TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Desarrollo de herramientas de gestión para clústeres basados en ARM

Curso: 2020/2021

Alumno/a:

Jerónimo Sánchez García

Director/es:

Juana López Redondo

Gracia Ester Martín Garzón

A mi familia, amigos y mi Rosita.

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mis tutoras, Juani y Ester, no solo por guiarme a lo largo de este trabajo, sino por mostrarme la beca **Summer of HPC** y apoyarme en su solicitud. Gracias a esta oportunidad he podido comprobar que mi lugar está en la investigación y en la innovación.

Agradezco al EPCC, en especial a Nick Johnson por enseñarme y guiarme a lo largo de mi estancia en este centro, y a la Drª Michèle Weiland, por cederme las imágenes originales del problema de afinidad.

También quiero agradecer a mis familiares y amigos por no molestarse con mi desaparición de los últimos meses y comprender que esto no es más que algo temporal. Nos queda poco para volver a reunirnos.

Finalmente, quiero agradecer a mi pareja por interesarse y comprender las ideas de este trabajo para así ayudarme en los momentos que más me ha costado expresarme, dándome consejos muy útiles; y a mi hermano, por ayudarme con la creación de algunas de las figuras.

ÍNDICE GENERAL

Página

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

LISTADOS

Resumen y Abstract

Este trabajo explora y da solución software a dos de los problemas que surgen de usar un supercomputador basado en procesadores ARM para la Computación de Alto Rendimiento (HPC). En concreto, se tratan los problemas relacionados con la afinidad de tareas y la automatización de informes de rendimiento de red. Los desarrollos y las evaluaciones de las herramientas se han realizado gracias al apoyo ofrecido por la Universidad de Almería y la organización PRACE con su programa *Summer of HPC*. Esta beca ha permitido al autor a participar en proyectos HPC del EPCC [1]. Parte de los resultados obtenidos en este TFG han sido publicados en el Congreso Español De Informática (CEDI) [2].

This work studies and solves, through software, two of the problems that arise from using an ARM-based supercomputer for High-Performance Computing (HPC). To be precise, it targets the issues relating to task affinity and the automation of a network's performance reports. The development and evaluation of such tools has been carried out thanks to the support of the University of Almería and the Summer of HPC program, organized by PRACE. This scholarship has allowed the author to participate in HPC projects of the EPCC [1]. A subset of the results obtained in this TFG have been published in the Spanish Computer Science Congress (CEDI) [2].

ABREVIATURAS

API	Application Programming Interface
CEDI	Congreso Español De Informática
CF	CPU Física
CISC	Complex Instruction Set Computer
CL	CPU Lógica
CLI	Command Line Interface
CPU	Central Processing Unit
EPCC	Edinburgh Parallel Computing Centre
GPU	Graphics Processing Unit
HPC	High Performance Computing
HT	HyperThreading
HWT	HardWare Thread
IDE	Integrated Development Enviroment
ILP	Instruction Level Parallelism
IO	Input/Output
ISA	Instruction Set Architecture
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
RISC	Reduced Instruction Set Computer
SMT	Simultaneous MultiThreading
TFG	Trabajo Fin de Grado
TLP	Task Level Parallelism
UMA	Uniform Memory Access
UML	Unified Modeling Language
VCS	Version Control System
XP	eXtreme Programming

1 Introducción

A lo largo de este capítulo se describen tanto el contexto como la justificación de este trabajo. Además, se muestran los objetivos en los que el trabajo se descompone, la planificación temporal de estos y la metodología de desarrollo seguida. Finalmente, se mencionan las herramientas software utilizadas para la creación de las herramientas de este trabajo, como las utilizadas para la comunicación.

1.1 Contexto y justificación del trabajo

A mediados de 1978, Intel lanza su procesador **Intel 8086**, un procesador CISC (*Complex Instruction Set Computer*) de 16-bits y primer procesador **x86**. De hecho, el término **x86** aparece debido a que los sucesores de este procesador terminan todos en "*86*". En la actualidad, el término **x86** se refiere a un ISA (*Instruction Set Architecture*) compatible con el del procesador **Intel 80386** (i386), un procesador de 32-bits lanzado en 1985 [3].

La popularización de esta arquitectura como líder en la fabricación de procesadores surge en 1981, cuando IBM creó el **IBM PC**, uno de los computadores personales más populares. Para este PC (*Personal Computer*), IBM escogió el procesador **Intel 8088**, una variante del **Intel 8086** con un bus de 8-bits en vez del bus de 16-bits del **8086** [4]. Como parte del trato con Intel, esta compañía se comprometía a la existencia de otro fabricante de los chips **8086** y **8088**, para que así IBM no dependiera únicamente de Intel para la fabricación de los chips de su computador. AMD (Advanced Micro Devices) fue la compañía elegida para este trato entre Intel e IBM. Entre los fabricantes de silicio (Intel y AMD), se firmó un acuerdo por el que cualquiera de las partes que quisiera fabricar un componente de la otra tendría que ceder un componente suyo a modo de intercambio. Unos años más tarde, en 1984, ambas compañías modificaron este acuerdo para permitir que AMD pudiera fabricar los sucesivos procesadores que conforman **x86** a cambio de *royalties* (tasas de regalías).

A los pocos años del lanzamiento del **IBM PC**, este fue perdiendo su influencia en el mercado a favor de los clones. A pesar de ello, Intel vio como su arquitectura se convertía en "estándar", ya que dichos clones usaban chips Intel compatibles con el **8088**. Esto provocó que Intel no tuviera motivos para continuar su trato con AMD. En un principio, Intel se negó a ceder su tecnología existente a AMD basándose en el pretexto de que no querían tecnología de esta empresa a cambio. Por ello, AMD pidió arbitraje acerca del contrato que tenía firmado con Intel [5].

La estrategia de Intel de negarse a ceder nueva tecnología a AMD, específicamente información del procesador **Intel 80386**, provocó que AMD estuviera fuera del mercado en un

momento de pleno crecimiento gracias a la popularización de los clones del **IBM PC**. AMD, con la intención de volver a ser competitiva, realizó ingeniería inversa al i386 y creó su propio clon compatible: el **AM386**. Durante este periodo, el arbitraje pedido por AMD se resolvió a favor de esta compañía. La resolución indicaba que Intel tenía que indemnizar a AMD con más de 10 millones de dólares y con acceso permanente y sin *royalties* a cualquier patente de Intel contenida dentro la versión de AMD realizada mediante ingeniería inversa del i386 [6].

Ya a las puertas del lanzamiento comercial del **AM386**, Intel demandó a AMD por el nombre de este procesador, lo que bloqueó la salida de este al mercado hasta que el conflicto estuviera resuelto. Tras el pleito, AMD pudo lanzar dicho procesador, ya que se determinó que Intel no puede proteger un número como marca registrada [7]. Este fue un éxito comercial debido a que era más barato e igual de rápido que las alternativas de Intel. Finalmente, en 1995, ambas compañías acordaron un alto el fuego judicial mediante el que AMD recibiría una licencia perpetua al microcódigo de los procesadores Intel 80386 y 80486 (i486) pero no podría copiar ningún microcódigo más de Intel [5].

Durante todo este periodo, a **x86** se añadieron nuevas extensiones, siempre de una manera que respetase la compatibilidad hacia atrás (*backwards compatibility*), permitiendo de este modo a **x86** obtener mejoras y modernizarse. Una de estas extensiones, la creada por **AMD** en 1999, llamada **x86-64** (conocida como **AMD64**), permite al procesador usar una mayor cantidad de memoria virtual que cualquiera de las extensiones de **x86** desarrolladas hasta el momento. Tras ello, Intel desarrolló su propia extensión **x86-64** llamada **Intel 64** con el objetivo de competir con la extensión desarrollada por AMD.

Paralelamente a este desarrollo entre Intel y AMD respecto a **x86**, en 1985, nace el procesador **ARM1**, un procesador RISC (*Reduced Instruction Set Computer*) de 32-bits y primer procesador **ARM**. Este fue desarrollado por Acorn Computers, compañía que pasó a llamarse ARM Ltd.

Este procesador **ARM1** surge como un coprocesador del computador **BBC Micro** [8], aunque dicho procesador nunca fue comercializado. Posteriormente, a finales del 1986, el **ARM2** fue comercializado. Este incluía nuevas características, como un nuevo algoritmo para la multiplicación de enteros, nuevos modos de interrupción y 27 registros de propósito general. El rendimiento de este procesador es equivalente al del **Intel 80386**.

Al poco tiempo de la salida de dicho procesador, la división encargada del diseño de este se separó de la compañía matriz para formar otra que pudiera colaborar de una manera mucho más efectiva con Apple. Esta compañía es **ARM Ltd.** Esta no se dedica a la fabricación y venta de chips como Intel y AMD, sino que ofrece licencias por el uso de su ISA para que así, el cliente pueda crear chips compatibles con ARM enfocados a sus necesidades. Además, **ARM Ltd.** ofrece el diseño de sus núcleos de referencia, los núcleos Cortex, para que el cliente pueda agilizar el diseño del procesador.

Desde la creación de esta nueva compañía, se han diseñado diferentes versiones del ISA inicial ARMv1, diferentes extensiones de este ISA y nuevos ISAs compatibles con ARM. Uno de los ISA compatibles con ARM es **Thumb**. Este es un subconjunto del ISA ARM de 16-bits. Las instrucciones de estas tienen equivalencia directa con las presentadas por ARM. Esta reducción en el tamaño del ISA viene dada porque estas instrucciones usan de manera

implícita operandos en las instrucciones ARM, lo que favorece que el procesador tenga un menor consumo energético para operaciones simples.

Otra de las técnicas implementadas para conseguir un ahorro energético es la creación de una arquitectura de procesadores heterogénea llamada **big.LITTLE**. Esta combina, en el mismo procesador, núcleos relativamente pequeños para ahorrar consumo (**LITTLE**), con otros mucho más grandes y potentes, pero con mayor gasto energético (**big**). La idea es crear un procesador multinúcleo que pueda ajustarse mejor a las necesidades de computación dinámica y utilizar menos energía. Como resultado, los procesadores **ARM** se han convertido en los dominantes dentro del mercado de la electrónica móvil e integrada.

Dicho esto, las mejoras que ARM brinda no solo sirven para dispositivos con limitaciones de consumo, sino que también son muy interesantes para otros contextos como los servidores, los centros de procesamiento de datos y los clústeres HPC (*High Performance Computing*), los cuales usan de manera general procesadores x86 [9, 10, 11, 12]. Es en estos últimos donde las mejoras en eficiencia energética pueden ayudar a reducir los costes de servicio y mantenimiento [13, 14]. No obstante, en estos escenarios, se usa una arquitectura de procesadores ARM multinúcleo homogénea, para facilitar así el equilibrio de la carga computacional y obtener eficiencia energética y mejora del rendimiento de las aplicaciones de requerimientos computacionales muy elevados.

La arquitectura ARM también ha integrado la técnica **SMT** (*Simultaneous Multithreading*). El **SMT** es un proceso en el que una CPU divide cada uno de sus núcleos físicos en núcleos virtuales, denominados hilos (*threads*). Esto permite ejecutar varios flujos de instrucciones o tareas de manera concurrente. Esta tecnología también se ha integrado en arquitecturas Intel, aunque se conoce por el término *HyperThreading*. Para conseguir un aumento del rendimiento en arquitecturas con **SMT** activo, es necesario descomponer la aplicación en tareas independientes que puedan ejecutarse simultáneamente y distribuirlas de forma equilibrada en las unidades de procesamiento. En este proceso es muy importante que se controle qué tarea va a realizar cada núcleo virtual. El cómputo, el intercambio de mensajes, los accesos a memoria y el balanceo de la carga son ejemplos claros de acciones que deban realizarse específicamente por una unidad de procesamiento concreto.

El objetivo es explotar eficientemente los recursos que ofrecen los clústeres de procesadores multinúcleos modernos y posibilitar el enorme procesamiento asociado. Esto no debería revertir problema alguno para el programador experimentado, salvo por el hecho de que la numeración de los núcleos virtuales sigue un criterio distinto en arquitecturas Intel y ARM.

En arquitecturas Intel, las CPUs lógicas se numerarán por el orden de la CPU física a la que pertenecen. Esta metodología de asignación es a la que denominaremos nomenclatura Intel. Por el contrario, en arquitecturas ARM se prioriza que las CPUs lógicas pertenezcan a diferentes CPUs físicas durante la numeración. Este procedimiento es al que denotaremos como nomenclatura ARM. La correspondencia entre núcleos físicos, núcleos virtuales y numeración se conoce con el término ***afinidad***.

La ***afinidad*** distinta provoca que la transición de aplicaciones paralelas entre arquitecturas Intel y ARM no sea automática. Además, ARM tiene sus propios conjuntos de instrucciones, mientras que Intel utiliza los de las arquitecturas x86-64. Por tanto, en muchos casos, es necesario desarrollar o modificar el software existente para evitar problemas de compatibili-

dad, para mejorar la eficiencia o porque simplemente necesita de parámetros específicos para funcionar correctamente [15]. Uno de los escenarios donde es preciso asegurar la afinidad, es en aplicaciones donde se utiliza OpenMPI, una de las implementaciones más conocidas y usadas del estándar MPI y desarrollada para arquitecturas x86. En este contexto controlar la afinidad es muy importante para obtener ejecuciones paralelas eficientes, ya que OpenMPI permite al programador repartir la carga computacional en los diferentes nodos de un clúster. OpenMPI hace, por defecto, la numeración de CPUs en nomenclatura Intel, independientemente de la arquitectura subyacente. En los sistemas ARM, puede provocar que tareas que deberían ejecutarse en CPUs físicas diferentes, acaben haciéndolo en la misma CPU. Esto podría conllevar errores de coherencia y desbalanceo de la carga y, por tanto, un decremento de la eficiencia y rendimiento.

Además de lo comentado previamente, existe otro escenario, dentro de los clústeres de HPC con arquitecturas ARM, donde se necesita supervisión y adaptación. Concretamente los drivers de componentes externos, como pueden ser las tarjetas de red. Estos componentes son relativamente noveles, por lo que no han alcanzado un nivel de perfeccionamiento tan grande como los desarrollados para arquitecturas Intel, por lo que es necesario comprobar que actúan como se espera, y que el ancho de banda entre los nodos del clúster es correcto. Esto es de especial interés en aplicaciones paralelas donde se requiere intercambio de información mediante paso de mensajes, ya que un ancho de banda irregular podría suponer un decremento en la eficiencia y redimiendo del sistema.

Por estas razones, el objetivo de este proyecto es doble. Por un lado, se desarrollará un planificador que permitirá asignar un banco de tareas a unidades de procesamiento preestablecidas en un entorno de programación MPI. Por otro, se automatizará el procesamiento y generación de informes de rendimiento acerca del ancho de banda de red entre los diferentes nodos de computación dentro del clúster de computadoras ARM, con el fin de determinar si los drivers de red para la arquitectura ARM están a la par en x86

1.2 OBJETIVOS DEL TRABAJO

Como se ha indicado en la Sección 1.1, el objetivo de este trabajo es el de desarrollar dos aplicaciones que ayuden tanto al correcto uso y aprovechamiento de las unidades de cómputo de un clúster ARM como en el análisis del rendimiento de los componentes red de este. Estas herramientas se llamarán **demangler** e **iorbenchtool** respectivamente.

Este objetivo doble se puede descomponer en los siguientes objetivos parciales:

- Familiarización con el flujo de trabajo en un clúster de altas prestaciones
- Obtención de conocimientos acerca de la arquitectura **ARM** y sus principales diferencias respecto a **x86**
- Familiarización con conceptos y software de la computación paralela
- Desarrollo de herramientas de gestión para:
    1. asignar tareas **MPI** a unidades de procesamiento del clúster **ARM**

2. automatizar la generación de informes que evalúen el rendimiento de la red de interconexión del clúster **ARM**

1.3 METODOLOGÍA DE TRABAJO

Debido a la naturaleza cambiante de los requerimientos del trabajo y la inclusión de nuevas características a medida que el software es utilizado, es necesario seguir una metodología ágil la cual permita la incorporación de características nuevas sin apenas coste y que tenga en cuenta en todo momento la realimentación por parte del cliente de una manera muy rápida [16].

Es por ello por lo que **Extreme Programming** (XP) es la metodología de desarrollo utilizada en este trabajo. **XP** obliga a mantener una estructura del software sencilla y limpia, fácilmente extensible; y, además, gracias al continuo *feedback*, también permite solventar errores tempranos encontrados en el software desarrollado.

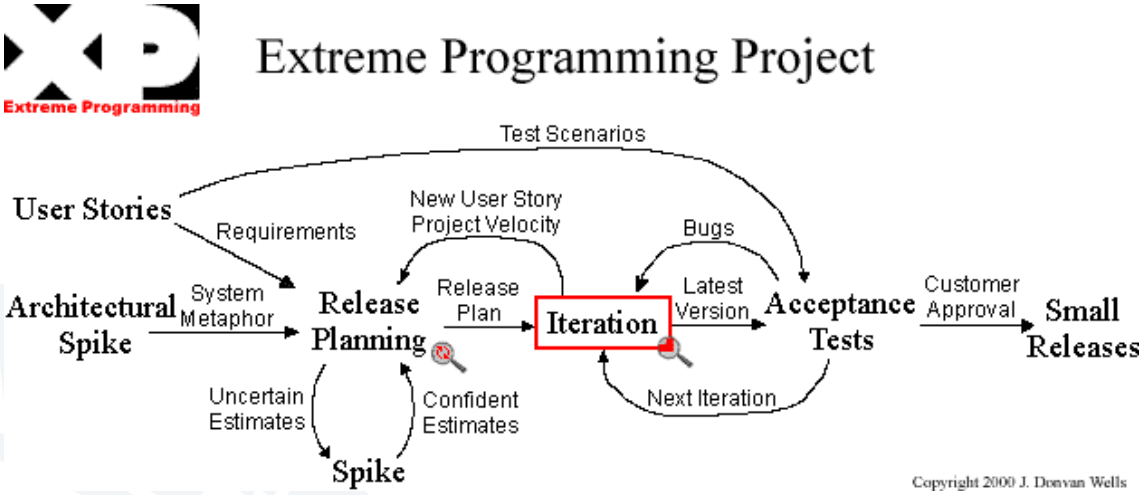


Figura 1.1: Flujo de trabajo de **XP** [16]

En la Figura 1.1 se observa que una parte esencial del flujo de trabajo en **XP** es la rápida iteración de cambios en el software, enfocados en crear pequeñas versiones de la aplicación. Además, se observa que el cliente o usuario es también clave en esta metodología, estando presente tanto al inicio del proyecto como durante el desarrollo de este.

1.4 PLANIFICACIÓN TEMPORAL DEL TRABAJO

Las fases por realizar para alcanzar los objetivos propuestos de este trabajo son las siguientes:

1. **Reuniones con los tutores (10 horas):** Reuniones con los tutores de la universidad de Almería. En estas se actualiza el progreso del trabajo y se utilizan para resolver dudas.
2. **Definición de las especificaciones del software a desarrollar (10 horas):** Una serie de reuniones con el EPCC en las que se definen los requisitos del software que se de-

sarrolla, y las posibles modificaciones y actualizaciones de estos requisitos para añadir nuevas características a este.

3. **Estudio de las instalaciones y arquitectura del hardware (30 horas):** Se estudia cómo son las instalaciones del EPCC tanto en organización e interconexión de los nodos de computación, como la arquitectura y diseño de los chips ARM que se encuentran en estos.

4. **Diseño del software a realizar (10 horas):** Creación de un diseño software flexible y simple, el cual permita la incorporación de variedad de funcionalidades.

5. **Desarrollo del software (180 horas):** Programación e implementación de los requisitos definidos en el Elemento 2.

6. **Evaluación del software realizado y cambios finales (15 horas):** El EPCC evalúa si el software realizado cumple con los requisitos definidos en el Elemento 2 y está listo para sus necesidades. Además, se contemplará la opción de añadir cambios / nuevas características de última hora y se pulirán los errores que se encuentren en el software.

7. **Redacción de la memoria final (45 horas):** Elaboración de una memoria del trabajo realizado en la que se explica todo el proceso de desarrollo, características y resultados obtenidos.

El total de horas son 300, organizadas según muestra la Tabla 1.1.

Tabla 1.1: Planificación temporal del proyecto. Cada rectángulo representa cinco horas

Reuniones con los tutores	**10 horas**	
Definición especificaciones del software	**10 horas**	
Estudio de instalaciones y arquitectura hardware	**30 horas**	
Estudio instalaciones (topología)	10 horas	
Estudio del hardware disponible	20 horas	
Diseño del software a realizar	**10 horas**	
Investigación herramientas más adecuadas	2 horas	
Modelado del software	8 horas	
Desarrollo del software	**180 horas**	
Automatización de informes de rendimiento	50 horas	
Pinning de tareas a hilos específicos	130 horas	
Evaluación del software realizado y cambios finales	**15 horas**	
Reuniones para evaluación del software	5 horas	
Cambios de última hora y corrección de errores	10 horas	
Redacción de memoria final	**45 horas**	

1.5 MATERIALES

Durante el desarrollo de este trabajo, se han usado diversos programas, los cuales se dividen en las siguientes categorías. Para cada uno de ellos se dará una breve descripción a continuación. Las categorías son seis: **lenguajes de programación**, **entornos de desarrollo**, **herramientas para control de versiones**, **herramientas del clúster**, **herramientas para la creación de documentación** y **herramientas de comunicación**.

1.5.1 Lenguajes de programación

En esta subsección de describen los lenguajes de programación utilizados durante el desarrollo del software del proyecto, así como de los lenguajes utilizados en la creación de la documentación de este:

C#: Es un lenguaje desarrollado por **Microsoft**. Se caracteriza por ser un lenguaje estáticamente tipado (aunque se puede optar por un tipado dinámico) orientado a objetos. Es también un lenguaje compilado, el cual crea una representación llamada *código intermedio* (IL, *Intermediate Languaje*) la cual posteriormente se ejecuta sobre una máquina virtual (CLI, *Common Language Infrastructure*), la cual transforma el código IL a código máquina gracias a un compilador JIT (*Just In Time*) sobre el computador en el que se ejecuta el CLR [17].

Rust: Lenguaje desarrollado inicialmente por **Mozilla** y que actualmente continua su desarrollo bajo la fundación Rust en conjunto con la comunidad *Open Source*. Este es un lenguaje fuertemente tipado; enfocado a la programación de programas CLI, generación de *WebAssembly*, aplicaciones de red y programación de sistemas embebidos; y, que, además, ofrece garantías acerca de la gestión de la memoria sin necesidad de un recolector de basura (GC, *Garbage Collector*). Gracias a su sistema de tipos, **Rust** detecta errores del uso de paralelismo en tiempo de compilación. Este es un lenguaje compilado, el cual hace uso de la estructura **LLVM** (`https://llvm.org/`) para compilar en las diferentes arquitecturas y sistemas operativos.

Markdown: Es un lenguaje basado en etiquetas (como *HTML*) utilizado para la creación de texto con formato. Muchos procesadores de texto enfocados a código hacen uso de **Markdown** para redactar su documentación ya que es un lenguaje simple.

LaTeX: Lenguaje compuesto de macros de **TeX** para la escritura de texto con formato, enfocado principalmente en la producción de documentación científica y técnica.

1.5.2 Entornos de desarrollo

En el desarrollo de este trabajo, dos IDEs (Integrated Development Enviroment) se han utilizado: **Visual Studio** y **Visual Studio Code**.

Visual Studio: IDE creado por **Microsoft Windows**. Este está fuertemente integrado con el entorno de desarrollo de **Microsoft**. Este permite trabajar con *C*, *C++* y *C#* principalmente. Debido a su excelente integración con este último lenguaje, **Visual Studio** será usado para el desarrollo bajo este lenguaje.

Visual Studio Code: Editor de código con capacidad de *IDE* a través de extensiones. Desarrollado por **Microsoft** y la comunidad *Open Source* para *Windows*, *macOS* y *GNU/Linux*. Gracias a la integración de *plugins* de terceros, será usado como IDE de **Rust**.

1.5.3 Herramientas para control de versiones

En el desarrollo de este proyecto se ha usado **GIT** como herramienta VCS (*Version Control System*).

GIT: Herramienta VCS distribuida, inicialmente creada para ayudar en el desarrollo del kernel *Linux*. Su funcionamiento se basa en grafos, en los que cada nodo representa un estado del proyecto. Gracias a este diseño, se pueden crear copias del proyecto independientes para probar cambios sin afectar al código original, el cual es mantenido.

GitHub: Plataforma VCS basada en **GIT** de **Microsoft**. Esta se ha utilizado para alojar el código fuente de las herramientas de este proyecto, manteniéndolas fácilmente accesibles y seguras en caso de pérdida del código local.

1.5.4 Herramientas del clúster

Con la finalidad de probar las herramientas en el clúster **Fulhame**, se han utilizado herramientas tanto para acceder a él, como para modificar su configuración para que esta sea la más adecuada para las pruebas. Las principales herramientas utilizas son:

SLURM: Es una utilidad para la gestión y planificación de tareas sobre un clúster o nodos de este. El uso de **SLURM** se realiza a través de *scripts* **bash** en los que se sigue la siguiente estructura:

1. Comprobar si el clúster o los nodos que se desean usar están en uso.
2. Configurar los nodos que se van a utilizar (configuración **SMT**, por ejemplo).
3. Configurar el entorno en el que se va a ejecutar software, cargando las librerías, parámetros del software, etc.
4. Planificar la ejecución del entorno (mediante una cola de espera).
5. Esperar el resultado de la ejecución.

SSH: Es un protocolo de red criptográfica para operar servicios de manera segura en un canal no seguro. Este permite realizar acciones como si se estuviera conectado al ordenador remoto, además de permitir la posibilidad de enviar archivos entre origen y destino [18].

1.5.5 Herramientas para la creación de documentación

A continuación, se describen las herramientas utilizadas durante la creación de la documentación del software de este trabajo, esta memoria, y los blogs y otros requerimientos del programa **Summer Of HPC**.

Zotero: Herramienta para la gestión bibliográfica. Gracias a su integración con el navegador (mediante el uso de una extensión), es muy útil para mantener la bibliografía consultada durante la creación de esta memoria sin mayor esfuerzo.

Zettlr: Editor de **Markdown** enfocado a la creación de notas y proyectos de escritura gracias a su integración con herramientas de citado como **Zotero**. Se ha utilizado para la creación de los ficheros **Markdown** que explican el uso del software desarrollado en este trabajo de fin de estudios y para tomar notas acerca de las reuniones con el **EPCC**.

