pocas palabras, GrPPI aboga por una interfaz de patrones paralelos utilizable, simple, genérica y de alto nivel, lo que permite a los usuarios implementar aplicaciones paralelas sin tener una comprensión profunda de los marcos de programación paralelos actuales y las interfaces de terceros.

FastFlow [7, 43] es un framework de programación paralela de C ++ que aboga por la programación paralela basada en patrones de alto nivel. Principalmente es compatible con streaming y el paralelismo de datos, y se dirige a plataformas heterogéneas compuestas por clusters de plataformas de memoria compartida, posiblemente equipadas con aceleradores como NVidia GPGPU, Xeon Phi y Tilera TILE64.

Al igual que otros marcos de programación de alto nivel, como Intel TBB y OpenMP, simplifica el diseño y la ingeniería de aplicaciones paralelas portátiles. Sin embargo, tiene una ventaja clara en

términos de expresividad y rendimiento con respecto a otros marcos de programación paralelos en escenarios de aplicación específicos, que incluyen, entre otros:

- Paralelismo de grano fino en plataformas de memoria compartida coherentes a la caché.
- Aplicaciones de streaming.
- Uso acoplado de múltiples núcleos y aceleradores.

En otros casos, FastFlow es típicamente comparable a (y es en algunos casos ligeramente más rápido que) frameworks de programación paralelos de última generación como Intel TBB, OpenMP, Cilk, etc.

SkePU [9] es un enfoque en el cual se escribe una aplicación con la ayuda de esqueletos. Un esqueleto es un componente genérico predefinido como *map, reduce, scan, Farm, Pipeline*, etc. que implementa un patrón específico común de cómputo y dependencia de datos, y que se puede personalizar con parámetros de código definidos (definidos por el usuario). Los esqueletos proporcionan un alto grado de abstracción y portabilidad con una interfaz de programación casi secuencial, ya que sus implementaciones encapsulan todos los detalles de bajo nivel y específicos de la plataforma, como paralelización, sincronización, comunicación, administración de memoria, uso de aceleradores y otras optimizaciones.

SkePU es un framework de programación de código abierto para CPU multinúcleo y sistemas multi-GPU. Es una biblioteca de plantillas C ++ con seis esqueletos paralelos a datos y uno paralelo a tareas, dos tipos genéricos de contenedores y soporte para ejecución en sistemas multi-GPU con CUDA [44] y OpenCL [45].

Eden [46], es una extensión que proporciona patrones en Haskell que da soporte para entornos paralelos y distribuidos. En esta biblioteca, los procesos se comunican a través de canales unidireccionales definidos por los programadores al tiempo que especifican dependencias de datos. De forma similar, JaSkel [47] proporciona versiones secuenciales, concurrentes y distribuidas de los esqueletos de Pipeline y Farm, siendo posible implementarlos en las infraestructuras de clúster y grid.

Alternativamente, también encontramos frameworks de programación de patrones paralelos que hacen uso de MPI para focalizar plataformas distribuidas. Un ejemplo es SkeTo [48], una biblioteca C ++ acoplada con MPI que ofrece operaciones para estructuras de datos paralelos, como listas, árboles y matrices, que sin embargo, carece de patrones orientados a streaming. De forma similar, la biblioteca de esqueletos de Muesli [49] ofrece una gran colección de patrones a través de métodos C ++ implementados en OpenMP y MPI, para plataformas multi-core y cluster, respectivamente. Los principales patrones admitidos son matrices y matrices distribuidas para el paralelismo de datos; y Pipeline y Farm para el paralelismo orientado a streaming. Otra contribución es MALLBA [50], una biblioteca que proporciona una colección de esqueletos de alto nivel para la optimización combinatoria que trata el paralelismo de una manera fácil de usar y eficiente. MALLBA aprovecha NetStream, una capa de abstracción MPI personalizada que se encarga de la clasificación y sincronización primarias del tipo de datos entre los

procesos que se ejecutan en máquinas distribuidas. Finalmente, destacamos DASH [51], una biblioteca de plantillas C ++ que ofrece estructuras de datos distribuidas y algoritmos de biblioteca de plantillas estándar (STL) en paralelo a través de un enfoque de espacio de direcciones globales particionado (PGAS) sin compilador.

Capítulo 3

Patrones de paralelismo

Este capítulo presenta los patrones de paralelismo, qué son y cuales son sus principales casos de uso. Primero, explicamos que son los patrones de paralelismo (Sección 3.1). Después, presentamos los principales patrones y su formalización matemática (Sección 3.2). Finalmente, se describen los patrones implementados en GrPPI y las posibilidades de composición entre patrones que ofrece GrPPI (Sección 3.3).

3.1 Introducción

Los patrones pueden definirse vagamente como estrategias comúnmente recurrentes para tratar problemas particulares. Esta metodología ha sido ampliamente utilizada en múltiples áreas, como arquitectura, programación orientada a objetos y arquitectura de software. En nuestro caso, aprovechamos los patrones para el diseño de software paralelo, ya que ha sido reconocido como una de las mejores prácticas de codificación. Esto se debe principalmente a que los patrones proporcionan un mecanismo para encapsular características algorítmicas, haciéndolas más robustas, portátiles y reutilizables, mientras que si se adaptan, pueden lograr una mejor escalabilidad y ubicación de datos. En general, los patrones paralelos se pueden categorizar en dos grupos: patrones paralelos de streaming, por ejemplo, Pipeline, Farm y Filter; y datos paralelos, por ejemplo, Map, Reduce y MapReduce. Siguiendo esta clasificación, describimos formalmente los patrones soportados por nuestra interfaz en la siguiente Sección.

3.2 Descripción de los patrones y formalización matemática

En esta Sección describimos formalmente los patrones paralelos de streaming (*Pipeline, Farm, Filter y Accumulator*) y los patrones de datos paralelos (*Map, Reduce, Stencil, MapReduce y Divide&Conquer*), todos ellos incluídos en GrPPI.

3.2.1 Patrones de streaming

- **Pipeline**: Este patrón procesa los elementos que aparecen en el flujo de entrada en varias etapas paralelas. Cada etapa de este patrón procesa los datos producidos por la etapa anterior en la tubería y entrega los resultados a la siguiente. Siempre que la etapa i-ésima en un pipe con n-etapas compute la función $f_i : \alpha \rightarrow \beta$, el Pipeline entrega el elemento x_i a la secuencia de salida aplicando la función $f_n(f_{n-1}(\ldots f_1(x_i)\ldots))$. El principal requisito de este patrón es que las funciones relacionadas con las etapas deben ser puras, es decir, pueden computarse en paralelo sin efectos secundarios.

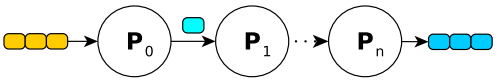


Figura 3-1: *Patrón Pipeline.*

- **Farm**: Este patrón computa en paralelo la siguiente función: $f : \alpha \rightarrow \beta$ sobre todos los items que aparecen en la secuencia de entrada. Por lo tanto, para cada elemento x_i en la secuencia de entrada, el patrón Farm entrega el elemento a la secuencia de salida como $f(x_i)$. En este patrón, los cálculos realizados por f para los elementos en el flujo de entrada deben ser completamente independientes entre sí, de lo contrario no pueden procesarse en paralelo.

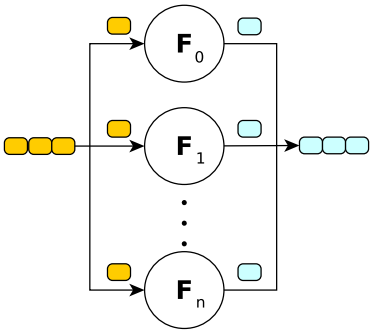


Figura 3-2: *Patrón Farm.*

- **Filter**: Este patrón calcula en paralelo un filtro sobre los elementos que aparecen en la secuencia de entrada, pasando solo a la secuencia de salida aquellos elementos que satisfacen la función booleana "filter" (o predicado) $\mathcal{P} : \alpha \rightarrow \{true, false\}$. Básicamente, el patrón recibe una secuencia de elementos de entrada $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ y produce una secuencia de elementos de salida del mismo tipo pero con diferente cardinalidad. La evaluación de la función de filtrado en un elemento de entrada debe ser independiente de cualquier otra, es decir, el predicado debe ser una función pura.

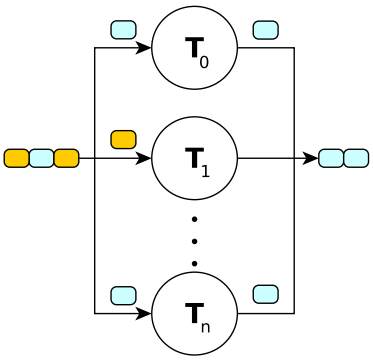


Figura 3-3: *Patrón Filter.*

- **Accumulator**: Este patrón colapsa los elementos que aparecen en la secuencia de entrada y entrega estos resultados a la secuencia de salida. La función utilizada para contraer los valores de elemento $\oplus$ debe ser una función binaria pura de tipo $\oplus : \alpha \times \alpha \rightarrow \alpha$, siendo generalmente asociativa y conmutativa. Básicamente, el patrón calcula la función $\oplus$ sobre una secuencia finita de elementos de entrada $\ldots, x_{i+1}, x_i, x_{i-1}, \ldots$ para producir un elemento contraído en la secuencia de salida . La cantidad de elementos que se acumulará depende del tamaño de ventana configurado como parámetro.

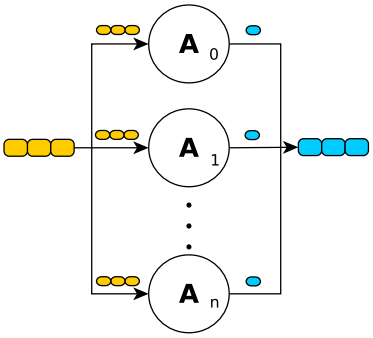


Figura 3-4: *Patrón Accumulator.*

3.2.2 Patrones de datos

- **Map**: Este patrón paralelo de datos calcula la función $f : \alpha \rightarrow \beta$ sobre los elementos de la colección de datos de entrada, donde los elementos de entrada y salida son de los tipos α y β respectivamente. El resultado de salida es la colección de elementos $y_1, y_2, \ldots, y_N$, donde $y_i = f(x_i)$ para cada $i = 1, 2, \ldots, N$ y x_i el el elemento i-ésimo de la colección de entrada. El único requisito del patrón map es que la función f sea pura.

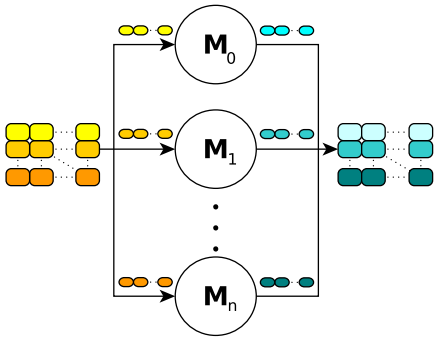


Figura 3-5: *Patrón Map.*

- **Reduce**: Este patrón paralelo de datos agrega los elementos de la colección de datos de entrada de tipo α utilizando la función binaria $\oplus : \alpha \times \alpha \rightarrow \alpha$, que generalmente es asociativa y conmutativa. Finalmente, el resultado del patrón se resume en un solo elemento y del tipo α que se obtiene al realizar la operación $y = x_1 \oplus x_2 \oplus \ldots x_N$, donde x_i es el elemento i-ésimo de la colección de datos de entrada. La principal restricción de este patrón es que la función binaria debe ser pura.

