TRABAJO FIN DE GRADO

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Planificador Adaptativo del Kernel

Mención en Sistemas de Información/Tecnologías de la Información

Grado en Ingeniería Informática

Curso 2017/2018

Alumno:

Rodrigo Espeso Cano

Director:

Dr. Juan Francisco Sanjuan Estrada

Universidad de Almería

PLANIFICADOR ADAPTATIVO DEL KERNEL

Trabajo fin de grado realizado por:

Rodrigo Espeso Cano

dirigido por:
Dr. Juan Francisco Sanjuan Estrada

A Tone

Prólogo

En estos último años, la evolución del hardware ha estado y está predominada por la paralelización: arquitecturas *multicore*, *CMP (Chip-Multiprocessors)*, *manycore*, sistemas *HPC (High Performance omputing)*, etc. Acompañada de la arquitectura, el software ha tenido que experimentar un gran avance para aprovechar esta tecnología y hacerla eficiente. La programación paralela y los algoritmos multihebrados sustentan las aplicaciones que se ejecutan en este tipo de arquitecturas para aumentar su rendimiento y aprovechar todo el potencial computacional que ofrecen. Sin embargo, el hardware avanza a mayor velocidad que el software, y por ello se necesitan herramientas para ayudar al desarrollador de aplicaciones. Lenguajes de programación especializados, APIs y otros tipos de herramientas de programación están actualmente en su apogeo.

El presente Trabajo Fin de Grado se centra en el estudio práctico del *Gestor del nivel de Paralelismo (GP)* a nivel de kernel sobre una arquitectura con más del triple de procesadores disponibles del que inicialmente fue creada el GP [8]. Este GP a nivel de kernel requiere hacer uso de una API que provee la librería *IGP (Interfaz del Gestor de Paralelismo)*. De tal forma, que al integrar *IGP* en una aplicación multihebrada, emplazando las llamadas a las funciones de la librería en lugares concretos del código, se consigue que la aplicación sea capaz de llevar una gestión interna automática de las hebras en ejecución en función de los requerimientos de la aplicación y del *Sistema Operativo (SO)*. Elegir de antemano un gran número de hebras puede resultar ineficiente debido a la sobrecarga en el paralelismo. La gran ventaja del *GP*, es que puede decidir en tiempo de ejecución qué cantidad de hebras deben estar activas para mejorar el rendimiento, teniendo en cuenta que el rendimiento suele estar asociado a un número óptimo de hebras.

Índice general

Índice de Algoritmos

Índice de Figuras

Parte I

Base teórica, objetivos y herramientas

Capítulo 1

Introducción

El planificador del sistema operativo (SO) tiene la función de repartir los recursos del sistema entre todas las aplicaciones en ejecución. En el caso de ejecutar aplicaciones multihebradas en multiprocesadores, el planificador decide qué tarea (proceso o hebra) debe ejecutarse en cada procesador, así como el tiempo de asignación. El criterio de decisión por el cual selecciona la tarea que debe ejecutar un procesador, viene establecido por el tipo de planificador.

Actualmente el sistema operativo Linux dispone, por defecto, del planificador *CFS (Completely Fair Scheduler)* implementado en todas las versiones de Linux superiores a 2.6.23. Este planificador básicamente trata de realizar un reparto equitativo del tiempo de utilización de los procesadores entre las tareas en ejecución. Sin embargo, el número de tareas en ejecución depende del número de hebras activas de las aplicaciones multihebradas. En este sentido, estimar el número óptimo de hebras activas de las aplicaciones en ejecución es crucial para que las aplicaciones multihebradas puedan mejorar su eficiencia.

El kernel del sistema operativo dispone de información sobre el rendimiento instantáneo de cada aplicación, por lo que una modificación en el código fuente del kernel permitiría estimar la eficiencia del sistema, a partir del rendimiento instantáneo de cada aplicación. Por otro lado, el usuario suele lanzar las aplicaciones multihebradas creando un número fijo de hebras, normalmente, el número de procesadores disponibles en el sistema. Sin embargo, el rendimiento óptimo de la aplicación no tiene porque producirse con ese número fijo de hebras, pues depende de muchos factores, tales como, carga de trabajo de la aplicación a resolver, el impacto de otras aplicaciones en ejecución o incluso ante la aparición de fallos esporádicos en el sistema [1].

En este sentido, el *Gestor del nivel de Paralelismo a nivel de kernel* será el encargado de estimar el rendimiento instantáneo de cada aplicación en ejecución, e interactuar, tanto con cada una de las aplicaciones multihebradas, como con el planificador del sistema operativo para establecer dinámicamente el número óptimo de hebras, de tal forma que se mejore el rendimiento [2].

En este capítulo se realizará una breve introducción al kernel de Linux utilizando como base [3]. Además, pretende clarificar las diferencias entre procesos y hebras [4]. Posteriormente se introduce el planificador del sistema operativo [5, 6] y se acaba realizando una primera aproximación al *Gestor del nivel de Paralelismo (GP)* [8].

1.1. ¿Qué es el kernel de Linux?

Al pensar en un computador, lo más seguro es que uno se imagine un conjunto de dispositivos físicos interconectados (procesador, placa base, memoria, disco duro, teclado, etc.) que permiten hacer las tareas simples como escribir un e-mail, ver una película o navegar por la web. Entre este hardware y las aplicaciones existe una capa de software responsable de hacer que todo el hardware trabaje de manera eficiente y de construir una infraesctructura de alto nivel para que las aplicaciones que usamos puedan funcionar. Esta capa de software es el *Sistema Operativo (SO)*, y su núcleo se denomina coloquialmente como *kernel*.

En los sistemas operativos modernos, el kernel es responsable realizar tareas que normalmente se dan por sentado: gestionar la memoria virtual, acceso a disco, manejo de entrada/salida, entre otras. Generalmente más grande que la mayoría de aplicaciones de usuario, el kernel es un complejo y fascinante código escrito en una mezcla de lenguaje ensamblador (lenguaje de bajo nivel para la máquina) y lenguaje C. Además, el kernel usa algunas propiedades de arquitecturas subyacentes para separarse del resto de programas en ejecución. De hecho, la mayoría de instrucciones *ISA (Instruction Set Architecture)* provee al menos dos modos de ejecución: modo *privilegiado*, en el cual todas las instrucciones a nivel máquina son completamente accesibles, y un modo *no-privilegiado*, en el cual solo un subconjunto de instrucciones son accesibles. Por otra parte, el kernel se protege a sí mismo de las aplicaciones de usuario implementando una separación en el nivel de software. Cuando hay que establecer el subsistema de memoria virtual, el kernel asegura que puede acceder al espacio de dirección(es) (p. ej., el rango de direcciones de memoria virtual) de cualquier proceso, y que ningún proceso puede referenciar directamente a la memoria del kernel. Nos referimos a la memoria visible sólo por el kernel como *espacio del kernel* y a la memoria que ve una aplicación de usuario como *espacio de usuario*, como se aprecia en la figura 1.1. El código que se ejecuta en el espacio del kernel lo hace con todos los privilegios y puede acceder a cualquier dirección de memoria válida del sistema, mientras que el código que se ejecuta en el espacio de usuario está sujeto a todas la limitaciones descritas anteriormente. Esta separación basada en hardware y software es obligatoria para proteger el kernel de daños accidentales, manipulación por mal comportamiento o de aplicaciones de espacio de usuario maliciosas. En definitiva, separar y proteger el kernel de otros programas en ejecución es el primer paso hacia un sistema seguro y estable [3].

1.2. Procesos y hebras

Una vez se ha introducido brevemente el kernel del sistema operativo, en esta sección se van a resaltar las semejanzas y diferencias entre dos conceptos importantes para el desarrollo del trabajo: los procesos y hebras del SO.

Procesos Se puede contemplar un proceso desde diferentes prespectivas:

- Una instancia de un programa en ejecución.
- Una entidad que puede ser asignada al procesador para ser ejecutada.

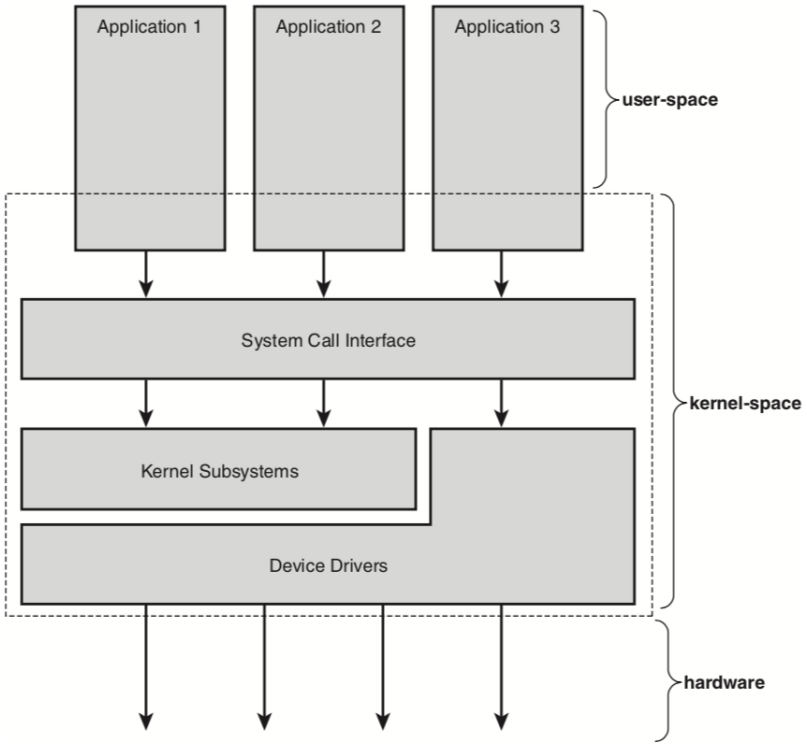


Figura 1.1: Relación entre aplicaciones, kernel y hardware (Fuente: [9]).

- Una unidad de actividad caracterizada por la ejecución de una secuencia de instrucciones, un estado actual y unos recursos del sistema asociados.

Los procesos son independientes, de manera que pueden existir varios procesos ejecutando una misma aplicación, pero cada uno distinto, al igual que una aplicación puede ejecutar dos procesos diferentes (esto se conoce como *multitasking*). Cada proceso tiene un código de programa y un conjunto de datos asociado a este código [4].

Hebras Una hebra es una secuencia más pequeña de instrucciones programadas que puede ser gestionada independientemente por una parte muy importante del sistema operativo: el *planificador*. En la mayoría de los sistemas operativos las hebras son componentes de los procesos, aunque la implementación de procesos y hebras difiere según el SO. Además, múltiples hebras pueden existir dentro de un proceso ejecutándose concurréntemente y compartiendo recursos, mientras que los diferentes procesos no comparten recursos.

Al igual que los procesos, las hebras tienen diferentes estados. Estos son:

- *Creación:* típicamente, cuando se crea un nuevo proceso también lo hace una hebra para ese proceso. Posteriormente, una hebra dentro de un proceso puede crear otras hebras.
- *Bloqueo:* cuando una hebra necesita esperar a un evento, esta se bloquea. El procesador puede ejecutar otra hebra del mismo proceso o de otro diferente.
- *Desbloqueo:* cuando ocurre el evento que bloquea a una hebra, esta se desbloquea.
- *Finalización:* cuando una hebra termina su ejecución en el sistema.

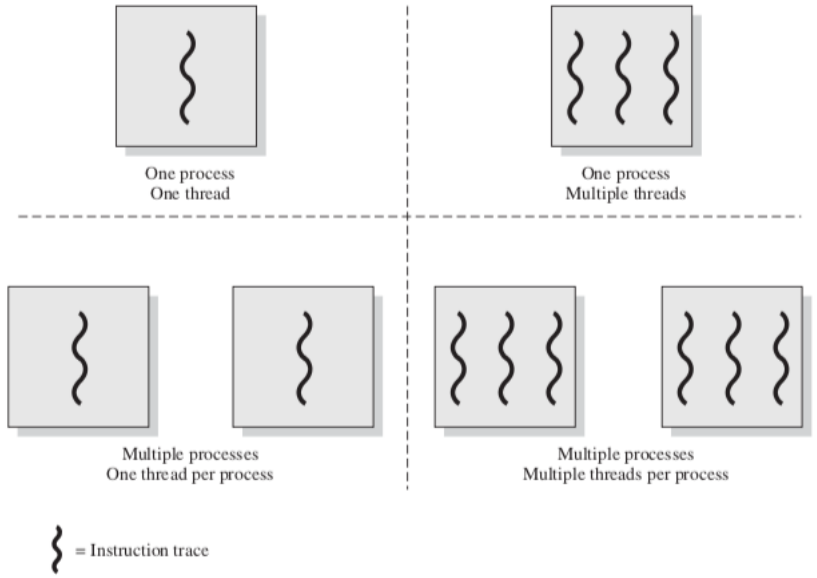


Figura 1.2: Procesos vs hebras (Fuente: [4]).

En los sistemas multihebrados conviven procesos y hebras, como se aprecia en la figura 1.2, un proceso puede tener una o más hebras cada una con su propio estado de ejecución, contexto de la CPU, pila de usuario y variables locales, entre otras cosas. Además, todas las hebras de un mismo proceso comparten información del mismo: el PCB del proceso, el estado del proceso, los recursos y memoria del proceso, el mismo espacio de direcciones y los datos del proceso. Existen hebras a nivel de usuario (ULT) y hebras a nivel de kernel (KLT) [4].

1.3. El planificador del sistema operativo

Definición La planificación es el método mediante el cual las hebras, los procesos o los flujos de datos tienen acceso a los recursos, por ejemplo, tiempo de procesador, ancho de banda en la comunicación, entre otros. El **planificador** es el componente del sistema operativo encargado de la planificación. La necesidad de algoritmos de planificación surgió con la aparición de sistemas operativos multitarea.

Completely Fair Scheduler (CFS) Desde la versión 2.6.23 el planificador tradicional de Linux fue reemplazado por el CFS. El 80 % del diseño de CFS fundamentalmente modela un procesador multitarea "ideal", con las siguientes características:

- Busca mantener el balance en el tiempo de procesador que se asigna a los procesos, de tal forma, que cada proceso debe recibir un tiempo equitativo.
- Cuando un proceso esta "fuera de balance" se le asigna tiempo de ejecución en el procesador. Para determinar el balance, CFS mantiene la cantidad de tiempo que se le ha asignado a un proceso en lo que llaman "Virtual Runtime". El proceso con menor "Virtual Runtime" es el próximo a ser ejecutado.
- CFS mantiene un árbol "rojo-negro" ordenado por tiempo, como se aprecia en la figura 1.3.

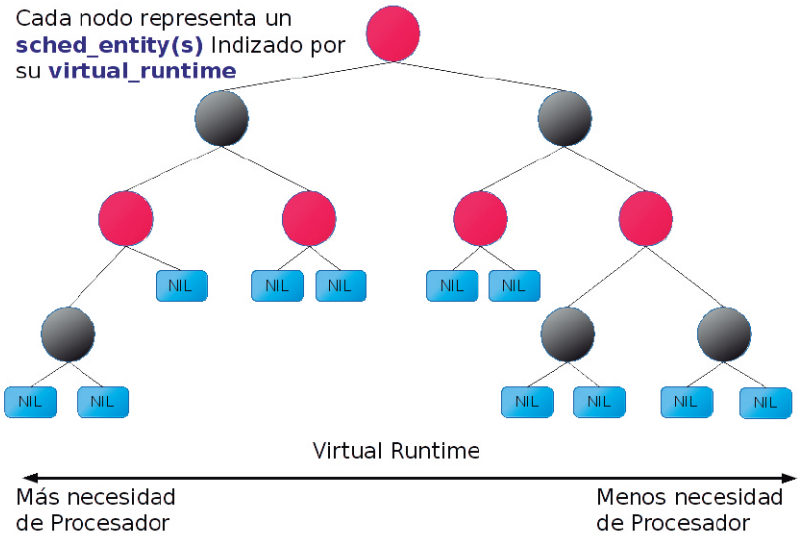


Figura 1.3: Arbol Rojo-Negro de CFS (Fuente: [6]).

Políticas de planificación Se utiliza una técnica de tiempo compartido: a cada proceso se le asigna un *quantum* de tiempo para ejecutarse en el procesador. La planificación se realiza según un ranking de prioridad: utiliza prioridades dinámicas que son actualizadas periódicamente. Los procesos que no han sido ejecutados por el procesador en un periodo largo de tiempo aumentan su prioridad, mientras que los que han sido ejecutados más tiempo, reducen su prioridad. Se utiliza la expropiación de procesos: un proceso se expropia cuando se acaba su quantum de tiempo asignado entra en ejecución un nuevo proceso con mayor prioridad que este.

Rutinas del planificador Como se representa en la figura 1.4, el planificador de Linux ejecuta varios métodos para cumplir su función, entre los principales están: *scheduler_tick()*, *try_to_wake_up()*, *recalc_task_prio()*, *load_balance()* y *schedule()*. Esta última rutina es una función muy importante del planificador, responsable de elegir el próximo proceso a ser ejecutado cuando: un proceso cede voluntariamente el procesador, un proceso se detiene debido a que recibe una señal para que duerma, un proceso agota su tiempo de ejecución, entre otros casos.

Estas rutinas del planificador se encuentran en el directorio *linux-<version>/kernel* del código fuente del kernel que se puede descargar de la página web *https://www.kernel.org/*. Algunos de los archivos principales son:

- *sched.c*. Contiene el código del planificador genérico. Las políticas de gestión están implementadas en otros archivos.
- *sched_fair.c*. Contiene el código del planificador *CFS* y provee las políticas de planificación para los procesos "Interactive" y "Batch".
- *sched_rt.c*. Provee las políticas usadas para los procesos "Real Time" [6].