Overleaf: Editor de LaTeX online. Gracias a su sistema de colaboración resulta muy útil para mantener un desarrollo fluido en conjunto con los tutores de este trabajo. Además, siempre está accesible, lo que puede resultar muy cómodo para trabajar en cualquier momento y desde cualquier lugar.

WordPress: Para cumplir con el requisito de elaborar un blog como parte del programa **Summer Of HPC**, este gestor de contenido ha sido utilizado para la elaboración de los ya mencionados blogs.

1.5.6 Herramientas para la comunicación

Esta subsección enumera las herramientas utilizadas para la comunicación del estudiante con los tutores, el EPCC y PRACE.

Correo electrónico: El correo electrónico es la herramienta más transversal utilizada para la comunicación entre las diferentes partes que han realizado este trabajo. Esta herramienta ha permitido que el alumno contacte con los tutores y con el **EPCC**; y poder compartir información como enlaces de reuniones; necesarios para la realización de este trabajo.

Google Meet: Herramienta que permite la creación de reuniones virtuales a través de videollamadas. Se ha utilizado para las reuniones con los tutores de este trabajo.

Zoom: Al igual que **Google Meet**, esta es una herramienta de videoconferencias en la cual se puede compartir pantalla, uso del puntero en el ordenador de otro participante y otro tipo de soluciones para permitir una comunicación fácil y natural. **Zoom** se ha utilizado para las reuniones semanales de actualización de progreso del programa **Summmer of HPC**.

Skype: Herramienta similar a **Google Meet**. Utilizada para participar en las reuniones del **EPCC** con otros centros de computación del **U.K**.

Slack: Entorno de colaboración virtual para un grupo o empresa. En esta aplicación se crean diferentes canales en los que los usuarios pueden entrar dependiendo de sus roles/permisos, permitiendo así una comunicación fluida y organizada incluso en grupos grandes de personas.

Mattermost: Herramienta similar a **Slack** la cual, de igual manera, permite la comunicación de grupos dentro de una misma organización y entre los usuarios de la plataforma.

Microsoft Teams: Herramienta de colaboración muy similar a **Slack**, que, además, integra funcionalidades de videollamadas; integrando en una misma plataforma los tipos de comunicación virtual más utilizados. Ha sido utilizada únicamente para las videollamadas con el **EPCC**, ya que la comunicación vía chat se ha realizado mediante **Mattermost**.

2 ARQUITECTURAS PARA LA SUPERCOMPUTACIÓN

A lo largo de este capítulo se explican conceptos clave para comprender el trabajo desarrollado. Para ello, se describen los diferentes tipos de clasificación de las arquitecturas de computadores en función del paralelismo que ofrecen, cómo son sus instrucciones, y la organización de su memoria. Además, se habla acerca de los diferentes niveles de paralelismo dentro de un procesador, y finalmente, se explican y se muestran ejemplos de los paradigmas de programación más importantes y usados para explotar los diferentes tipos de paralelismo que ofrecen los supercomputadores.

2.1 CLASIFICACIÓN DE ARQUITECTURAS DE COMPUTADORES

Esta sección trata acerca de las diferentes clasificaciones de computadores que existen. Estas clasificaciones no son mutuamente excluyentes, sino que un computador puede adscribirse a varias o todas ellas. Los computadores pueden clasificarse en función del flujo de instrucciones y datos, la organización de su memoria, y según la forma de sus instrucciones.

2.1.1 Clasificación según el flujo: Taxonomía de Flynn

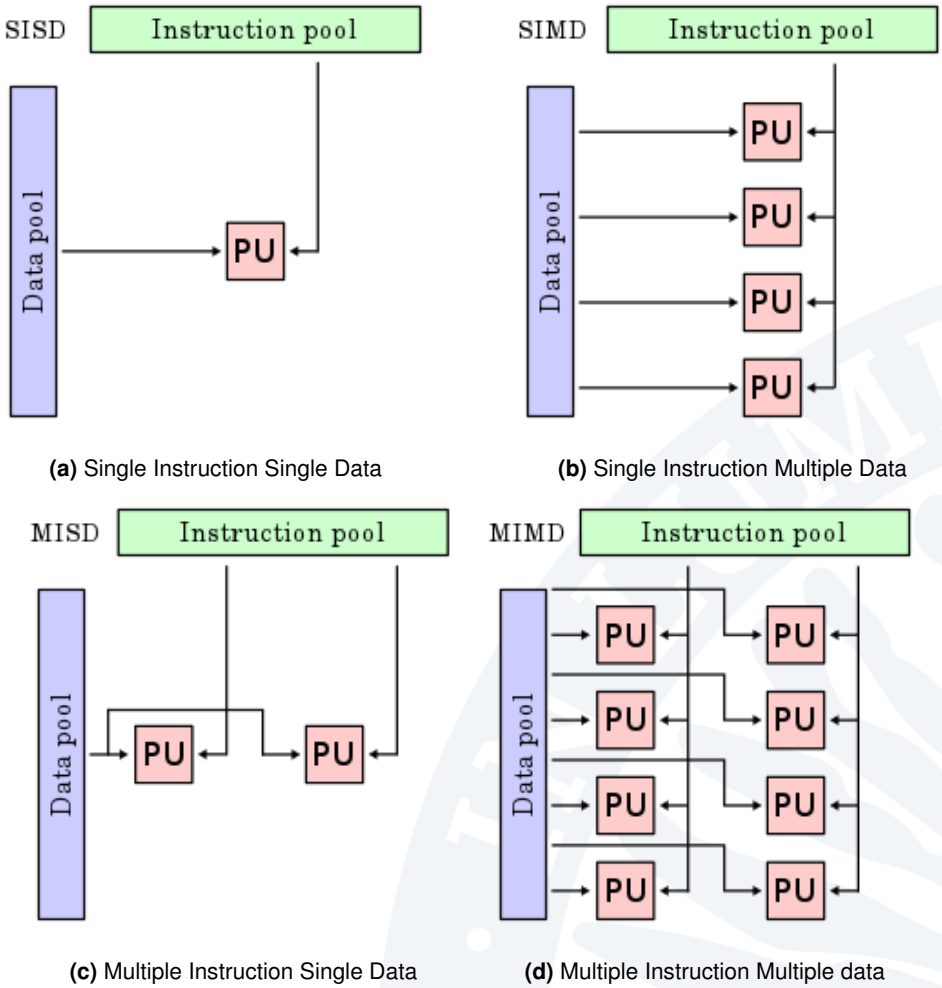
La taxonomía de Flynn es un modelo de clasificación de arquitecturas de computadores. Este indica que un computador puede ser catalogado en función de dos dimensiones independientes: el flujo de instrucciones y el flujo de datos, dando lugar a cuatro separaciones posibles [19, 20]. Este método de clasificación ha sido usado en el diseño de las funcionalidades de procesadores modernos [21]. Las diferentes clasificaciones se pueden observar en la Figura 2.1.

- **SISD**: *Single Instruction stream, Single Data stream* (Flujo único de instrucciones, flujo único de datos). Corresponde con un computador no paralelo. Este solo puede ejecutar un flujo de instrucciones y datos en una única unidad de procesamiento. Corresponde con la Figura 2.1a.
- **SIMD**: *Single Instruction stream, Multiple Data stream* (Flujo único de instrucciones, múltiples flujos de datos). Todas las unidades de procesamiento del computador ejecutan el mismo flujo de instrucciones en un determinado ciclo de reloj. Sin embargo, el flujo de datos puede ser diferente. Corresponde con la Figura 2.1b. Las GPUs son un computador **SIMD**. De igual manera, los procesadores modernos cuentan con exten-

siones de su ISA (*Instruction Set Architecture*) para soportar instrucciones SIMD: SSE y AVX en x86 [22]; y NEON y SVE en ARM [23].

- **MISD**: *Multiple Instruction stream, Single Data stream* (Múltiples flujos de instrucciones, flujo único de datos). Las unidades de procesamiento del computador pueden ejecutar diferentes flujos de instrucciones únicamente utilizando el mismo flujo de datos. Corresponde con la Figura 2.1c. Este tipo de computadores es poco común. Suele utilizarse en computadores con sistema de tolerancia de fallos.

- **MIMD**: *Multiple Instruction stream, Multiple data stream* (Múltiples flujos de instrucciones, múltiples flujos de datos). Cada unidad de procesamiento puede ejecutar diferentes flujos de instrucciones sobre diferentes flujos de datos. Corresponde con la Figura 2.1d. Los procesadores multi-núcleo y sistemas distribuidos pertenecen a esta clasificación [24].



(a) Single Instruction Single Data

(b) Single Instruction Multiple Data

(c) Multiple Instruction Single Data

(d) Multiple Instruction Multiple data

Figura 2.1: Comparativa de la taxonomía de Flynn [21]. *Instruction pool* se refiere al conjunto de instrucciones. *Data pool* se refiere al conjunto de datos. *PU* se refiere a una Unidad de Procesamiento (*Processing Unit*, PU)

2.1.2 Clasificación según la organización de su memoria

A lo largo de esta subsección, se introducen y explican los diferentes tipos de arquitecturas de computadores que existen según la organización de la memoria de estos. Hay tres grupos principales: Arquitecturas de memoria compartida, arquitecturas de memoria distribuida y arquitecturas híbridas. Todas ellas se pueden clasificar, según la taxonomía de Flynn, como computadores **MIMD** [25].

Arquitecturas de memoria compartida

En este tipo de arquitecturas, todos los procesadores del sistema comparten la memoria existente. Esto permite que cualquier región de memoria sea accesible desde cualquier procesador que forme parte del computador. [25].

Estas arquitecturas buscan explotar el paralelismo local del procesador utilizando interfaces de programación como **POSIX Threads** [26] y **OpenMP** [27].

Existen, principalmente, tres grupos de arquitecturas de memoria compartida: De acceso uniforme a memoria (*Uniform Memory Access*, UMA), de acceso no uniforme a memoria (*Non Uniform Memory Access*, NUMA) e híbridas UMA-NUMA.

- **Arquitecturas de acceso uniforme a memoria**: Pertenecen a UMA aquellas arquitecturas en las que todos los procesadores del sistema compartan una única memoria principal a la que pueden acceder directamente a través de un *bus*. Esto provoca que los diferentes procesadores del sistema tengan que competir por el acceso al *bus*, como se puede observar en la Figura 2.2, reduciendo el ancho de banda efectivo a medida que el número de procesadores crece.

  Como ventaja, todos los procesadores acceden a la memoria de una manera *uniforme* (en la práctica con latencias muy similares entre procesadores).

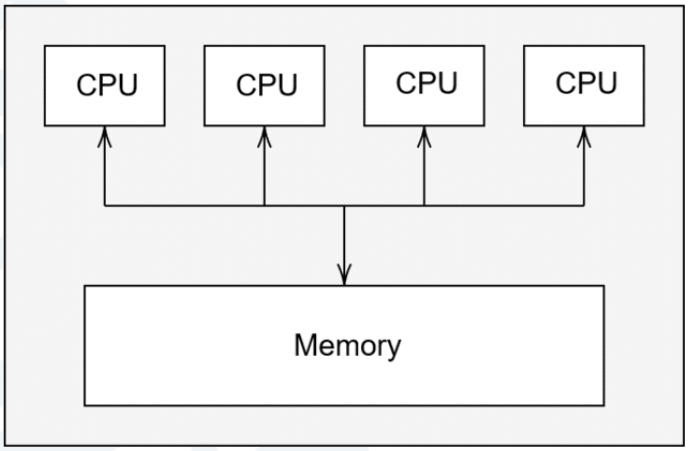


Figura 2.2: Diagrama de UMA

- **Arquitecturas de acceso no uniforme a memoria**: A NUMA pertenecen aquellas arquitecturas en las que los diferentes procesadores tienen su propia memoria local, pero, aun así, pueden acceder a la memoria del resto de CPUs mediante un *bus*. NUMA surge para reducir la complejidad de crear un *bus* eficiente para muchos procesadores en UMA.

Como se puede observar en la Figura 2.3, los procesadores están conectados entre sí mediante un *bus*, lo que hace que en la práctica este sea un sistema de memoria compartida. Por lo tanto, la penalización por acceder a la memoria solo aparecerá cuando un procesador necesite acceder a la memoria de otro, es decir, cuando los datos que necesita están en su memoria local, el acceso a la memoria será mucho más rápido que en un sistema UMA.

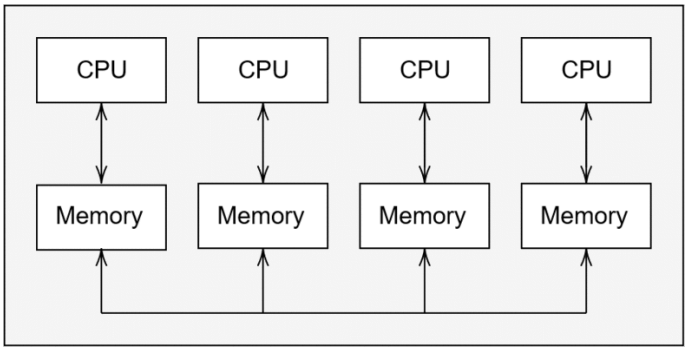


Figura 2.3: Diagrama de NUMA

- **Arquitecturas híbridas de memoria compartida**: En la actualidad, los procesadores multi-núcleo mezclan los conceptos de UMA y NUMA, formando lo que se denomina una arquitectura híbrida de memoria compartida o híbrido UMA-NUMA. Bajo este tipo de arquitecturas aparecen los denominados *nodos*, los cuales son grupos de procesadores bajo un diseño UMA. Estos *nodos* están interconectados entre sí permitiendo que exista un intercambio de memoria entre ellos, como si de una arquitectura NUMA se tratase [28]. Esto se puede observar en la Figura 2.4.

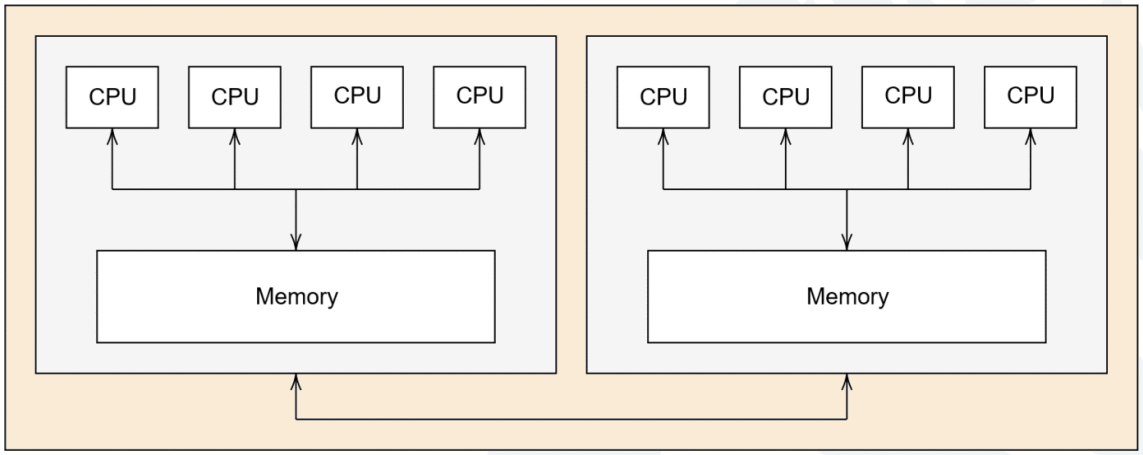


Figura 2.4: Diagrama de arquitectura híbrida de memoria compartida

Esta arquitectura híbrida combina las ventajas de UMA y NUMA para crear un sistema que pueda escalar en número de procesadores sin sacrificar el rendimiento global del sistema.

Arquitecturas de memoria distribuida

Como se ha visto en la Subsubsección 2.1.2.1, las arquitecturas de memoria compartida se basan en que los todos procesadores de un mismo computador pueden acceder al mismo bloque de memoria, el cual comparten. Las arquitecturas de memoria distribuida, en cambio, son aquellas en las que cada procesador tiene su propia memoria local a la que solo él puede acceder. Por tanto, todas las computaciones que este realice solo pueden usar datos que estén almacenados en dicha memoria local. La Figura 2.5 muestra un diagrama acerca de esta arquitectura.

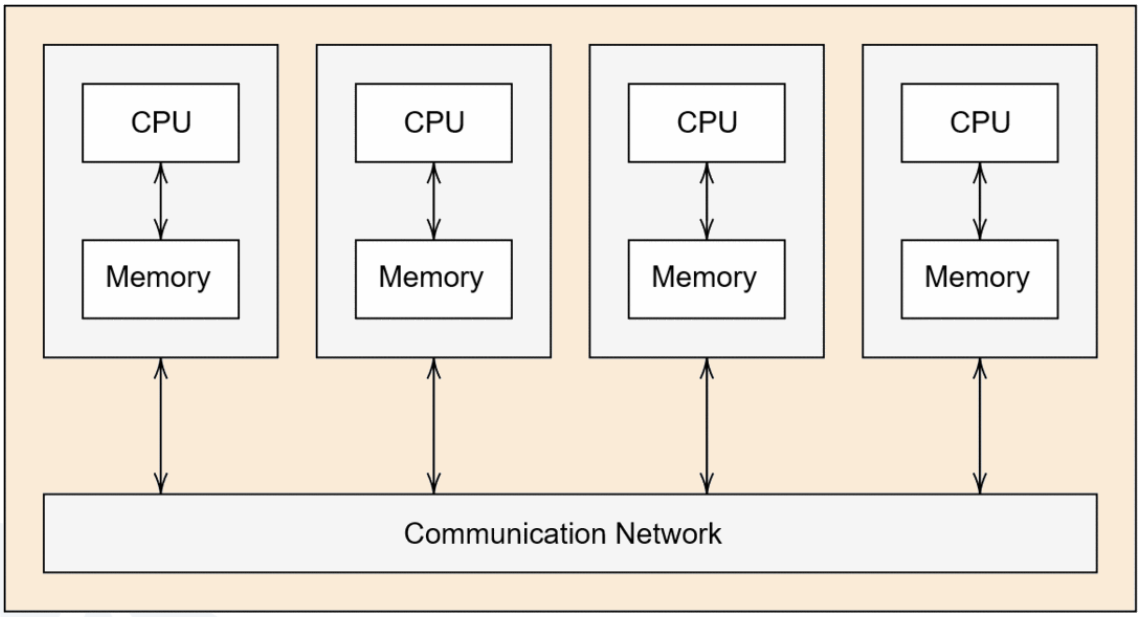


Figura 2.5: Diagrama de memoria distribuida

Sin embargo, es posible poder utilizar datos que este procesador no tiene en su memoria local. Para ello, primero tendrá que pedirlos a otro procesador mediante APIs como **MPI** [29], aumentando las latencias para realizar ese cálculo, ya que las comunicaciones con otros procesadores se realizan a través de interconexiones de red.

El problema de estas arquitecturas es que el rendimiento de las computaciones que se llevan a cabo pueden llegar a estar limitadas por el ancho de banda de las interconexiones de red, por lo que conectores como *Gigabit Ethernet* pueden resultar limitantes en ciertas configuraciones (ningún supercomputador del 10 top de **TOP500** usa *Gigabit Ethernet*, sino que usan una conexión propietaria como *Tofu Interconnect* o *Infiniband* [30]).

Arquitecturas híbridas

Estas arquitecturas son aquellas en las que diferentes computadores de memoria compartida se conectan formando un computador de memoria distribuida, tal y como se puede ver en la Figura 2.6.

De esta manera, una computación puede repartirse entre los diferentes computadores o *nodos* de memoria distribuida usando alguna API de paso de mensajes (ej. **MPI**) y en cada *nodo* explotar el paralelismo local haciendo uso de APIs de memoria distribuida (ej. **POSIX Threads**).

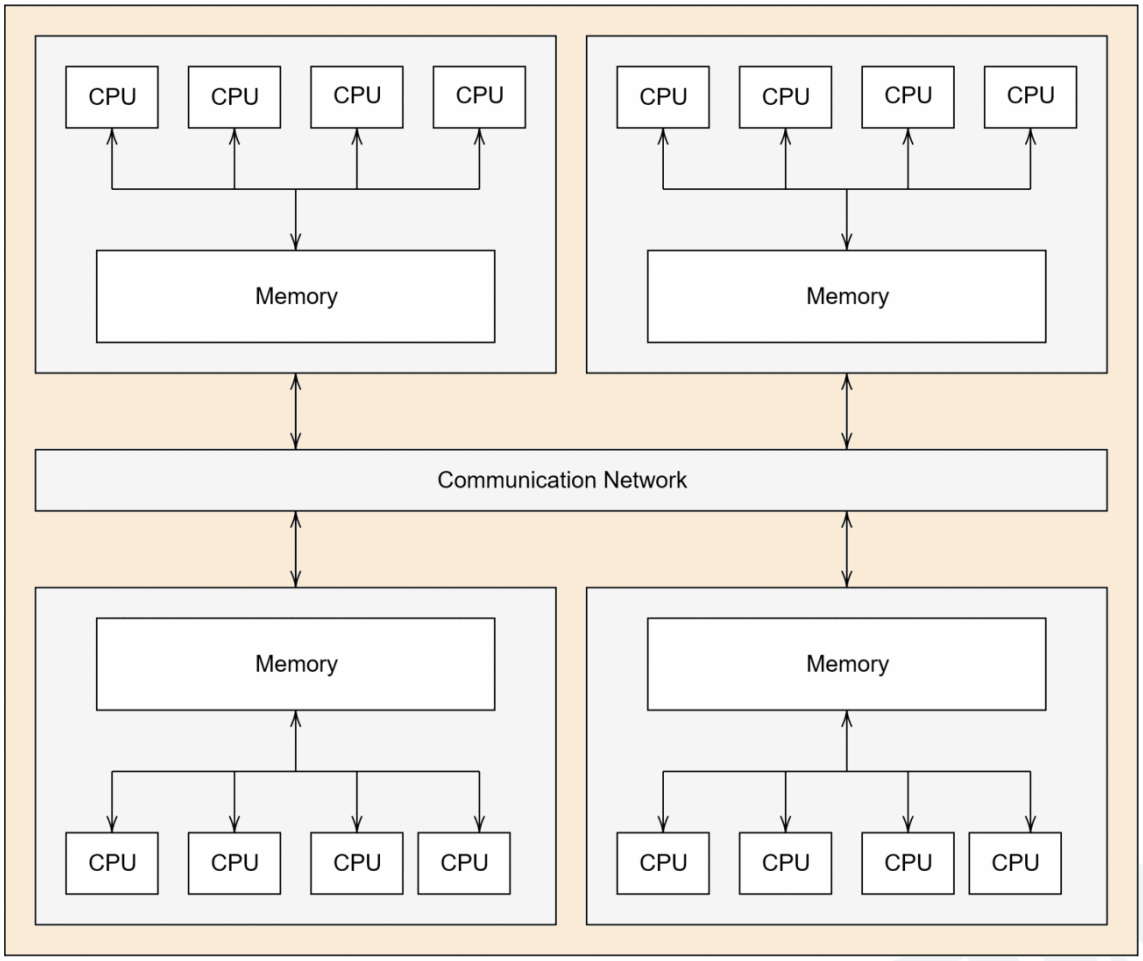


Figura 2.6: Diagrama de arquitectura híbrida

Este tipo de arquitecturas son extremadamente útiles para el reparto de computaciones, pero ponen mucho peso sobre el programador, ya que es este el encargado de determinar cómo repartir la información y cómo sincronizarla entre *nodos* e *hilos* para sacar el mayor partido al computador.

2.1.3 Clasificación según las instrucciones de un procesador

Independientemente del nivel de paralelismo que un procesador implemente (Sección 2.2), este solo puede ejecutar las instrucciones que conozca en un determinado formato binario. A este conjunto de instrucciones que un procesador puede ejecutar se denomina *Intruction Set Architecture* (ISA). El ISA de un procesador define la forma de las instrucciones que el procesador puede interpretar y ejecutar, por lo que realmente un ISA es la interfaz entre hardware y software de un procesador. Normalmente, las instrucciones de un ISA tienen un código de operación (opcode), el cual define la acción a realizar, y ciertos operandos que pueden indicar cuales son las entradas de datos de dicha operación y dónde almacenar el resultado [31]. Existen gran variedad de ISAs entre los que se incluyen: MIPS, PowerPC, x86 y ARM.

Dependiendo de las acciones que una determinada instrucción del ISA puede realizar, estos se pueden clasificar en dos grupos: CISC (*Complex Instruction Set Computer*) y RISC (*Reduced Instruction Set Computer*). Cada una de estas filosofías tiene diferentes metas:

Un procesador **CISC** trata de que sus programas (en ensamblador) requieran del menor número de instrucciones para llevar a cabo su cometido. Para ello, el hardware ha de poder ejecutar varias operaciones de bajo nivel en una sola instrucción. Esto permite que los programas para procesadores CISC sean más pequeños (comparados con programas RISC) y con menos accesos a memoria, lo que puede favorecer más aciertos en la caché [32]. Uno de los mayores problemas de los diseños CISC es respecto al ILP (*Instruction Level Parallelism*). Debido a que las instrucciones complejas son más difíciles de segmentar, este tipo de procesadores no aprovecha al máximo, en las instrucciones más grandes, la eficiencia que ILP provee [1].

Por el contrario, un procesador **RISC** trata de que las instrucciones que ofrecen sean muy simples y realicen una sola operación de bajo nivel. Esto ofrece muchas ventajas en lo que se refiere ILP respecto a CISC ya que permite que la segmentación de instrucciones sea más eficaz, haciendo que todas las instrucciones de un procesador RISC puedan ser ejecutadas en un solo ciclo de reloj [32]. Sin embargo, debido a que los programas son más largos, puede provocar una mayor tasa de fallos en la caché.

2.2 NIVELES DE PARALELISMO

Las operaciones definidas por el ISA de un procesador se ejecutan dentro de lo que se conoce como *hilo de ejecución*. Este es un conjunto de instrucciones que se ejecutan de manera secuencial en un procesador. Aunque originalmente, los procesadores solo podían ejecutar una instrucción de un *hilo de ejecución* de manera simultánea (en los procesadores primordiales), estos han ido evolucionando hasta alcanzar dos tipos diferentes y coexistentes de paralelismo:

- **Paralelismo a nivel de instrucción (*Instruction Level Parallelism*, ILP)**: Se refiere a la ejecución simultánea de instrucciones dentro de un mismo *hilo de ejecución*.
- **Paralelismo a nivel de tareas (Task Level Parallelism, TLP)**: Se refiere a la ejecución simultánea de diferentes *hilos de ejecución*.