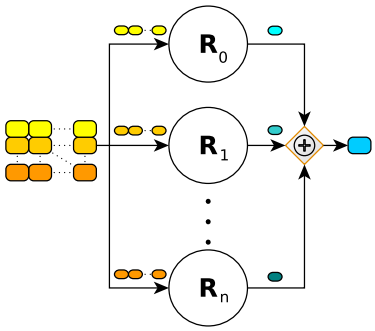


Figura 3-6: *Patrón Reduce.*

- **Stencil**: Este patrón es una generalización del patrón map en el cual una función elemental puede acceder, no solo a un elemento único en una colección de entrada, sino también a un conjunto de vecinos. La función $f : \alpha* \rightarrow \alpha$ utilizada por el patrón stencil recibe el elemento de entrada y un conjunto de vecinos ($\alpha*$) y produce un elemento de salida del mismo tipo. El principal requisito de este patrón es que la función f debe ser pura.

- **MapReduce**: Este patrón calcula, en una primera etapa, un patrón tipo map, una función de valor de clave sobre todos los elementos de una colección de entrada, y entrega, en una segunda etapa, un patrón de reducción, un conjunto de pares de valores de clave únicos donde el valor asociado a la clave es la "suma" de los valores emitidos para la misma clave. Para hacerlo, el patrón mapreduce calcula en la función map $f : \alpha \rightarrow \{\alpha, Key\}$ los elementos en la colección de entrada; luego usa la

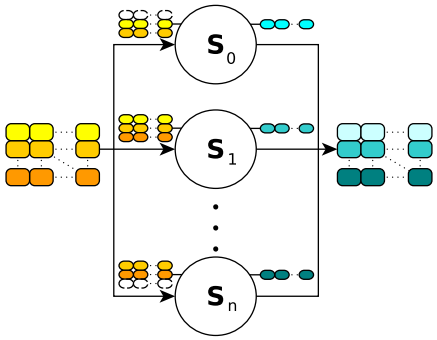


Figura 3-7: *Patrón Stencil.*

función binaria de reducción $\oplus : \beta \times \beta \rightarrow \beta$ para resumir los resultados parciales con la misma clave. El resultado de este patrón es una colección de elementos de datos de tipo β, uno por clave. Los requisitos del patrón mapreduce es que las funciones relacionadas tanto map como reduction deben ser puras.

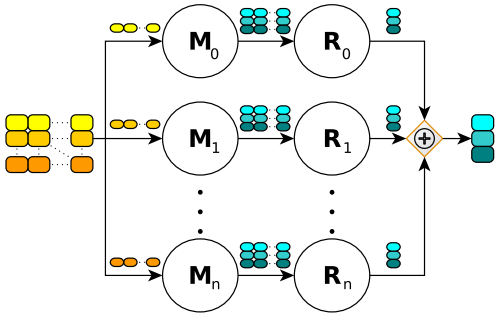


Figura 3-8: *Patrón MapReduce.*

- **Divide&Conquer**: Este patrón calcula un problema dividiéndolo en dos o más subproblemas del mismo tipo hasta que se alcanza el caso base y se resuelve directamente. Después, las soluciones de los subproblemas se combinan para proporcionar una solución al problema original. En otras palabras, este patrón aplica la función $f : \alpha* \rightarrow \beta*$ en una colección de elementos de tipo α y produce una colección de elementos de tipo β. Una función de división $\mathcal{D}$ se usa primero para dividir la colección en distintas particiones hasta el tamaño del problema base, que se puede resolver directamente aplicando f. Finalmente, los resultados parciales de los problemas de base se combinan de acuerdo con una función de combinación $\mathcal{M}$ para construir la colección de salida final. Los requisitos del patrón dac es que las funciones f, $\mathcal{S}$ y $\mathcal{M}$ deben ser puras.

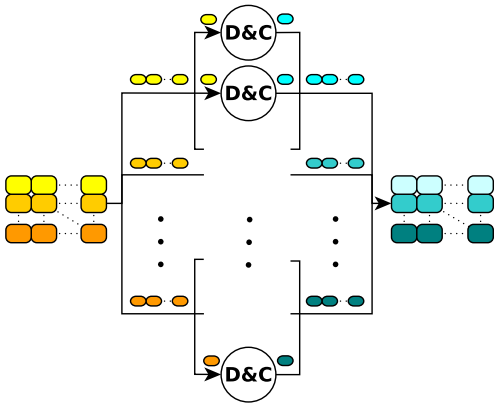


Figura 3-9: *Patrón Divide&Conquer.*

3.3 Interfaz de los patrones en GrPPI

En esta sección se presenta la interfaz de patrones paralelos genérica y reutilizable (GrPPI) para aplicaciones C ++. GrPPI aprovecha al máximo las características modernas de C ++, los conceptos de metaprogramación y la programación genérica para actuar como un interruptor entre los modelos de programación paralelos OpenMP, hilos C ++, Intel TBB y CUDA Thrust. Su diseño permite a los usuarios aprovechar los frameworks de ejecución antes mencionados solo en una interfaz única y compacta, ocultando la complejidad detrás del uso de mecanismos de concurrencia. Además, la modularidad de GrPPI permite integrar fácilmente nuevos patrones, combinándolos para organizar otros más complejos. Gracias a esta propiedad, GrPPI se puede utilizar para implementar una amplia gama de aplicaciones existentes de procesamiento de datos y de flujo de datos con esfuerzos relativamente pequeños, teniendo como resultado códigos portátiles que se pueden ejecutar en múltiples frameworks. La Figura 3-10 representa la vista general de la biblioteca GrPPI.

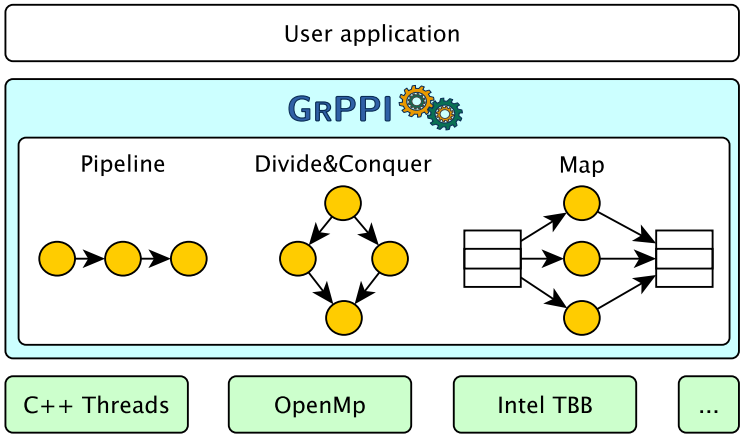


Figura 3-10: *Arquitectura de GrPPI.*

A continuación, describimos en detalle las interfaces de los patrones paralelos ofrecidos por GrPPI y demostrar su capacidad de compilación a través de diferentes ejemplos simples.

3.3.1 Descripción de las interfaces

GrPPI ofrece patrones de streaming y patrones de datos con una única interfaz cuidadosamente diseñada para permitir la composición y admitir múltiples back-ends de implementación.

Patrones de streaming

Los patrones de streaming incluídos en GrPPI son Pipeline, Farm, Filter y Accumulator.

- **Pipeline**: La interfaz GrPPI diseñada para el patrón Pipeline recibe el modelo de ejecución y las funciones (`in` y `stages`) relacionadas con sus etapas. Como se puede ver en el Listado 3.1, su interfaz C ++ usa plantillas, haciéndola más flexible y reutilizable para cualquier tipo de datos. Tenga en cuenta también el uso de plantillas variadicas, lo que permite que una tubería tenga un número arbitrario de etapas al recibir una colección de objetos invocables pasados como argumentos. En GrPPI, la implementación paralela de este patrón se lleva a cabo utilizando un conjunto de entidades concurrentes, cada una de las cuales se ocupa de una sola etapa. Esto se controla a través del parámetro del modelo de ejecución, que se puede configurar para operar en secuencia o en paralelo, a través de los diferentes frameworks compatibles; p.ej. para usar OpenMP, el parámetro debe establecerse en `parallel_execution_omp`.

Listado 3.1: *Interfaz Pipeline.*

```
1 template <typename ExecMod, typename InFunc, typename ... Arguments>
2 void Pipeline( ExecMod m, InFunc const &in, Arguments ... stages );
```

- **Farm**: De forma similar, la interfaz del patrón Farm, que se muestra en el Listado 3.2, recibe el modelo de ejecución y tres funciones (`in`, `Farm` and `out`) que son a cargo de *i)* consumir los artículos del flujo de entrada, *ii)* procesarlos individualmente, y *iii)* entregar los resultados al flujo de salida. Tenga en cuenta que la función `Farm` se ejecutará en paralelo por las diferentes entidades concurrentes. En este caso, el modelo de ejecución puede recibir opcionalmente, como argumento, el número de entidades que se utilizarán para la ejecución paralela, por ejemplo, `parallel_execution_omp{6}` utiliza 6 hilos de trabajo OpenMP. Si no se proporciona este argumento, la interfaz toma por defecto el número de hilos establecidos por la plataforma subyacente.

Listado 3.2: *Interfaz Farm.*

```
1 template <typename ExecMod, typename InFunc, typename TaskFunc, typename OutFunc>
2 void Farm( ExecMod m, InFunc const &in, TaskFunc const &Farm, OutFunc const &out );
```

- **Filter**: La interfaz para el patrón filter, que se describe en el Listado 3.3, recibe el argumento del modelo de ejecución, seguido de un consumidor de flujo (`in`), filtro (`filter`) y productor (`out`). Específicamente, la función `in` lee elementos de la secuencia de entrada y los reenvía a la función `filter` que es responsable de determinar si un elemento debe ser aceptado o no. Posteriormente, los elementos que satisfacen la rutina de filtrado son recibidos por la función `out` para entregarlos al flujo de salida. Tenga en cuenta que es obligatorio que la función `filter` devuelva una expresión booleana. La implementación paralela de este patrón aplica la función de filtro usando un conjunto de entidades concurrentes, que se pueden configurar en el parámetro del modelo de ejecución.

Listado 3.3: *Interfaz Filter.*

```
template <typename ExecMod, typename InFunc, typename FilterFunc, typename OutFunc>
void Filter( ExecMod m, InFunc const &in, FilterFunc const &filter, OutFunc const &out );
```

- **Accumulator**: El patrón accumulator tiene como objetivo reducir, utilizando una función de reducción específica (`redop`) los elementos que aparecen en el flujo de entrada. De forma similar a las otras interfaces, la interfaz del acumulador, como se muestra en el Listado 3.4, recibe el modelo de ejecución; la función de consumidor de flujo (`in`) el tamaño de la ventana, es decir, el número de elementos que formarán parte de cada operación de reducción; el desplazamiento, que determina el número de elementos superpuestos entre las ventanas; el operador de reducción; y una función de productor (`out`) responsable de entregar los artículos al flujo de salida. En este caso, las entidades concurrentes en la implementación paralela son responsables de procesar individualmente la acumulación de las ventanas de flujo de entrada.

Listado 3.4: *Interfaz Accumulator.*

```
template <typename GenFunc, typename ReduceOperator, typename OutFunc>
void Accumulator( ExecMod m, GenFunc const &in, int windowsize, int offset, ReduceOperator const &redop, OutFunc
     const &out );
```

Patrones de datos

Los patrones de datos incluídos en GrPPI son Map, Reduce, Stencil, MapReduce y Divide&Conquer.

- **Map**: La interfaz GrPPI para el patrón map, que se muestra en el Listado 3.5, ecibe los siguientes parámetros de entrada: el modelo de ejecución, las referencias al primer y último elemento de las colecciones de datos de entrada y la función kernel (`map`). Después del cálculo, el resultado del patrón del mapa se deja en la posición correspondiente del conjunto de datos de salida. Dado que cada elemento de la colección de datos de entrada es independiente entre sí, la ejecución

paralela del patrón de mapa se puede realizar de la siguiente manera. Primero, la colección de entrada se divide por igual entre las entidades concurrentes disponibles. Posteriormente, estas entidades ejecutan en paralelo la función kernel `map` y escriben los resultados en los segmentos correspondientes de la recopilación de datos de salida.