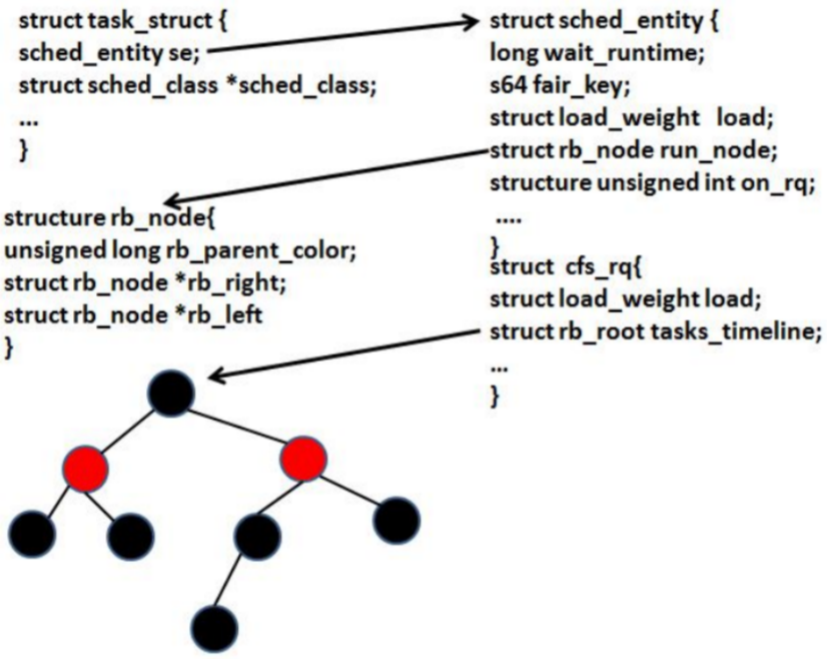


Figura 1.4: Rutinas del planificador del kernel de linux (Fuente: [6]).

1.4. Introducción al gestor del nivel de paralelismo

La computación paralela suele hacer uso de librerías y herramientas que facilitan la programación paralela, sin embargo, estas librerías no terminan de resolver correctamente el problema que plantean las aplicaciones multihebradas en sistemas no dedicados, dejando toda la responsabilidad al sistema operativo [8].

Problema El problema que motiva este trabajo se basa en las sobrecargas debidas al paralelismo en aplicaciones multihebradas, que suelen crecer con el número de hebras. Existen librerías y herramientas que facilitan la programación de aplicaciones paralelas. Normalmente el nivel de paralelismo de la aplicación es determinado de antemano. Por ejemplo, el número de hebras suele ser estático y seleccionado al inicio de la ejecución por el usuario. Estas librerías no modifican el número de hebras en tiempo de ejecución, y por lo tanto, no pueden mantener el nivel de eficiencia de la aplicación cuando se ejecuta en entornos no dedicados, donde varias aplicaciones compiten por los recursos existentes.

El presente Trabajo Fin de Grado (TFG) está basado en [8] donde se modificó el kernel del Sistema Operativo Linux para integrar un *Gestor del nivel de Paralelismo (GP)* a nivel de kernel que informe cuando las aplicaciones multihebradas pueden o no crear nuevas hebras. Se analizaron distintos métodos que permitían al SO cooperar con las aplicaciones multihebradas informándoles periódicamente del mejor número de hebras a utilizar. Sin embargo, los experimentos se realizaron sobre una arquitectura con 16 procesadores, mientras que en este TFG se pretende experimentar sobre una arquitectura con 57 procesadores, pero que permite hasta 228 hilos de ejecución simultáneos.

Las aplicaciones deben adaptar su nivel de paralelismo para mantener un buen rendimiento, teniendo en cuenta la disponibilidad de recursos en tiempo de ejecución, especialmente en sistemas no dedicados. Uno de los factores que inciden directamente en el tiempo total de ejecución de la aplicación en un sistema multicore es el número de hebras. Tradicionalmente, el número de hebras es establecido por el usuario al inicio de la ejecución de la aplicación. Generalmente, el comportamiento de la aplicación depende directamente del número de hebras, que se mantiene invariable durante todo el tiempo de ejecución. El GP determina periódicamente el número de hebras óptimo e informa a la aplicación para que pueda adaptarse durante la ejecución haciendo frente a las variaciones de las cargas computacionales en el sistema, producidas por la propia aplicación, o incluso por otras aplicaciones que se ejecutan en el sistema no dedicado. Los GPs que se propusieron analizan el comportamiento de las aplicaciones multihebradas en ejecución y adaptan el número de hebras automáticamente, manteniendo el máximo rendimiento posible. Esta cooperación entre el GP, la aplicación y el SO será beneficiosa para una amplia variedad de aplicaciones, entre ellas, aquellas que utilizan técnicas de B&B. En [8] se proponían distintos tipos de GP a nivel de usuario y a nivel de kernel.

Los GPs a nivel de usuario se caracterizan por su fácil implementación, pero poseen ciertas limitaciones respecto a los criterios de decisión que pueden utilizar para estimar el número óptimo de hebras. Mientras que los GPs a nivel de kernel ofrecen diseños de mayor complejidad, en función de la entidad encargada de estimar el número óptimo de hebras. La utilización de los distintos GPs, tanto a nivel de usuario como a nivel de kernel, requiere el manejo de la librería *Interfaz del Gestor de Paralelismo (IGP)* que facilita al programador la implementación de aplicaciones multihebradas y cuya API fue diseñada en el Trabajo Fin de Grado [7].

1.5. Estructura del trabajo

La memoria del presente TFG se ha organizado en dos bloques temáticos diferentes:

El primer bloque está compuesto por cuatro capítulos: En el capítulo primero se hace una introducción que abarca la descripción del kernel del SO Linux, conceptos de procesos, hebras y planificador, y termina aproximándose al gestor del nivel de paralelismo. En el segundo capítulo se esbozan los objetivos que persigue este trabajo. En el capítulo tercero se describen los GPs centrándose en el GP a nivel de kernel *KST (Kernel Decides on Sleeping Threads)*. Mientras que en el capítulo cuarto se muestran los materiales y métodos que se utilizarán y se describe el coprocesador Intel Xeon Phi sobre el que se realizará la implementación.

En el segundo bloque temático se exponen cinco capítulos donde se detallan cada una de las fases de desarrollo que se han seguido para realizar el trabajo: una primera fase de familiarización con la arquitectura Xeon Phi y actualización del software; una segunda fase que ha consistido en modificar el código fuente del kernel del coprocesador Phi, donde se explican los procesos de compilación del kernel, su modificación y las herramientas usadas para ello; una tercera fase que consiste en aplicar el GP a nuestra aplicación de benchmark y donde se describe de forma más técnica el funcionamiento de la librería *IGP*

y el programa *N-BodyIGP*; en la cuarta fase se analiza el comportamiento de *N-BodyIGP* mediante los distintos experimentos realizados; y por último en la quinta fase se recoge la difusión del trabajo.

Esta memoria finaliza con un capítulo donde se exponen la conclusión y el trabajo futuro.

1.6. Descripción y objetivos del trabajo

El presente Trabajo Fin de Grado se centra en la integración del GP a nivel de kernel desarrollado en [8] en una arquitectura Xeon Phi con 57 núcleos en el coprocesador Phi. Esta integración se basa en actualizar el software de gestión del coprocesador Phi y modificar el código fuente del kernel de Linux, de tal forma, que las aplicaciones multihebradas estén permanentemente informadas del instante más adecuado para crear cada hebra.

La experimentación realizada hace uso de la aplicación multihebrada *Parallel-N-Body-Simulation* [17], que simula la interacción entre partículas según las leyes de gravitación universal de Newton, que se ha utilizado de base para implementar un benchmark (*N-BodyIGP*) integrando la librería *IGP* para poder hacer uso del GP a nivel de kernel. Los objetivos fundamentales de este trabajo son:

1. *Adaptar el código fuente del kernel de linux.* Para ello se parte de un kernel 2.6.29 modificado en [8], y de un kernel 2.6.38 modificándolo de forma similar. Finalmente, este kernel modificado se instala sobre el software que integra el coprocesador Xeon Phi.

2. *Crear la aplicación de benchmark N-BodyIGP.* Partiendo de una aplicación ya existente *Parallel-N-Body-Simulation* [17], sobre la que se integró la librería *IGP* para crear *N-BodyIGP.*

3. *Verificación de funcionamiento del GP.* Se realizan varios experimentos que permiten evaluar el rendimiento del benchmark N-BodyIGP y analizar los resultados obtenidos mediante gráficas.

Estos objetivos se desarrollarán en diferentes fases expuestas en los capítulos sucesivos de este TFG.

1.7. Planificación temporal del TFG

El trabajo fin de grado fue planificado inicialmente para realizarse en 320 horas según el siguiente esquema:

1. **Conocer el funcionamiento de la Xeon Phi: 10 h**
   En esta primera etapa, a través de los diversos manuales [12, 13, 15] se ha analizado el funcionamiento de la Xeon Phi ejecutando aplicaciones de prueba, además de actualizar la pila de software (MPSS) a la versión 3.8.3. Se ha seleccioando la configuración más adecuada para conseguir un rendimiento óptimo [12].

2. **Modificar el planificador del kernel:** **100 h**
   Introducir las modificaciones adecuadas en el planificador del kernel de la Xeon Phi que permitan implementar el GP a nivel de kernel, según se describe en [8].

3. **Crear un módulo del kernel para añadir un GP a nivel de kernel:** **50 h**
   Esta fase pretendía inicialmente, que a partir del código fuente del kernel anteriormente modificado, extraer el código que permita gestionar el nivel de paralelismo (*GP*) a nivel de kernel, y se integrará en un módulo cargable del kernel, acompañado de un script que modifique y recompile el kernel del SO si fuese necesario. Así como, adecuar la librería *IGP* para utilizar el módulo del kernel. Sin embargo, esta fase no se ha podido implementar satisfactoriamente, ya que se ha detectado que el código de GP en el kernel requiere variables propias del kernel, por lo que no ha sido posible crear un módulo del kernel, sin modificar el código fuente del propio kernel.

4. **Seleccionar diferentes benchmarks para Xeon Phi:** **10 h**
   Buscar código fuente de aplicaciones multihebradas que se puedan ejecutar en Xeon Phi, y realizar una selección de distintas aplicaciones que en su conjunto abarquen un completo abanico de diferentes situaciones, decantándose finalmente por la aplicación *N-Body* [17].

5. **Analizar el comportamiento del módulo sobre un conjunto de benchmarks:** **100 h**
   Diseñar experimentos sobre distintos entornos de ejecución en sistemas dedicados, donde se ha analizado el comportamiento de *N-Body*, comparando las mejoras obtenidas. Los parámetros analizados han sido el número de hebras, tiempo total de ejecución, speed-up y eficiencia, entre otros.

6. **Difusión del TFG:** **50 h**
   Selección de distintas alternativas (diseño de página web, comentarios en foros y difusión en portales especializados) que permitan dar a conocer este módulo de kernel por Internet, ofreciendo tanto el código libre como la ayuda en español e inglés, para su correcta utilización. En este apartado, también se incluye la redacción de la memoria del TFG y la elaboración de presentaciones y vídeos para la defensa del TFG.

Capítulo 2

Gestores del nivel de paralelismo (GP)

Los gestores de paralelismo informan a las aplicaciones multihebradas sobre el número de hebras que dan lugar a un uso más eficiente de los recursos del sistema. Entre las distintas tipologías de gestores de paralelismo (GP) propuestos, se dividen en los GPs a nivel de usuario y los GPs a nivel de kernel.

2.1. Procedimiento de uso de un GP

El gestor del nivel de paralelismo inicia la monitorización cuando entra en ejecución la primera hebra de la aplicación multihebrada, siempre que esta hebra haya realizado una unidad de trabajo (por ejemplo, una iteración de un bucle). Independientemente de los parámetros escogidos por el GP para establecer el número de hebras adecuado, se deben establecer los procedimientos que los algoritmos multihebrados deben seguir después de ser informados por el GP sobre su nivel de paralelismo. Particularmente, en la creación dinámica de hebras es recomendable que la hebra con una mayor carga de trabajo sea la responsable de crear una nueva hebra, siempre que el número de hebras activas propuesto por el GP sea inferior al valor de *MaxThreads*[1]. En este TFG se ha escogido que sea siempre el proceso principal el cree las hebras, cuando se haya decidido así por el gestor, y no la hebra más cargada.

La figura 2.1 muestra el esquema de las distintas fases de funcionamiento en las que puede estar trabajando el GP. El proceso principal y las hebras de la aplicación multihebrada informan al GP a través de la fase de *Inicialización*, su deseo de ser gestionados por el GP. Posteriormente, en la fase de *Información*, cada una de las hebras computacionales informan al GP de los datos requeridos por el criterio de decisión, normalmente las unidades de trabajo realizadas desde el último informe. Una vez que todas las hebras activas han informado, el GP ejecuta la fase de *Evaluación* donde valora el criterio de decisión seleccionado, que puede informar sobre crear una nueva hebra, detener una hebra activa o no hacer nada. La decisión tomada es notificada a la hebra seleccionada por el GP en la fase de *Notificación*. En este caso se notificará al proceso principal, que es el único habilitado para crear hebras. A continuación, en la fase de *Restablecimiento*, el GP resetea todos los datos de la última evaluación realizada antes de comenzar un nuevo intervalo de evaluación con la fase de *Información*. El GP ejecuta la fase de *Terminación*, una vez que,

[1] *MaxThreads* es el valor que inicializa el usuario para indicar al GP cual es el número máximo de hebras que se permiten ejecutar simultáneamente.

tanto las hebras activas, como el proceso principal, han terminado su carga de trabajo e informan al GP su deseo de terminar. A continuación, se detallan las distintas fases en las que puede encontrarse el gestor del nivel de paralelismo para mantener un nivel de eficiencia adecuado:

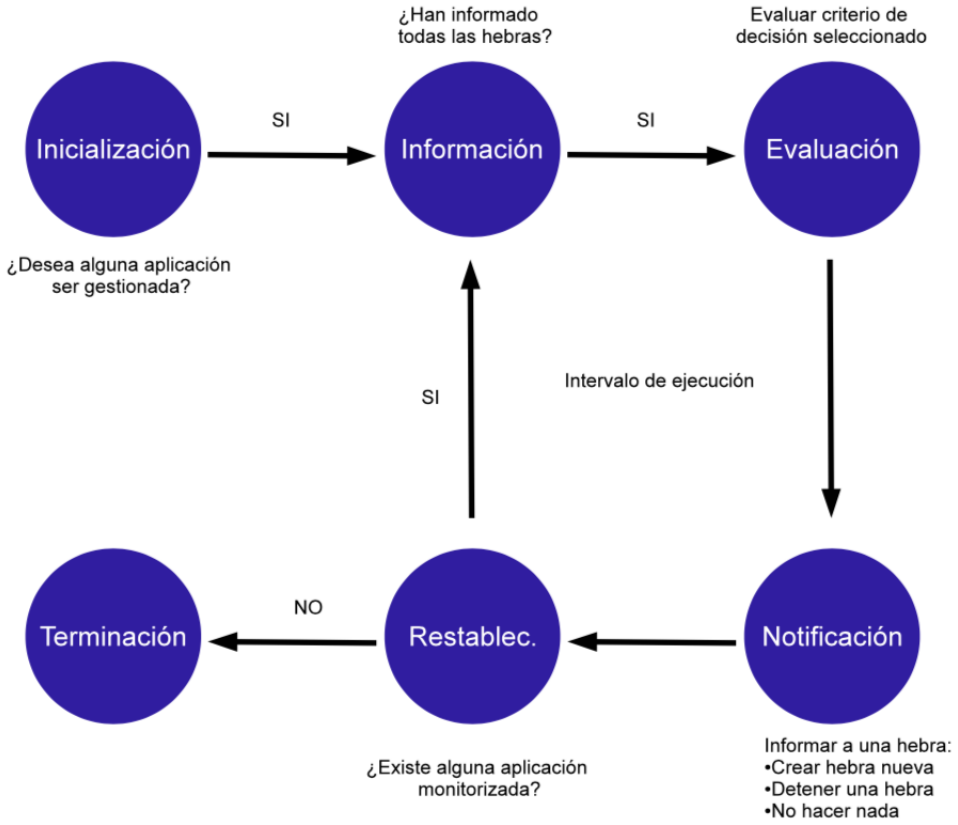


Figura 2.1: Diagrama de fases de funcionamiento de un GP (Fuente: [8]).

1. *Fases de Inicialización y Terminación*: La fase de *Inicialización* permite al proceso principal de una aplicación multihebrada activar el GP para que gestione la ejecución adaptativa de las hebras. En la fase de *Inicialización*, cada una de las hebras computacionales tienen que notificar al GP cual es la carga de trabajo asignada inicialmente. Sin embargo, si la aplicación crea otras hebras satélites sin carga computacional, destinadas a realizar otro tipo de tareas (por ejemplo, operaciones de entrada/salida, rebalanceo de carga, etc...), estas no tienen que ejecutar la fase de *Inicialización*.
   Por otro lado, cuando una hebra o el proceso principal finaliza su ejecución, debe ejecutar la fase de *Terminación* para indicar al GP que ya no tiene que ser monitorizada.

2. *Fase de Información*: Cada una de las hebras activas, debe informar al GP periódicamente, por ejemplo, en cada iteración del bucle, sobre la carga de trabajo realizada desde la última vez que informó. Esto permitirá al gestor calcular la carga de trabajo realizada por unidad de tiempo. El GP permanecerá en esta fase hasta que todas las hebras gestionadas hayan informado, pues debe asegurarse de que todas las hebras están ejecutando el respectivo bucle computacional. En esta fase, el

GP puede además recabar otra información adicional, no disponible por las hebras a nivel de usuario aunque si a nivel del SO, que puede ser utilizada posteriormente por el criterio de decisión seleccionado.
La fase de *Información* es la más crítica de todo el GP, dado que es la fase que más tiempo tarda en completarse, pues su tiempo de ejecución depende directamente del mecanismo utilizado por el GP para obtener la información utilizada en la fase de *Evaluación*.