ILP hace uso de técnicas que permiten a una CPU ejecutar el mayor número posible de instrucciones de un mismo *hilo de ejecución* por ciclo de reloj, intentando que la CPU tenga el menor tiempo de inactividad posible. Entre las técnicas utilizadas en ILP se encuentran [24]:

- **Instruction pipelining**: Segmentación de instrucciones. Consiste en dividir la ejecución de una instrucción en diferentes etapas, de manera que diferentes instrucciones pueden ejecutarse simultáneamente en diferentes etapas. En el mejor de los casos, esto permitirá la ejecución de una instrucción por ciclo de reloj.

[1] los procesadores CISC modernos generan μops o *micro-operaciones* de sus instrucciones más complejas (no accesibles desde el ISA) permitiendo que estas sean segmentadas y puedan aprovechar el ILP del procesador

- **Superscalar execution**: Ejecución superescalar. Consiste en crear nuevas líneas de ejecución idénticas de manera que diferentes instrucciones pueden ser ejecutadas al mismo tiempo. Esto unido a **Instruction pipelining** permite la ejecución de más de una instrucción por ciclo de reloj bajo circunstancias ideales.

- **Out-of-order execution**: Ejecución fuera de orden. Permite a la CPU reordenar las instrucciones para evitar que las dependencias entre estas provoquen paradas de la CPU. Esta técnica ayuda a mejorar la eficiencia de las técnicas anteriores.

- **Branch prediction & expeculative execution**: Predicción de saltos y ejecución especulativa. Estas técnicas permiten a la CPU predecir que ramificación de instrucciones el programa seguirá y ejecutar dichas instrucciones para adelantar trabajo.

TLP, por el contrario, no se centra en ejecutar el máximo de instrucciones por ciclo de reloj, sino en aumentar la cantidad de *hilos de ejecución* que un procesador ejecutar de manera simultánea. Bajo la taxonomía del Flynn (Subsección 2.1.1), un procesador que implemente alguna forma de TLP es clasificado como un computador MIMD. Existen diferentes maneras de incrementar la cantidad de *hilos de ejecución*, las dos principales son las siguientes:

- **Incrementar CPUs**: Al incrementar las CPUs de un procesador se aumentan el número de *hilos de ejecución* que el procesador puede soportar al mismo tiempo.

- **Implementar SMT**: Al multiplicar parte del hardware de las CPUs, estas pueden ejecutar instrucciones de múltiples *hilos de ejecución* al mismo tiempo. **SMT** explota de manera simultánea el paralelismo ILP y TLP [12, 25].

2.3 Paradigmas de programación paralela

Como se cuenta en la Sección 2.2, los procesadores han evolucionado de una manera que les permite poder ejecutar más *hilos de ejecución* al mismo tiempo, bien sea porque estos han visto aumentado su número de núcleos o porque implementan alguna forma de **SMT**.

Este incremento en número de *hilos de ejecución* provoca que la potencia bruta de los procesadores haya crecido mucho durante las últimas dos décadas. Aun así, un programa no hace uso, por defecto, de más de un *hilo de ejecución* al mismo tiempo. Por ello nacen las interfaces de programación paralela, las cuales permiten a los diseñadores y desarrolladores explotar toda la potencia que las técnicas de TLP han añadido a los procesadores.

Existen gran variedad de interfaces de programación paralelas, pero todas ellas pueden agruparse en dos grandes categorías según la organización de la memoria del sistema seguida en la Subsección 2.1.2: Interfaces de memoria compartida e interfaces de memoria distribuida.

Cabe destacar que no todas las cargas de trabajo son compatibles con una implementación paralela, ya sea en memoria compartida como distribuida, por lo que antes de usar estas APIs para resolver un problema es muy recomendable analizar cuál será el beneficio obtenido al usar estos métodos de programación paralela [26].

En esta sección se mencionan y explican las interfaces de programación de memoria compartida y memoria distribuida. Además, también se indica las interfaces de programación usadas en un clúster más allá de las utilizadas en los sistemas de memoria compartida y memoria distribuida.

2.3.1 Interfaces de programación de memoria compartida

Las interfaces de programación de memoria compartida se centran en explotar los recursos que ofrece un único computador, como por ejemplo el uso de sus múltiples CPUs.

Existen diferentes APIs para la programación en sistemas de memoria compartida, pero las más conocidas son **POSIX Threads** (**pthreads**) y **OpenMP**. A continuación, se expone un ejemplo de uso para ambas interfaces.

POSIX Threads

Esta interfaz permite transformar código que se ejecuta de manera secuencial en uno que puede hacer uso de las CPUs del computador para resolver el mismo problema de una manera que hace mejor aprovechamiento de los recursos del sistema de memoria compartida [33].

En **pthreads**, cada unidad de ejecución paralela se llama *thread* o *hilo*. Cada uno de estos hilos puede ejecutar un conjunto específico de instrucciones indicado por el programador. Esto es muy útil para repartir trabajo entre diferentes hilos, pudiendo reducir el tiempo que un algoritmo tarda en ejecutarse.

Uno de los problemas que conlleva el uso de APIs que permiten compartir el trabajo son las condiciones de carrera. Estas pueden darse si algún hilo intenta acceder a datos que otro hilo está sobrescribiendo o que aún no han sido creados, lo que puede causar que el algoritmo tenga una salida incorrecta o directamente no funcione.

Para solventar este tipo de problemas, **POSIX Threads** ofrece una serie de primitivas de sincronización mediante las cuales el programador indicará como se usa la memoria que estas primitivas protegen. Entre las primitivas que **pthreads** ofrece se encuentran:

- **Mutex**: Ofrece acceso exclusivo a la memoria que bloquea. Cuando un hilo accede a la memoria que un mutex protege, cualquier otro hilo que intente acceder a la misma memoria queda bloqueado hasta que el hilo inicial libere el mutex [34].
- **RWLock**: Similar a mutex, salvo que esta primitiva ofrece dos operaciones: **lectura** y **escritura**. Cuando la memoria contenida en un RWLock se adquiere en modo lectura, el resto de los hilos pueden acceder a esa misma memoria sin quedar bloqueados, siempre que también sea en modo lectura. Si, por el contrario, cuando esta región se adquiere en modo escritura, su funcionamiento es el mismo que el de un mutex [35].
- **Semaphore**: Primitiva similar a RWLock salvo que permite limitar el número de los hilos que pueden acceder a la memoria que protege. Esta cuenta con dos operaciones, **wait** y **signal**. La primera decrementa el contador interno del *semaphore* (implementado mediante un entero atómico) en una unidad. Signal realiza la operación contraria a wait, incrementa dicho contador en una unidad. Debido a esto pueden darse dos situaciones cuando se utiliza la operación wait.

- **El contador es negativo**: En este caso, el hilo que haga que el contador sea negativo queda bloqueado.
- **El contador no es negativo**: La ejecución del hilo no queda bloqueada.

Cuando un hilo deja de utiliza la memoria que el *semaphore* protege ha de usar la operación signal para permitir que los hilos que estén bloqueados puedan continuar con su ejecución.

Un mal uso de estas primitivas puede hacer que un programa/algoritmo quede bloqueado (dead lock).

Para ilustrar el uso de **POSIX Threads**, se muestra un fragmento de código, el Listado 2.1, el cual realiza la suma de los elementos de un vector de enteros repartiendo el trabajo entre diferentes *hilos*.

```
#include <pthread.h>
#include <stdlib.h>
#include <assert.h>

// Estructura de paso de argumentos a hilos
struct slice
{
    int *start;
    intlen;
} typedefslice;

// Función paralelizada de suma
void *suma(void *args)
{
    slice *arg = (slice *)args;
    intlocal = 0;
    for (size_ti = 0; i < arg->len; i++)
    {
        local += arg->start[i];
    }
    int *result = calloc(sizeof(int), 1);
    *result = local;
    returnresult;
}

int main(void)
{
    intsize = 1 << 30;                      // 2^30 elementos
    int n_threads = 4;                      // Numero de hilos
    int *vector = calloc(sizeof(int), size); // 4 GiB

    assert(vector != NULL);

    // Inicialización del vector
    for (size_ti = 0; i < size; i++)
    {
        vector[i] = 1;
    }

    // Ventana de cada hilo
    int partition = size / n_threads;

    pthread_t threads[n_threads];
    slice args[n_threads];
    for (size_ti = 0; i < n_threads; i++)
    {
```

```
47          slice arg = {.start = vector + (partition * i), .len = partition};
48
49          args[i] = arg;
50
51          // Creacion hilo con sus parametros
52          int error = pthread_create(&threads[i], NULL, suma, &args[i]);
53          assert(!error);
54      }
55
56      int resultado = 0; // Resultado final
57      for (size_ti = 0; i < n_threads; i++)
58      {
59          void *temp;
60          int error = pthread_join(threads[i], &temp); // Espera de hilos
61          assert(!error);
62
63          resultado += *(int *)temp; // Resultado final = suma resultadosparciales
64
65          free(temp);
66      }
67
68      assert(resultado == size);
69      free(vector);
70
71      return 0;
72  }
```

Listado 2.1: Ejemplo de suma de elementos de un vector en POSIX Threads

Como se puede observar en el Listado 2.1, para poder compartir información a los hilos, primero es necesario saber cómo se va a hacer el reparto y crear estructuras para almacenar la información que se va a compartir. Tras ello, se comparten estas estructuras y se realiza el cálculo. Finalmente, se sincronizan los hilos y se ponen en común los resultados, terminando así con la ejecución del programa.

POSIX Threads es una API en la que es necesario especificar hasta el más mínimo detalle acerca de la manera de compartir los datos, realizar las computaciones y sincronizar los hilos. Es por ello por lo que surge **OpenMP**.

OpenMP

Esta API tiene la misma finalidad que **POSIX Threads**, salvo que la manera en la que esta se utiliza es radicalmente diferente.

El Listado 2.2, creado usando **OpenMP**, es un código mucho más breve y conciso que el mostrado en el Listado 2.1, programado con **POSIX Threads**. Esto se debe a que **OpenMP** usa directivas de pre-compilado para generar el código necesario para el reparto de los datos entre diferentes hilos y posteriormente, sincronizarlos; aunque bajo las escenas, la implementación puede usar **pthreads** (o cualquier otra implementación adecuada según el sistema operativo).

```
1  #include <stdio.h>
2  #include <omp.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  int main()
7  {
8      int size = 1 << 30; // 2^30 elementos
9      int n_threads = 4;
```

```
10
11      omp_set_num_threads(n_threads); // Numero de hilos. Se puede hacer tmb mediante
        OMP_NUM_THREADS env.
12
13      int *vector = calloc(sizeof(int), size); // 4 GiB
14      assert(vector != NULL);
15
16      // Inicialización del vector
17      for (size_t i = 0; i < size; i++)
18      {
19          vector[i] = 1;
20      }
21
22      int result = 0;
23  #pragma omp parallel for reduction(+ \
24                                     : result)
25      for (size_t i = 0; i < size; i++)
26      {
27          result += vector[i];
28      }
29
30      assert(result == size);
31      free(vector);
32
33      return 0;
34  }
```

Listado 2.2: Ejemplo de suma de elementos de un vector en OpenMP

OpenMP es una API mucho más simple que **POSIX Threads**, pero mucho menos flexible, ya que todos los aspectos expuestos al programador en **pthreads** ahora están ocultos. **OpenMP** es muy útil cuando la computación que se quiere paralelizar es relativamente simple o no se quiere controlar hasta la última minucia.

2.3.2 Interfaces de programación distribuida

Este tipo de interfaces de programación se encargan de regir cómo diferentes computadores o nodos de memoria compartida se comunican entre ellos, proveyendo de una interfaz y *framework* para compartir computaciones y datos.

La interfaz más conocida y usada para la programación de sistemas de memoria distribuida es **MPI** [29]. A pesar de ello, esta es solamente un estándar, por lo que no se puede usar como tal. Por ello existen diferentes librerías que implementan este estándar, como **OpenMPI**.

OpenMPI

Esta librería es una de las implementaciones del estándar **MPI** más usada. Está mantenida por una comunidad de académicos, investigadores y personas de la industria [36].

OpenMPI, como parte del estándar **MPI**, ofrece un compilador (mpicc) para los códigos que hagan uso de la librería y un ejecutor (mpirun). Además de ello, el sistema ha tenido que ser previamente configurado para el uso de **MPI**.

```
1  #include <stdio.h>
2  #include <openmpi/mpi.h>
3  #include <assert.h>
4  #include <stdlib.h>
5
6  int main(int argc, char *argv[])
```

```
{
    MPI_Init(&argc, &argv); // Inicializacion de MPI

    int size = 1 << 28; // 2^28 elementos
    int world_rank, world_size = 0;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Obtención de número de tarea en MPI
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Obtención del número total de tareas en
    MPI

    int *vector = NULL;
    if (world_rank == 0) // Solo un proceso ha de inicializar el vector
    {
        vector = calloc(sizeof(int), size); // 1 GiB

        // Inicialización del vector
        for (size_t i = 0; i < size; i++)
        {
            vector[i] = 1;
        }

        assert(vector != NULL);
    }

    // Reparto del vector a todas las tareas de MPI
    int local_size = size / world_size;
    int *local_vec = calloc(sizeof(int), local_size);

    MPI_Scatter(vector, local_size, MPI_INT, local_vec, local_size, MPI_INT, 0,
    MPI_COMM_WORLD);

    // Suma local de un framento del vector
    int local_sum = 0;
    for (size_t i = 0; i < local_size; i++)
    {
        local_sum += local_vec[i];
    }

    // Suma global del conjunto de fragmentos del vector
    int global_result = 0;
    MPI_Reduce(&local_sum, &global_result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (world_rank == 0)
    {
        assert(size == global_result);
    }

    free(local_vec);
    free(vector);

    return 0;
}
```

Listado 2.3: Ejemplo de suma de elementos de un vector en OpenMPI

El ejemplo del Listado 2.3 sigue el mostrado en los Listados 2.1 y 2.2: un programa para repartir la suma de un vector de números entre diferentes unidades de trabajo. En este caso, usando MPI mediante **OpenMPI**.

Para ello, primero es necesario inicializar el entorno MPI usando la función MPI_Init(). Esto se debe a que como MPI se ejecuta sobre un grupo de computadores, todos tienen que

sincronizar su inicio. Además, esta función permite que el resto de las que ofrece MPI puedan ser usadas.

Para poder repartir trabajo entre diferentes tareas de MPI, es necesario conocer el ID de la tarea actual o "*rank*" y el número total de tareas. Tras ello, se pueden ocultar fragmentos de código para que estos solo puedan ser ejecutados por una tarea en específico, en el Listado 2.3, por la tarea 0. En el Listado 2.3, la inicialización del vector de datos solo la puede realizar la tarea 0. Tras ello, se reparte de manera igualitaria el tamaño del vector usando la función MPI_Scatter() y se suman localmente los elementos del fragmento del vector que cada tarea recibe. Es en este lugar, entre las líneas 38 y 42, donde **OpenMP** podría ser ejecutado para, además de utilizar diferentes computadores, aprovechar el paralelismo local de estos. Finalmente, se utiliza MPI_Reduce() para sumar todos los valores que las tareas MPI devuelven a la tarea que repartió el trabajo.

Como se puede comprobar, el funcionamiento de esta librería es una mezcla de POSIX Threads y OpenMP respecto a la complejidad del código. Sin embargo, conocer cómo funcionan los procedimientos que MPI ofrece es muy más complejo.

2.3.3 Interfaces de programación en clústeres

Un clúster no es más que un computador de arquitectura híbrida (Subsubsección 2.1.2.3), es decir, es un conjunto de computadores independientes o nodos conectados entre sí. Esto permite que las interfaces de programación mencionadas en las Subsecciones 2.3.1 y 2.3.2 sean utilizadas conjuntamente.

Por tanto, las APIs de memoria distribuida son utilizadas para compartir datos entre los diferentes nodos que conforman el clúster, y las APIs de memoria compartida y otras como CUDA u OpenCL (estas últimas cuando el nodo cuenta con aceleradores, como GPUs) se utilizan para explotar el paralelismo local de cada uno de los nodos.

Los desarrollos llevados a cabo en este TFG se ejecutan en un clúster ARM. En concreto, se desarrollan dos herramientas: una para controlar la afinidad de tareas **OpenMPI** y otra para generar informes de rendimiento a partir de mediciones del ancho de banda de las conexiones de red. El siguiente capítulo describe de manera detallada las herramientas desarrolladas para alcanzar estos objetivos.

3 HERRAMIENTAS DE GESTIÓN DE CLUSTER ARM

Este capítulo detalla las especificaciones, diseño e implementación de las herramientas que conforman este trabajo: **demangler** e **iorbenchtool**. La primera actúa como traductor entre las nomenclaturas de las CPUs lógicas de las arquitecturas x86 y ARM. La siguiente herramienta genera, a partir de los datos del ancho de banda de un nodo con el sistema de almacenamiento, un informe acerca del rendimiento de esta conexión.

Estas herramientas han sido creadas para solventar algunos problemas que surgen al portar o utilizar software diseñado para ser ejecutado en arquitecturas **x86** en **ARM**. Por ello resulta interesante conocer cómo estas arquitecturas están implementadas.

3.1 ARQUITECTURAS DE PROCESADORES

En esta sección se describen las arquitecturas **x86** y **ARM**. Es necesario entender correctamente porqué surgen los problemas de portabilidad de código entre ellas. Además, a lo largo de esta sección se habla de las tecnologías de paralelismo que implementan y el modelo de numeración de CPUs que la implementación de dichas tecnologías genera en cada una de las arquitecturas.

3.1.1 Arquitectura x86

A lo largo de esta subsección, se explica la implementación hardware de los procesadores x86 y la numeración que las CPUs de estos siguen.

Implementación de un procesador x86

Inicialmente, los procesadores **x86** no contaban con ni con técnicas de ILP ni de TLP. Sin embargo, las grandes ganancias de rendimiento de CPUs **x86** son generadas por la segmentación de instrucciones (*instruction pipelining*) y la creación de un procesador superescalar (*superscalar processor*), cuyo rendimiento se mejora con técnicas como la predicción de saltos (*branch prediction*) y la ejecución especulativa (*expeculative execution*).

El problema para la implementación de las técnicas ILP residía en una característica fundamental de los procesadores CISC **x86**: sus largas y dinámicas instrucciones. Para solventar este problema, Intel con su procesador Pentium Pro (i686) [1] de 1995 introdujo un

[1] Intel dejó nombrar a los procesadores por códigos porque los números no se pueden proteger como marca registrada. Intel Corp. v. Advanced Micro Devices, Inc., 756 F.Supp. 1292, 1293 (N.D. Cal. 1991) [7].

cambio fundamental en la microarquitectura de los procesadores **x86**. Cuando uno de estos procesadores obtiene una instrucción **x86**, esta se decodifica en micro-operaciones (μops), operaciones que se pueden ejecutar en un sistema segmentado. De esta manera se mantiene la compatibilidad hacia atrás, algo fundamental en **x86**, y los procesadores CISC **x86** pueden beneficiarse de las mejoras de rendimiento de un procesador segmentado [37]. Cabe destacar que estas μops no aparecen en el ISA, por lo que estas son un detalle interno de la implementación del procesador.

En el caso del procesador Pentium Pro, este está segmentado en 14 fases, siendo uno de estas la encargada de decodificar instrucciones **x86** a un conjunto de μops. Gracias a este diseño de microarquitectura, este procesador cuenta con OOO (*Out-Of-Order execution*), segmentación de instrucciones (*instruction pipelining*) y, además, es un procesador superescalar que permite hasta dos instrucciones de manera simultánea. Todas estas técnicas ILP se implementaron sobre el conjunto de las μops [38]. La implementación de estas μops se sigue utilizando en los procesadores **x86** actuales. El diagrama de bloques del núcleo de este procesador se puede ver en la Figura 3.1.

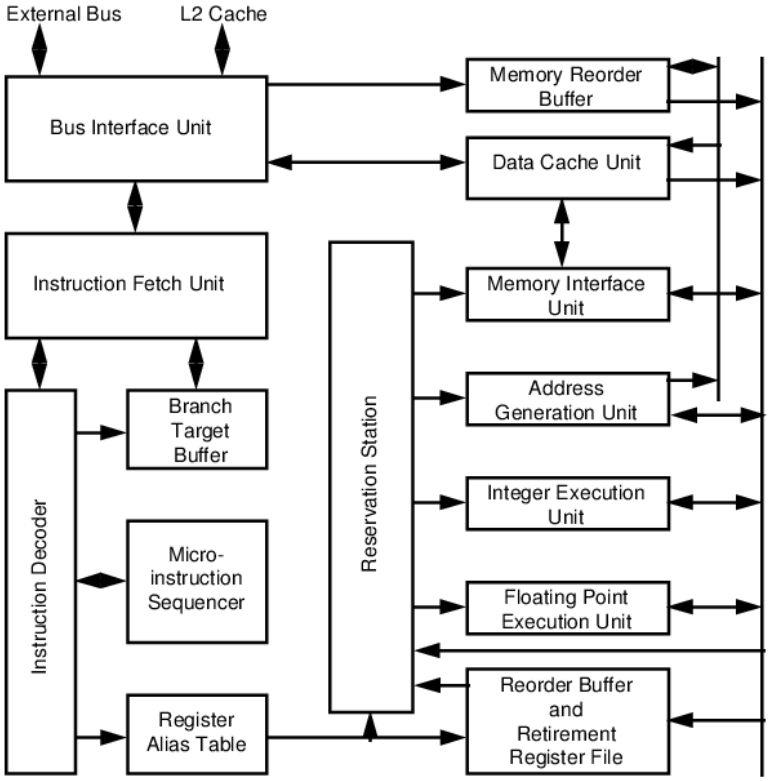


Figura 3.1: Diagrama de bloques del núcleo de Pentium Pro [39]

Tras las mejoras en ILP realizadas en el Pentium Pro, los procesadores comenzaron con la implementación de TLP mediante la adición de un núcleo más. Esto permite a los procesadores poder ejecutar diferentes *hilos de ejecución* de manera simultánea, siempre que estos sean aprovechados por el programador. Un ejemplo de un procesador multi-núcleo homogéneo se puede observar en la Figura 3.2.

Como muestra la Figura 3.2, este procesador cuenta con cuatro núcleos. Estos, además, implementan la técnica de TLP **SMT**. Esta permite que el ILP del procesador sea aprovechado

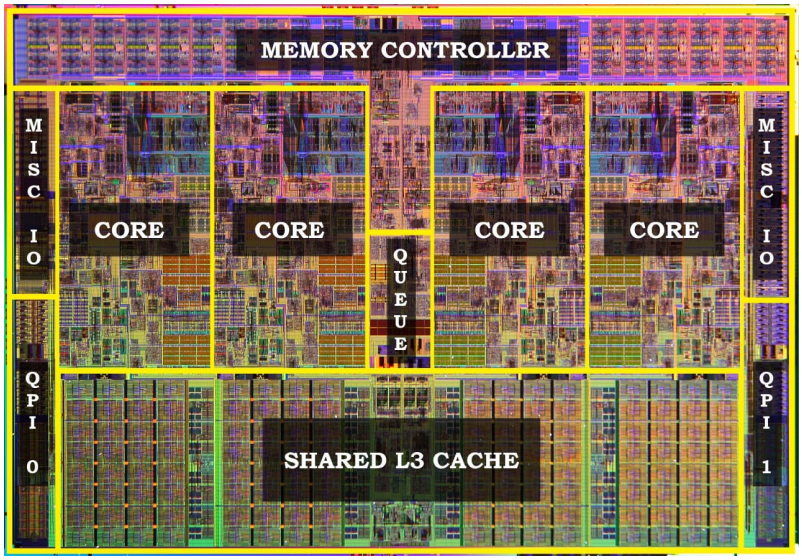


Figura 3.2: Diagrama de un procesador Intel Core de 6º generación

de una manera mucho más eficiente mediante la duplicación del hardware que guarda el estado de la arquitectura (principalmente registros de control y de propósito general) [40]. El nombre que la implementación de **SMT** tiene en procesadores Intel es *HyperThreading* (HT), incluida en algunas series de sus procesadores desde 2002.

La Figura 3.3 exhibe cómo el **SMT** es una técnica que permite un mejor aprovechamiento del ILP del procesador, prácticamente haciendo un uso completo de las capacidades ILP del procesador durante todo momento. Aun así, para las cargas de trabajo que provocan una invalidez de la cache de las otras tareas, el **SMT** puede conllevar pérdidas de rendimiento substanciales [41].

Numeración de CPUs lógicas en x86

Como se ha explicado anteriormente, que un procesador tenga habilitado el **SMT** puede suponer pérdidas del rendimiento si los trabajos que ejecuta compiten por los mismos recursos de la CPU [41]. Es por ello por lo que resulta fundamental poder indicar cómo se han de repartir las cargas de trabajo. Para permitir dirigir las tareas a unidades de procesamiento específicas nacen las nomenclaturas. En el caso de una CPU con el **SMT** deshabilitado, la numeración de las CPUs de un procesador sería similar al presentado en la Figura 3.4.

Como se puede comprobar en la Figura 3.4, esta numeración de CPUs es muy simple. Consiste en incrementar el índice de las CPUs consecutivas a partir de una inicial cuyo índice es 0.