Listado 3.5: *Interfaz Map.*

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename TaskFunc, typename ... MoreIn>
void Map(ExecMod m, InputIt first, InputIt last, OutputIt firstOut, TaskFunc const &map, MoreIn ... inputs );
```

- **Reduce**: La interfaz para el patrón reduce, como se describe en el Listado 3.6, toma el modelo de ejecución, una referencia al primer y último elemento de la colección de datos de entrada y al operador de reducción. El resultado de la reducción se escribe en el parámetro de salida pasado por referencia. De acuerdo con las propiedades del operador de reducción, el cálculo de reducción se puede realizar en paralelo. Por lo tanto, la recopilación de datos de entrada se divide en N trozos y se calcula en paralelo mediante N entidades concurrentes diferentes que producen un conjunto de resultados parciales. Finalmente, el resultado del patrón de reducción se calcula en serie por una de estas entidades.

Listado 3.6: *Interfaz Reduce.*

```
template <typename ExecMod, typename InputIt, typename Output, typename ReduceOperator>
void Reduce(ExecMod m, InputIt first, InputIt last, Output &out, ReduceOperator const &redop);
```

- **Stencil**: La interfaz GrPPI para el patrón stencil, presentado en el Listado 3.7, es bastante similar a la del patrón de mapa, con la excepción de que también recibe la función de vecindad (`nh`). Esta función es responsable de acceder a los vecinos en una determinada coordenada del conjunto de datos de entrada. La implementación paralela del patrón de esténcil es análoga a la del patrón de mapa. Sin embargo, el acceso a los vecinos en los límites de un conjunto de datos particionados podría requerir comparaciones adicionales entre las posiciones de los elementos.

Listado 3.7: *Interfaz Stencil.*

```
template <typename ExecMod, typename InputIt, typename OutputIt, typename TaskFunc, typename NFunc, typename ...
     MoreIn>
void Stencil(ExecMod m, InputIt first, InputIt first, InputIt last, OutputIt firstOut, TaskFunc const &stencil, NFunc
      const &nh, MoreIn ... inputs);
```

- **MapReduce**: La interfaz para el patrón mapreduce combina llamadas internas al mapa y reduce las interfaces del patrón GrPPI. En cuanto a los parámetros de entrada, recibe el modelo de ejecución, referencias al primer y último elemento de las colecciones de datos de entrada, la función

kernel (`map`) y el operador de reducción para el patrón reduce. El resultado finalmente se deja en una referencia al primer elemento de la colección de salida. La implementación paralela de este patrón en GrPPI explota el paralelismo ofrecido internamente por el mapa y reduce los patrones paralelos. El resultado de la operación del mapa se baraja y se reduce en paralelo, uno para todos los elementos con la misma clave. El resultado global para cada clave finalmente se reduce en serie por una de las entidades concurrentes.

Listado 3.8: *Interfaz MapReduce.*

```
template <typename ExecMod, typename InputIt, typename Output, typename MapFunc, typename ReduceOperator, typename
    ... MoreIn>
void MapReduce(ExecMod m, InputIt first, InputIt last, Output &out, MapFunc const &map, ReduceOperator const &redop,
    MoreIn ... inputs);
```

- **Divide&Conquer**: La interfaz diseñada para el patrón Divide&Conquer consta de los siguientes elementos: el modelo de ejecución, una referencia de la colección de datos de entrada y las funciones `divide`, `base_case` and `merge`. El resultado de este patrón se escribe en la recopilación de datos de salida pasada por referencia. La implementación paralela de este patrón en la interfaz GrPPI aprovecha primero el núcleo dividido para dividir el problema de manera constante en otros más pequeños. Esta operación es realizada por las entidades concurrentes disponibles hasta que se alcanza la dimensión de problema mínima y donde se aplica el kernel de solución de caso base. Tomando las soluciones parciales generadas, las entidades concurrentes fusionan los resultados en una estructura basada en árbol hasta que se obtiene la solución global. Tenga en cuenta que dado que el ancho del árbol puede crecer por encima del número máximo de entidades concurrentes especificadas, se usa un grupo de tareas para implementar un enfoque de programación dinámica.

Listado 3.9: *Interfaz Divide&Conquer.*

```
template <typename ExecMod, typename Input, typename Output, typename DivFunc, typename TaskFunc, typename MergeFunc>
void DivideAndConquer(ExecMod m, Input &problem, Output &out, DivFunc const &divide, TaskFunc const &base_case,
    MergeFunc const &merge);
```

3.3.2 Composición de patrones

Como se mencionó previamente, los patrones ofrecidos por GrPPI pueden componerse entre ellos para producir estructuras más complejas y para unir construcciones específicas presentes en aplicaciones secuenciales y paralelas de datos. Para demostrar esta característica, describimos tres ejemplos de compibilidad de patrones abordando cada una de las combinaciones factibles de paradigmas computacionales (streaming y datos) soportados por la interfaz GrPPI: stream-stream, data-data y composición stream-data.

Para la composición del patrón stream-stream, el código en el Listado3.10 implementa un Pipeline en el que la segunda etapa es un patrón Farm. Las etapas del Pipeline, pasadas como funciones lambda, realizan las siguientes tareas: *i)* leer las líneas de un archivo de entrada con valores separados por espacios en blanco y empaquetarlos en una estructura vectorial, *ii)* calcular el valor máximo de vectores entrantes usando el patrón Farm, y *iii)* imprime los valores máximos de los vectores en un flujo de salida. Dado que el Pipeline recibe el modelo de ejecución paralela OpenMP (línea 1), las etapas se calculan en paralelo por los 3 hilos de trabajo. De forma similar, el patrón de conjunto anidado se ejecuta mediante 6 hilos OpenMP. Tenga en cuenta también que las variables `std::optional`, de *c++ Library Fundamentals Extensions* (ISO / IEC 19568: 2015), se utilizan para marcar el final de las secuencias con un valor vacío. Denotamos esta composición de Pipeline-Farm `(p|f|p)`, siendo `p` y `f`, respectivamente, etapas secuenciales y basadas en Farm. Como se puede ver, gracias al uso de técnicas de metaprogramación, plantillas y expresiones lambda, es posible componer fácilmente patrones paralelos GrPPI para construir otros más complejos.

Listado 3.10: *Ejemplo de composición Pipeline-Farm.*

```
1  Pipeline( parallel_execution_omp,
2      // Stage 0: read values from a file
3      [&]() -> optional<vector<int>> {
4          auto r = read_list(is);
5          return ( r.size() == 0 ) ? {} : r;
6      },
7      // Stage 1: takes the maximum value of the vector
8      Farm(parallel_execution_omp{6},
9          []( vector<int> v ) {
10             return ( v.size() > 0 ) ?
11                 max_element(v.begin(), v.end()) :
12                 numeric_limits<int>::min();
13     }),
14     //Stage 2: prints out the result
15     [&os]( int x ) {
16         os << x << endl;
17     }
18 );
```

Con respecto a la composición del patrón de datos de datos, el Listado 3.11 muestra una construcción donde se compone un patrón map con una operación reduce. En este caso, la matriz de entrada en el patrón map se divide en particiones iguales entre los hilos del trabajador. A continuación, para cada fila en una partición, el patrón reduce anidado resume sus valores y almacena el resultado en la posición correspondiente del vector de salida, pasado como un argumento en la llamada a la función del map. Tenga en cuenta que el modelo de ejecución en paralelo para el patrón map es OpenMP, mientras que el patrón reduce anidado utiliza subprocesos de C ++, cada uno de ellos utilizando 6 subprocesos de trabajo. Denotamos esta composición como `m(r)`, siendo `m` y `r` los patrones map y reduce, respectivamente.

Listado 3.11: *Ejemplo de composición Map-Reduce.*

```
Map( parallel_execution_omp{6},
    // Input matrix
    mat_in.begin(), mat_in.end(),
    // Vector of accumulated values from matrix rows
    vec_out.begin(),
    // Map kernel: divide matrix into rows
    [&]( auto row_in, auto sum ) {
        // Reduce kernel: Sum up the values in a matrix row
        Reduce( parallel_execution_thr{6},
            row_in.begin(), row_in.end(),
            &sum,
            std::plus<double> );
        );
    }
);
```

Como se mencionó, también podemos componer stream con patrones de datos. Esta es una composición factible, dado que los elementos provenientes de un flujo se pueden procesar utilizando un patrón paralelo de datos. Sin embargo, lo contrario no es factible ya que los resultados generados en un patrón de datos no pueden transformarse en flujo y, por lo tanto, procesarse utilizando un enfoque de procesamiento de streaming. Para ilustrar una composición de patrón de datos de flujo, el Listado 3.12 muestra un ejemplo donde un patrón paralelo de flujo de granja se compone con un dato de Divide&Conquer. En este caso particular, el patrón de Farm lee constantemente los valores almacenados en un archivo y calcula, para cada uno de ellos, su correspondiente *i*-ésimo número de Fibonacci utilizando el patrón Divide&Conquer. Finalmente, los números de Fibonacci se imprimen para el usuario final. Como se muestra, la paralelización de la granja se realiza utilizando 6 hilos OpenMP, mientras que el patrón Divide&Conquer anidado usa 6 hilos C ++. Dado que cada uno de los subprocesos relacionados con la granja crean 6 anidados en C ++, el número total de subprocesos que computan esta composición es 36. Esta composición se denota como `f(d)`, siendo `f` y `d`, los patrones Farm y Divide&Conquer, respectivamente.

Listado 3.12: *Ejemplo de composición Farm-Divide&Conquer.*

```
Farm(parallel_execution_omp(6),
    [&]() -> optional<int> { // Read values from an input file
        auto value = read_value(is);
        return ( value > 0 ) ? value : {};
    },
    [&]( int value ) { // Compute the fibonacci number using a D&C pattern
        int fibonacci = 0;
        DivideAndConquer(parallel_execution_thr(6), value, &fibonacci,
            [&](auto &value){
                std::vector< int > subproblem;
                if( v < 2 ) subproblem.push_back(value);
                else subproblem.insert(subproblem.end(), { value-1, value-2 });
                return subproblem;
            },
            [&](auto &problem, auto &partial){
                partial = ( problem == 0 ) ? 0 : 1;
            },
```