3. *Fase de Evaluación*: En esta fase, el gestor debe analizar toda la información recogida en el fase anterior, y aplicar un criterio de decisión para obtener conclusiones sobre el nivel de eficiencia medido en el último intervalo de análisis. La elección de un criterio de decisión es de crucial importancia, para alcanzar niveles de eficiencia aceptables.
   El criterio de decisión determina la información que utilizará el GP en esta fase. El análisis del criterio de decisión puede desencadenar la creación de una nueva hebra, detención de alguna de las hebras activas, o simplemente no realizar nada pues se ha alcanzado un nivel óptimo de eficiencia.
   En entornos dedicados, el criterio de decisión más utilizado para aplicaciones HPC es el basado en el número de unidades de procesamiento. Sin embargo, este criterio de decisión es poco eficiente cuando trabajamos con entornos no dedicados, o las aplicaciones paralelas no tienen carga computacional suficiente, para que su ejecución en todos los procesadores del sistema sea eficiente.

4. *Fase de Restablecimiento*: Una vez tomada una decisión, el GP inicializa los contadores utilizados, como por ejemplo: la carga de trabajo realizada por cada hebra, el número de hebras activas, y el trabajo pendiente de cada hebra. Esta fase de reestablecimiento de contadores se va a ejecutar siempre que el número de hebras activas cambie, es decir, cada vez que se cree, detenga o termine una hebra. De esta forma, se garantiza que la fase de *Evaluación* analice el criterio de decisión con los parámetros recopilados en la fase de *Información*, es decir, durante el periodo de tiempo más reciente en el que no ha cambiado el número de hebras activas. La fase de *Restablecimiento* está íntimamente relacionada con la fase de *Evaluación*, dado que se debe de ejecutar una vez que el GP tome una decisión en la fase de *Evaluación*.

5. *Fase de Notificación*: En esta fase, el gestor del nivel de paralelismo notifica a cada una de las hebras activas la decisión tomada en la fase de *Evaluación*. En el supuesto de que el GP decida crear una nueva hebra, se debe seleccionar la hebra o bien la hebra más cargada, e informarle para que se divida, o notificar para que sea el proceso principal quien cree una nueva hebra. Una vez que la hebra más cargada detecta que tiene que dividirse, está crea una nueva hebra y le asigna la mitad de su carga de trabajo. Si se ha escogido que es el proceso principal el que se encarga de la creación de hebras, se creará una nueva hebra con una carga de trabajo previamente preestablecida. Si por el contrario, el gestor toma la decisión de detener o activar una hebra, entonces seleccionará una hebra activa o detenida, respectivamente. El mecanismo para seleccionar la hebra a detener es más complejo, ya que se debería detener aquella hebra que menos incida en la resolución del problema, por ejemplo, la hebra menos sobrecargada. Sin embargo, existen aplicaciones donde la hebra menos sobrecargada puede contener el trabajo pendiente que permita encontrar la solución

del problema, por lo que su detención aumentaría el tiempo total de ejecución.

2.2. GP a nivel de kernel: KST

Una de las principales dificultades para diseñar un GP consiste en seleccionar un criterio de decisión que garantice un nivel de eficiencia adecuado. Esto es aún más cierto en los gestores a nivel de usuario, por la dificultad de recopilar información para implementar algunos criterios de decisión. Existen otros criterios de decisión, que requieren de información no disponible a nivel de la aplicación, ni proporcionada por el sistema operativo de forma estándar a través de los pseudo-ficheros del directorio */proc*. Por este motivo, en esta sección se estudia la posibilidad de modificar el Kernel del sistema operativo.

En este caso, la interacción entre la aplicación multihebrada y el GP se realiza mediante una llamada al sistema, de tal forma que, cada una de las hebras gestionadas pueda ejecutar periódicamente la llamada al sistema para informar del trabajo realizado y conocer la decisión adoptada por el GP. El gestor KST es un GP a nivel de kernel basado en llamadas al sistema. Las llamadas al sistema son rutinas que se cargan en memoria durante el arranque del SO y permiten a las aplicaciones que se ejecutan a nivel de usuario acceder a la información almacenada a nivel del kernel. Una de las ventajas de esta versión es que no requiere de una profunda modificación del kernel.

La figura 2.2 muestra como se distribuye la ejecución de las distintas fases del gestor a nivel de kernel basado en llamadas al sistema, donde las fases de *Evaluación* y *Restablecimiento* se encuentran ahora a nivel de kernel. Mientras que la fase de *Información* se distribuye entre el nivel de usuario y el nivel de kernel, de tal forma que permite al GP recopilar información tanto desde la aplicación como desde el SO. Las fases de *Inicialización*, *Terminación* y *Notificación* son las únicas fases que se mantienen a nivel de usuario. Las fases de *Información* y *Notificación* se ejecutan directamente, o a través de librerías, dado que la llamada al sistema se puede realizar directamente en el código de la aplicación o a través de la librería IGP (Interfaz con el Gestor del nivel de Paralelismo). La característica principal de este gestor es que esta llamada al sistema se ejecuta a nivel de kernel, lo que le permite tener acceso a toda la información de todos los procesos, hebras y unidades de procesamiento del sistema. Por este motivo, el GP a nivel de kernel puede implementar cualquier criterio de decisión que haga uso de esta información en la fase de *Evaluación* [8].

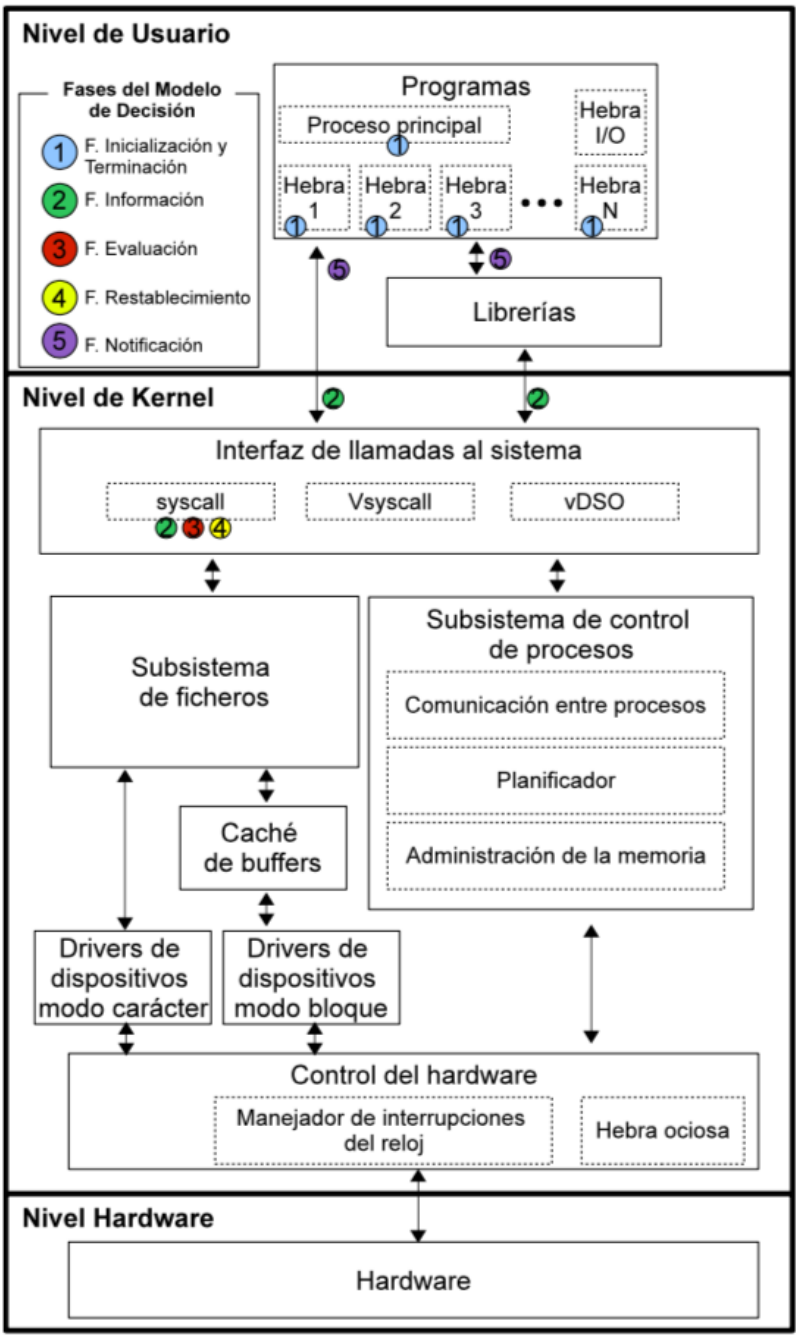


Figura 2.2: Ejecución de las fases del GP a nivel de kernel KST (Fuente: [8]).

Capítulo 3

Materiales y métodos

Materiales Se ha utilizado un equipo Dell PowerEdge T620 con procesador Intel Xeon E5-2609 v2 con 4 núcleos a 2,5 GHz de 64 bits y 10 MB de cache cada procesador, junto con un coprocesador Intel Xeon Phi 3120A con 57 núcleos de 4 hilos/núcleo, lo que hace un total de 228 hilos a 1,1 GHz de 64 bits y 512 KB de cache en cada núcleo (ver Figura 3.1) [11, 12].

Antes de la realización del trabajo, el equipo disponía del sistema operativo Red Hat Enterprise 7.1 sobre un kernel de Linux 3.10.0, mientras que el coprocesador Intel Xeon Phi tenía instalado un *Manycore Platform Software Stack (MPSS)* versión 3.5.1 sobre un kernel de Linux 2.6.38.8. Sin embargo, se ha actualizado el MPSS a la versión 3.8.3 de Linux. Además, ha sido utilizado el código fuente de linux versión 2.6.29 modificado [8] y la librería *IGP* para elaborar este trabajo.

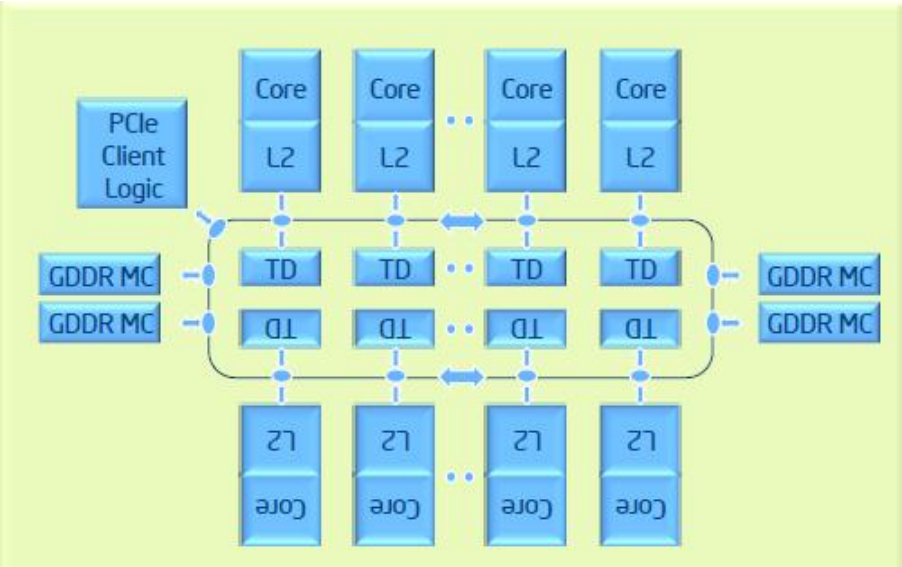


Figura 3.1: Arquitectura Knights Corner del coprocesador Phi (Fuente: [11]).

Herramientas Para trabajar con el equipo se ha utilizado el protocolo SSH (Secure SHell) bajo una VPN para poder conectarse a la red de la universidad, de manera que todo el trabajo práctico se ha hecho de manera remota. Se ha utilizado el compilador *icc (Intel C/C++ Compiler)* con el código fuente del kernel y del benchmark *N-BodyIGP* una

vez elaborado este. También se han creado diferentes scripts escritos en *Bash* que permiten agilizar las tareas de traslado de archivos, compilación de programas y del código fuente del kernel, ejecución del benchmark, entre otros.

3.1. Procesador Intel Xeon Phi

Xeon Phi es una serie de procesadores multinúcleo x86 destinados a ser usados en supercomputadores, servidores y computadores de alta gama (high-end workstations). Su arquitectura permite ejecutar aplicaciones programadas con APIs estándar, como PThread.

Originalmente fue diseñada como una GPU y comparte áreas de aplicación con las GPUs (Graphics Processing Units) pero la principal diferencia entre Xeon Phi y una GPGPU (General-Purpose GPU), como por ejemplo NVidia Tesla, es que Xeon Phi, con un núcleo compatible con x86, ejecuta software destinado a una CPU x86 estándar.
El modelo 3120A pertenece a la familia Knights Corner, segunda iteración de Intel MIC (Many Integrated Core). El dispositivo se trata de un periférico con interfaz PCIe 2.0. Es por ello que al procesador Xeon Phi se denomina *coprocesador* y la máquina a la que está conectado *host*, ya que pueden interpretarse como dos máquinas diferentes donde el host controla el coprocesador permitiendo comunicarse con él, configurarlo, ejecutar órdenes de apagado, reiniciado, etc. Esta jerarquía permite que los experimentos sean más fáciles de ejecutar, ya que si por ejemplo se ejecuta un programa en el coprocesador y éste se queda bloqueado, puede ser reiniciado con una orden desde el host.

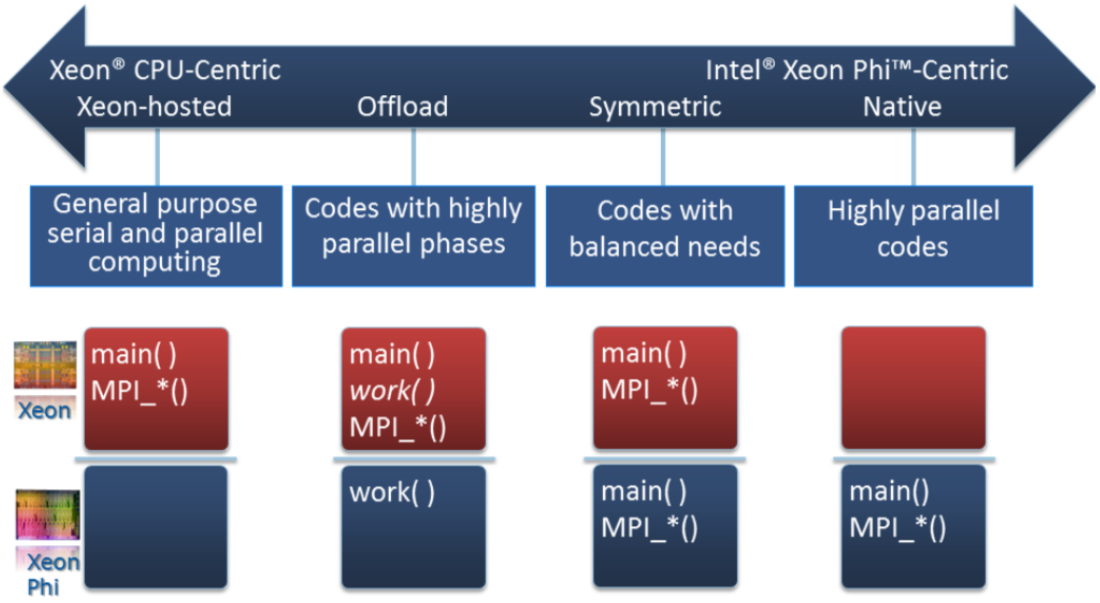


Figura 3.2: Modelos de programación (Fuente: [15]).

Intel MPSS O Intel Manycore Platform Software Stack es el conjunto de software necesario para utilizar el coprocesador Xeon Phi. Es como un "pequeño sistema operativo"

construido sobre el kernel de linux.

Este sistema Intel de *Host + Coprocesador* ofrece modelos de programación (ver figura 3.2) de los cuales se destacan dos:

- *Programación tipo offload*: se puede especificar en el código de un programa que parte de este se ejecute en el coprocesador, de forma que se delega computación aprovechando la arquitectura MIC. Esto se realiza mediante directivas offload o *pragmas*.
- *Programación nativa:* los programas se ejecutarán directamente en el coprocesador. Este es el modelo que se va a utilizar en la experimentación realizada en el presente TFG, ya que permite evaluar la aplicación multihebrada directamente sobre el coprocesador Xeon Phi [14, 15].

Parte II

Fases de desarrollo

Capítulo 4

Fase I. Interacción con la Xeon Phi

4.1. Utilización y ejecución de programas

Todo el trabajo fin de grado se ha realizado sobre un equipo con arquitectura Xeon Phi, conectado a Internet a través de la red privada de la Universidad de Almería. El acceso al equipo se ha realizado remotamente a través de una conexión SSH, y normalmente a través de la VPN de la Universidad.

Conexión SSH

1. Se precisaba conectarse a VPN (en este caso con la aplicación *TunnelBlick* recomendada por la universidad para equipos con Mac Os) usando un perfil proporcionado por la universidad.

2. En una ventana de terminal, se ejecuta el siguiente comando::

```
miusuario$ ssh usuario@ip_maquina
password:
[usuario@universidad-zahen]$
```

3. Se ha generado un par de claves pública y privada para poder conectarse sin necesidad de introducir la contraseña del usuario. En este caso, han sido creadas para los usuarios "rodrigo" y "root". Este paso es opcional.

Una vez conectado, ya se puede navegar por la máquina con los comandos propios de Linux.