HyperThreading, la implementación de **SMT** de Intel, permite a una CPU poder ejecutar de manera concurrente dos *hilos de ejecución*. Para ello, se duplica el hardware de la CPU encargado de guardar la información de los registros, pero no las unidades de ejecución. Esto permite a cada CPU (física) tener dos CPUs lógicas y poder contar con dos *hilos de ejecución*. Por ello, con la finalidad de aprovechar el rendimiento que HT puede ofrecer, resulta fundamental saber cómo estas CPUs lógicas están distribuidas entre las diferentes CPUs físicas del procesador, para así poder balancear correctamente la carga de trabajo que las

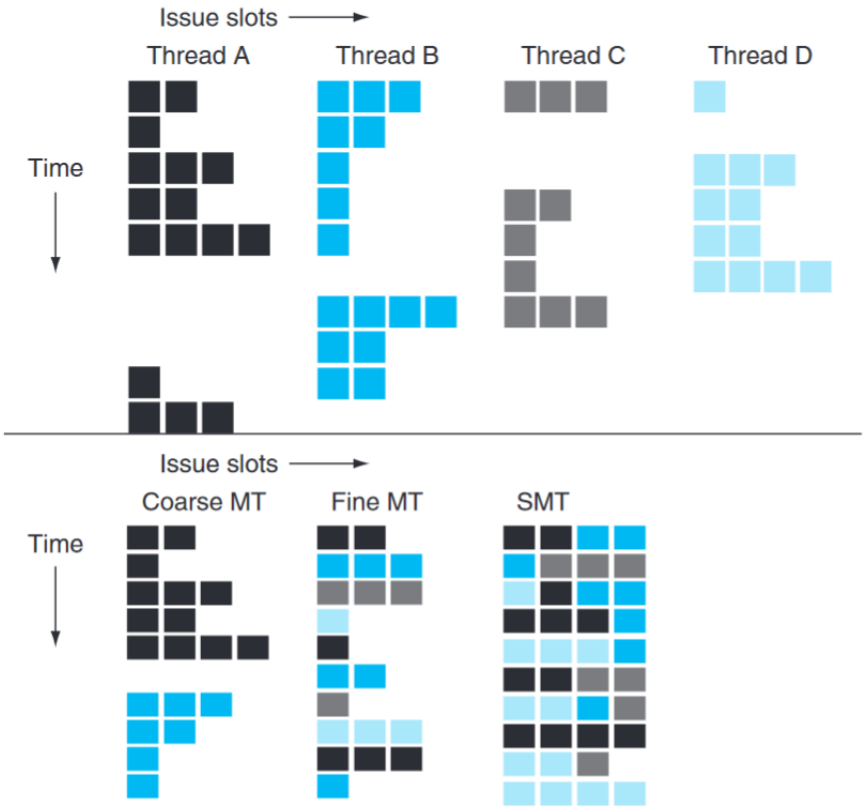


Figura 3.3: Diferentes técnicas de aprovechamiento del ILP del procesador con **SMT** 4. Los cuatro hilos (parte superior de la figura) se ejecutan solos en un procesador superescalar sin *multithreading*. La dimensión horizontal representa la capacidad de instrucciones por ciclo de reloj. La dimensión vertical representa la secuencia de ciclos de reloj. Un cuadradillo en blanco representa que ese *slot* no se usa en el ciclo de reloj al que pertenece [25].

CPUs lógicas reciben. Un procesador con HT numera sus CPUs lógicas de manera análoga a la presentada en la Figura 3.5.

La numeración de **x86** con HT activo consiste en numerar progresivamente las CPUs lógicas que pertenecen a una misma CPU física a partir de una CPU física inicial. De esta manera, como se observa en la Figura 3.5, un procesador de cuatro núcleos/CPUs físicas con HT activado tiene ocho CPUs lógicas numeradas como se muestra en dicha figura.

3.1.2 Arquitectura ARM

Esta subsección, trata, de igual manera que la subsección de x86, acerca de la implementación hardware y numeración de las CPUs de los procesadores ARM.

Implementación de un procesador ARM

Desde sus inicios, los procesadores **ARM** cuentan con ILP debido a que, al ser procesadores RISC, implementar técnicas como la segmentación de instrucciones es mucho más simple que en procesadores CISC. Además, debido a su sencillez respecto a las instrucciones (respecto a CISC), la decodificación de estas es muy rápida y no necesita hardware

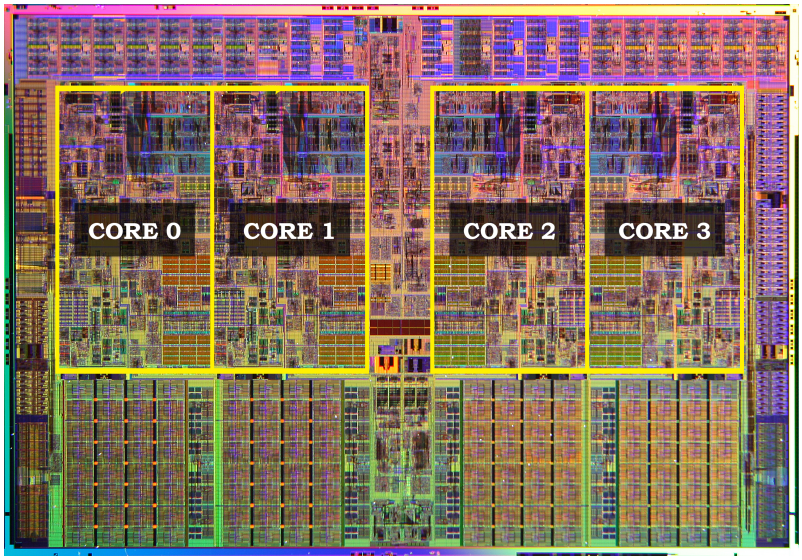


Figura 3.4: Numeración de las CPUs de un procesador

adicional como en los procesadores **x86**. El diagrama de bloques del procesador **ARM1** se puede observar en la Figura 3.6.

Respecto a la implementación de la segmentación, el procesador **ARM1** cuenta con tres etapas: *fetch*, *decode* y *execute*. En los procesadores **ARM** modernos esta segmentación varía desde nueve etapas hasta quince o más dependiendo del procesador. Generalmente, los procesadores **ARM** no cuentan con una implementación de **SMT**, debido a que esta tecnología no es adecuada para dispositivos de bajo consumo, en los cuales un diseño **big.LITTLE** resulta más apropiado. Ha sido recientemente cuando, gracias a la entrada de **ARM** en centros de datos y clústeres HPC, la tecnología **SMT** se ha aplicado a procesadores de esta arquitectura.

En ARM, **SMT** no tiene un nombre como HT para Intel. Sin embargo, la funcionalidad e implementación son las mismas.

Numeración de CPUs lógicas en ARM

Al igual que pasa con **x86**, cuando un procesador no tiene activado **SMT**, la manera en la que sus CPUs son numeradas es muy simple. Esto puede observarse en la Figura 3.4. En cambio, cuando este se activa, la nueva numeración es la que se muestra en la Figura 3.7.

Como se puede ver, la numeración de las CPUs lógicas en ARM prima que los índices consecutivos pertenezcan a CPUs físicas diferentes y contiguas.

3.2 Herramienta para control de afinidad bajo OpenMPI en ARM

Esta sección se divide en seis subsecciones en las que se muestra cuál es el problema que esta herramienta trata de resolver, las especificaciones y requisitos del software, su diseño y estructura, y, finalmente, su implementación.

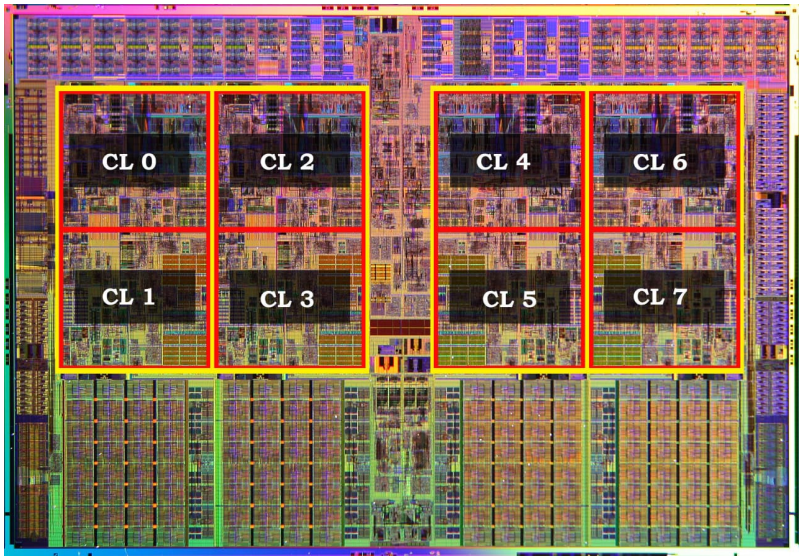


Figura 3.5: Numeración de las CPUs lógicas de un procesador x86 con *HyperThreading* activo. CL significa CPU lógica. Cada rectángulo amarillo representa una única CPU física con dos CPUs lógicas marcadas por rectángulos rojos.

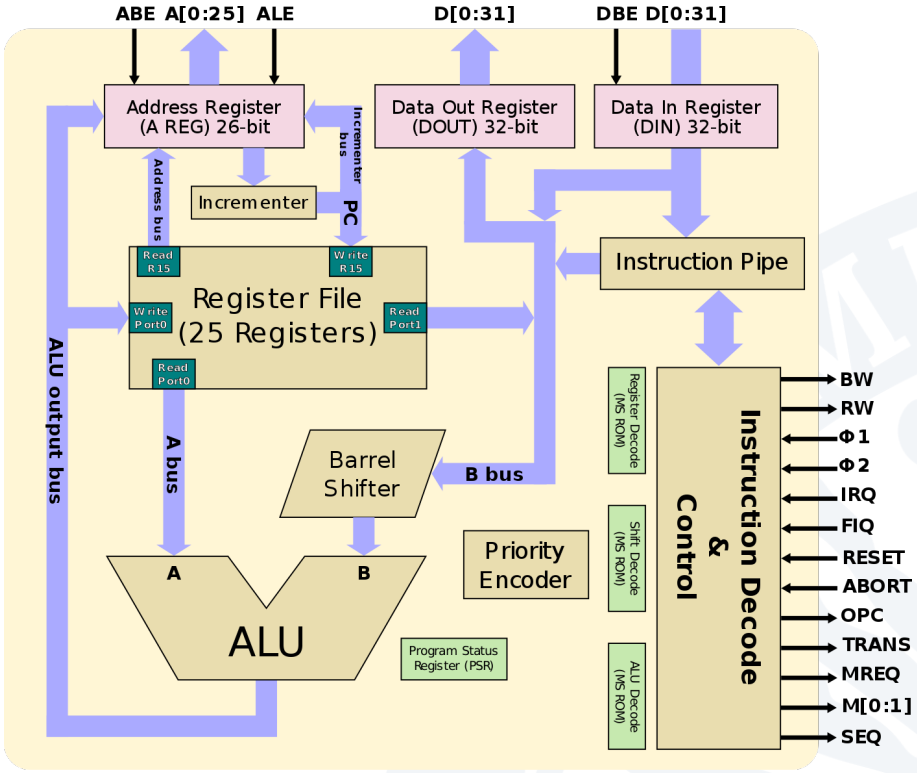


Figura 3.6: Diagrama de bloques del núcleo de ARM1 [8]

3.2.1 Definición del problema

Como se ha explicado en la Sección 3.1, cuando los procesadores tanto de Intel como de ARM tienen activados el **SMT** (una tecnología por la cual pueden multiplicar el número de CPUs lógicas que el procesador tiene, denominado *HyperThreading* para Intel y *SMT* para

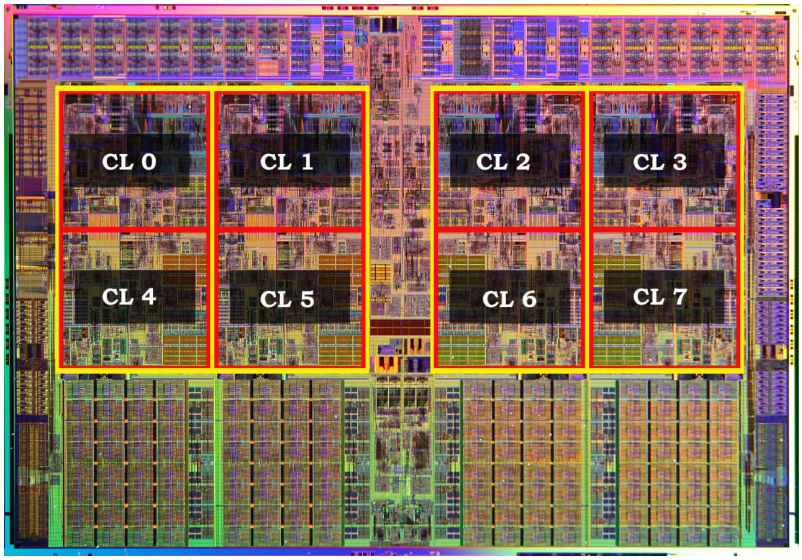


Figura 3.7: Numeración de las CPUs lógicas de un procesador ARM con **SMT** activo. CL significa CPU lógica. Cada rectángulo amarillo representa una única CPU física con dos CPUs lógicas marcadas por rectángulos rojos.

ARM) la numeración de sus CPUs lógicas varía. Una comparativa entre ambos estilos de numeración se puede ver en la Figura 3.8.

Los procesadores x86 con HT activado numeran sus CPUs lógicas siguiendo el modelo presentado en la Figura 3.8a. Este consiste en numerar progresivamente las CPUs lógicas que pertenecen a una misma CPU física a partir de un CPU lógica inicial. En cambio, para procesadores ARM con SMT activado, como se puede ver en la Figura 3.8b, su numeración prima que los índices consecutivos de CPUs lógicas pertenezcan a CPUs físicas diferentes y contiguas. A la primera de las numeraciones descritas, y presente en la Figura 3.8a, se denomina **nomenclatura Intel**. Al segundo modelo de nomenclatura, presentado en la Figura 3.8b, se denomina **nomenclatura ARM**.

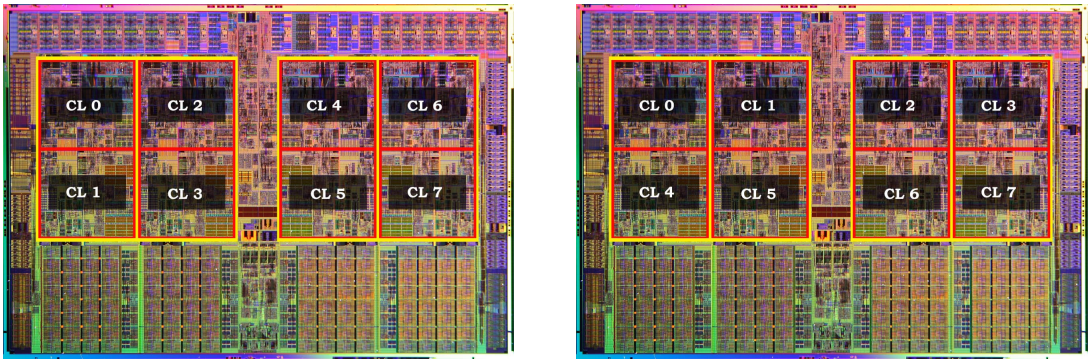
Esta diferencia en la numeración de las CPUs lógicas de los procesadores provoca que los programas que hacen uso de **OpenMPI** en sistemas de **nomenclatura Intel**, y son utilizados/portados a sistema de **nomenclatura ARM**, no se ejecuten sobre las CPUs lógicas inicialmente indicadas, lo que puede suponer pérdidas de rendimiento.

Este problema se puede observar en la Figura 3.9. Como se puede comprobar en la parte superior de dicha figura, se lanza el comando **OpenMPI** mostrado en el Listado 3.1, el cual ejecuta el programa **xthi** [2] sobre las CPUs lógicas 0, 1, 2, 128, 129 y 130 de este nodo, tal y como marca el rectángulo rojo de la esquina superior derecha. El resultado tras el lanzamiento de este comando muestra que el programa **xthi** ha sido ejecutado sobre las CPUs lógicas 0, 32, 64, 128, 160 y 192. Este fallo de asignación de CPUs lógicas para este programa no genera ninguna pérdida de rendimiento, pero muestra cuál es el problema.

```
1 mpirun -n 6 --report-bindings --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set
      0,1,2,128,129,130 ./xthi
```

Listado 3.1: Comando para ejecutar el programa xthi usando OpenMPI

[2] Este programa muestra la localización física de las CPUs lógicas donde un programa **OpenMPI** se ejecuta.



(a) Nomenclatura Intel **(b)** Nomenclatura ARM

Figura 3.8: Comparativa numeración CPUs lógicas entre **x86** y **ARM**. CL significa CPU lógica. Cada rectángulo amarillo representa una única CPU física con dos CPUs lógicas marcadas por rectángulos rojos.

Esto se debe a que los cálculos necesarios para situar físicamente una CPU lógica son diferentes entre las arquitecturas **x86** con HT activado y **ARM** con SMT. Para poder solventar este problema se requiere de una herramienta que, aplicando los cálculos correctos, pueda convertir desde la **numeración Intel** a la **numeración ARM**, tal y como se muestra en la Figura 3.10.

Una vez que se ha ilustrado el problema, se procede a explicar cómo se ha abordado. Primeramente, se ha calculado la correspondencia entre CPUs lógicas en arquitectura Intel y ARM. Las Ecuaciones 3.1, 3.2, 3.3 y 3.4 generan, a partir de una CPU lógica en **nomenclatura Intel** o **nomenclatura ARM**, la CPU física (CF) a la que corresponden (Ecuaciones 3.1 y 3.3) y el hilo hardware (*hardware thread*, HWT) al que pertenecen dentro de una CF (Ecuaciones 3.2 y 3.4).

$$C_{intel}(t_{intel}) = \left\lfloor \frac{t_{intel}}{SMT} \right\rfloor \tag{3.1}$$

$$T_{intel}(t_{intel}) = t_{intel} \text{ mód } SMT \tag{3.2}$$

$$C_{arm}(t_{arm}) = t_{arm} \text{ mód } CORES \tag{3.3}$$

$$T_{arm}(t_{arm}) = \left\lfloor \frac{t_{arm}}{CORES} \right\rfloor \tag{3.4}$$

donde t es una CPU lógica en **nomenclatura Intel** o en **nomenclatura ARM**, SMT es el **SMT** del nodo y $CORES$ es el número de CPUs físicas que tiene el primer socket del nodo.

A partir de las Ecuaciones 3.1 y 3.2 se puede crear una nueva expresión que realice la conversión de una CPU lógica en **nomenclatura Intel** a **nomenclatura ARM**. Esta es la Ecuación 3.5.

$$T_{arm}(t) = C_{intel}(t) + [T_{intel}(t) \cdot CORES] \tag{3.5}$$

donde t es una CPU lógica en **nomenclatura Intel**, $CORES$ es el número de CPUs físicas que tiene el primer socket del nodo y C_{intel} y T_{intel} son las Ecuaciones 3.1 y 3.2 respectivamente.

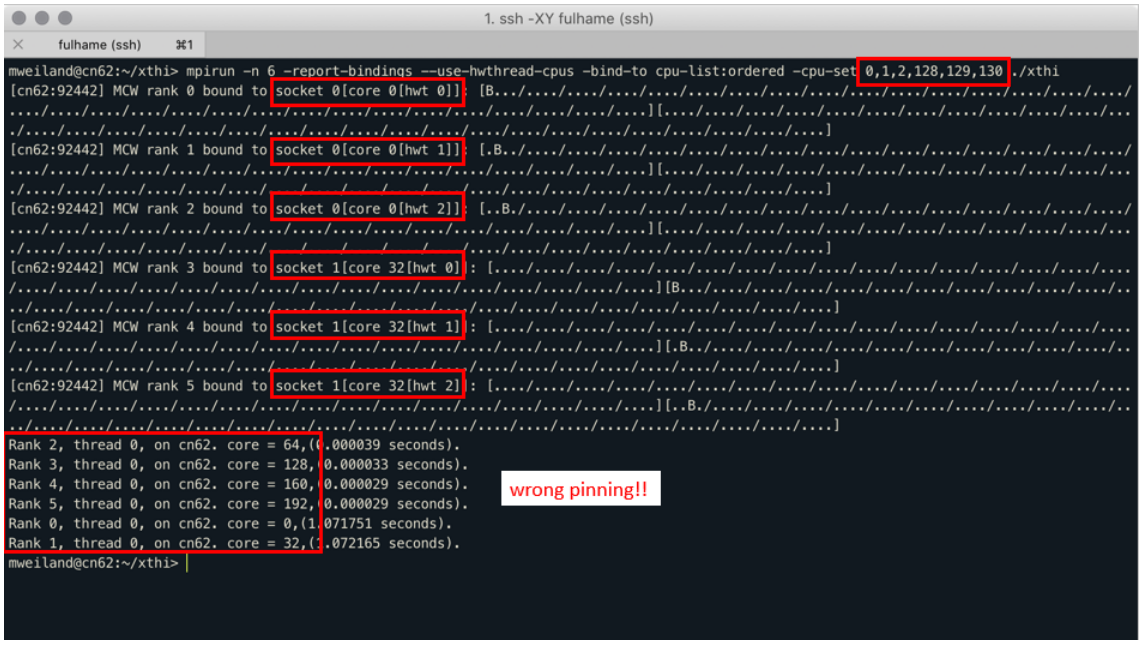


Figura 3.9: Afinidad incorrecta usando **OpenMPI**. Los rectángulos rojos inicial y final muestran cómo la afinidad inicial no se respeta

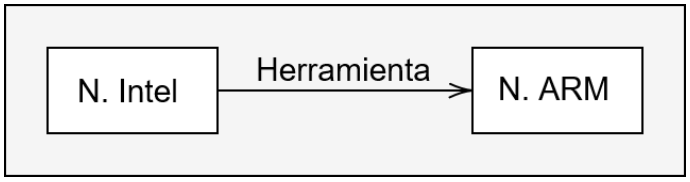


Figura 3.10: Diagrama de funcionamiento de la herramienta de traducción

A pesar de disponer de la Ecuación 3.5 para realizar la traducción de **nomenclatura Intel** a **nomenclatura ARM**, debido a las operaciones internas que **OpenMPI** utiliza para la asignación de la afinidad, si esta ecuación es usada, se seguirán generando índices de CPUs lógicas incorrectos. Para contrarrestar la conversión incorrecta de **OpenMPI**, hay que definir un formato de numeración de CPUs lógicas que al ser utilizado por **OpenMPI**, genere que la afinidad del programa final sea la correcta. La herramienta encargada de realizar esta conversión es **demangler**. La Figura 3.11 muestra el flujo que se ha de seguir para ejecutar correctamente un programa usando **OpenMPI** a partir de las CPUs en **nomenclatura Intel**.

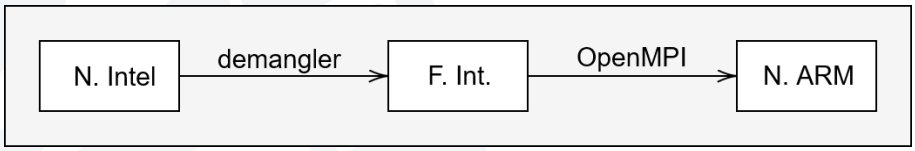


Figura 3.11: Diagrama de las conversiones necesarias para generar CPUs lógicas correctas. N.Intel y N.ARM significan **nomenclatura Intel** y **nomenclatura ARM** respectivamente. F.Int significa **formato intermedio**

Como se ha mencionado anteriormente, el **formato intermedio** se define como aquel que al ser utilizado como entrada en un comando **OpenMPI** permite que el programa que vaya a ser ejecutado lo haga sobre las CPUs lógicas indicadas inicialmente a **OpenMPI**. Este

formato está definido por como **OpenMPI** interpreta las CPUs lógicas de **nomenclatura Intel** de entrada a sus comandos. La Ecuación 3.6 muestra cómo, a partir de una CPU lógica en **nomenclatura Intel**, se genera su **formato intermedio**.

$$P_{int}(t) = C_{arm}(t) \cdot SMT + T_{arm}(t) \tag{3.6}$$

donde t es una CPU lógica en **nomenclatura Intel**, SMT es el **SMT** del nodo y C_{arm} y T_{arm} son las Ecuaciones 3.3 y 3.4 respectivamente. Esta ecuación es esencial para la implementación de esta herramienta.

También se puede definir la operación inversa a la mostrada en la Ecuación 3.6. Esta operación convierte una CPU lógica en **formato intermedio** a **nomenclatura Intel**. La Ecuación 3.7 muestra cómo se realiza esta operación.

$${P_{int}}^{-1}(t) = T_{intel}(t) \cdot CORES + C_{arm}(C_{intel}(t)) \tag{3.7}$$

donde t es una CPU lógica en **formato intermedio**, $CORES$ es el número de CPUs físicas que tiene el primer socket del nodo, C_{arm} es la Ecuación 3.3 y C_{intel} y T_{intel} son las Ecuaciones 3.1 y 3.2 respectivamente.

Ya definido el problema y las ecuaciones necesarias para su resolución, se procede a explicar las especificaciones del software, el diseño de los diferentes procedimientos de la herramienta y la implementación de estos.

3.2.2 Especificaciones del software

La herramienta **demangler** tiene como finalidad servir de capa traductora entre las diferentes nomenclaturas de CPUs lógicas de las arquitecturas x86 y ARM para que aplicaciones HPC que usen **OpenMPI** en sistemas x86 puedan ser ejecutadas (si el código de la aplicación lo permite) en un clúster ARM sin mayor problema. Este funcionamiento queda patente en la Figura 3.11.

Para ello, **demangler**, a partir de la entrada de un conjunto de CPUs lógicas en **nomenclatura Intel**, ha de generar otro listado con la traducción de dichas CPUs lógicas a un **formato intermedio**. Además, el listado de salida tiene que estar en forma de comando de **OpenMPI**, ya que esta herramienta tiene que integrarse dentro del sistema de gestión y planificación de tareas del clúster. El funcionamiento y un ejemplo de ello se muestra en la Figura 3.12.

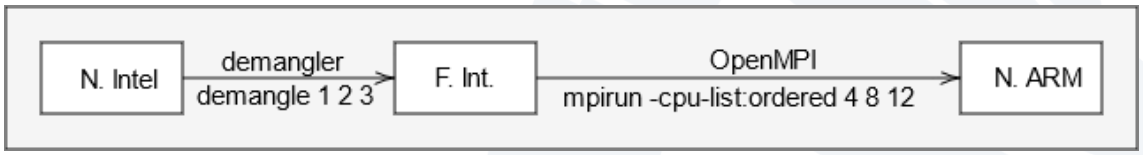


Figura 3.12: Diagrama de una traducción con ejemplo

Los comandos **OpenMPI** han de seguir el formato presentado en el Listado 3.2.

```
mpirun -n {0} --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set {1} {2}
```

Listado 3.2: Estructura del comando OpenMPI de salida de demangler

donde 0 es el número de CPUs lógicas introducidas, 1 es la lista de CPUs lógicas y 2 son las opciones extra para el comando **OpenMPI**.

La herramienta también ha de poder realizar la operación complementaria a la definida anteriormente. A partir de un listado de CPUs lógicas en **formato intermedio**, ha de generar un listado con dichas CPUs en **nomenclatura Intel**. Esta función servirá principalmente para realizar pruebas y comprobar si la implementación de las traducciones es correcta.

Finalmente, **demangler** ha de poder, a partir de ciertas indicaciones acerca de las CPUs del sistema (como la paridad de su índice), generar el listado de CPUs lógicas de entrada para la traducción de estas a **formato intermedio**.

Como se ha mencionado inicialmente, **demangler** es una herramienta cuya ejecución se realizará dentro de un clúster, por lo que la interfaz de esta tiene que adaptarse a este tipo de sistemas.