```
18            [&](auto & partial, auto & out){
19                out += partial;
20            }
21        );
22        return fibonacci;
23    },
24    [&]( int fibonacci ) { // Print the fibonacci values
25        cout << fibonacci << endl;
26    }
27 );
```

En general, las Tablas 3.1, 3.2 y 3.3 resumen las composiciones de patrones agrupadas por las tres posibles combinaciones de paradigmas computacionales compatibles con la interfaz GrPPI: stream-stream, data-data y stream-data. Tenga en cuenta que las filas y columnas en las tablas representan los patrones externos e internos involucrados en una composición determinada, respectivamente. Clasificamos cada composición de patrón específico con una de las siguientes cuatro categorías, desde menos hasta más restrictivas:

- **No-Factible**: Esta categoría representa una composición que no es compatible con GrPPI.
- **Factible**: Esta categoría denota una composición que se puede implementar en GrPPI.
- **Irreducible**: esta categoría es una composición factible que proporciona un patrón paralelo útil que no puede simplificarse más. Tenga en cuenta que las composiciones de patrones que se incluyen en esta categoría son compatibles de forma nativa con GrPPI.
- **Usable-Reducible**: Esta categoría es una composición factible que implementa una composición de patrones que puede simplificarse aún más, pero que, en algunos casos, proporciona un código más claro y más legible que su equivalente más simple.

Como se muestra en la Tabla 3.1, las composiciones de patrones stream-stream que involucran un Pipeline y otro patrón se clasifican como Irreducible (excepto aquellos con un patrón de accumulator externo), dado que no es posible obtener la misma construcción paralela usando un patrón más simple. Este tipo de composiciones son compatibles nativamente en GrPPI, como se muestra en el Listado 3.10. Cualquier otra composición se considera Factible ya que pueden simplificarse usando el patrón externo o interno con un mayor grado de paralelismo. Sin embargo, estas construcciones no proporcionan ninguna ventaja importante en comparación con la construcción más simple. Por otro lado, las composiciones que contienen un patrón de acumulador externo son No-Factible, ya que este patrón no recibe ninguna función de usuario para ejecutarse en paralelo.

Centrándose en las composiciones de datos de datos, como se muestra en la Tabla 3.2, las construcciones cuyo patrón exterior es similar a un map (map and stencil) se categorizan como Usable-Reducible. Esto se debe a que existe un equivalente más simple que usa solo el patrón externo tipo mapa. En cuanto

Tabla 3.1: *Composiciones stream-stream.*

	Pipeline	Farm	Filter	Accumulator
Pipeline	✓ (Factible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Farm	✓ (Irreducible)	✓ (Factible)	✓ (Factible)	✓ (Factible)
Filter	✓ (Irreducible)	✓ (Factible)	✓ (Factible)	✓ (Factible)
Accumulator	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)

al patrón de reducción, no se puede combinar con ningún otro interno. Las razones son las mismas que para el patrón del acumulador en las composiciones de flujo continuo. Otras composiciones cuyo patrón exterior es mapreduce o Divide&Conquer se clasifican como Factible, ya que pueden implementarse en GrPPI aunque no aportan ninguna ventaja importante.

Tabla 3.2: *Composiciones data-data.*

	Map	Reduce	Stencil	MapReduce	Divide&Conquer
Map	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)
Reduce	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)
Stencil	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)	✓ (Usable-Reducible)
MapReduce	✓ (Factible)	✓ (Factible)	✓ (Factible)	✓ (Factible)	✓ (Factible)
Divide&Conquer	✓ (Factible)	✓ (Factible)	✓ (Factible)	✓ (Factible)	✓ (Factible)

Finalmente, las composiciones de stream-data se resumen en la Tabla 3.3. Las composiciones cuyo patrón exterior es Pipeline o Farm se denotan como Irreducible. La combinación de dos paradigmas paralelos distintos (stream-data) hace que estas composiciones sean únicas e impide que se simplifiquen más. En cuanto a las composiciones con un patrón de filtro externo, la cardinalidad de salida de su patrón interno determina si la composición es Factible or Usable-Reducible. Esto se debe a que la salida de la función de filtro es booleana.

Tabla 3.3: *Composiciones stream-data.*

	Map	Reduce	Stencil	MapReduce	Divide&Conquer
Pipeline	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Farm	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)	✓ (Irreducible)
Filter	✓ (Factible)	✓ (Factible)*	✓ (Factible)	✓ (Factible)	✓ (Factible)*
Accumulator	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)	✗ (No-Factible)

Por ejemplo, en una composición map-filter, la cardinalidad de salida del patrón map es igual a la cardinalidad de entrada. De modo que, aunque el predicado del patrón filter se puede implementar

transformando el conjunto de datos de salida en un booleano, este caso no refleja una práctica común. Por lo tanto, clasificamos estas composiciones solo como Factible. Por otro lado, la cardinalidad de salida del patrón reduce en una composición filter-reduce es un elemento único. Por lo tanto, el predicado de filtro se puede implementar fácilmente transformando dicho elemento en un booleano. Por esta razón, categorizamos esta construcción como un caso especial de composición Factible. La combinación filter-Divide&Conquer también es un caso especial de composición Factible porque la cardinalidad de salida del patrón Divide&Conquer depende del algoritmo. Finalmente, el patrón accumulator no se puede componer y, por lo tanto, es clasificado como No-Factible.

Capítulo 4

Interfaz de patrones de programación paralela para memoria distribuida

Este capítulo presenta la implementación realizada de GrPPI para plataformas de memoria distribuida. En primer lugar se describe el problema (Sección 4.1) por el cual se decide realizar esta implementación. Posteriormente, se describe la política de ejecución MPI para GrPPI (Sección 4.2), describiendo la interfaz de usuario resultante (Sección 4.2.1), la implementación de las colas de comunicación entre procesos (Sección 4.2.2) y el algoritmo de distribución de operadores en procesos MPI (Sección 4.2.3).

4.1 Descripción del problema

En general, la implementación de un back-end distribuido mediante MPI se ha motivado principalmente por las necesidades de escalamiento y el desarrollo de nuevos modelos de programación para aplicaciones científicas DaSP. El interés de implementar este back-end proviene de la amplia adopción de MPI en las supercomputadoras de hoy, que actualmente no tiene soporte estándar para el procesamiento de streaming [36]. Debido a este motivo al crecimiento en los últimos años del paradigma del *Big Data*, creemos necesario presentar un back-end que pueda ser de ayuda en el desarrollo de aplicaciones de streaming en C ++.

Por otro lado, una vez analizado el estado de la cuestión y las herramientas y frameworks actualmente disponibles para este área, identificamos una brecha importante entre la comunidad de MPI y las necesidades de procesamiento de flujo de las aplicaciones científicas actuales [1]. En este sentido, el objetivo del nuevo back-end de GrPPI MPI es doble. Por un lado, GrPPI ofrece una interfaz C ++ de alto nivel de patrones paralelos que mejora tanto la flexibilidad de la aplicación como la legibilidad del código fuente. Por otro lado, el uso de GrPPI MPI permite de forma transparente la ejecución de la aplicación de C ++ en tiempo real en plataformas HPC distribuidas, ya que GrPPI oculta la complejidad

relacionada con la comunicación y la sincronización de procesos.

4.2 Política de ejecución MPI para GrPPI

Con respecto a la interfaz de patrones paralelos, hemos aprovechado GrPPI, una interfaz de patrones paralelos genérica y reutilizable para aplicaciones C ++. Esta interfaz aprovecha al máximo las características modernas de C ++, los conceptos de metaprogramación y la programación genérica para actuar como un interruptor entre , hilos de C ++ y los modelos de programación paralela de Intel TBB. Su diseño permite a los usuarios aprovechar los marcos de ejecución antes mencionados en una interfaz única y compacta, ocultando la complejidad detrás del uso de mecanismos de concurrencia. Además, la modularidad de GrPPI permite integrar fácilmente nuevos patrones, combinándolos para organizar construcciones más complejas. Además, permite la integración de nuevas políticas de ejecución basadas en modelos de programación distribuidos y de memoria compartida. Gracias a estas propiedades, GrPPI se puede utilizar para implementar una amplia gama de aplicaciones existentes de procesamiento de datos y de streaming con esfuerzos relativamente pequeños, teniendo como resultado códigos portátiles que se pueden ejecutar en múltiples plataformas.

Como se indicó anteriormente, el objetivo de GrPPI es acomodar una capa de patrones paralelos entre desarrolladores y marcos de programación paralelos existentes. Hasta ahora, GrPPI solo admite políticas de ejecución de memoria compartida, como C + + Threads, e Intel TBB. Para extender GrPPI a fin de admitir plataformas distribuidas, hemos incorporado la política de ejecución de MPI que, en este momento, permite a los usuarios ejecutar los patrones de oleoductos y flujos de granja en clusters de múltiples núcleos. En esta sección, describimos en detalle los elementos básicos de la política de ejecución de MPI: *i)* interfaz de usuario; *ii)* canales de comunicación interna; y *iii)* algoritmo de distribución de operadores.

4.2.1 Interfaz de usuario

Dado el diseño GrPPI, donde cada política de ejecución se implementa utilizando una clase C ++, para la nueva política MPI también diseñamos su clase. Esto se debe a que cada una de estas clases contiene las implementaciones de patrones específicos del marco junto con los parámetros de configuración para esa política. Básicamente, las interfaces de patrones GrPPI están sobrecargadas con una implementación diferente para cada una de las políticas de ejecución disponibles. Con él, cuando se compila el código de usuario, la implementación del patrón específico se selecciona dependiendo de la política de ejecución pasada como argumento (ver parámetro `execution_policy` en la interfaz del patrón Pipeline en la Figura 4.3). El Listado 4.1 muestra un extracto de la clase de política de ejecución de MPI. Tenga en cuenta que hemos utilizado la biblioteca Boost MPI [52] como interfaz de C ++ MPI.

Como se observa, los constructores de políticas de ejecución reciben los argumentos del programa o

el comunicador MPI directamente. Para admitir escenarios híbridos, ambos constructores pueden recibir una política de ejecución de memoria compartida local. Esta política local se usará para ejecutar múltiples etapas de Pipeline o réplicas de Farm (también denominadas operadores de flujo) dentro del mismo proceso. Por lo tanto, los operadores asignados al mismo proceso se ejecutarán en forma secuencial o en paralelo según la política de ejecución de la memoria compartida seleccionada.

Listado 4.1: *Clase de política de ejecución MPI en GrPPI.*

```
template<typename LocalPolicy = parallel_execution_native>
class parallel_execution_mpi {
 namespace mpi = boost::mpi;
 ...
 public:
  parallel_execution_mpi(int argc, char** argv,
         LocalPolicy local_exec_policy = parallel_execution_native{});
  parallel_execution_mpi(mpi::communicator mpi_comm = mpi::communicator{}, LocalPolicy local_exec_policy =
       parallel_execution_native{});
  ...
};
```

Como se muestra en el Listado 4.2, el patrón de Pipeline GrPPI aprovecha la nueva clase de política de ejecución MPI para su ejecución. Tenga en cuenta que la política se construye en la declaración utilizando los argumentos `argc` y `argv`. Esta canalización consta de tres etapas en la forma `(p|f|p)`, donde la primera y la tercera etapa se ejecutan en serie, mientras que la segunda ejecuta dos réplicas del mismo operador utilizando el patrón de la granja. Suponiendo que el programa se ejecute mediante 4 procesos de MPI, el primero y el último respectivamente ejecutan las etapas de generador y consumidor, mientras que el segundo y el tercero calcularán cada una de las réplicas de conjunto. Es importante destacar que los artículos que transitan de una etapa a otra se envían y reciben a través de las colas de comunicación MPI implementadas dentro de la política de respaldo.

Listado 4.2: *Ejemplo de Pipeline GrPPI distribuido.*

```
grppi::parallel_execution_mpi ex{argc, argv};
grppi::Pipeline(ex,
  [x=1,n]() mutable -> optional<double> {
    if (x<=n) return x++;
    else return {};
  },
  grppi::Farm(2,[](double x) {
    return x*x;
  }),
  [](double x) { cout << x << endl; }
);
```

Los Listados 4.3, 4.4 muestra los prototipos C ++ GrPPI MPI de los patrones Pipeline y Farm. Tenga en cuenta el uso de `templates` con parámetros únicos y múltiples (pack) como referencias universales, lo que permite pasar múltiples objetos invocables (por ejemplo, un functor o expresión lambda) como

para los operadores de patrones. Tenga en cuenta también que el primer parámetro indica la política de ejecución que se utilizará para ejecutar los operadores.

Listado 4.3: *Interfaz patrón Pipeline en GrPPI MPI.*