Compilación y ejecución de programas Para compilar programas en la máquina Xeon Phi se puede utilizar el tradicional *gcc (GNU C Compiler)* o el compilador *icc (Intel C/C++ Compiler)*. Aunque ambos compiladores pueden funcionar, para ejecutar aplicaciones en el coprocesador Phi *gcc* no es el compilador adecuado ya que no incluye soporte para las instrucciones "Knights Corner vector" y las mejoras de rendimiento relacionadas. Sólamente es usado *gcc* para compilar el kernel y las herramientas relacionadas, no para compilar aplicaciones. Así que, para asegurar la estabilidad en las aplicaciones utilizadas, se usa el compilador *icc* de Intel.

Para testar nuestra máquina por primera vez, se ha compilado y ejecutado los siguientes programas básicos de Hola Mundo.

(a) *hello_ world.c*:

```
#include <stdio.h>
#include <stdlib.h>
void main(){
  printf("Hello World! \n");
}
```

Este primer ejemplo se trata de un Hola Mundo, que se compila de forma nativa en el host y se ejecuta en el coprocesador. Para ello, tras activar el compilador habilitando las variables de entorno, se compila, se envía el ejecutable mediante el comando `scp` al coprocesador y se ejecuta mediante `ssh`, tal y como se muestra a continuación.

```
$ . /opt/intel/parallel_studio_xe_2017.1.043/
   compilers_and_libraries_2017/linux/pkg_bin/compilervars.sh intel64
$ icc -nmic hello_world.c -o hello_world
$ scp hello_world mic0:
$ ssh mic0:/home/rodrigo/hello_world
Hello World!
$
```

(b) *hello_ offload.c*:

```
#include <stdio.h>
#include <stdlib.h>
void main(){
  #pragma offload target (mic:0){
    printf("Hello World from offloaded code running
      on the coprocessor!" \n");
  }
}
```

En este caso se usan las directivas *offload* para indicarle al compilador que partes del código se ejecutarán en el coprocesador, aunque la orden de ejecutar el programa se haga desde el host. El proceso sería activar las variables de entorno, compilar y lanzar el ejecutable en el host, tal y como se muestra a continuación:

```
$ . /opt/intel/parallel_studio_xe_2017.1.043/
   compilers_and_libraries_2017/linux/pkg_bin/compilervars.sh intel64
$ icc -offload hello_offload.c -o hello_world
$ ./hello_world mic0:
Hello World from offloaded code running on the coprocessor!
$
```

(c) *hello_ world_ offload.c*:

```
#include <stdio.h>
#include <stdlib.h>
```

```
3  void main(){
4    printf("Hello World from host...\n");
5    #pragma offload target (mic:0)
6    {
7      printf("and Hello World from offloaded code running on the
           coprocessor! \n");
8    }
9  }
```

En este ejemplo conviven las directivas *offload* o `pragma`. El proceso de compilación y ejecución es el mismo que en el caso anterior.

Este apartado ha permitido demostrar el procedimiento seguido para compilar una aplicación sencilla para ejecutarse en la Xeon Phi. Sin embargo, en este TFG se trabajará con la aplicación *N-BodyIGP*, por lo que se deberá compilar y ejecutar *N-BodyIGP* de forma nativa como en el primer ejemplo.

4.2. Actualizar la pila de software de la Xeon Phi (MPSS)

El coprocesador Phi estaba provisto inicialmente del software Intel MPSS en su versión 3.5.2 y se ha actualizado a la 3.8.3, siendo esta la última versión disponible al inicio del trabajo fin de grado. Actualmente, está disponible la versión 3.8.4, sin embargo, el procedimiento de actualización es el mismo. Para realizar la actualización se han seguido las indicaciones del fichero *readme.txt*, disponible en la documentación complementaria al código fuente de esta versión [16]. Todos los comandos se deben ejecutar en la máquina host con permisos de superusuario. A continuación, se describen los pasos seguidos para la actualización.

1. Descargar con `wget` el código fuente de la web de Intel en la carpeta personal

```
usuario # wget http://registrationcenter-download.intel.com/akdlm/
    irc_nas/11120/mpss-3.8-linux.tar
...
usuario #
```

   y descomprimir el fichero descargado.

```
usuario# tar -xvf mpss-8.8-linux.tar
...
usuario#
```

2. Desactivar el servicio MPSS.

```
# systemctl stop mpss
# modprobe -r mic
#
```

3. Consultar la versión del sistema operativo para saber qué comandos utilizar en el proceso (ya que varían según el SO).

```
# cat /etc/redhat-release
Red Hat Enterprise Linux Server release 7.1 (Maipo)
#
```

4. Desde la carpeta donde está localizado el código del MPSS actual, ejecutar el script `uninstall.sh` para desinstalarlo. En este caso, el código MPSS se encuentra en el directorio `/home/juan/mpss-3.5.17`

```
mpss 3.5.1# ./uninstall.sh
...
mpss 3.5.1#
```

   Al finalizar, comprobar que no se haya quedado instalado ningún paquete relacionado con MPSS.

```
mpss 3.5.1 # rpm -qa | grep -e intel-mic -e mpss
mpss 3.5.1 #
```

5. En Red Hat se recomienda desactivar `NetworkManager` y cambiar al daemon de red para evitar problemas con las interfaces virtuales de red.

```
# chkconfig NetworkManager off
Nota: Reenviando peticion a 'systemctl disable NetworkManager.service
   '.
# chkconfig network on
# service NetworkManager stop
Redirecting to /bon/systemctl stop NetworkManager.service
# service network start
Starting network (via systemctl):     [ OK ]
#
```

6. Desde el directorio donde se ha descargado el código fuente de la nueva versión, copiar los paquetes relacionados con nuestro SO e instalarlos.

```
mpss-3.8# cp ./modules/*'uname -r'*.rpm .
mpss-3.8# yum install *.rpm
...
mpss-3.8#
```

7. Cargar el driver *mic.ko* (proporciona configuración e información de control al software del host a través del sistema de archivos Sysfs de Linux) e inicializar la configuración inicial.

```
# modprobe mic
# micctrl --initdefaults
#
```

8. Actualizar el firmaware Flash y SMC. Primero comprobar el estado del coprocesador, a través del siguiente comando:

```
# micctrl -s
mic0: ready
#
```

si no se encontrara en estado `ready`, se debe de cambiar con el comando:

```
# micctrl -rw
```

y una vez listo ejecutar el comando:

```
#/usr/bin/micflash -update -device all -smcbootloader
No image path specified - Searching: /usr/share/mpss/flash
mic0: Flash image: /usr/share/flash/EXT_HP2_C0_0391-02.rom.smc
micflash: mic0: No compatible SMC boot-loader image found
#
```

Llegado este punto el readme advierte:
NOTE: If using C0 stepping, or 5110P B1 SKUs with a TA of G65758-253 or higher, or if the SMC boot loader version is 1.8, use this command in step 2...

step 2 se refiere a este paso 8 así que, como el anterior comando no ha funcionado, se debe ejecutar el siguiente comando:

```
# /usr/bin/micflash -update -device all
No image path specified - Searching: /usr/share/mpss/flash
mic0: Flash image: /usr/share/flash/EXT_HP2_C0_0391-02.rom.smc
mic0: Flash update started
mic0: Flash update done
mic0: SMC update started
mic0: SMC update done
mic0: Transitioning to ready state

Please restart host for flash changes to take effect
#
```

9. Reiniciar la máquina host con el comando:

```
# reboot
```

y una vez reiniciado el sistema activar el servicio MPSS, además de establecer que se active al inicio.

```
# systemctl start mpss
# systemctl enable mpss
#
```

10. Configuración de acceso SSH: La comunicación con el sistema operativo Linux del coprocesador Xeon Phi se realiza mediante una interfaz de red estándar. La interfaz usa un driver de red virtual sobre el bus PCIe, lo que requiere utilizar una conexión SSH entre el host y el coprocesador.

El coprocesador Xeon Phi soporta el acceso usando claves SSH, autentificación de contraseña, o autentificación basada en host. En la fase de configuración inicial del MPSS se crean usuarios para cada coprocesador basados en los IDs de los usuarios actuales en el archivo `/etc/passwd` del host. Para cada usuario en `/etc/psswd` (incluído el host) si existen archivos de claves SSH en el directorio `.ssh/` del usuario, estas claves son extendidas al sistema de archivos del coprocesador Xeon Phi.

Los administradores que prefieran proveer sólamente un limitado conjunto de credenciales de usuario (root, sshd, micuser, nobody and nfsnobody) lo pueden hacer ejecutando estos comandos después de la configuración inicial.

```
# systemctl stop mpss
# micctrl --userupdate=none
# systemctl start mpss
```

Finalmente, se recomienda tener instalada la última versión de *OpenSSH* en el coprocesador que se encuentra entre los paquetes del código fuente.

Capítulo 5

Fase II. Modificar el código fuente del kernel

Este capítulo pretende mostrar las modificaciones realizadas en el código fuente del kernel, de tal forma, que permita al lector comprender mejor el funcionamiento del GP y facilite futuras mejoras y actualizaciones. Para ello, se parte del código fuente del kernel de Linux versión 2.6.29 modificado en [8]. Éste se va a denominar *kernel inicial* o kernel *2.6.29-GP*.

El coprocesador Xeon Phi se ha actualizado con el MPSS versión 3.8.3, descrito en el capítulo anterior. Esta versión trabaja sobre el kernel de linux versión 2.6.38, por lo que se ha adaptado el código fuente del kernel 2.6.38 con las modificaciones propuestas en el kernel *2.6.29-GP*. Como resultado se ha obtenido un kernel estable que llamaremos *kernel final* o kernel *2.6.38-GP*.

5.1. El kernel inicial

A continuación, se exponen los diferentes ficheros del kernel *2.6.29-GP* modificados o creados, de tal forma, que se describe brevemente la función de cada fichero y se muestra la parte del código modificado. Las modificaciones en el kernel abarcan un total de 19 ficheros modificados y la creación de 4 ficheros nuevos.

- *kernel/adaptive.h*, *kernel/adaptive.c* y *kernel/define_ adaptive.h*:

  Estos ficheros implementan las llamadas al sistema, funciones, variables, estructuras de datos, etc, que conforman el gestor de paralelismo a nivel de kernel. Mientras que *adaptive.c* y *adaptive.h* implementan el grueso del GP, *define_ adaptive.h* recoge las contantes para identificar los métodos y criterios de adaptación. Estos 3 archivos han sido añadidos directamente al kernel original del 2.6.38 modificando el *Makefile* para su compilación.

- *kernel/Makefile*:

  Makefile para compilar el kernel de linux al que se le ha añadido `adaptive.o` para compilar *adaptive.c*.

```
obj-y = sched.o  adaptive.o  fork.o exec_domain.o panic.o printk.o
    cpu.o exit.o itimer.o time.o softirq.o resource.o
    sysctl.o capability.o ptrace.o timer.o user.o
    signal.o sys.o kmod.o workqueue.o pid.o
    rcupdate.o extable.o params.o posix-timers.o
    kthread.o wait.o kfifo.o sys_ni.o posix-cpu-timers.o mutex.o
    hrtimer.o rwsem.o nsproxy.o srcu.o semaphore.o
    notifier.o ksysfs.o pm_qos_params.o sched_clock.o cred.o
    async.o
```

- *include/linux/sched.h*:

  Archivo de cabecera del planificador de linux.

```
...

struct sched_entity {
...
   u64 uninterruptible_time;
   u64 interruptible_time;
   u64 waiting_time;
...
```

Las estructuras *struct sched_ entity* y *struct task_ sctruct* recopilan datos de las hebras, a través de las siguientes variables:

uninterruptible_time: tiempo que la hebra ha estado en estado ininterrumpible.

interruptible_time: tiempo que la hebra ha estado en estado interrumpible.

waiting_time: tiempo que la hebra ha estado en estado waiting.

```
struct task_struct {
...
   u64 runtime;
   u64 memsleeptime;
   u64 blocksleeptime;
   u64 waiting;
   u64 sleeptime;

   bool detenida;
   bool detenible;
   bool detenida_ultimointervalo;
   bool terminada;

   unsigned long *pointer_var_done_work;
   unsigned long done_work;
   unsigned long total_done_work;
   unsigned long *pointer_var_remaining_work;
   unsigned long remaining_work;
   int id_proc_acs;
   int id_thd_acs;
}; // Fin de task_struct
```

runtime: tiempo que la tarea ha estado en ejecución desde la última comprobación del criterio de decisión por el GP.

memsleeptime: tiempo que la tarea ha estado dormida en el intervalo entre dos consultas consecutivas del criterio de decisión por el GP, provocado por la reserva de memoria y swaping.

blocksleeptime: tiempo que la tarea ha estado dormida en el intervalo entre dos consultas consecutivas del criterio de decisión por el GP, provocado por el bloqueo de hebras debido a secciones críticas e operaciones de entrada/salida.

waiting: tiempo que la tarea ha estado esperando en cola de ejecución entre dos consultas consecutivas del criterio de decisión por el GP.

sleeptime: tiempo que la tarea ha estado detenida en el intervalo entre dos consultas consecutivas del criterio de decisión por el GP, por cualquier motivo.

detenida: indica si la hebra ha sido detenida a petición del GP.

detenible: indica que la hebra puede ser detenida.

detenida_ultimointervalo: indica si la hebra ha sido detenida en el último intervalo.

terminada: indica si la hebra ha terminado su ejecución.

pointer_var_done_work y **done_work:** contabilizan el trabajo realizado desde la última consulta del criterio de decisión por el GP.

pointer_var_remaining_work y **remaining_work:** contabilizan la predicción del trabajo pendiente por realizar.

id_proc_acs: identificador de la aplicación adaptativa.

id_thd_acs: identificador de la hebra activa en la aplicación.
Todos estos datos son utilizados periódicamente por el GP para evaluar el criterio de decisión seleccionado.

- *kernel/sched.c*:

  Este fichero implementa el planificador de linux, donde se han incluido al inicio de *sched.c* algunas variables de *adaptive.c* que se utilizarán en el planificador.

```
1  #include "adaptive.h"
2
3  extern int activar_detencion_hebras;
4  extern int cont;
5  extern int ejecutar_analizar;
6  extern bool creando_hebra;
7  ...
```

  También se incluye la función *sched_fork* que se encarga de inicializar las variables para cada nuevo proceso o hebra creada.

```
1  static void __sched_fork(struct task_struct *p)
2  {
3    p->se.exec_start = 0;
4    p->se.sum_exec_runtime = 0;
5    p->se.prev_sum_exec_runtime = 0;
6    p->se.last_wakeup = 0;
7    p->se.avg_overlap = 0;
```

```
8
9   #ifdef CONFIG_SCHEDSTATS
10    p->se.wait_start = 0;
11    p->se.sum_sleep_runtime = 0;
12
13     p->se.interruptible_time = 0;
14     p->se.uninterruptible_time = 0;
15     p->se.waiting_time = 0;
16     p->detenida = false;
17     p->detenible = false;
18     p->detenida_ultimointervalo = false;
19     p->terminada = false;
20     p->total_done_work = p->done_work = p->remaining_work = 0;
21     p->pointer_var_done_work = p->pointer_var_remaining_work = NULL;
22     p->id_proc_acs = p->id_thd_acs = -1;
23  ...
```

La funcion *scheduler_tick* se ejecuta cada tick de reloj, por lo que se ha utilizado para incluir las versiones más avanzadas de GP (KITST y SST).

```
1   void scheduler_tick(void)
2   {
3     int cpu = smp_processor_id();
4     struct rq *rq = cpu_rq(cpu);
5     struct task_struct *curr = rq->curr;
6
7     unsigned long work_done, remaining_work;
8
9     sched_clock_tick();
10    spin_lock(&rq->lock);
11    update_rq_clock(rq);
12    update_cpu_load(rq);
13    curr->sched_class->task_tick(rq, curr, 0);
14
15    if((ejecutar_analizar==KITST || ejecutar_analizar==SST)
16      &&!creando_hebra){
17      if(current->id_proc_acs!=-1) {
18        if(current->id_thd_acs!=-1) {
19        // Leer el trabajo realizado y trabajo pendiente
20          if(copy_from_user(&work_done, current->pointer_var_done_work,
21            sizeof(unsigned long)))
22            printk(KERN_INFO "%llu ms: ERROR no puede copiar trabajo
23              hecho.\n",sched_clock()/1000000);
24
25          current->done_work=work_done-current->total_done_work;
26
27          if(current->done_work>0){
28            if(copy_from_user(&remaining_work,
29              current->pointer_var_remaining_work,
30              sizeof(unsigned long)))
31              printk(KERN_INFO "%llu ms: ERROR no puede copiar trabajo
32                realizado.\n",sched_clock()/1000000);
33            current->remaining_work=remaining_work;
34            if(ejecutar_analizar==SST) Analizar();
35            // Llamada realizada desde el scheduler
36          }
37        }
```

```
38      }
39    }
40
41    spin_unlock(&rq->lock);
42  #ifdef CONFIG_SMP
43    rq->idle_at_tick = idle_cpu(cpu);
44    trigger_load_balance(rq, cpu);
45  #endif
46  }
```

- *kernel/sched_rt.c*:

  Este fichero implementa la "Real-Time Scheduling Class".

```
1  /*static*/ void deactivate_task(struct rq *rq, struct task_struct *p,
       int sleep);
```

  Se ha comentado "static" para poder usar esta función en otra parte del código, ya que puede ser de utilidad para el GP.