Debido a que los clústeres no utilizan, de manera general, entornos gráficos, **demangler** tiene que ser una herramienta **CLI** (Command Line Interface). Esto supone ventajas para el diseño de la herramienta, ya que las aplicaciones CLI suelen tener una interfaz de usuario más sencilla y rápida de diseñar. Además, que **demangler** sea una herramienta CLI permite que esta sea utilizada por otras herramientas y *scripts* del clúster, permitiendo que esta se integre de manera adecuada en el ecosistema de un clúster, justo como se indica al comienzo de esta sección.

A modo de resumen, las especificaciones generales del software son las siguientes:

- Traducir de **nomenclatura Intel** a **formato intermedio**.
- Traducir de **formato intermedio** a **nomenclatura Intel**.
- Generar listado de CPUs lógicas mediante filtros y traducirlo.
- Generar comando **OpenMPI** para las CPUs lógicas traducidas.
- Implementar CLI para integrar la herramienta en el ecosistema del clúster.

3.2.3 Diseño de la interfaz de usuario

Esta herramienta es una CLI, y, por tanto, para que sea usable, las opciones y comandos que esta ofrece han de ser pocos, simples y bien definidos. Estos se muestran en la Figura 3.13.

demangler ofrece dos *flags* para obtener información general acerca de la herramienta:

- **–help**: Muestra por consola los datos mostrados en la Figura 3.13.
- **–version**: Muestra la versión que se está ejecutando de **demangler**.

```
dmglr 1.1.0
Jerónimo Sánchez <jsg568@inlumine.ual.es>
Perform operations regarding mangled ARM cores when using OpenMPI.

USAGE:
    dmgler [OPTIONS] <SUBCOMMAND>

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information

OPTIONS:
    -m, --mpi-program <program>    Append 'program' to MPI recommended command. [default: <program>]
        --smt <smt>                SMT configuration (2 or 4). [default: 4]

SUBCOMMANDS:
    demangle    Demangle a set of Intel-styled threads
    get         Display a set of demangled threads that satisfy certain condition(s)
    help        Prints this message or the help of the given subcommand(s)
    mangle      Reverse demangle. From a set of demangled Intel-styled threads, return the threads in which the
                program will be finally executed
```

Figura 3.13: Interfaz de consola de **demangler**

Además, para permitir la configuración de estado global de ella, también ofrece dos opciones:

- **-m, –mpi-program <program>**: Añade al final del comando **OpenMPI** generado el texto de *<program>*. Esto corresponde con la variable "*2*" en el Listado 3.2.
- **–smt**: Configura el **SMT** para realizar los cálculos. Por defecto es **4**.

Los comandos que **demangler** ofrece son los siguientes cuatro:

- **demangle**: Este comando se encarga de convertir un conjunto de CPUs lógicas en **nomenclatura Intel** a formato intermedio, generando como salida un comando **OpenMPI**.
- **get** : Este comando es complementario al anterior. A partir de un conjunto de CPUs lógicas en **formato intermedio**, genera la traducción de estas a **nomenclatura ARM**.
- **help**: Muestra por consola los datos mostrados en la Figura 3.13 o la ayuda del subcomando indicado.
- **mangle**: Este comando permite seleccionar un conjunto de CPUs lógicas sin la introducción manual de estas haciendo uso de diferentes filtros. Finalmente, las CPUs lógicas filtradas se traducen a **formato intermedio** y se genera con ellas un comando **OpenMPI**.

3.2.4 Estructura de organización del código fuente

El código fuente del proyecto se estructura conforme la Figura 3.14. Esta estructura es la que Rust genera por defecto. El significado y utilidad de los archivos y directorios mostrados se explica a continuación:

- **.cargo**: Contiene información necesaria para encontrar las herramientas necesarias para compilar el proyecto para Aarch64.
- **.vscode**: Contiene los archivos de configuración de Visual Studio necesarios para realizar *debugging*.

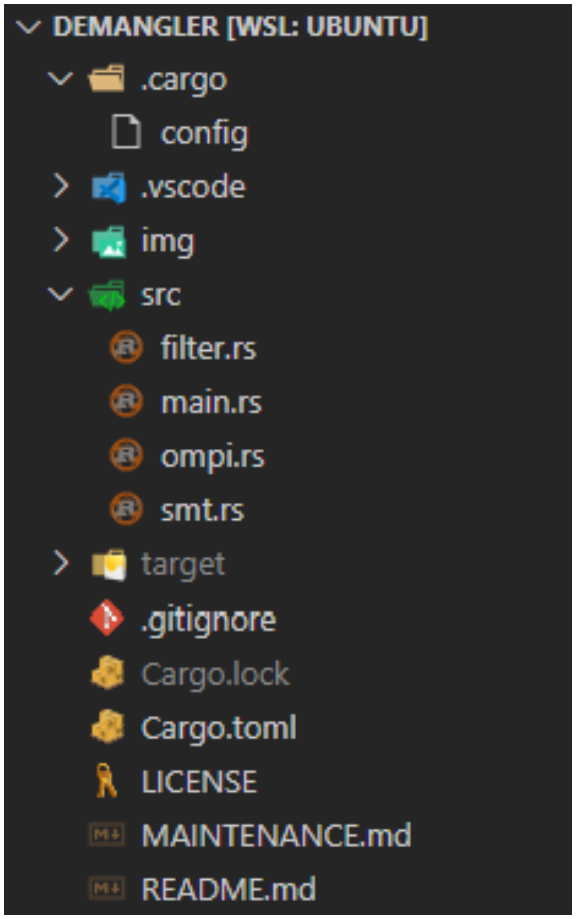


Figura 3.14: Estructura organizativa del código fuente de **demangler**

- **img**: Contiene las imágenes que se utilizan en el archivo *README.md*.
- **src**: Contiene los archivos de código fuente que conforman el programa. Estos tienen la siguiente utilidad:
  - ***filter.rs***: Contiene las funciones necesarias para dar funcionamiento al comando **get**.
  - ***main.rs***: Contiene el punto de entrada de la herramienta. Además, contiene las funciones y estructuras necesarias para verificar las entradas de consola y la lógica ejecución en función de dichas entradas.
  - ***ompi.rs***: Contiene las funciones relacionadas con los comandos **demangle** y **mangle**.
  - ***smt.rs***: Contiene utilidades para poder utilizar la configuración **SMT** de manera conveniente en cualquier parte del código.
- **target**: Contiene los artefactos generados durante la compilación del proyecto y el ejecutable final.
- ***.gitignore***: Este archivo indica a GIT que directorios y archivos ignorar. Son los mismos que aparecen en un tono gris semitransparente en la Figura 3.14.

- ***Cargo.toml y Cargo.lock***: El primero se encarga de indicar a la configuración del proyecto y librerías que este usa. El segundo guarda qué versión de cada una de las librerías (y librerías usadas por estas) está siendo usada, permitiendo que si el proyecto se vuelve a compilar estas versiones sean usadas. Esto se hace para permitir construcciones reproducibles del proyecto.
- ***LICENSE***: Este archivo es la licencia sobre la que el proyecto se desarrolla y distribuye. Apache-2.0 es la licencia escogida.
- ***MAINTENANCE.md***: Contiene una pequeña descripción de que hace cada módulo del código fuente para permitir que este sea modificado sin ayuda del autor inicial.
- ***README.md***: Contiene información acerca de cómo compilar el proyecto y una explicación básica de su funcionamiento.

3.2.5 Diseño de la herramienta

Durante este subapartado de diseño se explica el flujo de información general que la herramienta sigue para cada uno de sus comandos. Los detalles de las implementaciones de dichos comandos se muestran en la Subsección 3.2.6.

Cuando **demangle** recibe un comando por consola, la forma de este es validada para poder generar una estructura con todos los datos que han sido introducidos, como el nivel de **SMT**. Esta estructura tiene los siguientes campos:

- **Configuración SMT**
- **Configuración OpenMPI**
- **Comando**

Comando, según la Subsección 3.2.2, puede tener tres valores diferentes, **demangle**, **mangle** y **get**. Estos comandos son igualmente verificados para crear estructuras con los datos introducidos para su ejecución. En el caso de los dos primeros comandos mencionados, se comprueba que los datos introducidos correspondan con índices de CPUs lógicas, aunque en este paso no se compruebe su validez en el contexto del sistema. En el caso del comando **get**, se comprueba que las opciones del filtrado estén entre las siguientes: Socket, Paridad y Rango **SMT**. La forma y relación de estas estructuras se puede observar en la Figura 3.15.

Una vez creada una instancia de *CliInit*, la estructura que contiene toda la información acerca del comando introducido por consola, se procede a determinar qué comando ha sido ejecutado. Para ello, solamente hace falta comprobar el valor del enumerado *CliCommand* almacenado en la variable ***cmd*** de la instancia de *CliInit*, tal y como se muestra en el Listado 3.3. El pseudocódigo de **EjecucionDemangle**, **EjecucionMangle** y **EjecucionGet** se encuentran definidos y explicados en la Subsección 3.2.6.

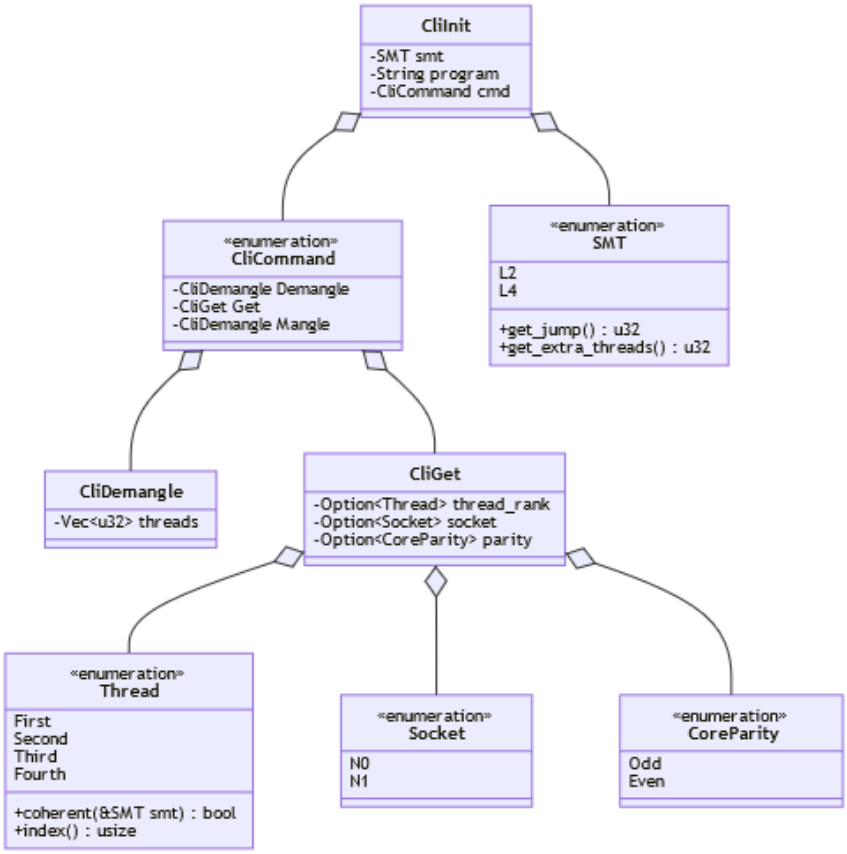


Figura 3.15: Diagrama UML para la verificación de los comandos de **demangler**

```
1 Segun CliInit ->cmd
2     Si Demangle Hacer EjecucionDemangle
3     Si Mangle Hacer EjecucionMangle
4     Si Get Hacer EjecucionGet
5 FinSegun
```

Listado 3.3: Pseudocodigo de seleccion del comando en demangle

Tras ser ejecutado el procedimiento correspondiente dependiendo del estado del enumerado *CliCommand*, los resultados de dichos procedimientos son mostrados por consola dentro un comando **OpenMPI** con la forma indicada en el Listado 3.2.

3.2.6 Implementación de la herramienta

A lo largo de esta Subsección se explica de una manera detallada la implementación de los procedimientos más importantes para el funcionamiento de la herramienta **demangler**. Estos ya se han mencionado con anterioridad y son los siguientes:

- **EjecucionDemangle**: Es el procedimiento encargado de realizar las traducciones de CPUs lógicas de **nomenclatura Intel** a **formato intermedio**. Es la implementación del comando **demangle**.

- **EjecucionMangle**: Este procedimiento se encarga de traducir las CPUs lógicas en **formato intermedio** a **nomenclatura ARM**. Es la implementación del comando **mangle**.

- **EjecucionGet**: Genera un listado de CPUs lógicas en **nomenclatura Intel** a partir de unos filtros y posteriormente los traduce a **formato intermedio**. Es la implementación del comando **get**.

La descripción de dichos procedimientos se realiza en el orden mostrado en la enumeración superior.

Primero, el procedimiento **EjecucionDemangle**, cuya implementación en pseudocódigo se muestra en el Listado 3.4, como se ha mencionado, traduce CPUs lógicas de **nomenclatura Intel** a **formato intermedio**.

```
1  Procedimiento EjecucionDemangle(input: Entero, smt: SMT)
2   Const
3       CORE_NUMBER: Entero = 32
4   Var
5       smt_val: Entero
6       jump: Entero
7       thread_value: Entero
8       col_index: Entero
9       row_index: Entero
10      output: Entero
11      flag: Booleano
12
13  smt_val <- smt como Entero
14  Si input > (smt_val * 2 * CORE_NUMBER) - 1 Hacer
15        FinProdecimiento error
16  FinSi
17
18  jump <- smt.get_extra_threads()
19  flag <- false
20
21  Si input >= jump Hacer
22    flag <- true
23    thread_value <- input - jump
24  Sino
25    thread_value <- input
26  FinSi
27
28  col_index <- thread_value / CORE_NUMBER
29  row_index <- thread_value mod CORE_NUMBER
30
31  output <- smt_val * row_index + col_index
32  Si flag = true Hacer
33    output <- output + jump
34  FinSi
35 FinProcedimiento output
```

Listado 3.4: Pseudocodigo del procedimiento EjecucionDemangle de demangler

Para ello, este procedimiento necesita como entradas un entero "*input*" que representa el índice de la CPU lógica en **nomenclatura Intel** y la configuración **SMT** "*smt*" configurada en la aplicación.

Tras ello, convierte a entero el valor de "*smt*" y comprueba que el índice "*input*" sea válido dentro de la configuración indicada a **demangler**.

A continuación, procede a obtener el número de CPUs lógicas extra que el valor **SMT** configurado genera en el sistema. En caso de **SMT 2** se añadirán 64 CPUs lógicas, 128 en caso de **SMT 4**, etc. Ya obtenido el número de CPUs lógicas extra que cada valor de **SMT** añade al sistema, el índice de CPU lógica de "*input*" se "mueve" al primer socket del sistema, indicando en la variable "*flag*" si este índice pertenece al segundo socket. Este índice se guarda en la variable "*thread_value*".

Seguidamente, se calculan los valores de las variables "*col_index*" y "*row_index*" los cuales corresponden a la división y módulo del valor de "*thread_value*" con *CORE_NUMBER*.

Finalmente, se multiplican "*smt_val*" y "*row_index*" y posteriormente se añade a este valor "*col_index*". Este cálculo corresponde a la implementación de la Ecuación 3.6, la cual realiza la traducción entre **nomenclatura Intel** a **formato intermedio**. Tras ello, este valor se almacena en la variable "*output*". Para reajustar el índice según el socket al que pertenecía inicialmente, si el valor de "*flag*" es verdadero, se añade "*jump*" al valor de "*output*" y se devuelve este como resultado de la traducción.

Respecto a la implementación de **EjecucionMangle**, esta se muestra en el Listado 3.5.

```
1  Procedimiento EjecucionMangle(input: Entero, smt: SMT)
2  Const
3     CORE_NUMBER: Entero = 32
4  Var
5     smt_val: Entero
6     row_index: Entero
7     col_index: Entero
8     start: Entero
9     jump: Entero
10
11 smt_val <- smt como Entero
12 row_index <- input / smt_val
13 col_index <- input mod smt_val
14
15 start <- col_index * smt-val
16 jump <- smt.get_jump()
17
18 Si input >= CORE_NUMBER * smt_val Hacer
19    start <- start + jump
20 FinSi
21
22 Segun smt
23    Si SMT::L2 Hacer FinProcedimiento start + row_index
24    Si SMT::L4 Hacer FinProcedimiento start + (row_index mod CORE_NUMBER)
25 FinSegun
26 FinProcedimiento
```

Listado 3.5: Pseudocodigo del procedimiento EjecucionMangle de demangler

La implementación de este procedimiento es muy similar a la ofrecida acerca **EjecucionDemangle** respecto la definición de entradas, aunque el algoritmo es muy distinto. La funcionalidad de este es la de, a partir de una CPU lógica en **formato intermedio**, generar la CPU lógica en la que se ejecutará. Para ello, este algoritmo sigue una serie de pasos.

Primero, el procedimiento requiere como entradas, al igual que la implementación de *EjecucionDemangle*, un entero "*input*" el cual representa el índice de la CPU lógica en **formato intermedio** y la configuración **SMT** "*smt*" que ha sido configurada en la aplicación.

Tras ello, convierte a entero el valor de "*smt*". Sin embargo, no se comprueba que el índice "*input*" sea válido dentro de la configuración indicada a **demangler** ya que inicialmente, la implementación de este procedimiento sirvió como referencia para el desarrollo del resto de ecuaciones presentes en la Subsección 3.2.1. De hecho, este procedimiento es la implementación de la Ecuación 3.7. Seguidamente, se calculan la división y módulo de "*input*" con "*smt_val*" y se almacenan en las variables "*row_index*" y "*col_index*" respectivamente.

A continuación, se calcula el valor de inicio para realiza el cálculo necesario para el valor final. Además, se obtiene el salto en número de índices que se realiza en caso de que la CPU lógica de entrada pertenezca al segundo socket. Si se da esta situación, se añade "*jump*" al valor de "*start*".

Finalmente, dependiendo del valor del **SMT**, se devuelve el valor de "*start*" más "*row_index*" o el cálculo completo representado por la Ecuación 3.7.

La implementación del procedimiento **EjecucionGet** es el más simple de los tres mostrados y el más largo. En líneas generales, la implementación sigue el flujo mostrado en el Listado 3.6.

```
1  Procedimiento EjecucionGet(smt: SMT, th: Thread, sck: Socket, p: CoreParity)
2  Var
3      table: Vector<Entero>
4      tmp: Vector<Entero>
5      smt_val: Entero
6
7      table <- GenerarTabla(smt)
8      smt_val <- smt como Entero
9
10     Si sck presente Hacer
11         Segun smt
12             Si SMT::L2 Hacer table <- table(..smt.get_extra_threads())
13             Si SMT::L4 Hacer table <- table(smt-get_extra_threads()..)
14         FinSegun
15     FinSi
16
17     Si th presente Hacer
18         Si !th.coherent() Hacer
19             FinProcedimiento error
20         FinSi
21
22         tmp <- new Vector()
23         Para chunk Con table.bloques(smt_val)
24             tmp.add(chunk(th.index()))
25         FinPara
26     FinSi
27
28     Si p presente Hacer
29         Segun p
30             Si CoreParity::Odd Hacer table <- table.filtrar(elemento %2 == 1)
31             Si CoreParity::Even Hacer table <- table.filtrar(elemento %2 == 0)
32         FinSegun
33     FinSi
34
35 FinProdecimiento EjecucionDemangle(smt, table)
```

Listado 3.6: Pseudocodigo del procedimiento EjecucionGet de demangler

Como se puede observar en el Listado 3.6, el procedimiento sigue un flujo muy simple. Primero genera una tabla donde las filas son los índices de los HWT y las columnas los índi-

ces de las CPUs físicas. Según si están presentes los diferentes tipos de filtros, se realizan diferentes operaciones sobre ella para crear un conjunto de CPUs lógicas con los requerimientos introducidos.

Primero, se filtra en función de la pertenencia al socket. Tras ello, se filtra según la pertenencia de la CPU lógica al *hardware thread* 0, 1, 2, 3, etc. Finalmente, se filtra según si el índice de la CPU lógica es par o impar. Estos filtros son acumulativos en el orden mostrado y, además, se pueden ignorar, por lo que no es obligatorio tener que filtrar por las tres presentadas al mismo tiempo. Una vez filtrada la tabla y obtenidas las CPUs lógicas en **nomenclatura Intel** correspondiente a dichos filtros, se utiliza la implementación del procedimiento **EjecucionDemangle** para convertir dichas CPUs lógicas en **nomenclatura Intel** a **formato intermedio**.

Con la implementación del procedimiento **EjecucionGet** quedan explicados los procedimiento más importantes y esenciales para el funcionamiento de **demangler**, así de una idea general del flujo de datos que el programa sigue para realizar su función.

3.3 Herramienta para la generación de informes de rendimiento de red

Al igual que la sección anterior, esta se divide en seis subsecciones en las que se muestra cuál es el problema que esta herramienta trata de resolver, las especificaciones y requisitos del software, su diseño y estructura, y, finalmente, su implementación.

3.3.1 Definición del problema

Durante las últimas décadas, los clústeres HPC han crecido tanto en número de núcleos por socket como en número de nodos disponibles. Esto último se ha posibilitado, en parte, no solo gracias a estándares como MPI, los cuales han dado sentido a este tipo de crecimiento, sino también gracias a las interconexiones de red que los nodos de los clústeres tienen. Por ello resulta de gran interés monitorizar las diferentes métricas existentes del rendimiento de la red, ya que de ello depende gran parte de la potencia que estos clústeres ofrecen. Las métricas más importantes para el monitorizar el rendimiento de la red son las siguientes [42]:

- **Latencia**: Mide el tiempo que pasa entre el envío y recibo de un paquete de información.
- **Pérdida de paquetes**: Mide el número o porcentaje de paquetes de información que se han perdido durante una comunicación.
- **Ancho de banda**: Mide la cantidad de datos, expresados en potencias de bits, que son transmitidos durante un periodo temporal.

Las interconexiones de red han de ser lo más rápidas posibles para no suponer un factor limitante en la potencia de los supercomputadores, por ello, estas han de estar diseñadas en consonancia con el resto de la arquitectura del clúster. Esta es la razón por la que algunos fabricantes de clústeres han decidido diseñar sus propias interconexiones, como Fujitsu en el caso de Fugaku con las conexiones *Tofu interconnect* [43]. También existen fabricantes que se dedican únicamente al diseño y fabricación de este tipo de interconexiones. Uno de ellos es **Mellanox** [44], responsable de las interconexiones del clúster **Fulhame** [45].

Actualmente, una de las herramientas más utilizadas para realizar este tipo de mediciones es **IOR** [3]. Esta es un *benchmark* paralelo de IO utilizado para medir el ancho de banda de los sistemas paralelos de almacenaje, haciendo uso de diferentes interfaces y diferentes patrones de acceso [46].

Debido a que **ARM** no tiene mucho recorrido en el ámbito del HPC, los drivers de algunos componentes de los clústeres que no han sido diseñados junto al clúster pueden no funcionar correctamente. Esto puede deberse a que la implementación de los drivers de estos componentes no esté muy pulida. Por ello, es vital monitorizar si el rendimiento esperado para ese mismo componente es distinto al que tiene en un sistema con otra arquitectura (principalmente x86).

El problema es que para generar datos que sean fiables y representativos han de ejecutar muchas iteraciones y diferentes variaciones de los test de acceso al sistema de almacenamiento, generando como resultado una ingente cantidad de archivos con los datos necesarios para medir el rendimiento de las interconexiones. Un archivo de IOR se divide en cuatro secciones: Información del sistema, configuración del *benchmark*, resultados y resumen de los resultados. Estas secciones se muestran en los Listados 3.7, 3.8, 3.9 y 3.10 respectivamente.

```
1  IOR-3.2.1: MPI Coordinated Test of Parallel I/O
2  Began               : Mon Jul 20 10:49:35 2020
3  Command line        : /lustre/hdd/fh04/fh04/jero/ior -v -w -r -i 4 -F -o /lustre/hdd/fh04
   /fh04/jero/iortmp/ior-test.file -t 1m -b 512g
4  Machine             : Linux cn01
5  Start time skew across all tasks: 0.00 sec
6  TestID              : 0
7  StartTime           : Mon Jul 20 10:49:35 2020
8  Path                : /lustre/hdd/fh04/fh04/jero/iortmp
9  FS                  : 85.4 TiB   Used FS: 0.8%   Inodes: 229.1 Mi   Used Inodes: 0.2%
10 Participating tasks: 1
```

Listado 3.7: Archivo IOR - Informacion del sistema

```
1  Options:
2  api                 : POSIX
3  apiVersion          :
4  test filename       : /lustre/hdd/fh04/fh04/jero/iortmp/ior-test.file
5  access              : file-per-process
6  type                : independent
7  segments            : 1
8  ordering in a file  : sequential
9  ordering inter file : no tasks offsets
10 tasks               : 1
11 clients per node    : 1
12 repetitions         : 4
13 xfersize            : 1 MiB
14 blocksize           : 512 GiB
15 aggregate filesize  : 512 GiB
```

Listado 3.8: Archivo IOR - Configuracion benchmark

[3] esta herramienta está accesible en la dirección `https://github.com/hpc/ior`

```
Results:

access    bw(MiB/s)  block(KiB) xfer(KiB)  open(s)    wr/rd(s)   close(s)   total(s)
 iter
------    ---------  ---------- ---------  --------   --------   --------   --------
 ----
Commencing write performance test: Mon Jul 20 10:49:35 2020
write     804.48     536870912  1024.00    0.001373   651.70     0.002082   651.71     0
Commencing read performance test: Mon Jul 20 11:00:26 2020
read      1101.85    536870912  1024.00    0.000280   475.83     0.000719   475.83     0
remove    -          -          -          -          -          -          22.23      0
Commencing write performance test: Mon Jul 20 11:08:44 2020
write     819.75     536870912  1024.00    0.000747   639.57     0.000850   639.57     1
Commencing read performance test: Mon Jul 20 11:19:24 2020
read      1100.00    536870912  1024.00    0.000302   476.62     0.000997   476.62     1
remove    -          -          -          -          -          -          21.58      1
Commencing write performance test: Mon Jul 20 11:27:42 2020
write     822.75     536870912  1024.00    0.000718   637.23     0.000927   637.24     2
Commencing read performance test: Mon Jul 20 11:38:19 2020
read      1092.81    536870912  1024.00    0.000484   479.76     0.000958   479.76     2
remove    -          -          -          -          -          -          20.66      2
Commencing write performance test: Mon Jul 20 11:46:40 2020
write     801.47     536870912  1024.00    0.000742   654.15     0.000998   654.16     3
Commencing read performance test: Mon Jul 20 11:57:34 2020
read      1109.44    536870912  1024.00    0.002137   472.57     0.000430   472.57     3
remove    -          -          -          -          -          -          23.76      3
Max Write: 822.75 MiB/sec (862.72 MB/sec)
Max Read:  1109.44 MiB/sec (1163.34 MB/sec)
```

Listado 3.9: Archivo IOR - Tabla de resultados

```
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev   Max(OPs)   Min(OPs)  Mean(OPs)
   StdDev    Mean(s) Test# #Tasks tPN reps fPP reord reordoff reordrand seed segcnt
 blksiz    xsize aggs(MiB)   API RefNum
write         822.75     801.47     812.11       9.26     822.75     801.47     812.11
     9.26  645.66791      0       1    1     4    1      0        1         0    0       1
 549755813888  1048576  524288.0 POSIX       0
read         1109.44    1092.81    1101.03       5.92    1109.44    1092.81    1101.03
     5.92  476.19484      0       1    1     4    1      0        1         0    0       1
 549755813888  1048576  524288.0 POSIX       0
Finished              : Mon Jul 20 12:05:50 2020
```

Listado 3.10: Archivo IOR - Resumen de resultados

La gran cantidad de archivos y la estructura de estos provoca que analizar toda esta información de manera manual sea muy tedioso. Es por ello por lo que el disponer de una herramienta capaz de procesar todos estos informes y generar archivos de hojas de cálculo es muy importante. Este es el nacimiento de **oirbenchtool**.

iorbenchtool ha de ser una herramienta simple de usar que pueda generar archivos .xlsx (Excel) con las estadísticas generadas por **IOR** y una gráfica de ellas demostrando los puntos clave del test. Además, debe poder procesar grupos de pruebas en conjunto y permitir, aunque sea de una manera primitiva, definir plantillas para la generación de dichos informes.