```
template <typename E, typename G, typename ... O>
void Pipeline(E execution_policy, G && generator, O && ... operators);
```

Listado 4.4: *Interfaz patrón Farm en GrPPI MPI.*

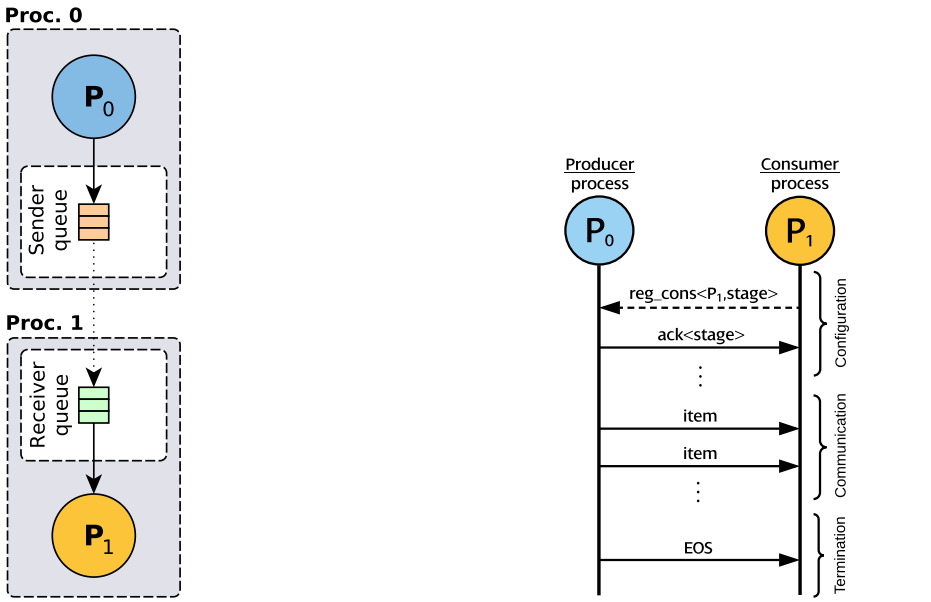
```
template <typename O>
void Farm(int replicas_replicas, O && operators);
```

4.2.2 Colas de comunicación

Los canales de comunicación son fundamentales en las aplicaciones de DaSP distribuidas. Por ejemplo, el operador de flujo que se ejecuta en un nodo necesita recibir elementos del operador anterior mapeado en otro nodo, realizar algunos cálculos sobre ellos y enviarlos al próximo operador. Por lo tanto, para habilitar el paralelismo de flujo entre nodos en la política de ejecución MPI, determinamos la necesidad de canales de comunicación en forma de colas Primero en entrar, primero en salir (FIFO). Para eso, dado que MPI no admite de forma nativa las colas distribuidas, hemos proporcionado una cola de comunicación unidireccional sobre las primitivas MPI `send` y `receive`, es decir, las comunicaciones a dos caras. Básicamente, estas colas encapsulan las comunicaciones entre procesos MPI que ejecutan operadores de flujo. De esta forma, las implementaciones de patrones solo hacen uso de las funciones públicas de cola, por ejemplo, el lado del productor de cola MPI llama `push` para poner en la cola un elemento que debe enviarse al siguiente operador; mientras que el lado del consumidor llama a `pop` para desencolar un elemento al recibirlo del operador anterior.

Teniendo en cuenta las posibles disposiciones entre patrones GrPPI Pipeline y Farm, identificamos cuatro escenarios de comunicación diferentes:

- *Single-Producer, Single-Consumer* (SPSC): dos etapas Pipeline consecutivas `(p|p)`.
- *Single-Producer, Multiple-Consumer* (SPMC): una etapa de Pipeline a un conjunto de réplicas de Farm `(p|f)`.
- *Multiple-Producer, Single-Consumer* (MPSC): un conjunto de réplicas de Farm a una etapa de Pipeline `(f|p)`.
- *Multiple-Producer, Multiple-Consumer* (MPMC): un conjunto de réplicas de Farm a otro conjunto de réplicas de Farm `(f|f)`.



(a) *Diagrama de colas SPSC.* (b) *Protocolo de comunicación de colas SPSC.*

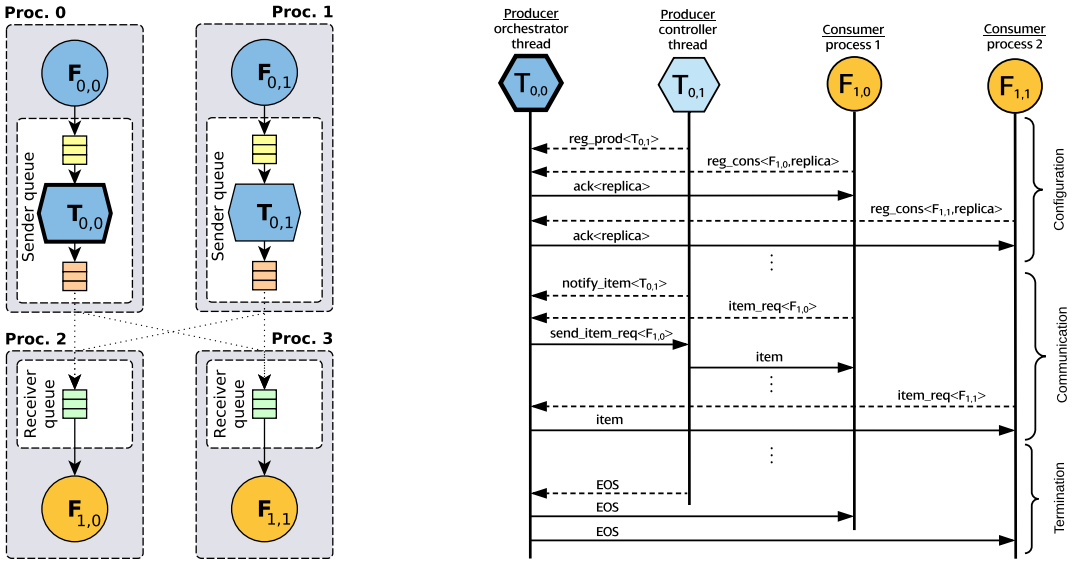
Figura 4-1: *Diagrama y protocolo de comunicación de colas SPSC.*

El lado izquierdo de las Figuras 4-1 y 4-2 muestran, respectivamente, los diagramas de cola en los escenarios de SPSC y MPMC.[1] Como se muestra en el lado del productor de colas MPMC, la cola crea un subproceso de controlador por proceso de productor (T) para solapar las comunicaciones con los cálculos del operador. La cola también selecciona un hilo del controlador que actúa como el orquestador, que es responsable de gestionar la cola haciendo un seguimiento de los artículos disponibles en cada productor y reenviando las solicitudes de artículos de consumo a un productor determinado. Sin embargo, para la cola de SPSC, dado que los procesos de productor y consumidor son conocidos de antemano, las comunicaciones pueden superponerse con cálculos utilizando directamente primitivas de envío/recepción asíncronas de MPI. Por lo tanto, no se necesitan subprocesos de controlador en el lado del productor para administrar la cola cuando se trata de escenarios de SPSC.

Para implementar el comportamiento mencionado anteriormente, los lados de cola de MPI del productor y del consumidor se comunican siguiendo un protocolo específico dependiendo del modo de cola ((ver lado derecho de las Figuras 4-1 y 4-2 para SPSC y MPMC, respectivamente). Este protocolo de comunicación se compone de las tres fases siguientes:

- **Fase de configuración**: el objetivo de esta fase es determinar si la cola debe ejecutarse en modo SPSC o MPMC. Si solo hay un productor y un consumidor, entonces la cola se configura como SPSC; de lo contrario, se establece como MPMC. Al comienzo de esta fase, tanto productores como consumidores inicializan sus lados de cola de acuerdo con el tipo de operador (etapa de Pipeline o

[1]Consideramos el SPMC y MPSC los escenarios se implementan utilizando el modo de cola MPMC, ya que estos casos se pueden ver como especializaciones de MPMC.



(a) *Diagrama de colas MPMC.* (b) *Protocolo de comunicación de colas MPMC.*

Figura 4-2: *Diagrama y protocolo de comunicación de colas MPMC.*

réplica de Farm). A partir de este momento, todos los procesos intercambian diferentes mensajes de configuración con el proceso del orquestador para seleccionar el modo de cola. Primero, los consumidores y productores respectivamente envían los mensajes `reg_recv<id, type>` y `reg_send<id>` para registrarse en el orquestador. Además, los consumidores indican el orquestador su `type`, es decir, la etapa de Pipeline o la réplica de Farm. A continuación, el orquestador configura la cola como SPSC o MPMC según la cantidad de consumidores y productores registrados, y les comunica el modo de cola final (`ack<mode>`). Finalmente, los procesos de los productores inician los hilos de controlador correspondientes si la cola está configurada como MPMC.

- **Fase de comunicación**: se usa un protocolo de comunicación diferente dependiendo del modo de cola. Si la cola está configurada como SPSC, el proceso del productor envía los artículos de manera asíncrona al consumidor. De lo contrario, si la cola está configurada en modo MPMC, el hilo del orquestador espera mensajes provenientes de productores y consumidores. Los productores envían mensajes `notify_item<id>` para informar sobre un nuevo artículo disponible, mientras que los consumidores envían `item_req<id>` para pedirle al orquestador un artículo. Cuando el orquestador recibe una nueva solicitud, sirve el artículo directamente o reenvía la solicitud a otro productor, siguiendo el mismo orden en el que llegaron las notificaciones del productor.

- **Fase de finalización**: Similar a la fase anterior, se usa un protocolo de terminal diferente dependiendo del modo de cola. Para el modo SPSC, tan pronto como el productor finaliza su operación, envía el mensaje *end-of-stream* (`EOS`) al consumidor. Por el contrario, para el modo MPMC, el orquestador

espera mensajes de `EOS` de todos los productores, incluido su propio elemento de terminación. Cuando esto sucede, el orquestador envía a cada uno de los consumidores el mensaje `EOS` para finalizar la comunicación.

Gracias a estas colas, los operadores de stream ejecutados por procesos MPI pueden transmitir elementos de acuerdo con el flujo de stream dictado por una construcción de Pipeline concreta, que puede estar compuesta por diferentes patrones de Farm. Tenga en cuenta que si un proceso utiliza varias instancias de cola simultáneamente, las etiquetas MPI, en lugar de múltiples comunicadores, se utilizan para hacer referencia a cada una de ellas.

4.2.3 Algoritmo de distribución de operadores en procesos MPI

La política de ejecución de MPI también incorpora un algoritmo de asignación para asignar operadores de flujo en procesos MPI. Básicamente, este algoritmo calcula al comienzo de la ejecución de la tubería la cantidad total de operadores involucrados en ella, considerando tanto el número de etapas como las réplicas de la granja. Luego, calcula la cantidad de operadores que se deben ejecutar por proceso de MPI (opp). Por defecto, los operadores se distribuyen homogéneamente usando la fórmula:

$$opp = \frac{num\_ops}{num\_procs} \tag{4.1}$$

Sin embargo, el usuario puede anular este valor llamando a la función `set_grouping_granularity(int ops_per_proc)`, parte de la clase de política de ejecución de MPI. A continuación, cada uno de los procesos de MPI calcula, utilizando su rango, el rango de operadores que deberían ejecutarse con la siguiente fórmula:

$$Range(rank) = \begin{cases} \{rank * opp, ((rank+1) * opp) - 1\} & \text{if } rank \neq num\_procs - 1 \\ \{rank * opp, num\_ops - 1\} & \text{if } rank = num\_procs - 1 \end{cases} \tag{4.2}$$

De acuerdo con esta fórmula, si el valor de opp se establece de manera predeterminada, es decir, con Ec. 4.1, todos los procesos ejecutan el mismo número de operadores excepto el último, que ejecuta todos los operadores restantes. De lo contrario, si opp ha sido establecido por el usuario y es mayor que el establecido por defecto, entonces algunos últimos procesos podrían no ejecutar ningún operador. Tenga en cuenta también que el algoritmo de distribución selecciona operadores consecutivos para mapearlos en procesos MPI, es decir, siguiendo el mismo orden en el que aparecen en el patrón de Pipeline.

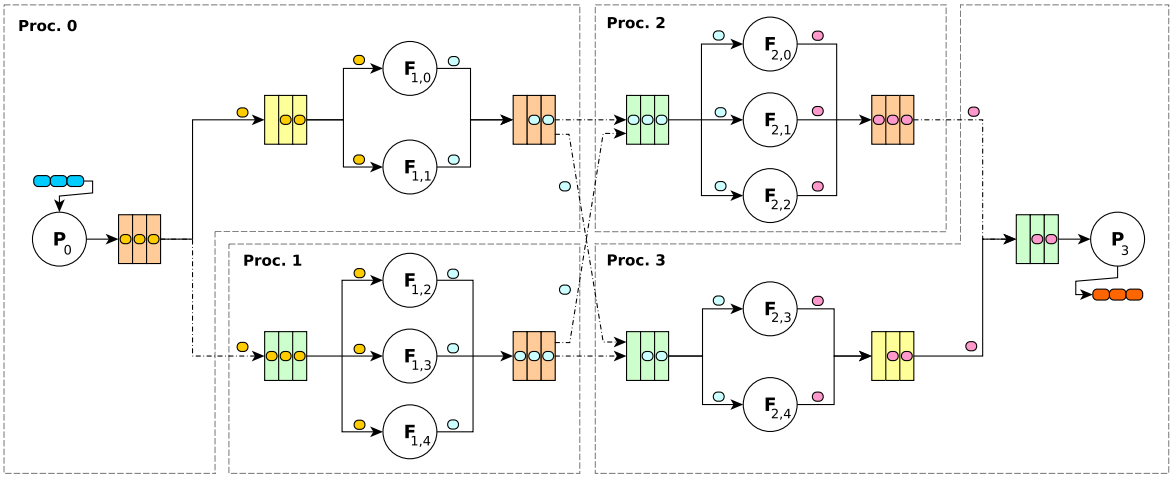


Figura 4-3: *Distribución de operadores de stream en procesos con* $opp = 3$.

La Figura 4-3 representa una tubería de cuatro etapas compuesta por dos patrones de granja en la segunda y tercera etapas, que se ejecuta con 5 réplicas cada una. En este caso, la granularidad de agrupación o *opp* se ha calculado de forma predeterminada utilizando Ec. 4.1. Dado que cada proceso ejecuta los 3 operadores consecutivos correspondientes a los devueltos por Ec. 4.2. Por lo tanto, el mismo proceso de MPI puede ejecutarse, utilizando la política de memoria compartida local, en etapas de Pipeline y/o réplicas de Farm.

En general, esta política de ejecución de MPI permite que los patrones GrPPI se ejecuten en plataformas distribuidas de múltiples núcleos, aprovechando el paralelismo inter e intranodo. Además, gracias a su algoritmo de distribución de operadores, es capaz de distribuir automáticamente operadores siguiendo el orden lógico de transmisión.

Capítulo 5

Evaluación experimental

Este capítulo presenta la evaluación experimental de GrPPI para plataformas de memoria distribuida. En primer lugar se describen los experimentos y la plataforma sobre la cual se han realizado (Sección 5.1). Posteriormente se realiza un estudio de usabilidad (Sección 5.2). Por último se realiza un análisis de rendimiento de la nueva implementación.

5.1 Descripción de los experimentos

En esta sección, llevamos a cabo una evaluación experimental del back-end GrPPI MPI desde los puntos de vista de la usabilidad y el rendimiento. Para esta evaluación, empleamos los siguientes componentes de hardware y software:

- **Plataforma de destino**. La evaluación se ha llevado a cabo en un clúster homogéneo de ocho nodos, cada uno compuesto por un procesador Intel Xeon Broadwell E5-2603 v4 con 12 cores a 1.70 GHz, 15 MB de caché L3 y 128 GB de RAM DDR3. El sistema operativo es Linux Ubuntu 16.04.3 LTS con el kernel 4.4.0-97. Los nodos están interconectados mediante un switch Ethernet de un Gigabit.

- **Software**. Aprovechamos el nuevo back-end GrPPI MPI construido sobre GrPPI v0.4 [10], junto con los respectivos back-end de memoria compartida y memoria distribuida, hilos de C ++11 y MPI-3.1, implementados por MPICH v3.2.1. Tenga en cuenta que el back-end de MPI se implementó con Boost MPI v1.66.0. El compilador de C ++ utilizado para ensamblar GrPPI fue GCC v7.2.0 que ya admite el estándar C ++ 14.

- **Caso de uso**. Para evaluar los patrones distribuidos de Pipeline y Farm, aprovechamos una aplicación de transmisión que se encarga del cómputo de Mandelbrot, de forma que se crean imágenes para montar una animación de zoom fractal y aplicar el filtro de desenfoque gaussiano (Blur filter) en cada uno de los fotogramas generados. Concretamente, esta aplicación basada en Pipeline

consta de las siguientes etapas:

Generador devuelve valores de zoom crecientes monótonamente lineales, que se pasan a la etapa de Mandelbrot.

Mandelbrot toma el zoom recibido por el generador y calcula la trama de Mandelbrot correspondiente a dicho valor de zoom y las coordenadas del punto de interés.

Gaussian blur toma el frame calculado por la etapa anterior y aplica el filtro de desenfoque gaussiano a ellos, usando un kernel de 3×3 píxels.

Consumidor guarda cada uno de los frames en el disco usando el formato BMP.

Listado 5.1: *Implementación Mandelbrot en Boost MPI.*