- *kernel/sched_fair.c*:

  Este fichero implementa la "Completely Fair Scheduling (CFS) Class". Se han añadido líneas en dos funciones para almacenar información extra.

```
1  static void
2  update_stats_wait_end(struct cfs_rq *cfs_rq, struct sched_entity *se)
3  {
4    schedstat_set(se->wait_max, max(se->wait_max,
5        rq_of(cfs_rq)->clock - se->wait_start));
6    schedstat_set(se->wait_count, se->wait_count + 1);
7    schedstat_set(se->wait_sum, se->wait_sum +
8        rq_of(cfs_rq)->clock - se->wait_start);
9
10   se->waiting_time +=rq_of(cfs_rq)->clock-se->wait_start;
11
12   schedstat_set(se->wait_start, 0);
13 }
```

```
1
2  static void enqueue_sleeper(struct cfs_rq *cfs_rq, struct sched_entity
       *se)
3  {
4  #ifdef CONFIG_SCHEDSTATS
5    if (se->sleep_start) {
6      u64 delta = rq_of(cfs_rq)->clock - se->sleep_start;
7      struct task_struct *tsk = task_of(se);
8      if ((s64)delta < 0)
9        delta = 0;
10     if (unlikely(delta > se->sleep_max))
11       se->sleep_max = delta;
12     se->sleep_start = 0;
```

```
13      se->sum_sleep_runtime += delta;
14
15      se->interruptible_time +=delta;
16
17      account_scheduler_latency(tsk, delta >> 10, 1);
18    }
19    if (se->block_start) {
20      u64 delta = rq_of(cfs_rq)->clock - se->block_start;
21      struct task_struct *tsk = task_of(se);
22
23      if ((s64)delta < 0)
24        delta = 0;
25      if (unlikely(delta > se->block_max))
26        se->block_max = delta;
27      se->block_start = 0;
28      se->sum_sleep_runtime += delta;
29
30      se->uninterruptible_time +=delta;
31
32      /*
33       * Blocking time is in units of nanosecs, so shift by 20 to
34       * get a milliseconds-range estimation of the amount of
35       * time that the task spent sleeping:
36       */
37      if (unlikely(prof_on == SLEEP_PROFILING)) {
38        profile_hits(SLEEP_PROFILING, (void *)get_wchan(tsk),
39             delta >> 20);
40      }
41      account_scheduler_latency(tsk, delta >> 10, 0);
42    }
43  #endif
44  }
```

- *arch/x86/include/asm/unistd_64.h* y *arch/x86/include/asm/vsyscall.h*:

  Este fichero contiene los números asociados a las llamadas al sistema y se han definido 6 nuevos para las llamadas al sistema de nuestro GP.

```
1  #define __NR_get_status          333
2  __SYSCALL(__NR_get_status, sys_get_status)
3  #define __NR_warns_status        334
4  __SYSCALL(__NR_warns_status, sys_warns_status)
5  #define __NR_new_thread_created    335
6  __SYSCALL(__NR_new_thread_created, sys_new_thread_created)
7  #define __NR_thread_end          336
8  __SYSCALL(__NR_thread_end, sys_thread_end)
9  #define __NR_stat_system        337
10 __SYSCALL(__NR_stat_system, sys_stat_system)
11 #define __NR_detener_hebra        338
12 __SYSCALL(__NR_detener_hebra, sys_detener_hebra)
```

- *arch/x86/kernel/syscall_table_32.S*:

  Este script recoge un listado de las distintas llamadas al sistema y se usa para la compilación del kernel. Se han añadido las 6 nuevas llamadas al sistema.

```
...
  .long sys_get_status
  .long sys_warns_status
  .long sys_new_thread_created
  .long sys_thread_end
  .long sys_stat_system
  .long sys_detener_hebra
// Final del archivo
```

- *arch/x86/include/asm/vsyscall.h*:

  Archivo de cabecera con atributos y propiedades de las vsyscalls. En este archivo se han añadido variables de `kernel/adaptive.c` utilizados por el GP.

```
#define __section_acs_dec_proc
  __attribute__ ((unused, __section__ (".acs_dec_proc"), aligned(16)))
#define __section_acs_dec_thd
  __attribute__ ((unused, __section__ (".acs_dec_thd"), aligned(16)))
#define __section_acs_dec_id_new
  __attribute__ ((unused, __section__ (".acs_dec_id_new"),
    aligned(16)))

... // Codigo original

extern int __acs_dec_proc;
extern int __acs_dec_thd;
extern int __acs_dec_id_new;

...

extern int acs_dec_proc;
extern int acs_dec_thd;
extern int acs_dec_id_new;

...
```

- *arch/x86/kernel/vsyscall_64.c*:

  En este archivo se incluyen funciones y variables de las vsyscalls para arquitecturas de 64 bits. Se han añadido las siguientes líneas al inicio del fichero, que son relativas a la declaración de las variables utilizadas por el GP para identificar al proceso de la aplicación multihebrada, a cada una de las hebras y a la nueva hebra para su próxima creación, respectivamente.

```
int __acs_dec_proc __section_acs_dec_proc;
int __acs_dec_thd __section_acs_dec_thd;
int __acs_dec_id_new __section_acs_dec_id_new;
...
```

- *arch/x86/kernel/vmlinux_64.lds.S*:

Script para compilar el kernel para la arquitectura x86-64. Se han añadido la definición de las variables anteriores del GP.

```
...
.acs_dec_proc : AT(VLOAD(.acs_dec_proc)) { *(.acs_dec_proc) }
  acs_dec_proc = VVIRT(.acs_dec_proc);

.acs_dec_thd : AT(VLOAD(.acs_dec_thd)) { *(.acs_dec_thd) }
  acs_dec_thd = VVIRT(.acs_dec_thd);

.acs_dec_id_new : AT(VLOAD(.acs_dec_id_new)) { *(.acs_dec_id_new) }
  acs_dec_id_new = VVIRT(.acs_dec_id_new);
...
```

- *arch/x86/vdso/vextern.h*:

  Se han añadido las lineas siguientes necesarias para la declaración externa de las variables anteriores, de tal forma que, se permita acceder a su contenido desde cualquier fichero del código fuente del kernel.

```
VEXTERN(acs_dec_proc)
VEXTERN(acs_dec_thd)
VEXTERN(acs_dec_id_new)
```

- *arch/x86/vdso/vadaptive.c*:

  Este fichero ha sido creado y añadido al kernel, para permitir que las versiones avanzadas del GP (KITST y SST) puedan informar a la aplicación multihebrada directamente desde el kernel, a través de vDSO, minimizando el uso de las llamadas al sistema.

```
#include <linux/kernel.h>
#include <linux/getcpu.h>
#include <linux/jiffies.h>
#include <linux/time.h>
#include <asm/vsyscall.h>
#include <asm/vgtod.h>
#include "vextern.h"

notrace int __vdso_get_status(int pid_proc, int pid_thd) {
  int id_thd, id_proc;
  id_proc=*vdso_acs_dec_proc;
  id_thd=*vdso_acs_dec_thd;
  if(pid_proc==id_proc && pid_thd==id_thd)
      return(*vdso_acs_dec_id_new);
  else return(-1);
}
int get_status(int pid_proc, int pid_thd) __attribute__((weak,
    alias("__vdso_get_status")));
```

- *arch/x86/vdso/Makefile*:

Makefile para los ficheros del directorio *arch/x86/vdso* actualizado para compilar *vadaptive.c*.

```
vobjs-y := vdso-note.o vclock_gettime.o vgetcpu.o vvar.o vadaptive.o
```

- *arch/x86/vdso/vdso.lds.S*:

  Este es el linker script de Linux para las funciones vDSO. Es necesario añadir la función creada en *vadaptive.c*.

```
VERSION {
  LINUX_2.6 {
  global:
    clock_gettime;
    __vdso_clock_gettime;
    gettimeofday;
    __vdso_gettimeofday;
    getcpu;
    __vdso_getcpu;

    get_status;
    __vdso_get_status;
  local: *;
  };
}
```

- *arch/x86/kernel/process_ 64.c*:

  Este fichero contiene la función que ejecutan las hebras ociosas del kernel, de tal forma que aquellos procesadores que no tienen ninguna tarea en la cola de ejecución, ejecutan una hebra ociosa. En este fichero, se ha incluido modificaciones que permitan al GP, denominado KITST, ser ejecutado cada vez que una hebra ociosa entre en ejecución.

```
#include <../kernel/define_adaptive.h>

void Analizar(void);
extern int ejecutar_analizar;

...

void cpu_idle(void)
{
  current_thread_info()->status |= TS_POLLING;
  /* endless idle loop with no priority at all */
  while (1) {
    tick_nohz_stop_sched_tick(1);
    while (!need_resched()) {

    if(ejecutar_analizar==KITST) Analizar();
    // Llamada realizada desde la hebra ociosa
      rmb();
...
```

- *include/linux/resource.h* y *usr/include/linux/resource.h*:

  Se encarga la contabilidad o control de los recursos. Con este cambio se permite la medida de tiempos de ejecución de hebras.

```
#define RUSAGE_SELF 0
#define RUSAGE_CHILDREN (-1)
#define RUSAGE_THREAD (-2)
#define RUSAGE_BOTH (-3)
//#define RUSAGE_BOTH (-2) /* sys_wait4() uses this */
//#define RUSAGE_THREAD 1 /* only the calling thread */
```

- *kernel/sys.c*:

  Aquí se recogen propiedades de las llamadas al sistema. Los cambios permiten realizar medidas de tiempo de hebras.

```
static void k_getrusage(struct task_struct *p, int who, struct rusage
    *r)
{
  struct task_struct *t;
  unsigned long flags;
  cputime_t utime, stime;
  struct task_cputime cputime;

  memset((char *) r, 0, sizeof *r);
  utime = stime = cputime_zero;

  if (who == RUSAGE_THREAD) {
    utime = task_utime(current);
    stime = task_stime(current);
    accumulate_thread_rusage(p, r);
    goto out;
  }

  if (!lock_task_sighand(p, &flags))
    return;

  switch (who) {
    case RUSAGE_THREAD:
      utime = p->utime;
      stime = p->stime;
      r->ru_nvcsw = p->nvcsw;
      r->ru_nivcsw = p->nivcsw;
      r->ru_minflt = p->min_flt;
      r->ru_majflt = p->maj_flt;
      r->ru_inblock = task_io_get_inblock(p);
      r->ru_oublock = task_io_get_oublock(p);
      break;
...
```

- *arch/mips/kernel/mips-mt-fpaff.c*:

  Este código acoge rutinas generales de soporte MIPS MT, utilizados en kernels

AP/SP, SMVP o SMTC. Se ha comentado "static" en la siguiente función. Esta función devuelve un puntero a la estructura *task_struct* de un proceso a partir de su *pid*, y de este modo, al no ser ya una función static, puede ser visible por otras funciones fuera de este fichero, por ejemplo, en *kernel/adaptive.c*

```
1 /*static*/ inline struct task_struct *find_process_by_pid(pid_t pid)
2 {
3   return pid ? find_task_by_vpid(pid) : current;
4 }
```

5.2. Compilación del kernel y proceso de adaptación

Una vez realizadas todas las modificaciones descritas anteriormente sobre el código fuente del kernel 2.6.38, se procede a realizar la compilación de esta nueva actualización, que se denominará kernel *2.6.38-GP*. Sin embargo, en la práctica, no ha sido inmediato incluir estas modificaciones y se han tenido que realizar por partes.

5.2.1. Compilación del kernel

En este apartado, se va a describir el procedimiento utilizado para compilar el kernel, a partir del código fuente original *linux-2.6.38+mpss3.8.3*. Cabe destacar que este procedimiento se realiza en el host de la Xeon Phi, y no en el coprocesador (*mic0*). El código fuente oficial de MPSS se ha descargado de la página *https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss#lx38rel*.

Primeramente, en el directorio donde se encuentra el código fuente del MPSS 3.8.3, se descomprime el fichero *linux-2.6.38+mpss3.8.3.tar.bz2* y crear un directorio *newtests* que será el directorio de trabajo; mover aquí el código *linux-2.6.38+mpss3.8.3* oficial, y hacer una copia del mismo llamada *tfgkernel-0* que será el código que compilaremos. De este modo siempre se tendrá una copia del código original descomprimida sin alterar.

```
rodrigo# cd /home/rodrigo/mpss-3.8.3/src/
src# tar -xvf linux-2.6.38+mpss3.8.3.tar.bz2
...
src# mkdir /home/rodrigo/mpss-3.8.3/src/newtests
rodrigo# cp linux-2.6.38+mpss3.8.3/ newtests/tfgkernel-0/
rodrigo#
```

A continuación, se habilita el entorno para poder compilar, copiar el archivo de configuración *.config* del código original, ya que es una configuración conocida, y ejecutar el comando `make`, tal y como se muestra con los siguientes comandos:

```
newtests# cd tfgkernel-0/
tfgkernel-0# . /opt/mpss/3.8.3/environment-setup-k1om-mpss-linux
tfgkernel-0# cp /opt/mpss/3.8.3/sysroots/k1om-mpss-linux/boot/config
   -2.6.38.8+mpss3.8.3 .config
tfgkernel-0# make ARCH=k1om CROSS_COMPILE=k1om-mpss-linux- -j4
...
```

Los parámetros utilizados con el comando `make` son:

- `ARCH=k1om:` especifica la arquitectura (compatible con Knights Corner).
- `CROSS_COMPILE=k1om-mpss-linux-:` permite compilar únicamente aquellos ficheros, no compilados anteriormente, o modificados. Este parámetro permite reducir considerablemente el tiempo de compilación (varios segundos), ya que la compilación completa de todo el código fuente asciende a casi 3 minutos. Esto es muy útil cuando se editan unos pocos ficheros y se compila, puesto que make los detecta y no necesita compilar el código fuente entero.
- `-j4:` divide el trabajo de compilación en 4 procesadores del host, por lo que también, permite reducir el tiempo de compilación.

Una vez finalizada la compilación se muestra por pantalla el siguiente mensaje:

```
Root device is (253, 1)
Setup is 15256 bytes (padded to 15360 bytes).
System is 2666 kB
CRC 1487fdb0
Kernel: arch/x86/boot/bzImage is ready (#1)
tfgkernel-0#
```

A continuación, se debe apagar el coprocesador (o chequear primero su estado en caso de que estuviera apagado). Una vez apagado, cambiar la imagen de arranque del kernel y arrancar el coprocesador, a través de los siguiente comandos:

```
tfgkernel-0# micctrl --shutdown
...
tfgkernel-0# micctrl --osimage=/home/rodrigo/mpss-3.8.3/src/tfgkernel-0/
   arch/k1om/boot/bzImage --sysmap=/usr/share/mpss/boot/System.map-
   knightscorner
tfgkernel-0# micctrl --reboot
...
```

El arranque del coprocesador no es inmediato, suele tardar menos de 1 minuto, pero si se detecta que el coprocesador no se reinicia, se deberán ejecutar los siguientes comandos:

```
tfgkernel-0# micctrl --reset
...
tfgkernel-0# micctrl --boot
...
```

De todas formas, también se puede utilizar el siguiente comando, para detectar cuándo el coprocesador está operativo.

```
tfgkernel-0# micctrl --wait
...
```

Finalmente, se puede conocer el estado del coprocesador (*mic0*) y la versión de la imagen del kernel en ejecución, a través de los siguientes comandos:

```
tfgkernel-0# micctrl --status
mic0: online (mode: linux image: /home/rodrigo/mpss-3.8.3/src/newtests/
   tfgkernel-0/arch/k1om/boot/bzImage)
mic0# ssh mic0
mic0# uname -a
...
```

Si hubiera necesidad de restaurar la version anterior del *mic0* habría que apagarlo, cambiar la imagen y volver a arrancar. Si no se puede restaurar la versión anterior, se debe reinstalar MPSS, aunque esta situación no ha llegado ha suceder durante la realización de este trabajo.

5.2.2. Proceso de adaptación para instalar el kernel 2.6.38-GP

Anteriormente, se ha comentado el procedimiento de compilación e instalación del Kernel en el coprocesador a partir del código fuente del kernel oficial. Sin embargo, la compilación e instalación del kernel *2.6.38-GP* no ha sido sencilla, debido a la dependencias entre el elevado número de ficheros a modificar. A continuación, se detalla el procedimiento utilizado:

1. Como preparativo, se crea el directorio *linux-2.6.38+mpss3.8.3-mod* (copia de *linux-2.6.38+mpss3.8.3*) y se modifican los ficheros acorde a *2.6.29-GP* según nuestro criterio, emplazando las nuevas líneas de código donde se estime mejor y colocando los archivos nuevos en el mismo directorio que en *2.6.29-GP*. Esto solo tiene motivos prácticos, sin pretensión de conseguir directamente el kernel final estable y funcionando. Posteriormente, se copia el código *linux-2.6.38+mpss3.8.3-mod* en el directorio de trabajo *newtest/*, con el siguiente comando:

   ```
   rodrigo# cp -r linux-2.6.38+mpss3.8.3-mod/ /home/rodrigo/mpss-3.8.3/
      src/newtests/
   ```

   A continuación, se elabora un listado de los ficheros nuevos y modificados, extraído tras buscar en el código *linux-2.6.38+mpss3.8.3-mod* los archivos comentados por J. Sanjuan.