3.3.2 Especificaciones del software

Para poder llevar al cabo su cometido, **iorbenchtool**, a partir de las rutas de un conjunto de archivos de entrada, ha de generar uno o varios informes Excel acerca de los datos de

los archivos iniciales. Los archivos de salida Excel han de presentar los datos mediante una gráfica simple, además de contener los datos más importantes para la realización de dicha gráfica.

iorbenchtool ha de permitir, de manera dinámica, cambiar la plantilla con la que se crea el informe Excel, para así poder adaptar la salida de esta herramienta a otro tipo de representación de los datos de rendimiento del ancho de banda. También, ha de poder unificar todos los archivos Excel generados en un único informe Excel si así se desea.

Debido a que esta herramienta puede tardar mucho tiempo en procesar grandes cantidades de archivos, ha de tener un *log*, por consola y archivo para indicar el estado de la herramienta, así como de los problemas que puedan surgir durante su ejecución. Esto es bastante importantes debido que, si la versión de **IOR** se actualiza, puede cambiar el formato de sus archivos, haciendo que esta herramienta funcione incorrectamente o directamente no lo haga.

Finalmente, **iorbenchtool** ha de poder ejecutarse en un clúster, aunque no es un requerimiento necesario, puede ser muy útil para cuando el número de archivos sea muy grande para que copiarlos fuera del clúster sea muy costoso/tedioso. Por ello, **iorbenchtool** es una herramienta **CLI**. Como se menciona en la Subsección 3.2.2, una aplicación CLI supone ventajas tanto porque permite generar una interfaz de uso sencilla de manera rápida, y, además, permite que esta herramienta sea usada por otras herramientas y scripts del clúster, provocando que **iorbenchtool** se integre de manera adecuada dentro del ecosistema de un clúster, en este caso, en el del clúster **Fulhame**.

A modo de resumen, se muestran las especificaciones generales del software en el listado siguiente:

- Analizar archivos **IOR** y procesar sus datos.
- Permitir analizar y procesar archivos de manera recurrente, a partir de un directorio raíz.
- Generar informes Excel acerca de los datos procesados.
- Permitir cambiar la plantilla Excel utilizada para la generación de los informes.
- Capacidad de *logging* tanto por consola como mediante archivos de *log*.

3.3.3 Diseño de la interfaz de usuario

iorbenchool es una herramienta CLI, y, por tanto, los comandos y opciones que tiene que debe ofrecer para ser utilizable han de ser pocos, simples y bien definidos. Para la creación de dichos comandos, se utiliza de referencia los requisitos software definidos. Debido a que solo se define una acción principal, esta herramienta contará con un único comando, por lo que este se ejecutará siempre que la herramienta sea usada. Por tanto, las opciones de **iorbenchtool** son las que permiten a esta herramienta poder variar su funcionamiento dinámicamente. Se pueden definir ocho opciones para **iorbenchtool** usando los requerimientos software como referencia. Estas se pueden comprobar en la Figura 3.16.

Las opciones que **iorbenchtool** ofrece tienen las siguientes funciones:

```
USAGE:
Parse a set of files under a directory:
  ior-parser ../ior-files/
Parse a set of subdirectories with IOR files each:
  ior-parser --recursive ../folder-root/
Parse a set of subdirectories then, summarizes all the reports into a single one:
  ior-parser --recursive --unite ../folder-root/
Parse a set of IOR files with a custom Excel template:
  ior-parser --template ./ExcelTemplate.cs ../ior-files/

  -r, --recursive     Recursively search for IOR files under the path provided.

  --dump              Dump the content of the parsed IOR files.

  -v, --verbose       Display all the detailed log information.

  --log               Log information into a log file on directory path.

  -u, --unite         Unite all Excel reports into a single one. Only valid when --recursive is present.

  -t, --template      Template path for the Excel reports.

  --help              Display this help screen.

  --version           Display version information.

  DirPath (pos. 0)    Required. Directory path where IOR output file(s) can be found.
```

Figura 3.16: Interfaz de consola de **iorbenchtool**

- **-r, –recursive**: Esta opción indicará a la herramienta que realice una búsqueda recursiva en los directorios indicados.
- **–dump**: Genera un archivo de texto con la información interna del analizador de archivos.
- **-v, –verbose**: Indica a la herramienta que ha de mostrar información detallada acerca de su funcionamiento.
- **–log**: Indica a la herramienta que la información de *logging* ha de ser guardada en un archivo. Si esta opción no se especifica, los *logs* se muestran por consola.
- **-u, –unite**: Esta opción indica a **iorbenchtool** que todos los informes Excel que sean generados han de unificarse en un único archivo (sin eliminar los anteriores).
- **-t, –template**: Mediante esta opción se permite realizar un cambio de la plantilla usada para la generación de los informes Excel.
- **–help**: Muestra la información de ayuda de la herramienta. Esta opción genera la Figura 3.16.
- **–version**: Muestra la versión instalada de la herramienta.

3.3.4 Estructura de organización del código fuente del proyecto

El proyecto de la herramienta **iorbenchtool** se estructura conforme a la Figura 3.17. Cada una de las carpetas y archivos tienen las siguientes funciones:

- **img**: Contiene las imágenes que forman parte del archivo *README.md*.
- **src**: Este directorio aloja el código fuente del programa. Se divide en dos subdirectorios:

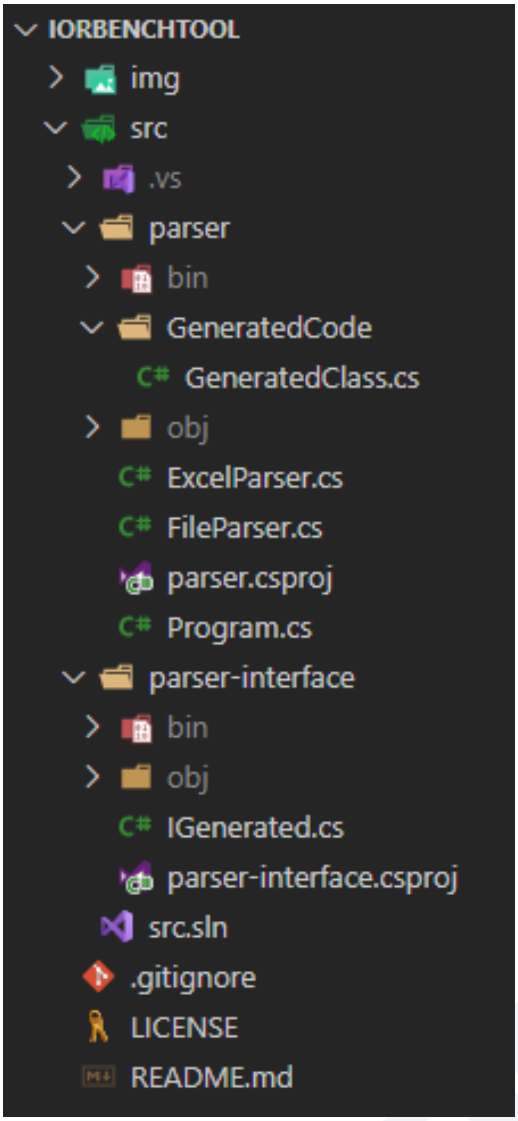


Figura 3.17: Estructura organizativa del código fuente de **iorbenchtool**

- **parser**: Contiene el código fuente de la herramienta.
  - **bin y obj**: Contienen los artefactos generados durante la compilación y ejecución en modo *debug* de la herramienta CLI.
  - ***Generated Code* / *GeneratedClass.cs***: Este archivo contiene la plantilla Excel generada mediante código
  - *ExcelParser.cs*: Contiene las funciones que serializan la información procesada de IOR en el archivo Excel siguiendo la plantilla.
  - *FileParser.cs*: Contiene las clases encargadas de contener la información de los archivos IOR, además de deserializarlos.
  - *Program.cs*: Punto de entrada del programa. Contiene las clases y funciones para serializar la entrada de comandos y ejecutar los códigos oportunos para las funciones indicadas por consola.

  - **parser-interface/IGenerated.cs**: Contiene la interfaz que las plantillas han de implementar para ser consideradas como tales.

- *.gitignore*: Indica qué archivos y carpetas son ignorados por GIT.
- *LICENSE*: Es la licencia del proyecto bajo la que se hace público y se distribuye.
- *README.md*: Incluye información básica acerca del uso de la herramienta, así de cómo crear nuevas plantillas (más detalles en una wiki).

3.3.5 Diseño de la herramienta

En este subapartado se explica el flujo de información de la herramienta sigue para llevar a cabo su funcionalidad. La implementación de estas funciones se detalla en la Subsección 3.3.6.

En el momento que esta herramienta recibe un comando por consola, este tiene que ser validado antes de poder ser usado. Para ello, se crea una estructura muy simple compuesta de opciones para cada una de las opciones definidas en la interfaz del software. Esta estructura es *CliOptions* y tiene la estructura mostrada en la Figura 3.18.

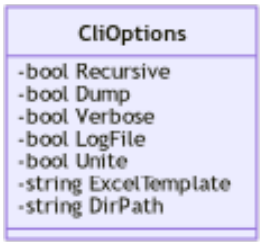


Figura 3.18: Diagrama UML para la verificación de los comandos de **iorbenchtool**

Los campos booleanos de la Figura 3.18 representan si la opción que representan con su nombre ha sido introducida por consola. El primero de los dos últimos (ExcelTemplate) es el correspondiente a la opción **–template**, en la que se indica la localización de la plantilla Excel que se desea usar. Por último, DirPath representa el directorio del cual se quieren analizar los archivos IOR contenidos.

Ya creada una instancia de la clase que contiene la información acerca de las opciones indicadas en el comando que se está ejecutando, se procede a inicializar la configuración del sistema de *logging*, definiendo el nivel de información dependiendo de la opción **–verbose**. Tras ello, si la opción **–log** se ha usado, la información de *logging* se guardará en un archivo con la fecha y hora de la ejecución de la herramienta. Tras crear la configuración, se inicia el sistema de *logging*. A continuación, se comprueba si la opción **–template** ha sido indicada. En caso afirmativo, se lee y compila el archivo indicado en esta opción, creando una plantilla Excel dentro de la memoria del programa. Si no es así o hay algún problema durante la compilación, se usa una plantilla por defecto. Además, también se comprueba si la opción **–recursive** ha sido indicada, obteniendo todos los archivos IOR que existan en los subdirectorios a partir de la indicada inicialmente si es así. Durante la configuración de esta

opción se comprueba si la opción **–unite** se ha indicado. Finalmente, se procesan todos los archivos IOR y se genera un archivo Excel o varios (dependiendo si **–unite** está presente). El pseudocódigo de esta descripción del funcionamiento de la herramienta se puede comprobar en el Listado 3.11.

```
1   Vars:
2       opts: CliOptions
3       logCfg: LoggerCfg
4       logger: Logger
5       template: ExcelTemplate
6       dirs: Lista<Directorios>
7
8   Si opts.Verbose Hacer
9       logCfg <- LoggerDetallado()
10  Sino
11      logCfg <- LoggerSimple()
12  FinSi
13
14  Si opts.LogFile Hacer
15      logCfg <- LoggerEnArchivo(logCfg)
16  FinSi
17
18  logger <- CrearLogger(logCfg)
19  template <- CargarPlantillaExcel(opts.ExcelTemplate)
20
21  Si opts.Recursive Hacer
22      dirs <- ListarDirectoriosRecursivamente(opt.DirPath)
23
24      Para dir : dirs
25          AnalizarDirectorio(dir)
26      FinPara
27
28      Si opts.Unite Hacer
29          UnificarExcels()
30      FinSi
31
32  Sino
33      AnalizarDirectorio(opts.DirPath)
34  FinSi
```

Listado 3.11: Pseudocodigo del funcionamiento general de iorbenchtool

3.3.6 Implementación de la herramienta

En esta subsección se muestra, de manera detallada, la explicación e implementación de los procedimientos más importantes para el funcionamiento de la herramienta **iorbenchtool**. Algunos de ellos se muestran en el Listado 3.11, y son los siguientes:

- **CargarPlantillaExcel**: Compila un archivo generando un objeto del lenguaje capaz de generar archivos Excel con un formato definido.
- **AnalizarDirectorio**: Se encarga de analizar, procesar y generar un archivo Excel a partir de un grupo de archivos IOR de un directorio.
- **AnalizarArchivo**: Este procedimiento se usa dentro de **AnalizarDirectorio** y el más importante de la herramienta. Se encarga de generar un objeto de C# con la información de un archivo IOR.

Estos procedimientos se explican en el orden del listado anterior.

El primer procedimiento, **CargarPlantillaExcel**, cuya implementación en pseudocódigo se muestra en el Listado 3.12, se encarga de compilar un archivo de código fuente en C# a un objeto propio del lenguaje para poder cargar dinámicamente sin que el usuario que haya creado la plantilla tenga que compilar el código.

```
1  Procedimiento CargarPlantillaExcel(path: Cadena)
2   Const
3      saveLocation: Cadena = "template.dll"
4   Var
5      templateText: Cadena
6      compiler: CSharpCompilation
7      templateType: Type
8      template: IGenerated
9
10  templateText <- Leer path
11
12  compiler <- NuevaUnidadCompilacion(templateText)
13  compiler.Emit(saveLocation)
14
15  templateType <- CargarEnsamblado(saveLocation)
16  template <- BuscarInterfaz(templateType)
17
18  Si template es null Hacer
19     FinProcedimiento PlantillaPorDefecto()
20  Sino
21     FinProcedimiento template
22  FinSi
23
24 FinProdecimiento
```

Listado 3.12: Pseudocodigo del procedimiento CargarPlantillaExcel de iorbenchtool

Para cumplir con su cometido, el procedimiento **CargarPlantillaExcel** necesita de entrada la localización del archivo de código fuente usado como plantilla. Este tiene que implementar la interfaz *IGenerated* provista por el archivo **parser-interface/*IGenerated.cs***. Dicha interfaz define una única función, la cual genera una hoja de un archivo Excel siguiendo una plantilla.

Una vez cargado el código fuente en la variable "*templateText*", se crea una instancia del compilador de C# en la variable "*compiler*". A este compilador, se le añaden las diferentes librerías que necesita para que la compilación sea exitosa, y posteriormente, utilizando la función ***Emit()***, se genera un archivo **.dll** (*Dynamic Link Library*) con el código compilado y se guarda dentro de la carpeta donde se encuentra el programa bajo el nombre ***template.dll***.

Posteriormente, este ensamblado se carga usando las funciones que C# ofrece para ello. Este ensamblado es un objeto del lenguaje el cual permite comprobar que tipos de datos contiene y las interfaces que estos implementan. Esto se utilizará en el procedimiento **BuscarInterfaz** el cual devuelve una instancia de la interfaz **parser-interface/*IGenerated.cs*** si ha encontrado algún tipo que la implemente dentro del ensamblado generado anteriormente. De esta manera, si la compilación falló y no se generó ningún tipo que implemente dicha interfaz, se puede controlar esta situación y utilizar una incluida por defecto dentro de la herramienta, devolviendo finalmente la plantilla compilada o la por defecto en caso de error de la primera

El siguiente procedimiento, **AnalizarDirectorio**, se encarga de generar un archivo Excel a partir de archivos analizados y procesados por el procedimiento **AnalizarArchivo**. El pseudocódigo **AnalizarDirectorio** se muestra en el Listado 3.13.

El procedimiento **AnalizarDirectorio** requiere como entrada de datos la ruta del directorio sobre el que analizar los archivos IOR. Primeramente, carga todos los archivos IOR del directorio en la variable "*files*" y para cada uno de ellos, los analiza usando el procedimiento **AnalizarArchivo** y guarda este resultado en un vector de resultados. Estos resultados son del tipo *Info*, el cual tiene la estructura mostrada en la Figura 3.19. En ella se puede ver la existencia de una clase principal con dos clases agregadas: *Options* y *Result*. Estas guardan los datos del bloque IOR de información del sistema (Listado 3.7), información del *benchmark* (Listado 3.8) y resultados (Listado 3.9), respectivamente.

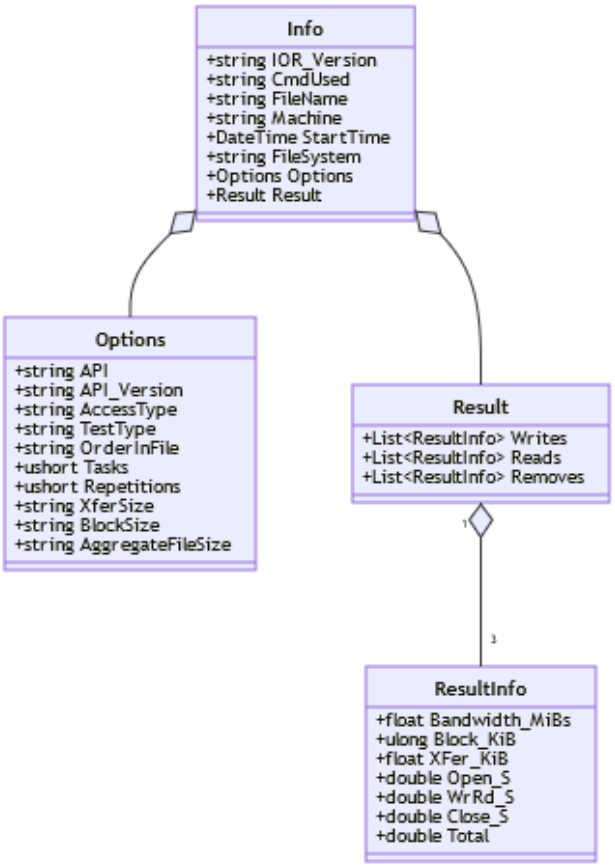


Figura 3.19: Diagrama UML de clases para procesamiento de información de IOR de iorbenchtool

Tras ello, crea un archivo Excel usando la plantilla que haya sido configurada mediante **–template** y la completa con los datos generados. Finalmente, guarda la hoja de cálculo y añade este Excel y la ruta donde ha sido guardado en un registro, utilizado en caso de que la opción **–unite** haya sido utilizada.

```
Procedimiento AnalizarDirectorio(path: Cadena)

 Var:
    files: Lista<Cadena>
    parsed: Lista<Info>
    excel: Excel

 files <- ObtenerArchivos(path)

 Para file : files Hacer
    parsed.Add(AnalizarArchivo(file))
 FinPara

 excel <- CrearExcel()
 CompletarDatosExcel(excel, parsed)

 excel.Save()
 AnadirExcelARegistro(excel)

FinProcedimiento
```

Listado 3.13: Pseudocodigo del procedimiento AnalizarDirectorio de iorbenchtool

A continuación, se procede a explicar el procedimiento **AnalizarArchivo**, el más importante de la herramienta. Este analiza y procesa la información de un archivo IOR a las estructuras que conforman la Figura 3.19. El pseudocódigo de este procedimiento se muestra en el Listado 3.14.

```
Procedimiento AnalizarArchivo(file: Cadena)
 Var
    texto: Cadena
    filas: Lista<Cadena>
    indexOpciones: Entero
    indexResultados: Entero
    info: Info
    opciones: Options
    resultado: Result

 texto <- Leer file
 filas <- Separar(texto, "\n")

 Para i = 0 Con i < filas.Length Paso i <- i + 1
    Si fila.EmpiezaPor("Options:") Hacer
        indexOpciones <- i
        info <- ProcesarInformacion(filas(..i))
    Sino fila.EmpiezaPor("Results:") Hacer
        indexResultados <- i
        opciones <- ProcesarOpciones(filas((indexOpciones + 1)..indexResultados))
        resultado <- ProcesarResultado(filas(indexResultados..))
    FinSi
 FinPara

 info.Options <- opciones
 info.Result <- resultado

FinProcedimiento info
```

Listado 3.14: Pseudocodigo del procedimiento AnalizarArchivo de iorbenchtool

Este procedimiento, para realizar su cometido, lee el archivo IOR presente en la ruta contenida en "*file*" dentro de la variable "*texto*". Posteriormente, divide el contenido por líneas y

guarda esta división dentro del vector "*filas*". A partir de aquí, para cada línea, se comprueba si esta comienza por *Options:* o por *Results:*. Si esta comienza por lo primero, significa que ya se ha delimitado el primer bloque del archivo IOR, el bloque de información del sistema (Listado 3.7), por lo, a continuación, este se procesa mediante el procedimiento **ProcesarInformacion**. Algo similar ocurre cuando una línea comienza por *Results:* salvo que en este caso indica que se han delimitado los bloques de configuración del *benchmark* (Listado 3.8) y resultados (Listado 3.9). Ahora, estos bloques, son procesados usando los procedimientos **ProcesarOpciones** y **ProcesarResultado** respectivamente. Finalmente, la instancia de la clase *Info* generada por **ProcesarInformacion** se completa con las clases *Options* y *Results* generadas por **ProcesarOpciones** y **ProcesarResultado**.

Los procedimientos **ProcesarInformación**, **ProcesarOpciones** y **ProcesaResultado** son muy similares. Estos necesitan de entrada un bloque de IOR (vector de líneas), el cual se *tokeniza* en un mapa de claves y valores. Tras el *tokenizado*, de manera diferente en cada uno de los procedimientos, se completa el tipo de dato buscando entre las claves el valor para una determinada variable de dicho tipo de datos. Finalmente, se devuelve el tipo de datos creado al procedimiento **AnalizarArchivo**.

Con la explicación del procedimiento **AnalizarArchivo** queda concluida la sección de implementación de los procedimientos más importantes para el funcionamiento de la herramienta **iorbenchtool** y se procede a evaluar ambas herramientas desarrolladas en este TFG.

4 Evaluación de las herramientas desarrolladas

En este capítulo se evalúan las herramientas desarrolladas en este trabajo para validar el correcto funcionamiento de estas. Las pruebas son realizadas en el clúster **Fulhame**, cuya descripción se encuentra en la Sección 4.1. Además, también se muestran ejemplos de uso de los diferentes comandos y opciones que ambas herramientas presentan, sirviendo este capítulo como una breve guía de uso de dichas herramientas.

Por tanto, este capítulo se divide en tres seccione: una para describir el clúster **Fulhame**, donde las pruebas se realizan y dos para las evaluaciones de las herramientas desarrolladas, una sección para cada una de ellas.

4.1 Características del clúster Fulhame

El clúster **Fulhame** nace de una colaboración entre HP, ARM, SUSE y tres universidades de UK, entre las que se encuentra el EPCC, siendo esta el emplazamiento escogido para instalar el sistema.

Figura 4.1: Cabina abierta mostrando algunos nodos de **Fulhame**

Fulhame cuenta con 64 nodos de computación HPE Apollo 70. Cada uno de ellos está formado por dos procesadores Cavium ThunderX2 con dos procesadores de 32 núcleos y 128GiB de memoria RAM (creando un sistema de 4096 núcleos y 8192 GiB de RAM). Todos los nodos están interconectados haciendo uso de las conexiones Inifiniband de Mellanox [45].

Bajo este clúster se realizan las todas las pruebas del software desarrollado en este trabajo. Por ello resulta de gran interés conocer en detalle los componentes de **Fulhame**.

4.1.1 Cavium - ThunderX2

Este es el nombre de la familia de procesadores desarrollados por Cavium (comprada por Marvell en 2018). Existen diferentes modelos, de los cuales, el utilizado en Fulhame es el más potente de ellos. Esta familia de procesadores está basada en la microarquitectura **Vulcan** de Cavium.

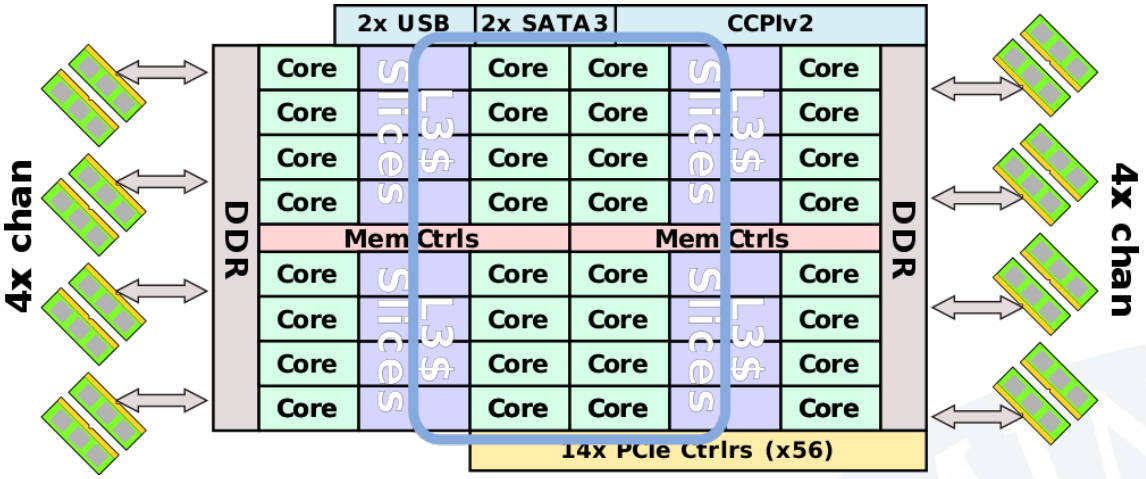


Figura 4.2: Diagrama general de un procesador ThunderX2 [47]

En la Figura 4.2 se puede observar el diagrama general de un procesador ThunderX2. Este cuenta con 32 núcleos basados en el ISA ARMv8.1-A, el cual hace uso de la extensión Aarch64, la cual ofrece un nuevo conjunto de instrucciones (A64) y, además, ofrece características SIMD mejoradas (NEON de 128-bits). Además, este implementa **SMT** hasta nivel 4, por lo que este procesador puede cuadruplicar el número de CPUs (lógicas) que ofrece .