```
int frame= 0;
int num_frames= 1000;
// Main Pipeline
grppi::Pipeline(mpi_exec,
  // Zoom generation
  [&]() -> std::experimental::optional<double> {
    if (frame++ == num_frames) return {};
      zoom-= zoom * 0.1;
    return zoom;
  },
  // Farm mandelbrot stage
  grppi::Farm(4,
    [&](auto zoom){
      return mandelbrot(height, width, poi_x, poi_y, zoom);
    }
  ),
  // Farm blur stage
  grppi::Farm(4,
    [&](auto image){
      return blur(height, width, kernel, image);
    }
  ),
  // Consuming stage
  [&](auto image){
    save_bmp(height, width, image);
  }
);
```

Como se sabe generalmente, el conjunto de Mandelbrot es el conjunto de números complejos c para los cuales la función $f_c(z) = z^2 + c$ no diverge cuando se itera desde $z = 0$, es decir, para el cual la secuencia $f_c(0)$, $f_c(f_c(0))$, etc., permanece limitada en valor absoluto. Por lo tanto, las imágenes de Mandelbrot pueden crearse muestreando los números complejos y determinando, para cada punto de muestra c, si el resultado de iterar la función anterior va al infinito. Luego, tratando las partes real e imaginaria de c como coordenadas de imagen $(x + yi)$ en el plano complejo, los píxeles pueden colorearse de acuerdo con la rapidez con la secuencia $z_n^2 + c$ diverge. Esta es la clave para comprender la naturaleza

heterogénea de esta operación, donde la velocidad de divergencia en el punto de interés y el valor del zoom dictan el número de iteraciones para calcular un frame dado. Por el contrario, el operador de desenfoque gaussiano tiene una carga de trabajo casi homogénea para procesar cada uno de los frames. En nuestro caso de uso específico, el caso de uso de Mandelbrot se configuró para generar 1,000 frames y hasta 1,000 iteraciones para calcular cada frame del Mandelbrot.

Listado 5.2: *Implementación Mandelbrot en GrPPI MPI.*