   ```
   rodrigo# cd /home/rodrigo/mpss-3.8.3/src/newtests/linux-2.6.38+mpss3
      .8.3-mod/
   ...-mod# grep -rl "Sanjuan" * > files.txt
   ```

   Esta fue la lista obtenida:

   ```
   arch/x86/include/asm/unistd_64.h
   arch/x86/include/asm/vsyscall.h
   arch/x86/kernel/process_64.c
   arch/x86/kernel/syscall_table_32.S
   arch/x86/kernel/vmlinux.lds
   arch/x86/kernel/process_64.c
   arch/x86/kernel/vmlinux_64.lds.S
   arch/x86/kernel/vsyscall_64.c
   arch/x86/vdso/Makefile
   ```

```
10 arch/x86/vdso/vadaptive.c
11 arch/x86/vdso/vdso.lds
12 arch/x86/vdso/vdso.lds.S
13 arch/x86/vdso/vextern.h
14 fs/proc/array.c
15 include/linux/resource.h
16 include/linux/sched.h
17 kernel/define_adaptive.h
18 kernel/adaptive.h
19 kernel/Makefile
20 kernel/sched.backup
21 kernel/sched.c
22 kernel/sched.cModificado.c
23 kernel/sched_copia.c
24 kernel/sched_fair.c
25 kernel/sched_mal.c
26 kernel/sched_rt.c
27 kernel/sys.c
28 kernel/adaptive.c.old
29 kernel/adaptive.c
30 sched.c
31 usr/include/asm/unistd_64.h
32 usr/include/linux/resource.h
```

En esta lista se aprecian más ficheros (32), de los descritos en el apartado anterior, y es debido a que contiene archivos que, o bien solo tienen comentarios, o eran copias para hacer pruebas (como por ejemplo *sched_ mal.c* o *adaptive.c.old*). Tras una análisis pormenorizado de todos los ficheros de la lista, se seleccionaron un total de 23 ficheros (resaltados anteriormente en negrita).

2. Se crea una copia de *tfgkernel-0* (kernel original compilado), denominada *tfgkernel-1*, y se copia un archivo de la lista en el lugar que le corresponde. A la pregunta de sobreescribir archivo responder que sí. Este código *tfg-kernel-1* conforma la primera versión parcial hasta llegar al kernel final.

   Version parcial del kernel: se utilizará este concepto para referirse a un kernel 2.6.38 con un(os) archivo(s) modificados. La creación de versiones parciales sucesivas concluyen en la obtención del kernel final.

```
newtests# cp -r tfgkernel-0 tfgkernel-1
newtests# cp linux-2.6.38+mpss3.8.3-mod/arch/mips/kernel/mips-mt-fpaff
    .c tfgkernel-1/arch/mips/kernel/
cp sobreescribir tfgkernel-1/arch/mips/kernel/mips-mt-fpaff.c? (s/n) s
newtests# cd tfgkernel-1
tfgkernel-1#
```

3. A continuación, dirigirse a este directorio y ejecutar los pasos, ya explicados en el apartado anterior, para compilar el código. Si la compilación termina correctamente pasar al siguiente paso, si hubiera errores resolverlos e intentar compilar de nuevo.

4. Con una compilación correcta del núcleo y su imagen creada, cambiar la imagen y reiniciar el *mic0* como se ha explicado en el apartado anterior.

5. Compilar y ejecutar *hello_ world_ offload.c* (ver apartado 4.1). Por comodidad, se puede copiar en este directorio *src/*, a través de los siguientes comandos:

```
rodrigo# cp hello_world_offload.c /home/rodrigo/mpss-3.8.3/src/
   newtests/
rodrigo# cd /home/rodrigo/mpss-3.8.3/src/newtests/
newtests# . /opt/intel/parallel_studio_xe_2017.1.043/
   compilers_and_libraries_2017/linux/pkg_bin/compilervars.sh intel64
newtests# icc -qoffload hello_world_offload.c -o hello_world_offload
newtests# ./hello_world_offload
```

   Si el sistema se queda bloqueado o el programa devuelve algo que no es lo esperado significa que el kernel no es estable y hay que volver al paso 2 probando otro archivo distinto. Si por el contrario, *hello_ world_ offload* compila y se ejecuta correctamente, es muy probable que el núcleo funcione como es debido. Marcar el archivo incluido en el kernel como "incluido" o simplemente lo eliminamos de lista anterior (denominada, *files.txt*) para seguir adelante con los restantes.

Este procedimiento (del paso 1 al 5) se repite copiando tfgkernel-1 como tfgkernel-2, sustituir en este código otro archivo de la lista, compilar, cambiar imagen del núcleo y testear con *hello_ world_ offload*. Finalmente, este proceso se repite hasta tener todos los archivos sustituidos y concluye con el kernel funcionando.

Para que esta fase del TFG no constituyera una labor tan larga y tediosa se implementaron varios scripts, denominados *maketfg.sh*, *changeos.sh* y *test_ hello_ world_ offload.sh)*, para agilizar el proceso.

En nuestro caso se han llegado a tener más 70 versiones parciales de lo que sería el kernel final, de las cuales solamente las primeras se hicieron de forma manual, mientras que las siguientes, hasta finalizar la fase de adaptación, se han hecho utilizando un método mucho más práctico. El principal problema ha surgido con las dependencias entre ficheros que ha ralentizado el proceso de integración.

5.2.3. Herramientas para la adaptación del código

Estas herramientas son scripts escritos en *Bash* que automatizan los pasos para crear versiones parciales del kernel final.

- *maketfg.sh*:

  Al ejecutar este script hay que introducir obligatoriamente 3 parámetros: la ultima versión estable, la línea de *files.txt* que corresponde al archivo nuevo a añadir a esta próxima versión, y el número de versión con el que identificaremos el nuevo kernel compilado. Un ejemplo de ejecución sería:

```
newtests# ./maketfg.sh 7 1 8
```

De esta forma, se crea el directorio *tfgkernel-8*, que será una copia de *tfgkernel-7*. A *tfgkernel-8* se le añadirá el fichero de la línea 1 de *files.txt* (*arch/mips/kernel/mips-mt-fpaff.c* en este caso) y se compilará *tfgkernel-8*. Con este script se ahorra mucho tiempo al no escribir tantos comandos por consola, ya que se ejecutan secuencialmente. De esta forma se ejecutan los pasos 2 y 3 vistos en el apartado anterior.

- *changeos.sh*:

  Este script apagará el *mic0*, cambiará la imagen actual del kernel a la del kernel introducido por parámetro y reiniciará el kernel. Por ejemplo, si actualmente el kernel en funcionamiento es *tfgkernel-7*, y se ejecuta el siguiente comando:

```
newtests# ./changeos.sh 8
```

  se cambiará a la imagen del núcleo a *tfgkernel-8* y se reinicia el *mic0*. Este script corresponde al paso 4 del apartado anterior.

- *test_hello_world_offload.sh*:

  Simplemente compila y ejecuta *hello_world_offload.c*, equivalente al paso 5 descrito en el apartado anterior.

Este trabajo ha requerido la creación de estos scripts para que el proceso de construcción del kernel final se agilice, ya que se realizan muchas operaciones juntas. El procedimiento se resume en ejecutar *maketfg.sh*, si la compilación sale correcta, *changeos.sh* y por último *test_hello_world_offload.sh* para verificar que la nueva versión parcial es estable. Aunque se ha dicho que es un método más práctico que introducir los comandos uno a uno, se ha invertido mucho tiempo en esta fase debido a los errores de compilación que iban surgiendo por dependencia de datos, o por no haber editado correctamente los ficheros.

5.3. El kernel final

Para conseguir una correcta compilación del kernel final se han añadido, tal cual, los 4 ficheros nuevos (*kernel/adaptive.h*, *kernel/adaptive.c*, *kernel/define_adaptive.c* y *arch/x86/vdso/vadaptive.c*) y se han editado los otros 19 de manera muy similar a la version *2.6.29-GP*. En la mayoría de los ficheros los cambios han sido directos, consistiendo en colocar las líneas de código nuevas en su lugar equivalente en el mismo archivo del código fuente de la versión *2.6.38*, ya que este no había cambiado demasiado tras la actualización oficial de la versión 2.6.29 a la 2.6.38. Sin embargo, se han detectado tres situaciones en los que se han hecho consideraciones adicionales debidas a cambios propios de la actualización oficial del kernel. A continuación, se exponen estos 3 de los 19 archivos del kernel *2.6.29-GP* y sus diferencias con respecto a la nueva versión final.

- *include/linux/sched.h*:

  En la versión 2.6.38 oficial del kernel de linux, fue creada una estructura *struct sched_statistics* que recogía gran parte de los parámetros de *struct sched_entity*, para que luego en *struct sched_entity* se creara una instancia de *struct sched_statistics* con esos datos.

Es por ello que los parámetros que fueron añadidos a *struct sched_entity* en el kernel *2.6.29-GP* (más concrétamente, *uninterruptible_time*, *interruptible_time* y *waiting_time*) fueron añadidos a *struct sched_statistics* en el kernel *2.6.38-GP*. Sin embargo, tras obtener errores en la compilación del kernel se optó por situar estos parámetros al final de *struct task_struct*, después de los parámetros que ya se añadieron en *2.6.29-GP* en dicha estructura y por tanto en *2.6.38-GP*.

```
struct task_struct {

... // Resto de parametros de 2.6.29-GP
  unsigned long total_done_work;
  unsigned long *pointer_var_remaining_work;
  unsigned long remaining_work;
  int id_proc_acs;
  int id_thd_acs;
  bu64      uninterruptible_time;
  bu64      interruptible_time;
  bu64      waiting_time;
};
```

- *kernel/sched.c*:

Debido a los cambios en el fichero *sched.h*, tanto oficiales como propios, se ha tenido que adaptar *sched.c*, de tal forma, que esta función cambió por la actualización oficial y fue adaptada de la siguiente forma:

```
static void __sched_fork(struct task_struct *p)
{
  p->se.exec_start = 0;
  p->se.sum_exec_runtime = 0;
  p->se.prev_sum_exec_runtime = 0;
  p->se.nr_migrations = 0;
/* Nuevo para 2.6.38 */
#ifdef CONFIG_SCHEDSTATS
  memset(&p->se.statistics, 0, sizeof(p->se.statistics));
#endif
/* Hasta aqui */

#ifdef CONFIG_SCHEDSTATS
  p->se.statistics.wait_start = 0; //Antes: p->se.wait_start
  //Antes: p->se.sum_sleep_runtime
  p->se.statistics.sum_sleep_runtime = 0;
  p->interruptible_time = 0;  // Antes: p->se.interruptible_time
  p->uninterruptible_time = 0; // Antes: p->se.uninterruptible_time
  p->waiting_time =0 ;  // Antes: p->se.waiting_time
  p->detenida = false;
  p->detenible = false;
  p->detenida_ultimointervalo = false;
  p->terminada = false;
  p->total_done_work = p->done_work = p->remaining_work = 0;
  p->pointer_var_done_work = p->pointer_var_remaining_work = NULL;
  p->id_proc_acs = p->id_thd_acs = -1;

  /* Modificado por Rodrigo Espeso.
   * Estos datos que antes estaban es sched_entity
```

```
30      * han sidoi ncluidos aqui para hacer esta
31      * funcion similar a 2.6.29-GP.
32      */
33     p->se.statistics.sleep_start = 0;
34     p->se.statistics.block_start = 0;
35     p->se.statistics.sleep_max = 0;
36     p->se.statistics.block_max = 0;
37     p->se.statistics.exec_max = 0;
38     p->se.statistics.slice_max = 0;
39     p->se.statistics.wait_max = 0;
40
41   #endif
42
43     INIT_LIST_HEAD(&p->rt.run_list);
44     p->se.on_rq = 0;
45     INIT_LIST_HEAD(&p->se.group_node);
46
47   #ifdef CONFIG_PREEMPT_NOTIFIERS
48     INIT_HLIST_HEAD(&p->preempt_notifiers);
49   #endif
50
51     //p->state = TASK_RUNNING; // Eliminado en 2.6.38
52   }
```

- *kernel/sched_fair.c*:

  Al igual que en *2.6.29-GP*, aquí hay cambios en dos funciones, pero en esta versión hay algunas diferencias debido a que en 2.6.29 no existía la *struct sched_ entity*.

```
1   static void update_stats_wait_end(struct cfs_rq *cfs_rq, struct
       sched_entity *se)
2   {
3     //Agregado por Rodrigo Espeso
4     struct task_struct *tsk = NULL;
5     schedstat_set(se->statistics.wait_max, max(se->statistics.wait_max,
6         rq_of(cfs_rq)->clock - se->statistics.wait_start));
7     schedstat_set(se->statistics.wait_count, se->statistics.wait_count +
          1);
8     schedstat_set(se->statistics.wait_sum, se->statistics.wait_sum +
9         rq_of(cfs_rq)->clock - se->statistics.wait_start);
10
11    /* Modificado por Rodrigo Espeso */
12    if (entity_is_task(se)) {
13      tsk=task_of(se);
14      tsk->waiting_time +=
15        rq_of(cfs_rq)->clock-se->statistics.wait_start;
16    }
17    /* Hasta aqui.
18     *    se->waiting_time +=rq_of(cfs_rq)->clock-se->wait_start;
19     */
20
21
22  /* Nuevo en 2.6.38 */
23  #ifdef CONFIG_SCHEDSTATS
24
25    if (entity_is_task(se)) {
```

```
26      trace_sched_stat_wait(task_of(se),
27        rq_of(cfs_rq)->clock - se->statistics.wait_start);
28    }
29
30    //Agregado por Rodrigo Espeso
31    tsk->waiting_time +=rq_of(cfs_rq)->clock-se->statistics.wait_start;
32
33  #endif
34  /* Hasta aqui */
35    schedstat_set(se->statistics.wait_start, 0);
36  }
```

```
1   static void enqueue_sleeper(struct cfs_rq *cfs_rq, struct sched_entity
        *se)
2   {
3   /* Nuevo en 2.6.38 */
4   #ifdef CONFIG_SCHEDSTATS
5     struct task_struct *tsk = NULL; /* Nuevo en 2.6.38 */
6
7     if (entity_is_task(se))
8       tsk = task_of(se);
9   /* Hasta aqui */
10    if (se->statistics.sleep_start) {  // 2.6.29: se->sleep_start
11      u64 delta = rq_of(cfs_rq)->clock - se->statistics.sleep_start;
12
13      if ((s64)delta < 0)
14        delta = 0;
15
16      if (unlikely(delta >se->statistics.sleep_max))
17        se->statistics.sleep_max = delta;  //2.6.29: se->sleep_max
18
19      se->statistics.sleep_start = 0; //2.6.29: se->sleep_start
20      se->statistics.sum_sleep_runtime += delta;
21      // 2.6.29: se->sum_sleep_runtime
22      /* Modificado por Rodrigo Espeso sobre codigo de J. Sanjuan */
23      tsk->interruptible_time +=delta;
24      // 2.6.29-GP: se->interruptible_time +=delta;
25      /* Hasta aqui */
26
27      if (tsk) { // Nuevo en 2.6.38
28
29        account_scheduler_latency(tsk, delta >> 10, 1); // Nuevo en
              2.6.38
30        trace_sched_stat_sleep(tsk, delta); // Nuevo en 2.6.38
31      }
32    }
33    if (se->statistics.block_start) {  // 2.6.29: se->block_start
34      u64 delta = rq_of(cfs_rq)->clock - se->statistics.block_start;
35      //struct task_struct *tsk = task_of(se); // Eliminado.
36
37      if ((s64)delta < 0)
38        delta = 0;
39
40      if (unlikely(delta >se->statistics.block_max))
41      // 2.6.29: se->block_max
42        se->statistics.block_max = delta;
43
```

```
44      se->statistics.block_start = 0;
45      se->statistics.sum_sleep_runtime += delta;
46      // 2.6.29: se->sum_sleep_runtime
47      /* Nuevo en 2.6.38 */
48      if (tsk) {
49        if (tsk->in_iowait) {
50          se->statistics.iowait_sum += delta;
51          se->statistics.iowait_count++;
52          trace_sched_stat_iowait(tsk, delta);
53        }
54      /* Hasta aqui */
55
56      /* Modificado por Rodrigo Espeso sobre codigo de J. Sanjuan */
57      tsk->uninterruptible_time +=delta;
58      //2.6.29-GP: se->uninterruptible_time +=delta;
59      /* Hasta aqui */
60        /*
61         * Blocking time is in units of nanosecs, so shift by
62         * 20 to get a milliseconds-range estimation of the
63         * amount of time that the task spent sleeping:
64         */
65        if (unlikely(prof_on == SLEEP_PROFILING)) {
66          profile_hits(SLEEP_PROFILING,
67              (void *)get_wchan(tsk),
68              delta >> 20);
69        }
70        account_scheduler_latency(tsk, delta >> 10, 0);
71      } //Fin if(tsk)
72    }
73  #endif
74  }
```

Conforme a los cambios realizados en estos 3 archivos también se han modificado los ficheros *adaptive.h* y *adaptive.c*. Después del análisis del kernel inicial y su fase de adaptación, se ha conseguido un kernel final integrado en el *2.6.38+mpss3.8.3* que se ha constituido como el kernel sobre el que se han ejecutado los distintos experimentos realizados en el presente trabajo. Esta tarea ha sido crucial, ya que sin un núcleo compatible con la arquitectura de la Xeon Phi, no se hubiera podido utilizar el gestor de paralelismo a nivel de kernel.

Capítulo 6

Fase III: Benchmark *N-BodyIGP*

En este capítulo se explica la aplicación *N-BodyIGP*, que se ha utilizado como benchmark, con la librería *IGP* integrada. Se parte del programa *Parallel-N-Body-Simulation* [17], al que se le ha incorporado la librería junto con otros cambios para convertirla en una aplicación multihebrada adaptativa, obteniendo el benchmark que se ha denominado *N-BodyIGP* y sobre el que se han realizado todos los experimentos del presente TFG.

A continuación, se explica el funcionamiento de *Parallel-N-Body-Simulation*[17], los métodos de la librería *IGP* y su relación con las fases del GP descritas anteriormente en el apartado 2.1, y por último se muestra el funcionamiento adaptativo de la versión final *N-BodyIGP*.