El conjunto de CPUs del procesado están interconectadas mediante un bus bidireccional en forma de anillo. Cada núcleo está unido a este mediante una memoria caché L3 distribuida de 32MiB, siendo 1MiB para cada núcleo.

En mitad de este bus, separando el conjunto de 32 núcleos en cuatro grupos de ocho CPUs se sitúan los controladores de memoria del procesador. Estas controladoras actúan como puerta al exterior de los núcleos, ya que el acceso a RAM, red e IO se hace a través de ellas.

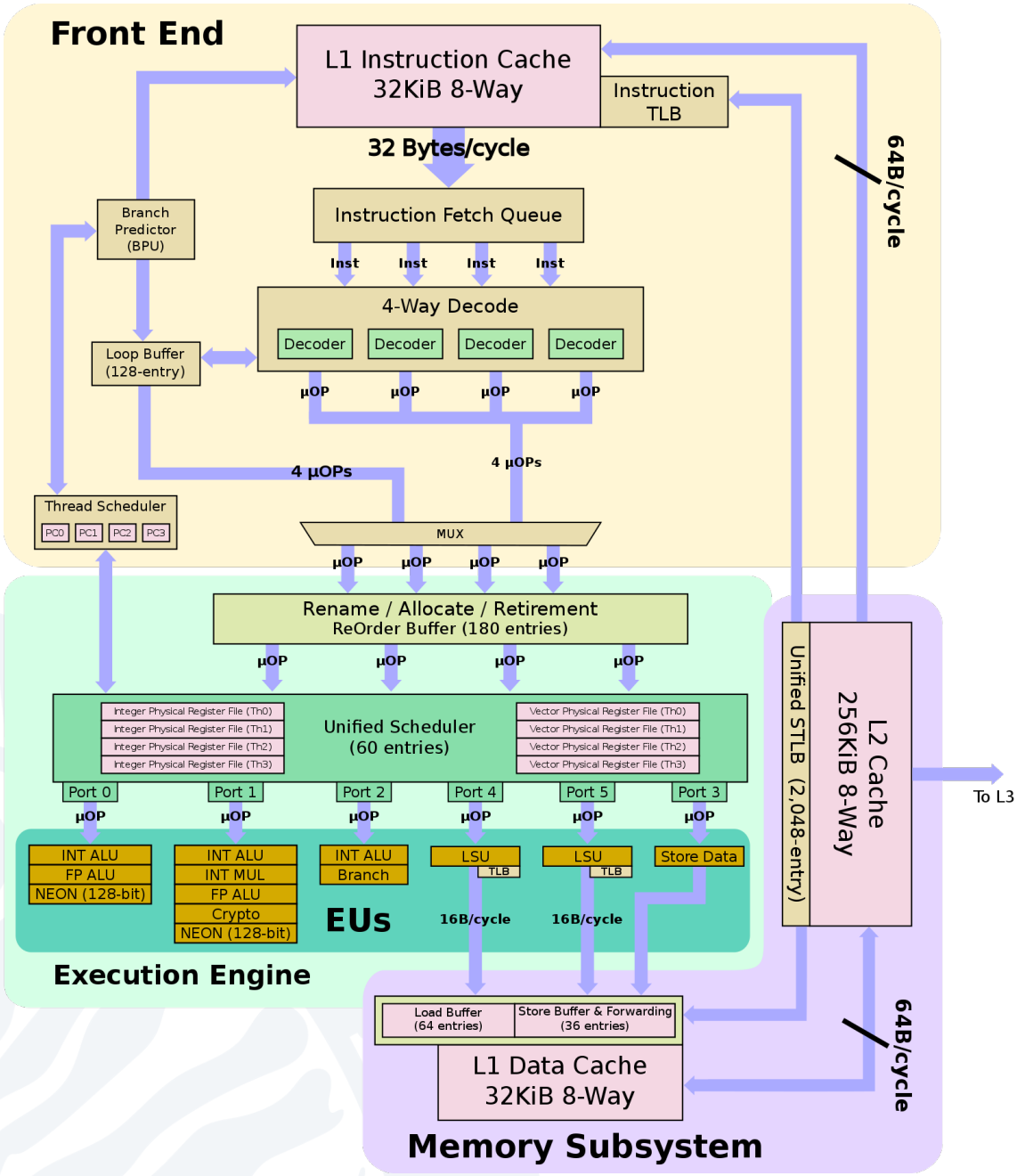


Figura 4.3: Diagrama bloques de un núcleo de ThunderX2 [47]

En la Figura 4.3 se muestra el diagrama de bloques de un núcleo del procesador. En él se puede comprobar las memorias caché internas del núcleo. Este cuenta con una caché de 256KiB L2 y con una memoria caché de 64KiB L1 separada en dos bloques de 32KiB, uno para instrucciones y otro para datos. Los núcleos basados en **Vulcan** están segmentados en 15 etapas. Además, la microarquitectura **Vulcan** decodifica las instrucciones ARM en μops e implementa hasta **SMT 4**.

Esto último puede observarse en la Figura 4.3, en el bloque de *frontend* (amarillo, abajo izquierda). Se puede ver que implementa un planificador de tareas que alimenta directamente al planificador unificado, el cual es finalmente ejecutado.

El diseño del ThunderX2 hace que este sea un procesador muy eficiente en cargas de trabajo limitadas por el ancho de banda de las memorias, y también se muestra competente para las tareas más computacionales [48].

4.1.2 Numeración de CPUs lógicas en Fulhame

Debido a que los nodos del clúster **Fulhame** están formados por procesadores ARM, la numeración que estos siguen es la misma que la mostrada y explicada en la Subsubsección 3.1.2.2. Esta consiste en numerar con índices consecutivos las CPUs lógicas priorizando que estas pertenezcan a CPUs físicas diferentes y consecutivas, como se puede comprobar en la Figura 3.7.

Esta numeración se ha mostrado numerando las CPUs lógicas de un nodo con un solo procesador, por lo que en caso de los nodos del clúster **Fulhame** esta varía ligeramente, ya que estos están compuestos de dos procesadores. Esta se puede observar en la Tabla 4.1.

Tabla 4.1: Disposición de los hilos en la **nomenclatura ARM** en clúster **Fulhame**. CF significa CPU física. HWT significa *hardware thread*.

	Socket 0					Socket 1				
CF / HWT	0	1	2	..	31	0	1	2	..	31
0	0	1	2	..	31	128	129	130	..	159
1	32	33	34	..	63	160	161	162	..	191
2	64	65	66	..	95	192	193	194	..	223
3	96	97	98	..	127	224	225	226	..	255

La Tabla 4.1 ha sido generada para un nodo configurado con **SMT 4**. Si la configuración de este fuera **SMT 2**, la Tabla 4.1 no tendría las últimas dos filas (correspondientes a CF 2 y 3). Aun así, los índices presentes en dicha tabla no cambiarían.

4.2 Evaluación de demangler

Esta sección trata acerca de la evaluación de la herramienta **demangler**. Dicha herramienta, la cual se encuentra explicada en la Sección 3.2, se encarga de traducir CPUs lógicas de **nomenclatura Intel** a **nomenclatura ARM** para que programas que hagan uso de la librería **OpenMPI** puedan ejecutarse correctamente bajo sistemas que hagan uso de esta última nomenclatura, como el clúster **Fulhame**.

Para verificar que esta herramienta cumple con su cometido, es necesario recordar cuales han de ser las funciones que esta realiza, las cuales se encuentran en la Subsección 3.2.2 y son las siguientes:

1. Traducir de **nomenclatura Intel** a **formato intermedio**. Corresponde con el comando **demangle**.
2. Traducir de **formato intermedio** a **nomenclatura Intel**. Corresponde con el comando **mangle**.
3. Generar listado de CPUs lógicas mediante filtros y traducirlo. Corresponde con el comando **get**.

Para verificar el funcionamiento del comando **demangle**, se hace uso de la Figura 3.9. En ella se puede comprobar que las CPUs lógicas iniciales, presentes en el rectángulo rojo superior derecho, no se corresponden con las que **OpenMPI** utiliza en la ejecución del programa **xthi** (rectángulo rojo inferior izquierdo). Inicialmente, se piden las CPUs lógicas 0, 1, 128, 129 y 130, las cuales se convierten en las CPUs lógicas 0, 32, 64, 128, 160 y 192 durante la ejecución de **OpenMPI**.

Por lo tanto, para comprobar y verificar el correcto funcionamiento del comando **demangle**, se introduce en la herramienta **demangler** el comando presente en la Figura 4.4. En este se puede comprobar que se introducen las CPUs lógicas iniciales mostradas en la Figura 3.9: 0, 1, 2, 128, 129 y 130. Además, se añade el programa que se desea ejecutar con el comando **OpenMPI**.

```
demangler on  master [!] is  v1.0.0 via  v1.51.0
❯ ./target/release/dmgler --smt 4 --mpi-program ./xthi demangle 0 1 2 128 129 130
mpirun -n 6 --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set 0,4,8,128,132,136 ./xthi
```

Figura 4.4: Ejecución del comando **demangle** de herramienta **demangler**

En la Figura 4.4, además del comando anteriormente descrito, se puede observar la salida de la ejecución de este en la parte inferior de la figura. En ella se puede ver que el comando **OpenMPI** generado indica que los índices de las CPUs lógicas en **formato intermedio** son: 0, 4, 8, 128, 132 y 136.

Finalmente, el comando **OpenMPI** mostrado en Figura 4.4 se ejecuta en un nodo del clúster **Fulhame**, generando del resultado mostrado en la Figura 4.5. Se puede comprobar que las CPUs lógicas resultado de utilizar **demangler** son utilizadas en la Figura 4.5 (rectángulo rojo esquina superior derecha).

Como se puede observar en la Figura 4.5, en el rectángulo rojo inferior izquierdo, las CPUs lógicas resultado de la ejecución son las indicadas inicialmente, tanto en la Figura 3.9, como en la Figura 4.4, por lo que el funcionamiento del comando **demangler** es correcto.

Ahora, se procede a verificar el funcionamiento del comando **mangle**. Este es el encargado de traducir CPUs lógicas de **formato intermedio** de vuelta a **nomenclatura Intel**. Su principal utilidad fue verificar la correcta implementación de la Ecuación 3.6.

Para realizar dicha verificación, se procede a convertir las CPUs lógicas en **formato intermedio** generadas por el comando **demangle** y mostradas en la Figura 4.4 a **nomenclatura**

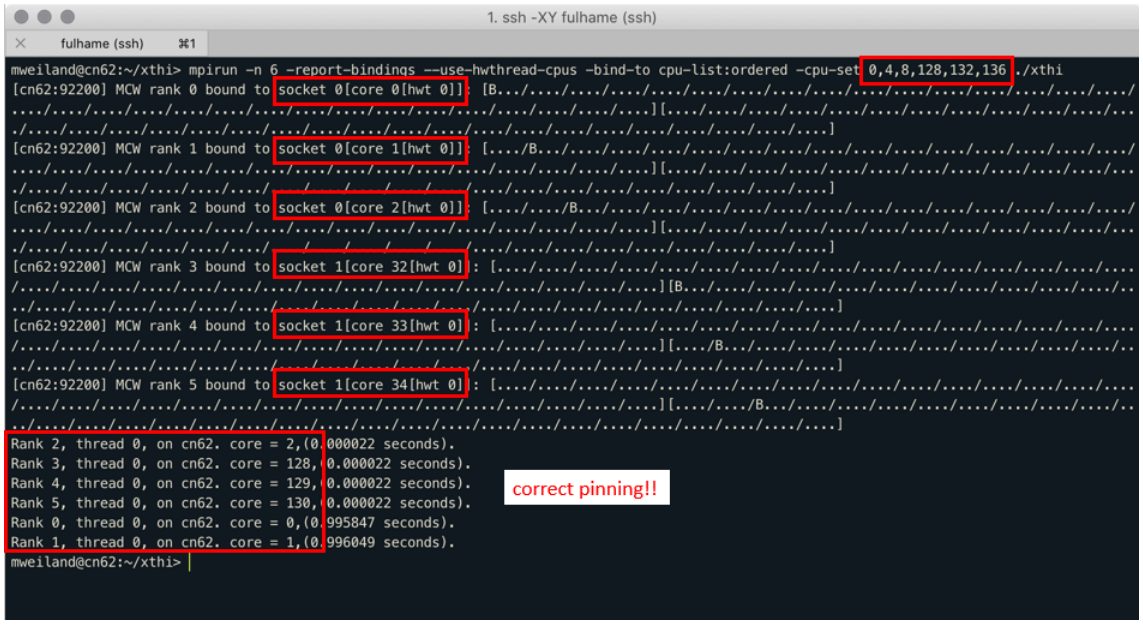


Figura 4.5: Afinidad correcta tras usar **OpenMPI** con CPUs lógicas generadas por **demangler**

Intel. Siguiendo este planteamiento, se introducen las CPUs lógicas 0, 4, 8, 128, 132 y 136 en el comando **mangle**, tal y como se muestra en la Figura 4.6. Como se puede observar en dicha figura, las CPUs lógicas generadas como resultado corresponden con las CPUs lógicas iniciales, presentes en el rectángulo rojo superior derecho en la Figura 3.9.

```
demangler/target/release on  master
❯ ./dmgler mangle 0 4 8 128 132 136
mpirun -n 6 --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set 0,1,2,128,129,130 <program>
```

Figura 4.6: Ejecución del comando **mangle** de herramienta **demangler**

Finalmente se procede a verificar el correcto funcionamiento del comando **get**. Puesto que este comando usa bajo las escenas la implementación de **demangle** para convertir las CPUs lógicas entre nomenclaturas, como se muestra en el Listado 3.6, solamente es necesario comprobar la exactitud del filtrado.

Para ello, se procede a obtener todas las CPUs lógicas que pertenezcan al HWT 1 del socket 1 de un nodo de **Fulhame**. El comando utilizado se puede observar en la Figura 4.7.

```
demangler/target/release on  master
❯ ./dmgler --smt 4 --mpi-program ./xthi get -s N0 -t Second
mpirun -n 32 --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61,65,69,7
3,77,81,85,89,93,97,101,105,109,113,117,121,125 ./xthi
```

Figura 4.7: Ejecución del comando **get** de herramienta **demangler**

Para comprobar que las CPUs obtenidas por el comando mostrado en la Figura 4.7 son las indicadas, se ejecuta este comando en el clúster **Fulhame** (aunque también se puede utilizar el comando **mangle**), y el resultado es el que se muestra en la Figura 4.8.

Como se muestra en el rectángulo rojo de la Figura 4.8, las CPUs lógicas obtenidas, aunque desordenadas, corresponden con la fila HWT 1 del socket 0 de la Tabla 4.1, justo las que se deseaba obtener.

```
jero@cn64:~> mpirun -n 32 --use-hwthread-cpus -bind-to cpu-list:ordered -cpu-set 1,5,9,13,17,21,25,29,33,37,41,45,49,53,
57,61,65,69,73,77,81,85,89,93,97,101,105,109,113,117,121,125 ./xthi
Hello from rank 1, thread 0, on cn64. (core affinity = 33)
Hello from rank 10, thread 0, on cn64 (core affinity = 42)
Hello from rank 2, thread 0, on cn64. (core affinity = 34)
Hello from rank 8, thread 0, on cn64. (core affinity = 40)
Hello from rank 11, thread 0, on cn64 (core affinity = 43)
Hello from rank 13, thread 0, on cn64 (core affinity = 45)
Hello from rank 9, thread 0, on cn64. (core affinity = 41)
Hello from rank 3, thread 0, on cn64. (core affinity = 35)
Hello from rank 12, thread 0, on cn64 (core affinity = 44)
Hello from rank 6, thread 0, on cn64. (core affinity = 38)
Hello from rank 0, thread 0, on cn64. (core affinity = 32)
Hello from rank 15, thread 0, on cn64 (core affinity = 47)
Hello from rank 5, thread 0, on cn64. (core affinity = 37)
Hello from rank 7, thread 0, on cn64. (core affinity = 39)
Hello from rank 27, thread 0, on cn64 (core affinity = 59)
Hello from rank 4, thread 0, on cn64. (core affinity = 36)
Hello from rank 23, thread 0, on cn64 (core affinity = 55)
Hello from rank 16, thread 0, on cn64 (core affinity = 48)
Hello from rank 18, thread 0, on cn64 (core affinity = 50)
Hello from rank 21, thread 0, on cn64 (core affinity = 53)
Hello from rank 25, thread 0, on cn64 (core affinity = 57)
Hello from rank 26, thread 0, on cn64 (core affinity = 58)
Hello from rank 20, thread 0, on cn64 (core affinity = 52)
Hello from rank 24, thread 0, on cn64 (core affinity = 56)
Hello from rank 17, thread 0, on cn64 (core affinity = 49)
Hello from rank 30, thread 0, on cn64 (core affinity = 62)
Hello from rank 14, thread 0, on cn64 (core affinity = 46)
Hello from rank 19, thread 0, on cn64 (core affinity = 51)
Hello from rank 28, thread 0, on cn64 (core affinity = 60)
Hello from rank 29, thread 0, on cn64 (core affinity = 61)
Hello from rank 31, thread 0, on cn64 (core affinity = 63)
Hello from rank 22, thread 0, on cn64 (core affinity = 54)
jero@cn64:~>
```

Figura 4.8: Obtención de CPUs lógicas correctas tras uso del comando **get** de la herramienta **demangler**

Con la verificación del correcto funcionamiento de los tres comandos de la herramienta **demangler**, su funcionamiento queda demostrado y se asegura que cumple con los requisitos definidos para dicha herramienta.

4.3 Evaluación de iorbenchtool

iorbenchtool es la segunda herramienta desarrollada en este trabajo. Esta se encarga de transformar un conjunto de archivos IOR, los cuales contienen información acerca del rendimiento de las conexiones de red de un nodo del clúster, a un archivo Excel personalizable con toda la información extraída.

Las pruebas de verificación de la herramienta se realizan sobre el directorio mostrado en la Figura 4.9. Este directorio raíz contiene cinco subdirectorios, cada uno de ellos con un conjunto de pruebas IOR de diferentes configuraciones. Bajo el directorio raíz, además, también se encuentra el ejecutable de la herramienta.

iorbenchtool solo tiene un comando principal, cuyo funcionamiento puede ser ligeramente modificado mediante el uso de los parámetros que la herramienta ofrece como parte de su interfaz de uso. Por lo tanto, para verificar el correcto funcionamiento de la herramienta, se realizarán las siguientes pruebas:

- No utilizar ninguna opción sobre la herramienta
- Utilizar la opción búsqueda de informes recursiva (**–recursive**)
- Utilizar la opción de unificación de informes (**–unite**). Esta solo funciona cuando **–recursive** ha sido indicada.

```
 Size  Name
    -  .
    -  ├── hdd-multiple
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_1_2020-07-20_10:49:34
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_2_2020-07-20_10:49:34
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_4_2020-07-20_10:49:34
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_8_2020-07-20_10:49:34
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_16_2020-07-20_10:49:34
3.5Ki  │   ├── IOR-RW-Multiple_Files-Common_Dir-c_1-s_32_2020-07-20_10:49:34
3.5Ki  │   └── IOR-RW-Multiple_Files-Common_Dir-c_1-s_64_2020-07-20_10:49:34
    -  ├── hdd-single
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_1_2020-07-17_10:10:41
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_2_2020-07-17_10:10:41
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_4_2020-07-17_10:10:41
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_8_2020-07-17_10:10:41
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_16_2020-07-17_10:10:41
3.5Ki  │   ├── IOR-RW-HDD-Single_File-c_1-s_32_2020-07-17_10:10:41
3.5Ki  │   └── IOR-RW-HDD-Single_File-c_1-s_64_2020-07-17_10:10:41
    -  ├── IOR_SSD_SMT1_64PROCS_LEICESTER
2.1Ki  │   ├── 3804.out
  460  │   └── ior-leicester.bash
    -  ├── ssd-multiple
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_1_2020-07-20_23:45:32
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_2_2020-07-20_23:45:32
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_4_2020-07-20_23:45:32
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_8_2020-07-20_23:45:32
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_16_2020-07-20_23:45:32
3.5Ki  │   ├── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_32_2020-07-20_23:45:32
3.5Ki  │   └── IOR-RW-SSD-Multiple_Files-Common_Dir-c_1-s_64_2020-07-20_23:45:32
    -  ├── ssd-single
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_1_2020-07-20_20:24:16
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_2_2020-07-20_20:24:16
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_4_2020-07-20_20:24:16
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_8_2020-07-20_20:24:16
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_16_2020-07-20_20:24:16
3.5Ki  │   ├── IOR-RW-SSD-Single_File-c_1-s_32_2020-07-20_20:24:16
3.5Ki  │   └── IOR-RW-SSD-Single_File-c_1-s_64_2020-07-20_20:24:16
 97Mi  └── iorbenchtool
```

Figura 4.9: Directorio de ejecución de las pruebas de la herramienta **iorbenchtool**

Puesto que el resto de las opciones de **iorbenchtool** no modifican el funcionamiento de la herramienta, sino que solamente ofrecen información extra acerca de la ejecución de esta, no se analizarán para la verificación de **iorbenchtool**.

La primera prueba que se realiza sobre la herramienta consiste en procesar un grupo de archivos sin ninguna configuración especial de **iorbenchtool**. El conjunto de archivos IOR elegidos es el que se encuentra bajo el directorio **hdd-single** de la Figura 4.9. Estos corresponden a test de IOR sobre un único HDD del sistema de almacenamiento paralelo, utilizando diferentes tareas paralelas, tanto para lectura como escritura.

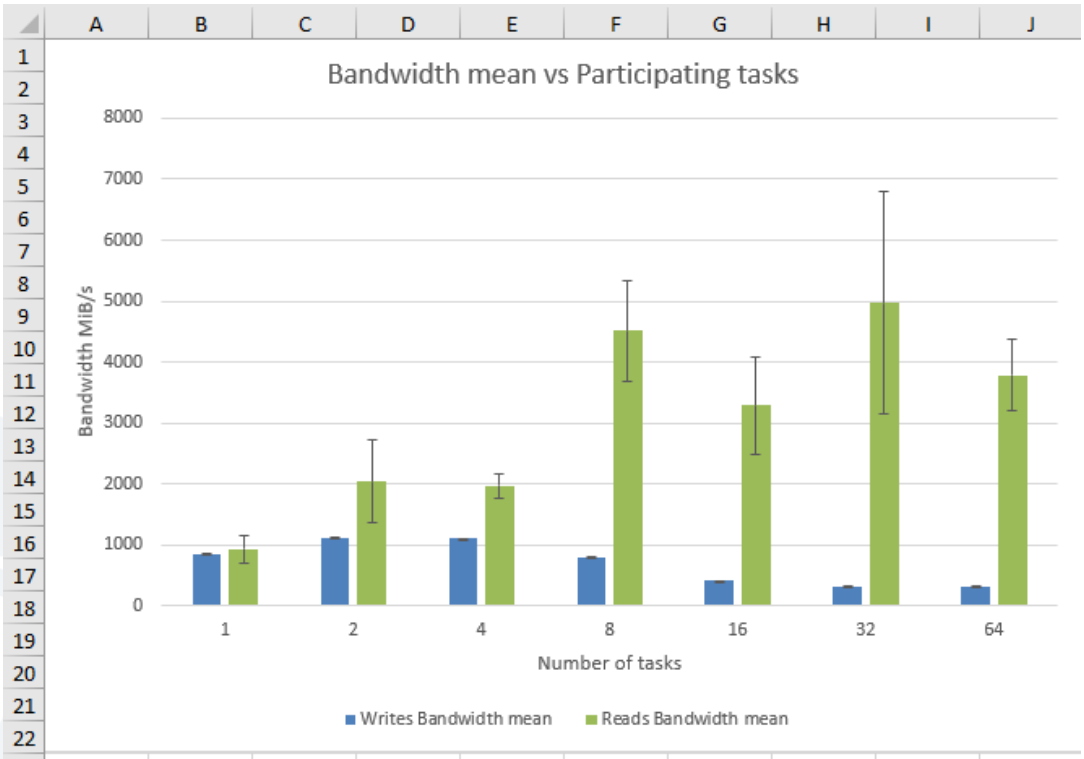
Para realizar esta prueba se ha ejecutado el comando **iorbentool** presentado en la Figura 4.10. Como se puede ver en dicha figura, este no utiliza ninguna opción.

En la Figura 4.10 se puede observar que la ejecución se ha realizado exitosamente. Además, como no se ha usado la opción **–verbose**, la salida por consola es muy breve y sin

```
Projects/SummerHPC/ior-results - tfg
› ./iorbenchtool hdd-single/
[20:23:55 INF] Parsing hdd-single directory.
[20:23:57 INF] Directory hdd-single parsing completed!
[20:23:57 INF] Done.
```

Figura 4.10: Ejecución sin uso de opciones de la herramienta **iorbenchtool**

apenas información. Como resultado de la ejecución presente en la Figura 4.10, se ha generado un archivo Excel en el directorio **hdd-single** con la forma mostrada en la Figura 4.11.