```
std::vector<color> image;
if (world.rank() == 0) { // Zoom generator
  for(int frame= 0; frame < num_frames; frame++) {
    zoom-= zoom * 0.1;
    world.send(world.rank()+1, 0, zoom);
  }
}
else if (world.rank() == 1) { // Mandelbrot stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, zoom);
    image = mandelbrot(height, width, poi_x, poi_y, zoom);
    world.send(world.rank()+1, 0, image);
  }
}
else if (world.rank() == 2) { // Blur stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, image);
    image = blur(height, width, kernel, image);
    world.send(world.rank()+1, 0, image);
  }
}
else if (world.rank() == 3) { // Consuming stage
  for(int frame= 0; frame < num_frames; frame++) {
    world.recv(world.rank()-1, 0, image);
    save_bmp(height, width, image);
  }
}
```

Dado el paralelismo natural de esta aplicación, donde cada uno de los fotogramas de animación se puede calcular de forma independiente, las etapas del Pipeline Mandelbrot y Gaussian blur pueden ser replicadas mediante el patrón Farm, por lo que pueden procesar fotogramas individuales. Por lo tanto, pueden surgir varias composiciones dependiendo de la paralelización de estas etapas, es decir, `(p|p|p|p)`, `(p|f|p|p)`, `(p|p|f|p)`, y `(p|f|f|p)`.

En las secciones siguientes, analizamos la usabilidad y el rendimiento de los patrones GrPPI Pipeline y Farm distribuidos utilizando el punto de referencia mencionado anteriormente con diferentes configuraciones de grado de paralelismo con respecto al número de procesos MPI y subprocesos de trabajo utilizados en las etapas de Farm. Además, evaluamos los costes del uso de GrPPI con respecto a la paralelización directa de la aplicación a través de MPI.

5.2 Estudio de la usabilidad

Para analizar la usabilidad y la productividad de la interfaz de patrones propuesta y el nuevo backend de MPI, utilizamos la herramienta analizadora Lizard [53] para obtener dos métricas bien conocidas: Líneas de código (LOCs) y el Número de Complejidad Ciclomática de McCabe's (CCN) [54]. Básicamente, utilizamos estas métricas para analizar las diferentes versiones de casos de uso, es decir, con y sin utilizar la interfaz GrPPI. La Tabla 5.1 resume el porcentaje de LOCs adicionales introducidos en el código fuente secuencial para implementar las versiones paralelas usando MPI y la interfaz GrPPI, junto con sus CCNs correspondientes. Como se observa, la implementación de composiciones más complejas a través de MPI conduce a códigos fuente más grandes y más complejos, mientras que para GrPPI el número de LOCs radicionales permanece constante. Esta diferencia se debe principalmente a las colas de comunicación requeridas para implementar el patrón Farm. Centrándonos en GrPPI, observamos que el esfuerzo de paralelización es casi insignificante: incluso la composición más compleja aumenta casi el 4.2 % de LOCs. Además, al cambiar a GrPPI para que use una política de ejecución particular, solo se necesita cambiar un único argumento en la llamada a la función de patrón. Con respecto a la complejidad ciclomática para MPI, observamos que sus CCNs son aproximadamente proporcionales al aumento del porcentaje de LOCs. . Por el contrario, la interfaz GrPPI tiene CCN constantes para todas las composiciones de Pipeline.

Tabla 5.1: *Porcentaje de LOCs adicionales con respecto a la versión secuencial y los CNNs para las composiciones de Pipeline.*

Composición Pipeline	**% LOCs adicionales**		**CCN**	
	MPI	GrPPI	MPI	GrPPI
`(p|p|p|p)`	+10.3 %	+4.2 %	9	7
`(p|f|p|p)`	+130.4 %	+4.2 %	38	7
`(p|p|f|p)`	+130.4 %	+4.2 %	38	7
`(p|f|f|p)`	+185.8 %	+4.2 %	58	7

Finalmente, realizamos una comparación lado a lado entre las interfaces Boost MPI y GrPPI implementando una versión simplificada del caso de uso de Mandelbrot. Como se puede ver en el Listado 5.1, la implementación de MPI distingue claramente las instrucciones y comunicaciones que deben ser ejecutadas por cada uno de los procesos. [1]. Por otro lado, el código GrPPI, que se muestra en el Listado 5.2, se centra más en la estructura del algoritmo de la aplicación que en las comunicaciones entre procesos. En pocas palabras, aunque ambas interfaces proporcionan interfaces de alto nivel, concluimos que las implementaciones de patrones ofrecidas por GrPPI ayudan a mejorar tanto la productividad como la

[1]Para mayor simplicidad, hemos reemplazado el uso de colas por comunicaciones síncronas, por lo que no hay cálculos que se superpongan a las comunicaciones en este caso.

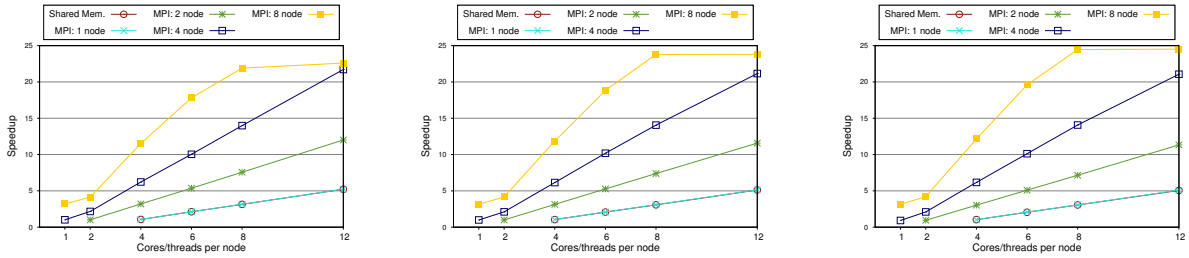
capacidad de mantenimiento.

5.3 Análisis de rendimiento

En esta sección, evaluamos el rendimiento y la escalabilidad de la aplicación Mandelbrot implementada con la interfaz GrPPI utilizando la política MPI junto con el back-end de memoria compartida basado en hilos C ++ 11. Evaluamos el rendimiento de la aplicación utilizando solo el back-end de memoria compartida. Además, hemos limitado la cantidad de núcleos por nodo para simular diferentes escenarios de clúster. Tenga en cuenta que solo nos centramos en la composición del Pipeline `(p|f|f|p)`, ya que ofrece el mejor rendimiento y puede escalarse mejor entre los nodos del clúster.

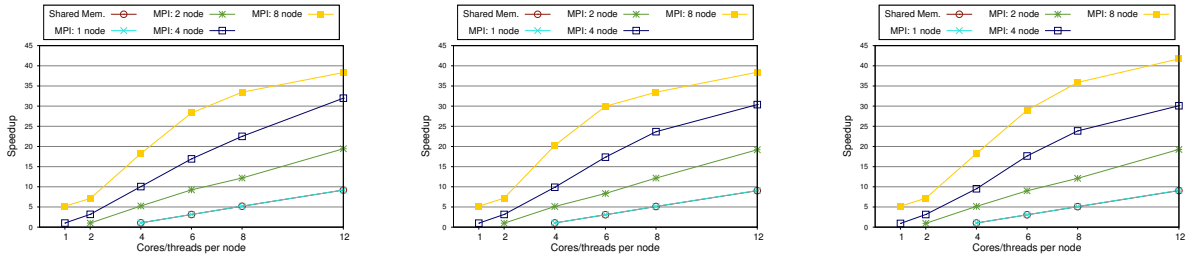
La Figura 5-1 representa la escala de speedup cuando se usa de 1 a 8 nodos de clúster y se ejecuta de 1 a 12 subprocesos por nodo para resoluciones de cuadros cuadrados de 200, 400 y 800 píxeles con respecto a la aplicación secuencial. Es importante destacar que cada experimento ejecuta tantos operadores (etapas de Pipeline y réplicas de Farm) como la cantidad total de subprocesos ejecutados en los nodos. Como se observa, el speedup logrado por MPI usando un solo nodo es igual a la memoria compartida; esto viene dado por el hecho de que el back-end MPI delega completamente en el back-end de la memoria compartida cuando se ejecuta en un nodo. En este primer intento del experimento, establecemos el mismo número de réplicas de Farm en las etapas de Mandelbrot y de Gaussian Blur. Como puede verse, para 1, 2 y 4 nodos, la aplicación escala linealmente con el número de subprocesos por nodo, con una eficiencia sostenida de aproximadamente 45 %. Sin embargo, para 8 nodos, el rendimiento se degrada desde 6 subprocesos por nodo, ya que la etapa de Mandelbrot causa un importante cuello de botella en la interconexión debido a los niveles de producción desequilibrados. La eficiencia máxima, en este caso, es 26 %. Esto se debe a que la carga de trabajo de Mandelbrot por cada frame es mucho mayor que la aplicación del operador Gaussian Blur.

Para lidiar con este desequilibrio, hemos calculado empíricamente la relación entre los rendimientos de las etapas Mandelbrot y Blur, que nos sirvió para determinar el número óptimo de réplicas en las etapas correspondientes de Farm. Básicamente, hemos calculado esta relación usando promedios de rendimiento de ambas etapas para las diferentes resoluciones de frame. Usando esta proporción, asignamos 1 réplica de Blur por cada 25 réplicas de Mandelbrot. Sin embargo, esta relación no ofrece rendimientos ideales dada la naturaleza heterogénea de la carga de trabajo del conjunto de Mandelbrot. La Figura 5-2 muestra el rendimiento logrado para la diferente cantidad de nodos y subprocesos por experimentos de nodo y el uso de la relación antes mencionada. Como se observa, de 1 a 4 nodos, la eficiencia mejora significativamente con respecto al experimento anterior, de 45 % a 68 %. Los resultados también revelan escalas de rendimiento relativamente buenas para todos los tamaños de frame. Por otro lado, para 8 nodos, la aplicación muestra un peor rendimiento cuando se utilizan más de 6 subprocesos por núcleo. Esta degradación se debe principalmente a la diferencia entre la carga de trabajo de la etapa y los gastos



(a) *Resolución de imagen 200×200.* (b) *Resolución de imagen 400×400.* (c) *Resolución de imagen 800×800.*

Figura 5-1: *Caso de uso de Mandelbrot con Pipeline* `(p|f|f|p)` *con mismo número de réplicas de Farm.*



(a) *Resolución de imagen 200×200.* (b) *Resolución de imagen 400×400.* (c) *Resolución de imagen 800×800.*

Figura 5-2: *Caso de uso de Mandelbrot con Pipeline* `(p|f|f|p)` *con un número ajustado de réplicas de Farm.*

generales de comunicación inherentes. Por esta razón, para tamaños de cuadros grandes, el rendimiento alcanzado es ligeramente mejor, ya que los cálculos paralelos pagan los tiempos de serialización y transferencia de datos requeridos. En cualquier caso, para 8 nodos, la eficiencia obtenida usando etapas de Farm equilibradas aumenta de 26 % a 45 %.

A partir de estos experimentos, podemos concluir que la interfaz GrPPI propuesta puede ayudar a implementar aplicaciones científicas de flujo distribuido a expensas de gastos indirectos insignificantes. En un experimento separado, evaluamos la sobrecarga introducida por GrPPI con respecto a utilizar MPI directamente. Esta sobrecarga fue menor a 0.1 %.

Capítulo 6

Planificación del proyecto

Este capítulo presenta una planificación detallada del proyecto. Este proyecto ha sido realizado en colaboración con el grupo de investigación ARCOS de la Universidad Carlos III de Madrid durante la realización del Máster en Ciencia y Tecnología Informática de la misma universidad. El proyecto comenzó el 8 de enero de 2018 y finalizó el 5 de septiembre de 2018, con un total de 9 meses de trabajo.

Dado que este proyecto de investigación incluye una parte de desarrollo, ambos lados, la investigación y el desarrollo tuvieron que fusionarse para obtener una metodología de investigación y planificación adecuadas. La metodología aplicada se muestra en la Figura6-1.

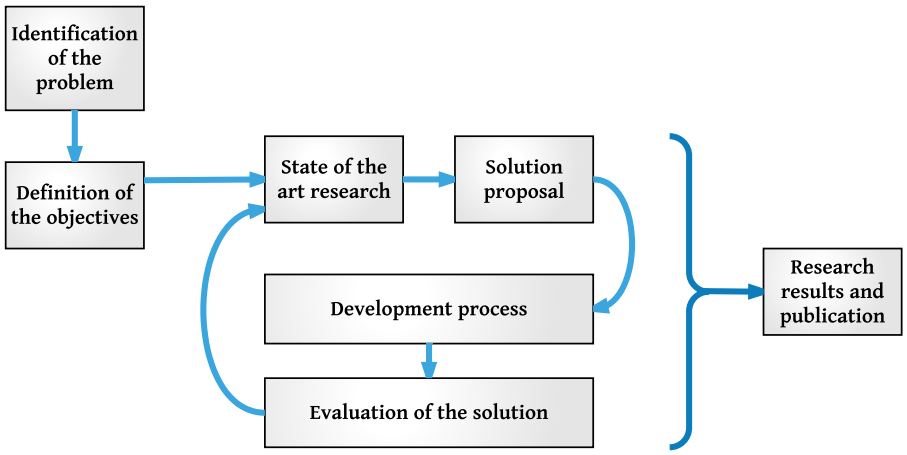


Figura 6-1: *Metodología de investigación empleada.*

Donde:

- **Identificación del problema** que estamos tratando de resolver.
- **Definición de los objetivos** del proyecto.

- **Estado de la cuestión a investigar**: sintetiza el más alto nivel del campo científico.
- **Propuesta de solución**: diseño y definición de la solución.
- **Proceso de desarrollo**: de la solución propuesta.
- **Evaluación de la solución**: analizar diferentes casos de estudio.
- **Resultados de investigación y publicación**: publica los resultados de la tesis de máster en varias revistas y conferencias.
- **Documentación de la tesis**: escribe el informe final.

El diagrama de Gantt (Figura 6-2) muestra todas las tareas realizadas durante el desarrrollo de este trabajo.

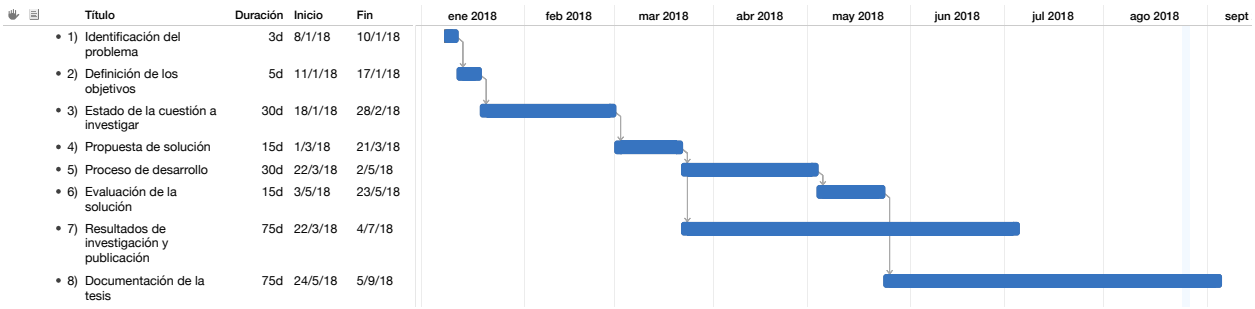


Figura 6-2: *Diagrama de Gantt.*

Capítulo 7

Conclusiones y trabajo futuro

Este capítulo presenta las conclusiones de este proyecto y el trabajo futuro. En primer lugar se exponen las conclusiones y los resultados obtenidos (Sección 7.1). Por último se expone el trabajo futuro a realizar en esta línea de investigación.

7.1 Conclusiones

En este trabajo, hemos ampliado GrPPI, una interfaz de patrones paralelos genérica y reutilizable, con un nuevo back-end MPI, que permite la ejecución de los patrones Pipeline y Farm en plataformas distribuidas. Para admitir escenarios híbridos, el back-end también combina una política de ejecución de memoria compartida intranodo que, si es necesario, se utiliza para ejecutar múltiples operadores de Pipeline y/o operadores de Farm dentro del mismo proceso MPI. En general, el diseño compacto de GrPPI facilita el desarrollo de aplicaciones de transmisión de datos, mejorando la flexibilidad y la portabilidad, a la vez que explota el paralelismo inter e intranodo.