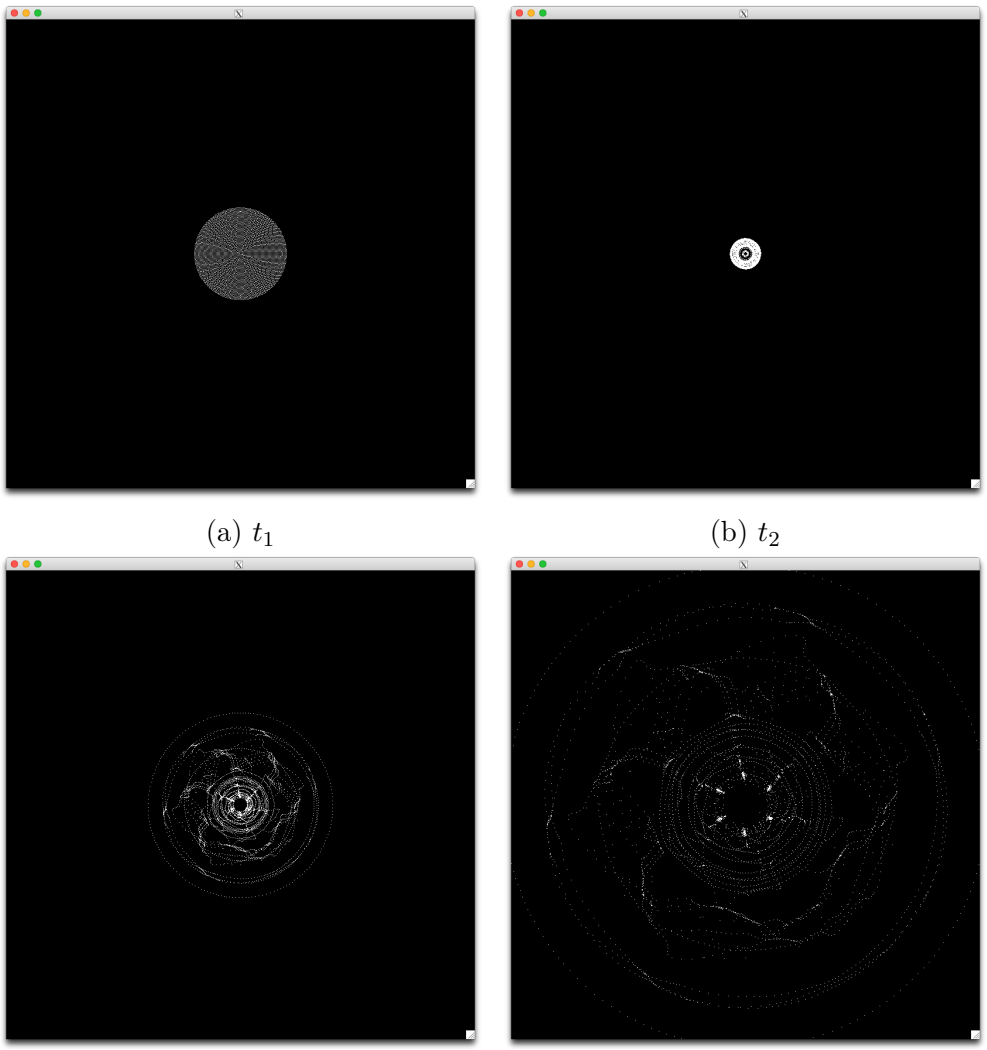
6.1. Programa *Parallel-N-Body-Simulation*

El problema de los N cuerpos trata de determinar los movimientos individuales de un grupo de partículas, o cuerpos, que interactúan mútuamente según la Ley de Gravitación Universal de Newton [10].

Datos de entrada En este programa, los N cuerpos se presentan inicialmente en posiciones *{x, y}* aleatorias de un escenario en 2D, moviéndose cada uno a una velocidad *{vx, vy}* también aleatoria, cuya masa m es la misma en todos ellos y se pretende calcular sus movimientos T veces, con un tiempo t entre cada iteración. Los datos de entrada *N, x, y, vx, vy* están recogidos en ficheros de test. Por ejemplo, en el fichero test1000.txt se almacenan los datos de 1000 cuerpos con el siguiente formato:

```
1000
-0.927726380304 0.191111912995 0.480017832135 0.18706239648
-0.749024108519 -0.270486385716 0.732610349458 0.992995512647
0.350119741719 -0.839116300913 0.89855730407 0.295953129302
0.43516995592 0.543163980272 0.296848012987 0.23176219823
-0.676579788764 -0.356005322202 0.415220142446 0.608855493978
...
```

Donde la primera línea es N y cada línea de las siguientes representa un cuerpo siendo las siguientes columnas x, y, vx y vy, respectivamente. El resto de datos de entrada necesarios se introducen como argumentos al ejecutar la aplicación. Por ejemplo, el siguiente



(a) t_1 (b) t_2

(c) t_3 (d) t_4

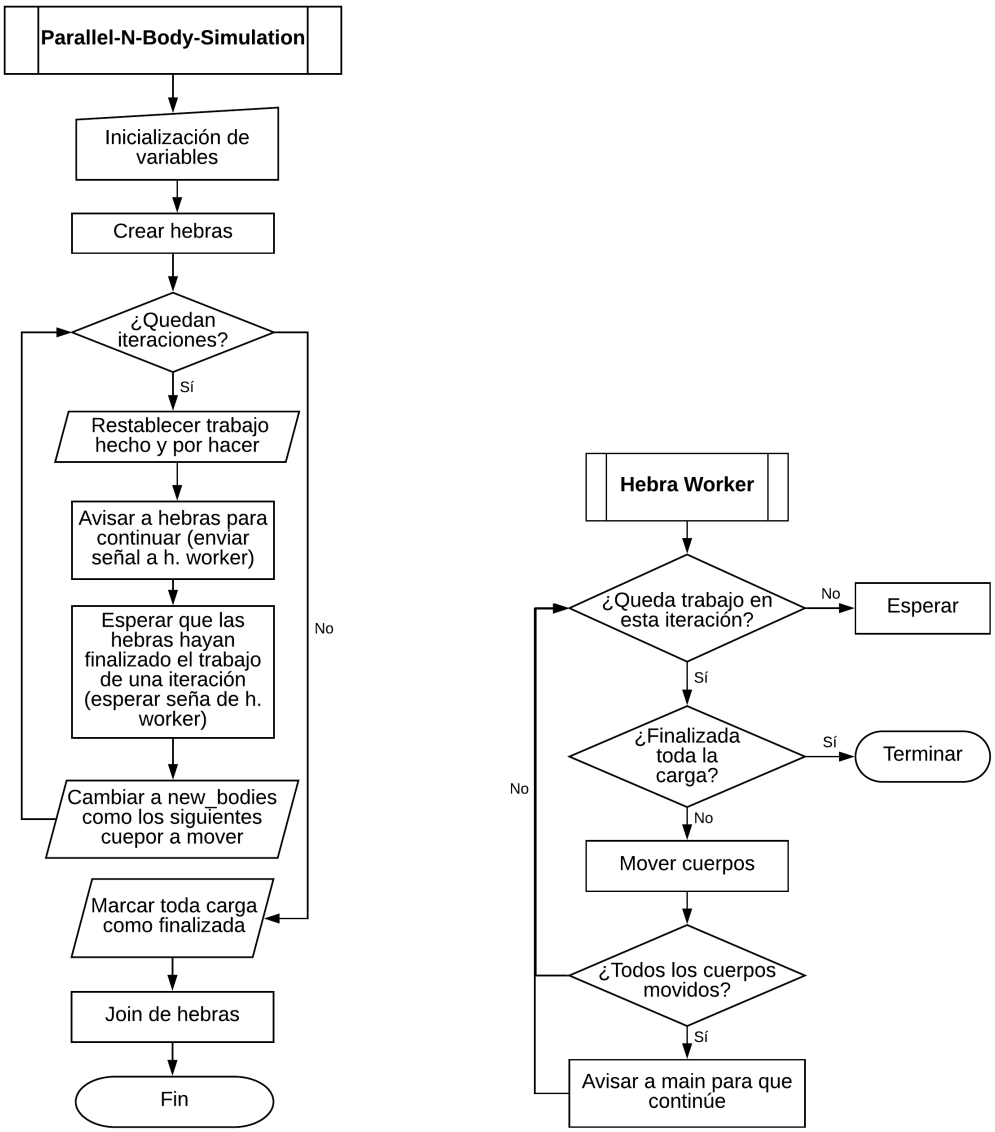
Figura 6.1: Salida de *Parallel-N-Body-Simulation*.

comando:

```
$ ./nbody_pthread 2 1000 3000 0.5 testcase/test2.txt 0.1 enable -2 -2 4 800
```

Donde los argumentos introducidos son, respectivamente: el número de hebras que utilizará la aplicación, igual a `2`, m = `1000`, T = `3000`, t = `0.5`, el archivo de test (`testcase/test2.txt` contiene 945 cuerpos), `enable` habilita la interfaz gráfica (con `disable` se inhabilita), *umbral* = `0.1`[1], `-2` y `-2` son las coordenadas de la esquina superior izquierda de la ventana de la interfaz gráfica, `4` es la longitud del eje de coordenadas y `800` la longitud de un lado de la ventana. Para este ejemplo se obtiene una animación del movimiento de 945 cuerpos, representada en la interfaz gráfica. En las figura 6.1 se muestra sucesivamente la salida en diferentes instantes de tiempo t_n, donde $t_1 < t_2 < t_3 < t_4$.

[1]Este valor es utilizado por *Parallel-N-Body-Simulation* en la versión que implementa el algoritmo *Barnes-Hut* [17]. En este caso se puede introducir cualquier valor, ya que no se utiliza.



(a) Proceso principal. (b) Función *worker*.

Figura 6.2: Diagramas de flujo de *Parallel-N-Body-Simulation*.

6.1.1. Algoritmo

El diagrama de la figura 6.2 describe el funcionamiento de la aplicación *Parallel-N-Body*. El proceso principal comienza con la lectura de argumentos e inicialización de variables, tras esto, crea las hebras que ejecutan la tarea de cómputo. Mediante un bucle que se realiza T veces, se reinicializan las variables que contabilizan el trabajo hecho y por hacer en una iteración, que se corresponden con el número de cuerpos movidos. Cuando se ha realizado el trabajo de una iteración, mover los N cuerpos, entonces la ejecución de este bucle continúa y establece que los cuerpos movidos son los siguientes que se van a mover. Finalmente, cuando se realizan las T iteraciones, finaliza la aplicación.

Por otro lado, cada hebra ejecuta el método *worker* en el cual primero comprueba si hay trabajo por hacer en esta iteración, si no lo hay la hebra espera, si por el contrario se

determina que sí queda trabajo, se comprueba si se ha finalizado toda la carga. Si se ha finalizado toda la carga de trabajo la hebra finaliza, si no, se ejecuta la función que realiza los cálculos para mover cada cuerpo. Cuando todos los cuerpos son movidos, se envía una señal al proceso principal para que desbloquee el bucle de intercambio de cuerpos. De esta manera la hebra continúa ejecutándose hasta que que se cumpla la condición de "Finalizada toda la carga".

6.2. Modificación de la librería *IGP*

IGP (Interfaz con el Gestor del nivel de Paralelismo) se creó para facilitar el diseño de aplicaciones multihebradas adaptativas, de tal forma que, proporciona funciones (en lenguaje *C*) para comunicar los algoritmos multihebrados con el GP seleccionado. Esta librería (versión 2.0) pretende simplificar el trabajo del programador de aplicaciones multihebradas ayudándole en la configuración y manejo del gestor, de tal forma que sea transparente al tipo de gestor utilizado.

1. Fase de *Inicialización*:

   - *IGP_Initialice()*: Esta función lee la configuración de un fichero llamado *IGP_config* y permite al proceso principal de una aplicación multihebrada solicitar al GP ser gestionado. Los distintos argumentos permiten configurar la gestión de la aplicación multihebrada:

     a) *Method*: Indica el tipo de GP escogido para la gestión de las hebras. Los valores permitidos oscilan en el rango de 0 a 7, entre los que podríamos destacar el valor 0 para seleccionar el método no adaptable, los valores del 1 al 3 son para gestores a nivel de usuario, por ejemplo, *ACW* o *AST*, y finalmente el rango de 4 a 7 están reservados para gestores a nivel de kernel, tales como *KST*, *KITST* y *SST*. Se han reservado los valores 3 y 7, en el nivel de usuario y kernel respectivamente, para implementaciones futuras.

     b) *Criterion*: Permite seleccionar el criterio de decisión que utilizará el GP escogido. Los valores permitidos oscilan en el rango entre 0 y 22, cuyos valores corresponden a diferentes criterios, tales como, número de procesadores ociosos, tiempo que duermen las hebras, tiempo interrumpible y no interrumpible, etc...

     c) *Period*: Establece el intervalo de tiempo que debe transcurrir entre dos ejecuciones consecutivas de la fase de *Evaluación* del GP, por ejemplo, 0.1 segundos.

     d) *Threshold*: Establece el umbral mínimo de eficiencia que debe obtener la aplicación multihebrada cuando es gestionada por ACW. Por ejemplo, 0.9 corresponde al 90 % de eficiencia.

     e) *Detention*: Este flag permite configurar el gestor de hebras para habilitar o no la detención temporal de las hebras.

     f) *ProcessIsAThread*: Si es *true*, se indica al gestor que el proceso principal crea las hebras. Si es *false* no es el proceso principal quien creará las hebras.

     g) *Maxload*: Si es *false* las hebras más cargadas no serán las que se dividan. Sí lo serán si es *true*.

- *IGP_ Begin_ Thread(IdThread, Workload)*: Esta función permite a cada una de las hebras de la aplicación informar al gestor que se acaba de crear una hebra con la carga de trabajo (*Workload*), y por lo tanto, esta hebra tiene que ser gestionada por el GP. Esta función debe ser ejecutada al inicio de todas las hebras de la aplicación que se desean gestionar.

2. Fase de *Información* y *Notificación*:

   - *new thread = IGP_ Get(IdThread)*: Esta función permite al gestor informar a la hebra de la decisión alcanzada en la fase de *Evaluación*: dividirse, detenerse o continuar ejecutándose fase de *Notificación*. Esta función debe ser ejecutada por cada hebra una vez completada una unidad de trabajo, por ejemplo, en cada iteración del bucle computacional.
   - *IGP_ Report(IdThread,Workdone)*: Esta función permite a cada hebra informar al gestor sobre la carga de trabajo realizada (*Workdone*) desde la última comunicación (fase de *Información*).

3. Fase de *Terminación*:

   - *IGP_ Finalize()*: El proceso principal de la aplicación multihebrada informa al gestor que va a terminar su ejecución y por lo tanto el GP seleccionado dejará de gestionar esta aplicación. Esta función se debe ejecutar una vez que todas las hebras de la aplicación han completado la carga de trabajo asignada.
   - *IGP_ End_ Thread(IdThread)*: Cada una de las hebras gestionadas debe ejecutar esta función antes de terminar, para indicarle al GP que ha completado toda la carga de trabajo asignada, y por lo tanto, desea no ser gestionada, pues va a terminar su ejecución.

IGP2.1 Sin embargo, en este TFG se han hecho algunas modificaciones sobre IGP2.0 para que funcione mejor con los gestores a nivel de kernel. A esta última librería editada, que será la utilizada en los experimentos, se ha denominado IGP2.1.

- *IGP_ Initialice()*: Cuando el gestor seleccionado es a nivel de kernel, lo que hace esta función es realizar la llamada al sistema *sys_ warns_ status(Method,Criterion, Detention,MaxThreads,ProcessIsAThread,Maxload)*. Han sido añadidos los flags *ProcessIsAThread* y *Maxload*.

- *IGP_ Get(IdThread,Workpend)* y *IGP_ Report(IdThread,Workdone,Workpend)*: Ha sido añadido el parámetro *Workpend* a ambas funciones, que representa el trabajo total del programa que queda por hacer, ya que el gestor a nivel de kernel necesita este valor para la llamada al sistema *sys_ get_ status(Workdone,Workpen,DifTime,use)*. Como esta llamada se realiza desde ambos métodos, el parámetro *use* también se ha añadido y distingue, dentro la llamada, si es realizada desde *IGP_ Get(IdTread, Workpend)* (*use = 0*) o desde *IGP_ Report(IdThread,Workdone,Workpend)* (*use = 1*) para limitar el código que se ejecuta en el kernel en ambos casos.

- *IGP_ Begin_ Thread(IdThread,Workload)*: Cuando se utiliza el gestor a nivel de kernel, faltaba por especificar que al notificar al gestor de que se creaba una hebra, su *id* debería ser marcado como reservado (ninguna hebra puede utilizarlo) y activo (representa una hebra en ejecución).

- *IGP_ End_ Thread(IdThread)*: Al igual que en *IGP_ Begin_ Thread(IdThread, Workload)*, faltaba especificar lo opuesto: el *id* de esta hebra que finalizaba debería ser marcado como no reservado (para poder asignarselo a otra hebra nueva) e inactivo (hebra que no está en ejecución).

Criterios de decisión

Como ya se ha comentado anteriormente (ver apartado 3.1), el GP depende de los criterios de decisión para que se comporte de una manera o de otra. Cada criterio especifica qué información se valora para realizar la fase de *Evaluación*. Por ello, la ejecución de una aplicación adaptativa con un criterio puede diferir de la ejecución de la misma aplicación con los mismos valores de entrada pero con un criterio diferente.

De los 21 criterios desarrollados en [8] que se recogen en el fichero *define_ adaptive.c*, en este trabajo se han utilizado los siguientes:

- **NON_ADAPTABLE.** Con este criterio el GP decide crear hebras siempre que el número de hebras activas sea inferior al número máximo de hebras establecido por el usuario.
- **NO_SLEEPTHREAD.** Este criterio permite crear una nueva hebra, si ninguna de las hebras activas se han detenido en el último intervalo de tiempo analizado.
- **NO_IBT.** Este criterio es idéntico al criterio *NO_ SLEEPTHREAD*, salvo que solo tiene en cuenta las detenciones producidas por las secciones críticas y operaciones de entrada/salida.
- **SLEEPTIMEvsINTERVAL.** Este criterio busca la hebra que mayor tiempo ha estado detenida y estima el número de hebras activas comparando el intervalo de tiempo desde el último análisis con el mayor tiempo de detención de esa hebra. De esta forma, el GP permite crear hebras si el número de hebras activas es inferior al numero de hebras estimado por este criterio.
- **MNT_IBTvsINTERVAL.** Este criterio busca la hebra que más tiempo ha estado detenida por secciones críticas y/o operaciones de entrada/salida, de tal forma, que estima el número de hebras activas comparando el tiempo de detección de esa hebra con el intervalo de tiempo desde el último análisis. De esta forma, el GP permite crear hebras si el número de hebras activas es inferior al numero de hebras estimado por este criterio.