	A	B	C	D	E
25	Participan	Writes Ba	Writes Ba	Reads Ban	Reads Bandwidth StdDev
26	1	846.51	4.86423	927.7625	223.2955
27	2	1118.963	9.65411	2037.083	674.804
28	4	1099.93	10.61211	1968.318	194.1344
29	8	794.6325	3.046166	4509.05	837.1304
30	16	407.675	9.64548	3284.433	798.2888
31	32	307.64	8.992541	4972.16	1819.541
32	64	321.79	7.891045	3785.998	589.3096

Figura 4.11: Informe Excel generado tras la ejecución de la herramienta **iorbenchtool** sin opciones

Este Excel contiene, en la parte inferior, los datos de los diferentes archivos IOR utilizados y que, además, han sido utilizados para la generación de la gráfica. Esta muestra una

```
Projects/SummerHPC/ior-results - tfg
❯ ./iorbenchtool --recursive ./
[20:24:32 INF] Parsing hdd-single directory.
[20:24:32 INF] Parsing ssd-single directory.
[20:24:32 INF] Parsing IOR_SSD_SMT1_64PROCS_LEICESTER directory.
[20:24:32 INF] Parsing ssd-multiple directory.
[20:24:32 INF] Parsing hdd-multiple directory.
[20:24:33 INF] Directory hdd-single parsing completed!
[20:24:33 INF] Directory IOR_SSD_SMT1_64PROCS_LEICESTER parsing completed!
[20:24:33 INF] Directory ssd-multiple parsing completed!
[20:24:33 INF] Directory hdd-multiple parsing completed!
[20:24:33 INF] Directory ssd-single parsing completed!
[20:24:33 INF] Done.
```

Figura 4.12: Ejecución con la opción **–recursive** de la herramienta **iorbenchtool**

comparativa del ancho de banda, tanto para escrituras, como para lecturas, en función de las tareas que acceden de manera concurrente al sistema de almacenamiento.

Como se puede observar la Figura 4.11, la creación de una gráfica permite obtener una rápida visualización de los resultados de las pruebas. En este caso, como es de esperar, a más tareas paralelas escribiendo en un mismo archivo (columnas azules), el ancho de banda medio se reduce. En cambio, en el caso de las lecturas (columnas verdes), a más tareas paralelas, el ancho de banda medio crece. La no linealidad de estos datos puede deberse a que otro usuario podría haber estado usando el sistema de almacenamiento paralelo o a fallos en la implementación de los drivers de red.

A continuación, se procede a verificar la opción **–recursive** de **iorbenchtool**. Esta opción busca archivos IOR en todos los subdirectorios a partir de uno raíz y para cada directorio válido genera un informe Excel. Para realiza esta prueba de verificación, el comando se ejecuta en el directorio raíz mostrado en la Figura 4.9. El resultado esperado de la ejecución de **iorbenchtool** con esta opción es la generación de un archivo Excel por cada uno de los subdirectorios presentes en la Figura 4.12. Esta opción es equivalente a ejecutar esta herramienta en cada uno de los subdirectorios.

Tras ello, bajo cada directorio presente en la Figura 4.9, se genera un informe Excel. Esto se puede comprobar en la Figura 4.13. Para la generación de esta figura se han filtrado todos los archivos que no son Excel. No se muestra ningún archivo Excel de la ejecución de la herramienta con la opción **–recursive** ya que los generados tienen el mismo formato que el mostrado en la Figura 4.11.

Finalmente se procede a comprobar el funcionamiento de la opción **–unite**. Esta unifica todos los archivos Excel generados al usar la opción **–recursive** bajo un único Excel en el di-

```
Projects/SummerHPC/ior-results - tfg
❯ exa -Tblh --group-directories-first --no-user --no-time --no-permissions -I="IOR*|*.out|*.bash"
Size Name
   - .
   - ├── hdd-multiple
12Ki │   └── Benchmark-hdd-multiple-03-7-2021--20-41-26.xlsx
   - ├── hdd-single
12Ki │   └── Benchmark-hdd-single-03-7-2021--20-41-26.xlsx
   - ├── ssd-multiple
12Ki │   └── Benchmark-ssd-multiple-03-7-2021--20-41-26.xlsx
   - ├── ssd-single
12Ki │   └── Benchmark-ssd-single-03-7-2021--20-41-26.xlsx
97Mi └── iorbenchtool
```

Figura 4.13: Directorio ejecución pruebas **iorbenchtool** tras ejecución con **–recursive** filtrado por archivos Excel

```
Projects/SummerHPC/ior-results - tfg
❯ ./iorbenchtool --recursive --unite ./
[20:47:27 INF] Parsing IOR_SSD_SMT1_64PROCS_LEICESTER directory.
[20:47:27 INF] Parsing ssd-single directory.
[20:47:27 INF] Parsing ssd-multiple directory.
[20:47:27 INF] Parsing hdd-multiple directory.
[20:47:27 INF] Parsing hdd-single directory.
[20:47:29 INF] Directory ssd-multiple parsing completed!
[20:47:29 INF] Directory IOR_SSD_SMT1_64PROCS_LEICESTER parsing completed!
[20:47:29 INF] Directory hdd-multiple parsing completed!
[20:47:29 INF] Directory ssd-single parsing completed!
[20:47:29 INF] Directory hdd-single parsing completed!
[20:47:29 INF] Summarizing all reports into single one.
[20:47:30 INF] Done.
```

Figura 4.14: Ejecución con la opción **–unite** de la herramienta **iorbenchtool**

```
Projects/SummerHPC/ior-results - tfg
❯ exa -Tblh --group-directories-first --no-user --no-time --no-permissions -I="IOR*|*.out|*.bash"
Size Name
   - .
   - ├── hdd-multiple
12Ki │   └── Benchmark-hdd-multiple-03-7-2021--20-47-27.xlsx
   - ├── hdd-single
12Ki │   └── Benchmark-hdd-single-03-7-2021--20-47-27.xlsx
   - ├── ssd-multiple
12Ki │   └── Benchmark-ssd-multiple-03-7-2021--20-47-27.xlsx
   - ├── ssd-single
12Ki │   └── Benchmark-ssd-single-03-7-2021--20-47-27.xlsx
97Mi ├── iorbenchtool
38Ki └── United-Benchmark-ior-results - tfg-03-7-2021--20-47-29.xlsx
```

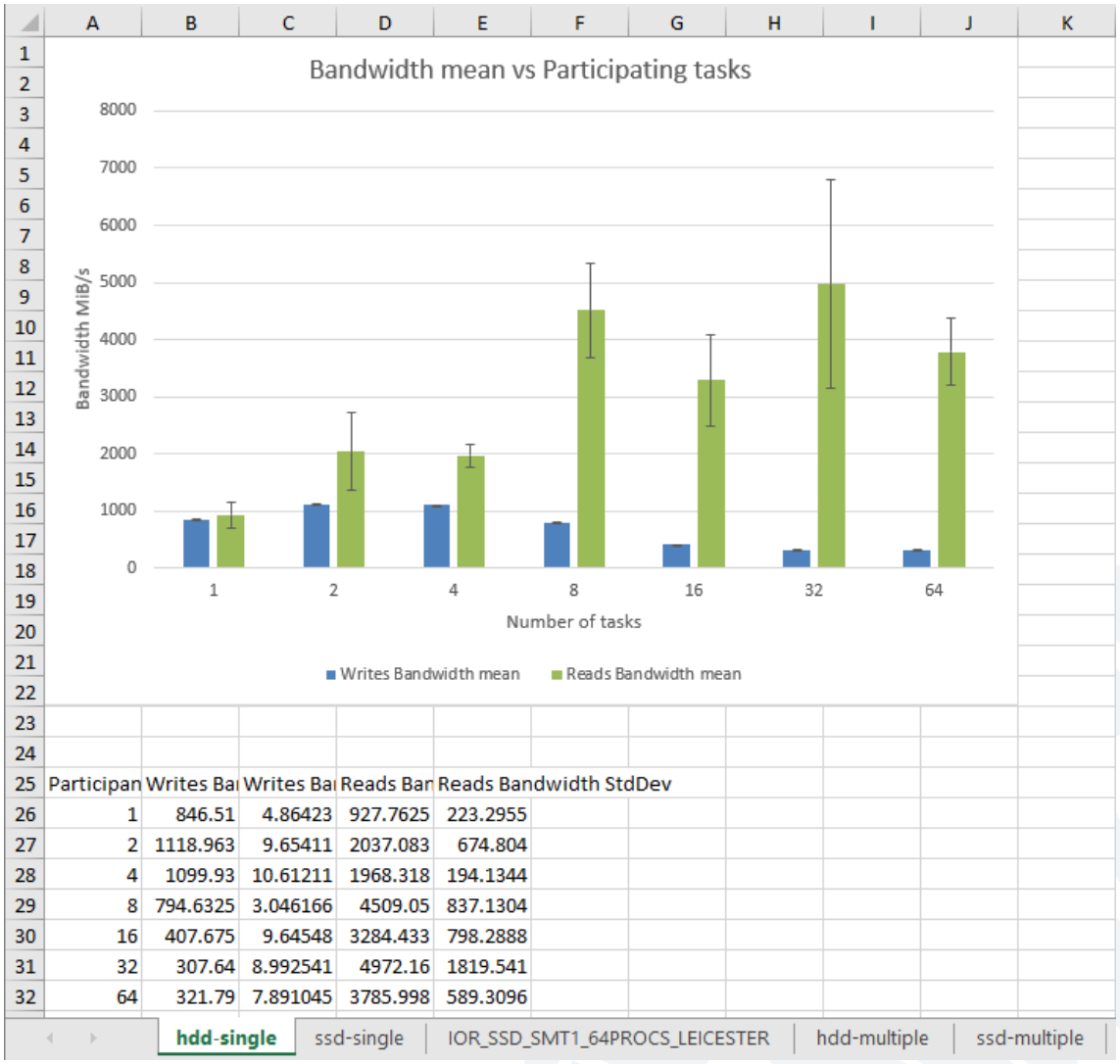
Figura 4.15: Directorio ejecución pruebas **iorbenchtool** tras ejecución con **–unite** filtrado por archivos Excel

rectorio raíz. El comando ejecutado se muestra en la Figura 4.14. Como se puede comprobar en esta figura, justo antes de terminar su ejecución, la herramienta indica que los archivos Excel están siendo unificados en uno solo.

Tras su ejecución, en la Figura 4.15 se puede comprobar que ha aparecido un nuevo archivo respecto a los generados con la opción **–recursive** (Figura 4.13). Este nuevo archivo se encuentra en el directorio raíz y su nombre comienza por ***"United-Benchmark"***, como se puede observar en la parte inferior de la Figura 4.15.

Este archivo Excel sigue la misma plantilla que el archivo mostrado en la Figura 4.11, salvo que este cuenta con una hoja extra por cada directorio que ha sido analizado, tal y como se puede comprobar en la parte inferior de la Figura 4.16.

Con la verificación del correcto funcionamiento de las opciones de la herramienta **iorbenchtool**, su funcionamiento queda demostrado y se asegura que cumple con los requisitos definidos para dicha herramienta.



Participan	Writes Ba	Writes Ba	Reads Ban	Reads Bandwidth StdDev
1	846.51	4.86423	927.7625	223.2955
2	1118.963	9.65411	2037.083	674.804
4	1099.93	10.61211	1968.318	194.1344
8	794.6325	3.046166	4509.05	837.1304
16	407.675	9.64548	3284.433	798.2888
32	307.64	8.992541	4972.16	1819.541
64	321.79	7.891045	3785.998	589.3096

Figura 4.16: Informe Excel generado tras la ejecución de la herramienta **iorbenchtool** usando la opción **–unite**

5 Conclusiones y trabajo futuro

Previo a las conclusiones, cabe indicar que los desarrollos y la evaluación de las herramientas de este trabajo se han realizado gracias al soporte ofrecido por la Universidad de Almería y por el programa **Summer of HPC** organizado por **PRACE**. Esta beca permite que un grupo selecto de alumnos de matemáticas, física e informática de toda Europa puedan participar en proyectos HPC de las universidades del continente con mayor prestigio en este campo. El autor fue escogido para trabajar en el **EPCC** [1], Escocia (Reino Unido) acerca de los problemas asociados al usar un clúster HPC basado en una arquitectura de procesadores **ARM**. Además, parte de los resultados obtenidos en este TFG, se han publicado en el Congreso Español De Informática [2] (CEDI), en concreto en las jornadas SARTECO de paralelismo y computación empotrada y reconfigurable [49].

El diseño y desarrollo de las herramientas que conforman este Trabajo Fin de Grado permite extraer las siguientes conclusiones respecto a los objetivos planteados:

- El tema del trabajo y la evaluación de las herramientas que lo conforman han permitido al autor aprender a utilizar los sistemas que gestionan y planifican las tareas/trabajos en un supercomputador, cómo lo hacen las librerías dentro de un clúster, cómo se traspasa información al supercomputador y cómo conectarse a él remotamente de una manera segura. Además, también han permitido al autor conocer los procesos burocráticos para obtener acceso a un sistema clúster.

- El desarrollo de las herramientas de este trabajo ha permitido al autor conocer el estado de la arquitectura **ARM** en el mundo del HPC, así como conocer de primera mano el trabajo que se está realizando para solventar los problemas y dificultades encontrados al hacer uso de esta arquitectura en este tipo de sistemas.

- Gracias también al tema del trabajo, el autor ha obtenido conocimientos acerca de las arquitecturas **x86** y **ARM**, tanto respecto a la historia de estas, como respecto al diseño e implementación de las diferentes tecnologías paralelas que forman parte de los procesadores modernos.

- El desarrollo de las herramientas de este trabajo ha permitido al autor aprender el argot utilizado en el campo de la programación de sistemas paralelos. También ha permitido al autor afianzar cuáles son los diferentes paradigmas para la programación paralela existentes, las principales diferencias entre ellos y cómo se usan de manera correcta y eficiente.

Asimismo, las herramientas desarrolladas no solo cumplen con los objetivos marcados al inicio de este trabajo, si no que estas, además, están siendo utilizadas por el **EPCC** en sus investigaciones acerca del uso de **ARM** en el HPC y la viabilidad y ventajas de ello. Además, estas herramientas podrían ser probadas en las plataformas **ARM** heterogéneas del grupo de **Supercomputación y Algoritmos** de la Universidad de Almería.

El uso de las herramientas desarrolladas puede suponer una mejora en el rendimiento de programas que hacen uso de **OpenMPI**. Por lo tanto, resultaría de gran interés contrastar el incremento del rendimiento obtenido como consecuencia de aplicar las herramientas de este TFG. Además, dentro de esta línea de investigación futura queda trabajo por realizar, ya que existe mucho software originalmente creado para **x86** que necesita ser portado a **ARM**.

Para concluir, este trabajo ha supuesto un gran enriquecimiento personal para el autor debido al aprendizaje de nuevos recursos, así como a la participación en un entorno innovador y multicultural.

BIBLIOGRAFÍA

[1] U. of Edinburgh. (s.f). "EPCC at The University of Edinburgh | EPCC," dirección: https://www.epcc.ed.ac.uk/ (visitado 18-03-2021).

[2] U. de Málaga. (s.f). "CEDI 20/21," dirección: https://congresocedi.es/ (visitado 08-07-2021).

[3] *X86*, en *Wikipedia*, Page Version ID: 1028036268, 11 de jun. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=X86&oldid=1028036268 (visitado 13-06-2021).

[4] *Intel 8088*, en *Wikipedia*, Page Version ID: 1028538882, 14 de jun. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Intel_8088&oldid=1028538882 (visitado 20-06-2021).

[5] Greg Tang. (4 de ene. de 2011). "Intel and the x86 architecture: A legal perspective," Harvard Journal of Law & Technology. col. de I. W. Brown, dirección: https://jolt.law.harvard.edu/digest/intel-and-the-x86-architecture-a-legal-perspective (visitado 20-06-2021).

[6] *Advanced micro devices, inc. v. intel corp. (1994)*. dirección: https://law.justia.com/cases/california/supreme-court/4th/9/362.html (visitado 29-06-2021).

[7] *Intel corp. v. advanced micro devices, inc. (1991)*. dirección: https://law.justia.com/cases/federal/district-courts/FSupp/756/1292/2291857/ (visitado 29-06-2021).

[8] *ARM1 - microarchitectures - acorn*, en *WikiChip*, 14 de ene. de 2021. dirección: https://en.wikichip.org/wiki/acorn/microarchitectures/arm1 (visitado 20-06-2021).

[9] M. Jarus, S. Varrette, A. Oleksiak y P. Bouvry, "Performance Evaluation and Energy Efficiency of High-Density HPC Platforms Based on Intel, AMD and ARM Processors," 1 de abr. de 2013, ISBN: 978-3-642-40516-7. DOI: 10.1007/978-3-642-40517-4_16.

[10] M. Hirki, Z. Ou, K. N. Khan, J. K. Nurminen y T. Niemi, "Empirical study of power consumption of x86-64 instruction decoder," pág. 6,

[11] J. Hashmi, S. Oh y G. Fox, "Evaluating ARM HPC clusters for scientific workloads," *Concurrency and Computation: Practice and Experience*, vol. 27, 1 de jul. de 2015. DOI: 10.1002/cpe.3602.

[12] H. Levy, J. L. Lo, J. Emer, R. Stamm, S. Eggers y D. Tullsen, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," en *23rd Annual International Symposium on Computer Architecture (ISCA'96)*, ISSN: 1063-6897, mayo de 1996, págs. 191-191. DOI: 10.1145/232973.232993.

[13] M. Awad, V. P. Infrastructure y Arm. (13 de jul. de 2020). “Arm-powered data centers optimized for cost and efficiency,” Arm Blueprint. Section: Arm Enables, dirección: https://www.arm.com/blogs/blueprint/optimizing-data-center (visitado 15-02-2021).

[14] M. U. Ashraf, F. A. Eassa, A. Ahmad y A. Algarni, “Empirical investigation: Performance and power-consumption based dual-level model for exascale computing systems,” *IET Software*, vol. 14, n.º 4, págs. 319-327, 2020, _eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/iet-sen.2018.5062, ISSN: 1751-8814. DOI: https://doi.org/10.1049/iet-sen.2018.5062. dirección: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2018.5062 (visitado 09-02-2021).

[15] C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein y T. Wettig, “ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX,” *arXiv:2103.03013 [hep-lat]*, 4 de mar. de 2021. arXiv: 2103.03013. dirección: http://arxiv.org/abs/2103.03013 (visitado 18-03-2021).

[16] Don Wells. (8 de oct. de 2013). “Extreme Programming: A Gentle Introduction.,” dirección: http://www.extremeprogramming.org/ (visitado 01-04-2021).

[17] *C sharp (programming language)*, en *Wikipedia*, Page Version ID: 1016891176, 9 de abr. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=C_Sharp_(programming_language)&oldid=1016891176 (visitado 13-04-2021).

[18] Tatu Ylonen. (s.f). “SSH (Secure Shell) Home Page,” dirección: https://www.ssh.com/ssh/ (visitado 01-04-2021).

[19] M. Flynn, “Very high-speed computing systems,” *Proceedings of the IEEE*, vol. 54, n.º 12, págs. 1901-1909, 1966, ISSN: 0018-9219. DOI: 10.1109/PROC.1966.5273. dirección: http://ieeexplore.ieee.org/document/1447203/ (visitado 19-06-2021).

[20] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers*, vol. C-21, n.º 9, págs. 948-960, sep. de 1972, ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071. dirección: http://ieeexplore.ieee.org/document/5009071/ (visitado 19-06-2021).

[21] *Flynn’s taxonomy*, en *Wikipedia*, Page Version ID: 1029163551, 18 de jun. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Flynn%27s_taxonomy&oldid=1029163551 (visitado 19-06-2021).

[22] Intel Corp. (s.f). “Intel® instruction set extensions technology,” Intel, dirección: https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html (visitado 23-06-2021).

[23] A. Ltd. (s.f). “SIMD ISAs – Arm Developer,” dirección: https://developer.arm.com/architectures/instruction-sets/simd-isas (visitado 23-06-2021).

[24] J. L. Hennessy, D. A. Patterson y K. Asanović, *Computer architecture: a quantitative approach*, 5th ed. Waltham, MA: Morgan Kaufmann/Elsevier, 2012, 493 págs., OCLC: ocn755102367, ISBN: 978-0-12-383872-8. (visitado 19-06-2021).

[25] D. A. Patterson, J. L. Hennessy y P. Alexander, *Computer organization and design: the hardware/software interface*, ARM® edition. Amsterdam ; Boston: Elsevier/Morgan

Kaufmann, 2017, 589 págs., OCLC: ocn946041146, ISBN: 978-0-12-801733-3. (visitado 18-06-2021).

[26] A. Park. (s.f). "Multithreaded Programming (POSIX pthreads Tutorial)," dirección: https://randu.org/tutorials/threads/ (visitado 05-04-2021).

[27] OpenMP. (s.f). "Homepage," OpenMP, dirección: https://www.openmp.org/ (visitado 03-04-2021).

[28] Intel Corp. (11 de feb. de 2011). "Optimizing applications for NUMA," Intel, dirección: https://www.intel.com/content/www/us/en/develop/articles/optimizing-applications-for-numa.html (visitado 03-04-2021).

[29] MPI Forum Contributors. (s.f). "MPI forum," dirección: https://www.mpi-forum.org/ (visitado 03-04-2021).

[30] T. Org. (nov. de 2020). "Highlights - November 2020 | TOP500," dirección: https://www.top500.org/lists/top500/2020/11/highs/ (visitado 03-04-2021).

[31] Y. Li, *Computer principles and design in Verilog HDL*. Solaris South Tower, Singapore: John Wiley y Sons, Inc, 2015, 1 pág., ISBN: 978-1-118-84112-9 978-1-118-84111-2. (visitado 19-06-2021).

[32] Crytal Chen, Greg Novick y Kirk Shimano. (s.f). "RISC vs. CISC," dirección: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/ (visitado 19-06-2021).

[33] *POSIX threads*, en *Wikipedia*, Page Version ID: 999779981, 11 de ene. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=POSIX_Threads&oldid=999779981 (visitado 03-04-2021).

[34] Rust Lang Contributors. (s.f). "std::sync::Mutex - Rust," dirección: https://doc.rust-lang.org/std/sync/struct.Mutex.html (visitado 05-04-2021).

[35] ——, (s.f). "std::sync::RwLock - Rust," dirección: https://doc.rust-lang.org/std/sync/struct.RwLock.html (visitado 05-04-2021).

[36] T. O. M. Project. (15 de ene. de 2021). "Open MPI: Open Source High Performance Computing," dirección: https://www.open-mpi.org/ (visitado 01-04-2021).

[37] L. B. Das, *The x86 microprocessors 8086 to Pentium, multicores, atom, and the 8051 microcontroller*. New Delhi, India: Dorling Kindersley (India), 2014, OCLC: 930869954, ISBN: 978-93-325-4079-8. dirección: http://0-proquest.safaribooksonline.com.fama.us.es//?uiCode=sevil&xmlId=9789332540798 (visitado 20-06-2021).

[38] *Pentium pro*, en *Wikipedia*, Page Version ID: 1028035995, 11 de jun. de 2021. dirección: https://en.wikipedia.org/w/index.php?title=Pentium_Pro&oldid=1028035995 (visitado 20-06-2021).

[39] Dileep Bhandarkar. (ene. de 1997). "Pentium ® pro processor block diagram," ResearchGate, dirección: https://www.researchgate.net/figure/Pentium-R-Pro-Processor-Block-Diagram_fig3_220267326 (visitado 20-06-2021).

[40] *Architectural state*, en *Wikipedia*, Page Version ID: 889277933, 24 de mar. de 2019. dirección: https://en.wikipedia.org/w/index.php?title=Architectural_state&oldid=889277933 (visitado 20-06-2021).

[41] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer y S. R. Kunkel, "Characterization of simultaneous multithreading (SMT) efficiency in POWER5," *IBM Journal of Research and Development*, vol. 49, n.º 4, págs. 555-564, jul. de 2005, ISSN: 0018-8646, 0018-8646. DOI: 10.1147/rd.494.0555. dirección: http://ieeexplore.ieee.org/document/5388823/ (visitado 20-06-2021).

[42] IANA. (18 de nov. de 2020). "Performance Metrics." col. de A. Morton y M. B. Braun, dirección: https://www.iana.org/assignments/performance-metrics/performance-metrics.xhtml (visitado 04-07-2021).

[43] FUJITSU. (s.f). "Specifications - Supercomputer Fugaku : Fujitsu Global," dirección: https://www.fujitsu.com/global/about/innovation/fugaku/specifications/ (visitado 09-02-2021).

[44] NVIDIA y Mellanox. (s.f). "Accelerated scientific innovation with InfiniBand," NVIDIA, dirección: https://www.nvidia.com/en-us/networking/products/infiniband/ (visitado 25-06-2021).

[45] University of Edinburgh. (s.f). "Fulhame | EPCC at The University of Edinburgh," dirección: https://www.epcc.ed.ac.uk/facilities/other-facilities/fulhame (visitado 10-02-2021).

[46] *hpc/ior*, original-date: 2015-06-23T15:56:39Z, 22 de ene. de 2021. dirección: https://github.com/hpc/ior (visitado 28-01-2021).

[47] *Vulcan - microarchitectures - cavium*, en *WikiChip*, 4 de oct. de 2019. dirección: https://en.wikichip.org/wiki/cavium/microarchitectures/vulcan (visitado 30-05-2021).

[48] A. Jackson, A. Turner, M. Weiland, N. Johnson, O. Perks y M. Parsons, "Evaluating the arm ecosystem for high performance computing," *arXiv:1904.04250 [cs]*, 8 de abr. de 2019. arXiv: 1904.04250. dirección: http://arxiv.org/abs/1904.04250 (visitado 09-02-2021).

[49] Universidad de Málaga. (s.f). "Jornadas Sarteco | Sociedad de Arquitectura y Tecnología de Computadores," dirección: http://www.jornadassarteco.org/ (visitado 08-07-2021).

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Este trabajo explora y da solución software a dos de los problemas que surgen de usar un supercomputador basado en procesadores ARM para la Computación de Alto Rendimiento (HPC). En concreto, se tratan los problemas relacionados con la afinidad de tareas y la automatización de informes de rendimiento de red. Los desarrollos y las evaluaciones de las herramientas se han realizado gracias al apoyo ofrecido por la Universidad de Almería y la organización PRACE con su programa *Summer of HPC*. Esta beca ha permitido al autor a participar en proyectos HPC del EPCC [1]. Parte de los resultados obtenidos en este TFG han sido publicados en el Congreso Español De Informática (CEDI) [2].

This work studies and solves, through software, two of the problems that arise from using an ARM-based supercomputer for High-Performance Computing (HPC). To be precise, it targets the issues relating to task affinity and the automation of a network's performance reports. The development and evaluation of such tools has been carried out thanks to the support of the University of Almería and the Summer of HPC program, organized by PRACE. This scholarship has allowed the author to participate in HPC projects of the EPCC [1]. A subset of the results obtained in this TFG have been published in the Spanish Computer Science Congress (CEDI) [2].

GRADO EN INGENIERÍA INFORMÁTICA

2020/2021