Como se demostró a lo largo de la evaluación experimental, el caso de uso de Mandelbrot, implementado con patrones de Pipeline y Farm distribuidos, logra ganancias de speedup considerables en comparación con la versión secuencial correspondiente. En cualquier caso, siempre es importante equilibrar las etapas de Pipeline de acuerdo con las cargas de trabajo de la etapa. También demostramos que el aprovechamiento de GrPPI reduce considerablemente el número de LOCs y la complejidad ciclomática con respecto a la implementación utilizando directamente MPI. Además, gracias a la comparación cualitativa de las dos interfaces de alto nivel, GrPPI y MPI, concluimos que GrPPI conduce a códigos más estructurados y legibles, y por lo tanto, mejora su mantenibilidad general. En general, la implementación de un back-end distribuido mediante MPI se ha motivado principalmente por las necesidades de escalamiento y el desarrollo de nuevos modelos de programación para aplicaciones científicas DaSP. Además, nuestro interés con este back-end proviene de la amplia adopción de MPI en las supercomputadoras

de hoy, que actualmente no tiene soporte estándar para el procesamiento de streaming [36]. Por estas razones, creemos que el back-end presentado puede ser de gran ayuda en el desarrollo de aplicaciones de streaming en C ++.

Por lo tanto, podemos concluir que se han logrado los objetivos citados en comienzo del documento:

- **O1**: Hemos presentado una nueva política de ejecución de GrPPI-MPI para entornos distribuidos e híbridos para los patrones paralelos *Pipeline* y *Farm*.
- **O2**: Se ha descrito el diseño de la interfaz GrPPI y las políticas internas de MPI para permitir la ejecución distribuida e híbrida de aplicaciones DaSP.
- **O3**: Se ha presentado un nuevo operador para patrones de *streaming* que, como un contenedor, permite a los usuarios reemplazar la política de ejecución predeterminada de un patrón.
- **O4**: Hemos analizado la usabilidad del patrón en términos de líneas de código y complejidad ciclomática, y realizando una comparación *side-by-side* de ambas interfaces de programación GrPPI y MPI.
- **O5**: Se han evaluado los patrones distribuidos *Pipeline* y *Farm* de *streaming* desde los puntos de vista de usabilidad y rendimiento usando una aplicación que renderiza frames de Mandelbrot bajo diferentes configuraciones híbridas.

Además, este trabajo ha dado como resultado una publicación en la conferencia internacional EuroMPI 2018 que se celebrará en Septiembre en Barcelona (España).

- **Supporting MPI-Distributed Stream Parallel Patterns in GrPPI**[55]. *Javier Fernández Muñoz, Manuel F. Dolz, David del Rio Astorga, Javier Prieto Cepeda and J. Daniel García*, EuroMPI 2018, Barcelona.

7.2 Trabajo futuro

Como trabajo futuro, planeamos admitir nuevos algoritmos para la distribución de operadores en procesos e introducir un nuevo operador para permitir a los usuarios reemplazar la política de ejecución predeterminada de un operador concreto en una etapa de Pipeline. También planeamos implementar otros patrones de streaming, como filter o Stream-Reduce, dentro de la política de ejecución de MPI y mejorar las colas de comunicación para usar comunicaciones MPI de un solo lado.

Acrónimos

API Application Programming Interface. 12–14

ARCOS Grupo de Aquitectura de Computadores, Universidad Carlos III de Madrid. 49

CPU Central Processing Unit. 15–17

CUDA Compute Unified Device Architecture. 17, 24

DaSP Data Stream Processing. 1, 2, 35, 38, 51, 52

GPGPU General-Purpose Computing on Graphics Processing Units. 16

GPU Graphics Processing Unit. 11, 16, 17

GrPPI Generic Reusable Parallel Pattern Interface. 2, 3, 16, 19, 24–29, 31, 32, 35–38, 42, 43, 45–48, 51, 52

HPC High-Performance Computing. 1, 2, 14, 15, 35

MIT Massachusetts Institute of Technology. 13

MPI Message Passing Interface. 1, 2, 14–17, 35–39, 41–48, 51, 52

NUMA Non-Uniform Memory Access. 6, 7

OpenMP Open Multi-Processing. 1, 2, 13, 15–17, 24, 25, 29, 30

RAM Random-Access Memory. 13, 37, 39, 41, 43

TBB Threading Building Blocks. 2, 13, 16, 17, 24, 36

UMA Uniform Memory Access. 6, 7

VLSI Very Large Scale Integration. 5

Bibliografía

[1] G. Fox, S. Jha, and L. Ramakrishnan, "Stream2016: Streaming requirements, experience, applications and middleware workshop," tech. rep., Lawrence Berkeley National Laboratory, 10 2016.

[2] S. Kamburugamuve, S. Ekanayake, M. Pathirage, and G. Fox, "Towards high performance processing of streaming data in large data centers," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, (Chicago, USA), pp. 1637–1644, IEEE, May 2016.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Massachusetts, USA: The MIT Press, 2014.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[5] F. Cappello and D. Etiemble, "Mpi versus mpi+openmp on ibm sp for the nas benchmarks," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, (Washington, DC, USA), pp. 12–12, IEEE Computer Society, 2000.

[6] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2012.

[7] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2013.

[8] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Münster Skeleton Library Muesli: A comprehensive overview," ERCIS Working Papers 7, University of Münster, European Research Center for Information Systems (ERCIS), 2009.

[9] A. Ernstsson, L. Li, and C. Kessler, "Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, vol. 46, pp. 62–80, Feb 2018.

[10] C. Computer Architecture and S. (ARCOS), "Generic Reusable Parallel Pattern Interface - GrPPI." `https://github.com/arcosuc3m/grppi/`, 2018. Online; accessed 5 May 2018.

[11] A. C. Sodan, "Message-passing and shared-data programming models-wish vs. reality," in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pp. 131–139, IEEE, 2005.

[12] B. Nitzberg and V. Lo, "Distributed shared memory: A survey of issues and algorithms," *Computer*, vol. 24, no. 8, pp. 52–60, 1991.

[13] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International*, pp. 149–160, IEEE, 2007.

[14] P. Rogers and C. FELLOW, "Amd heterogeneous uniform memory access," *AMD Whitepaper*, 2013.

[15] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, p. 40, 2013.

[16] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 15–26, IEEE, 1990.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the dash multiprocessor," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pp. 148–159, IEEE, 1990.

[18] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *ACM SIGARCH Computer Architecture News*, vol. 32, p. 102, IEEE Computer Society, 2004.

[19] S. Kleiman, D. Shah, and B. Smaalders, *Programming with threads*. Sun Soft Press Mountain View, 1996.

[20] S. Lee and J. S. Vetter, "Early evaluation of directive-based gpu programming models for productive exascale computing," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pp. 1–11, IEEE, 2012.

[21] H. E. Bal and M. Haines, "Approaches for integrating task and data parallelism," *IEEE concurrency*, no. 3, pp. 74–84, 1998.

[22] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[23] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. .ºReilly Media, Inc.", 2007.

[24] M. Schmidberger, M. Morgan, D. Eddelbuettel, H. Yu, L. Tierney, and U. Mansmann, "State-of-the-art in parallel computing with r," *Journal of Statistical Software*, vol. 47, no. 1, 2009.

[25] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[26] E. E. Hagersten and P. N. Loewenstein, "Multiprocessing computer system employing local and global address spaces and coma and numa access modes," Mar. 23 1999. US Patent 5,887,138.

[27] J. Rekimoto and M. Saitoh, "Augmented surfaces: a spatially continuous work space for hybrid computing environments," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 378–385, ACM, 1999.

[28] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pp. 427–436, IEEE, 2009.

[29] B. Nichols, D. Buttlar, J. Farrell, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. .ºReilly Media, Inc.", 1996.

[30] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," *Talk at HP-CAST*, vol. 18, p. 25, 2012.

[31] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, vol. 30. ACM, 1995.

[32] W. D. Gropp, W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[33] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[34] T. White, *Hadoop: The definitive guide*. .ºReilly Media, Inc.", 2012.

[35] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," in *Compiler Construction* (R. N. Horspool, ed.), (Berlin, Heidelberg), pp. 179–196, Springer Berlin Heidelberg, 2002.

[36] I. B. Peng, S. Markidis, R. Gioiosa, G. Kestor, and E. Laure, "MPI streams for HPC applications," *CoRR*, vol. abs/1708.01306, 2017.

[37] I. B. Peng, S. Markidis, E. Laure, D. Holmes, and M. Bull, "A data streaming model in mpi," in *Proceedings of the 3rd Workshop on Exascale MPI*, ExaMPI '15, (New York, NY, USA), pp. 2:1–2:10, ACM, 2015.

[38] K. Hamidouche, J. Falcou, and D. Etiemble, "A framework for an automatic hybrid mpi+openmp code generation," in *Proceedings of the 19th High Performance Computing Symposia*, HPC '11, (San Diego, CA, USA), pp. 48–55, Society for Computer Simulation International, 2011.

[39] Q. Meng, M. Berzins, and J. Schmidt, "Using hybrid parallelism to improve memory use in the uintah framework," in *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, (New York, NY, USA), pp. 24:1–24:8, ACM, 2011.

[40] B. Goglin and S. Moreaud, "Knem: A generic and scalable kernel-assisted intra-node mpi communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, 2013.

[41] T. Miyoshi, H. Irie, K. Shima, H. Honda, M. Kondo, and T. Yoshinaga, "Flat: A gpu programming framework to provide embedded mpi," in *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, (New York, NY, USA), pp. 20–29, ACM, 2012.

[42] T. Shimokawabe, T. Aoki, and N. Onodera, "High-productivity framework on gpu-rich supercomputers for operational weather prediction code asuca," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, (Piscataway, NJ, USA), pp. 251–261, IEEE Press, 2014.

[43] A. Marco, D. Marco, K. Peter, and T. Massimo, *Fastflow: High-Level and Efficient Streaming on Multicore*, ch. 13, pp. 261–280. Wiley-Blackwell, 2017.

[44] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pp. 836–838, IEEE, 2008.

[45] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[46] R. Loogen, Y. Ortega-mallén, and R. Peña marí, "Parallel functional programming in eden," *J. Funct. Program.*, vol. 15, pp. 431–475, May 2005.

[47] J. F. Ferreira, J. L. Sobral, and A. J. Proenca, "Jaskel: a java skeleton-based framework for structured cluster and grid computing," in *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1, (Singapore), pp. 4 pp.–304, IEEE, May 2006.

[48] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, (New York, NY, USA), ACM, 2006.

[49] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The münster skeleton library muesli: A comprehensive overview," Working Papers, ERCIS - European Research Center for Information Systems 7, Westf. Wilhelms-Univ., Münster, 2009.

[50] E. Alba, G. Luque, J. Garcia-Nieto, G. Ordonez, and G. Leguizamon, "MALLBA: a Software Library to Design Efficient Optimisation Algorithms," *Int. J. Innov. Comput. Appl.*, vol. 1, pp. 74–85, Apr. 2007.

[51] K. Fuerlinger, T. Fuchs, and R. Kowalewski, "Dash: A c++ pgas library for distributed data structures and parallel algorithms," in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, (Sydney, Australia), pp. 983–990, IEEE, Dec 2016.

[52] D. Gregor and M. Troyer, "Boost MPI," 2017.

[53] Terry Yin, "Lizard: an Cyclomatic Complexity Analyzer Tool." `https://github.com/terryyin/lizard`, 2018. Online; accessed 5 May 2018.

[54] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 308–320, July 1976.

[55] J. F. Muñoz, M. F. Dolz, D. del Rio Astorga, J. Prieto-Cepeda, and J. D. García, "Supporting mpi-distributed stream parallel patterns in grppi," *EuroMPI*, September 2018.