6.3. *N-BodyPhiIGP*

El código original *Parallel-N-Body-Simulation* fue editado para incorporar *IGP*. Para que el programa funcionara de forma adaptativa han sido necesarios otros cambios además de añadir las llamadas a los métodos de la librería.

- *División de carga de trabajo*. En la versión original, se crean primero todas las hebras deseadas y estas trabajan compartiendo el *TrabajoTotal* mientras que ahora cada hebra hará un bloque de trabajo. Así que, el primer cambio que se hizo en el programa fue dividir la carga de trabajo en bloques, por lo que se crea la variable

Nchunks que almacena el número de bloques de trabajo. La aplicación creará una hebra por cada bloque de trabajo, pero cada hebra ya tiene definido una porción de trabajo.
Para implementar la división de carga de trabajo, se crea el vector *ChunkWorkloads[Nchunks]*, que al inicio del programa almacenará las cantidades de trabajo a repartir. Así, por ejemplo, *T = 15 N = 100* y *Nchunks = 3*, y sabemos que *TrabajoTotal = iters * num_body = 1500*, así que *ChunkWorkloads[Nchunks] = [500 500 500]*. Esto quiere decir que cada bloque de trabajo tiene un trabajo = 500 y, por lo tanto, serán creadas 3 hebras que realizarán cada hebra una carga de trabajo igual a 500. De esta manera, se ha sustituído en el código original la variable que almacenaba el número de hebras a ejecutar (*NThreads*) por la variable *Nchunks*.

- *No interfaz gráfica*. Se ha eliminado el código relativo a la interfaz gráfica.
- *Estructura con parámetros para las hebras*. Cada hebra tiene que crearse y recibir 2 datos: el *id* que identificará esa hebra en el gestor, y su bloque de trabajo. Esta estructura se denomina *WorkerInfo* y contiene como miembros *idIgp* y *ChunkWork*.
- Variables *TotalWokdone* y *TotalWorkPend*. Almacenan el trabajo realizado y pendiente totales por la aplicación, respectívamente. Son necesarios debido al que el GP a nivel de kernel requiere estos datos.
- *Creación de la funcion Switch_bodies y la hebra switching*. Esta es una diferencia importante con respecto a la versión original. Se ha separado el código encargado de hacer el proceso de intercambiar los vectores que almacenaban los cuerpos a mover y los movidos. Los movidos pasarían a ser los que hay que mover en la siguiente iteración.
- *Condición Allfinnished*. Se utiliza para que al final del *main* comprobar que se haya realizado todo el trabajo y, en caso contrario, esperar a que finalicen todas las hebras.
- *Métodos IGP*. Se han añadido estos métodos en sitios específicos del código.
- *Ejecutable*. Como se ha modificado la aplicación para no hacer uso de la interfaz gráfica, no es necesario introducir los parámetros relacionados con esta. Además, tanto el argumento *m* y como el *t* tampoco son necesarios introducir los ya que se ha programado para que *m = 100* y *t = 0,1*. De esta forma para ejecutar *N-BodyIGP* habría que ejecutar el siguiente comando:

```
$ ./nbody_pthread <num-bloques-trabajo> <iteraciones> <fichero-test>
```

El algoritmo 6.1 muestra el código que ejecuta el proceso principal de *N-BodyIGP* diseñado en este TFG, donde se puede observar que junto con inicialización de variables (línea 2) se ha incluido el reparto del trabajo entre los bloques de trabajo. Se inicia el gestor (línea 5) antes de proceder a crear las hebras. El bucle finalizará cuando se hayan creado todas las hebras, una por bloque de trabajo. Dentro del bucle se ejecuta la función *IGP_Get(0, totalWorkpend)* (el valor 0 identifica al proceso principal) que devuelve *1* si la fase de *Evaluación* ha concluido positiva (crear hebras), *0* en caso de escoger detener hebras o *-1* si determina no hacer nada. En caso afirmativo se asignan los parámetros

Algoritmo 6.1 Proceso principal de N-BodyIGP.

```
 1: function Main(args)
 2:     InitEnvironment()                    ▷ Inicializa variables y reparte el trabajo.
 3:     Gmm = G * mass * mass;
 4:     Crear_hebra(&switching);
 5:     IGP_Initialice();
 6:     param_worker =reservaMemoria();      ▷ Parametros que recibe cada hebra
 7:     i = 0;
 8:     while (createdThreads < nChunks) do
 9:         idGivenIGP = IGP_Get(0,totalWorkpend);
10:         if idGivenIGP > 0 then
11:             param_worker-> id _igp = createdThreads + 1;
12:             param_worker-> chunk_work = chunkWorkloads[i];
13:             Crear_hebra(workers[i], Worker, param_worker);
14:             mutex_lock(&creating_thd);
15:             createdThreads++; i++;
16:             mutex_unlock(&creating_thd);
17:         end if
18:     end while
19:     if finnish == false then
20:         cond_wait(&allfinnished, &queuing);
21:     end if
22:     for (i = 0 → createdThreads) do
23:         join(workers[i]);
24:     end for
25:     IGP_Finalice();
26: end function
```

que recibirá la hebra a crear (línea 11 y 12). Hay una sección crítica en la línea 15 para hacer que el proceso principal espere o hasta que la hebra haya finalizado su creación (ver algoritmo 6.2). Al terminar el bucle, el proceso principal deberá esperar a que las hebras en ejecución finalicen su trabajo (líneas 19 y 20). Finalmente, tras hacer el join, se ordena al gestor que finalice su monitorización.

Para la función *Worker* del algoritmo 6.2 los cambios son estríctamente los relativos a la integración de *IGP*. Al principio hay una sección crítica que finaliza cuando los parámetros de entrada han sido copiados en otras variables. Los parámetros de entrada de esta función cambian para ser asignados a otra hebra durante la ejecución del *while* del proceso principal, pero si no se espera a que estén realmente asignados, pueden alterarse los de la hebra que "aún se está creando". En lugar de ejecutar un bucle infinito, cada hebra efectuará trabajo hasta cumplir con su límite (línea 8). Se ha añadido una variable local *workdone*, que cuenta el trabajo realizado por la hebra (línea 4), los contadores de *totalWorkpend* y *totalWorkdone* (líneas 21 y 22) y, lo más importante, se han añadido los métodos de *IGP* donde deben estar (líneas 7, 23 y 29). *IGP_Begin_Thread(id, workload)* al principio del método, al crearse la hebra; *IGP_Report(id, 1, totalWorkpend)* tras hacer un cómputo de trabajo; e *IGP_End_Thread(id)* al finalizar el trabajo de la hebra, que da paso a su terminación. El parámetro *id* en cada uno de estos últimos métodos identifica

Algoritmo 6.2 Método que ejecuta cada hebra, integrando *IGP*.

```
1: function Worker(param)
2:     mutex_lock(&creating_thd);                          ▷ Sección critica
3:     id = param->id_igp;
4:     workdone = 0;                                       ▷ Trabajo hecho por esta hebra
5:     workload = param->chunk_work;                       ▷ Trabajo que ha de hacer esta hebra
6:     mutex_unlock(&creating_thd);
7:     IGP_Begin_Thread(id, workload);                     ▷ GP: se inicia una hebra con
8:     while workdone < workload do
9:         mutex_lock (&queuing);
10:        while (!finnish && queuing_jobs <= 0) do
11:            cond_wait (&processing, &queuing);          ▷ La hebra espera sin trabajo
12:        end while
13:        i = --queuing_jobs;
14:        mutex_unlock (&queuing);
15:        if finnish then Break;
16:        end if
17:        Mover_cuerpos (i);
18:        workdone++;
19:        mutex_lock (&queuing);
20:        num_done++;                                     ▷ Se actualiza el trabajo hecho
21:        totalWorkpend--;
22:        totalWorkdone++;
23:        IGP_Report(id, 1, totalWorkpend)                ▷ GP: hebra ha hecho 1 trabajo
24:        if num_done >= num_body then
25:            cond_signal (&iter_fin);                    ▷ Los N cuerpos han sido movidos
26:        end if
27:        mutex_unlock(&queuing);
28:    end while
29:    IGP_End_Thread(id);                                 ▷ GP: esta hebra ha finalizado
30: end function
```

Algoritmo 6.3 Método (nuevo) para intercambio de cuerpos.

```
1: function Switch_bodies(param)
2:     for (i = 0 → iters) do
3:         mutex_lock(&queuing);
4:         queuing_jobs = num_body;
5:         num_done = 0;
6:         cond_broadcast (&processing);
7:         cond_wait (&iter_fin, &queuing);
8:         mutex _unlock (&queuing);
9:         Body t = new_bodies;                            ▷ Actualización de los cuerpos
10:        new_bodies = bodies;
11:        bodies = t;
12:    end for
13:    finnish = true;
14: end function
```

qué hebra los llama.

La función *Switch_ bodies* (algoritmo 6.3) que se ejecutará paralelamente en otra hebra (línea 4) no gestionada por el gestor, se encarga de intercambiar T (o *iters*) veces los vectores que almacenan los cuerpos a mover y los movidos, de manera que, los cuerpos movidos pasan a ser los cuerpos que hay que mover.

Interacción de N-BodyIGP con GP a nivel de kernel

En la figura 6.3 se muestra el funcionamiento del GP a nivel de kernel junto con *IGP* y el benchmark *N-BodyIGP*. Las figuras rojas representan funciones a nivel de kernel y llamadas al sistema, pertenecientes al fichero *adaptive.c* del código fuente modificado del kernel *2.6.38-GP*, mientras que las grises son funciones a nivel de usuario. Las hebras, como objetos paralelos, se representan en amarillo.

Cuando el proceso principal ejecuta la función *IGP_ Initialice*, ésta realiza una llamada al sistema a la función *sys_ wanrs_ status* que inicializa el GP a nivel de kernel y devuelve el identificador de la aplicación gestionada. En el proceso principal, en cada iteración del bucle en el que se crean las hebras, se ejecuta la función *IGP_ Get*, y ésta a su vez ejecuta la llamada al sistema de la función *sys_ get_ status* que devolverá la elección del GP durante la fase de *Evaluación*. En caso de "elegir crear hebra", el proceso principal creará una hebra que efectuará el cómputo de su bloque de trabajo asignado. La hebra notifica al gestor que ha sido creada mediante *IGP_ Begin_ Thread*, que efectuará la llamada al sistema de la función *sys_ new_ thread_ created*. Tras realizar trabajo, cada hebra notifica al GP mediante *IGP_ Report*, que efectúa la llamada *sys_ get_ status*. Cuando la hebra ha terminado su trabajo, lo último que ejecuta es la función *IGP_ End_ Thread*, y ésta a su vez realiza la llamada al sistema *sys_ end_ thread* para notificar al gestor que ya no necesita ser monitorizada y va a terminar su ejecución. Finalmente, cuando todas las hebras finalizan su trabajo, el proceso principal informa al GP mediante *IGP_ Finalice* antes de terminar su ejecución y esta función realiza la llamada al sistema *sys_ end_ thread* notificando que el termina el gestor.

6.4. Conclusión

Tras la integración de *IGP*, *N-BodyIGP* es en su esencia el mismo programa que el original pero gracias a estos pequeños cambios se consigue que esta aplicación pueda hacer uso del gestor del nivel de paralelismo.

En general, cualquier aplicación multihebrada puede utilizar el GP a nivel de kernel siguiendo este proceso de adaptación que, como se acaba de ver, tiene unas pautas bien definidas que se reducen a la colocación de los métodos en el lugar apropiado, junto a modificaciones requeridas. Precísamente en este caso de *Parallel-N-Body-Simulation*, debido a su diseño, se han tenido algunas consideraciones extra porque así lo requería el trabajo: la división del trabajo total en bloques, la creación del método *Switch_ bodies* y las precauciones con los espacios de memoria han sido las más importantes.

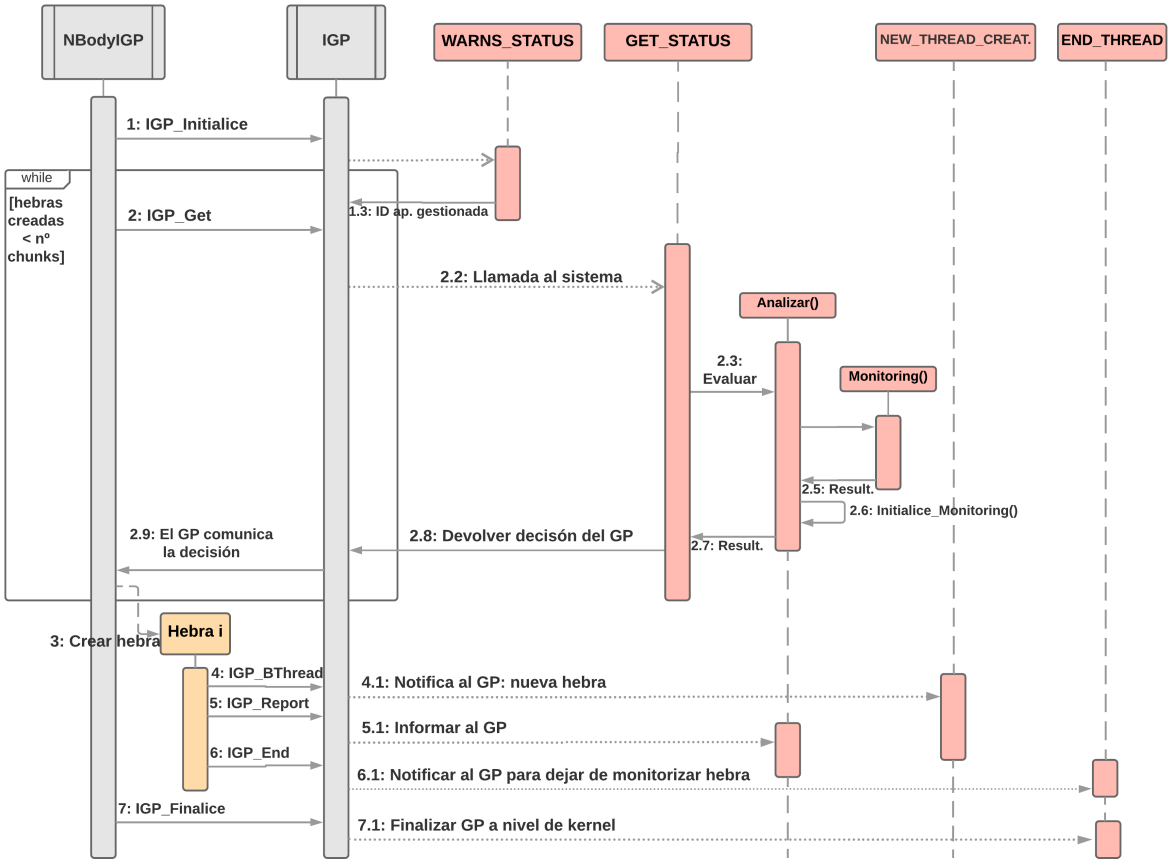


Figura 6.3: Flujo del GP a nivel de kernel: *N-BodyIGP*, *IGP* y *adaptive.c*.

Capítulo 7

Fase IV: Analizar el comportamiento del gestor del nivel de paralelismo sobre el benchmark

7.1. Métricas

Tras integrar la librería *IGP* se han realizado varios experimentos con la aplicación, que ha permitido obtener tiempos de ejecución, *speed-up* y *eficiencia* con diferentes números de hebras en ejecución y para diferentes cargas de trabajo. El rendimiento de una aplicación multihebrada se calcula comparando el tiempo de ejecución con distinto número de hebras, tal y como muestra la ecuación 7.1:

$$S_a(n) = \frac{T(1)}{T(n)} \tag{7.1}$$

Donde, $S_a(n)$ es el rendimiento, o *speed-up*, de la aplicación. Es la relación entre el tiempo real de ejecución de la aplicación ejecutada con 1 hebra y el tiempo real de ejecución de la aplicación con n hebras.

Mientras que la ecuación 7.2 muestra $E_a(n)$ que define la eficiencia de una aplicación. De tal forma, que se puede considerar que un número óptimo de hebras debe ser aquel que proporcione un nivel de eficiencia próximo a la unidad.

$$E_a(n) = \frac{S_a(n)}{n} \tag{7.2}$$

Los experimentos de *N-BodyIGP* sobre el equipo Xeon Phi, se han realizado utilizando una carga de trabajo específica, que consiste en $T = 1.000$ iteraciones y $N = 40.000$ cuerpos de tal forma, que la aplicación realiza 1.000 movimientos de 40.000 cuerpos. Además, se han testado diferentes criterios de decisión del GP a nivel de kernel.
Como se aprecia en la figura 7.1, se han medido los tiempos de ejecución de la aplicación *N-BodyIGP* en su versión *no adaptable*, es decir, el GP siempre decide crear hebras tras las fase de *Evaluación*, para lo cual se ha ejecutado la aplicación utilizando el criterio *KERNEL_NONADAPTABLE*. Cabe destacar que, en el eje de abscisas se representa el número máximo de hebras en ejecución de la aplicación, que coincide con el número de bloques de trabajo en el que se divide la carga total, asignando a cada hebra un bloque de

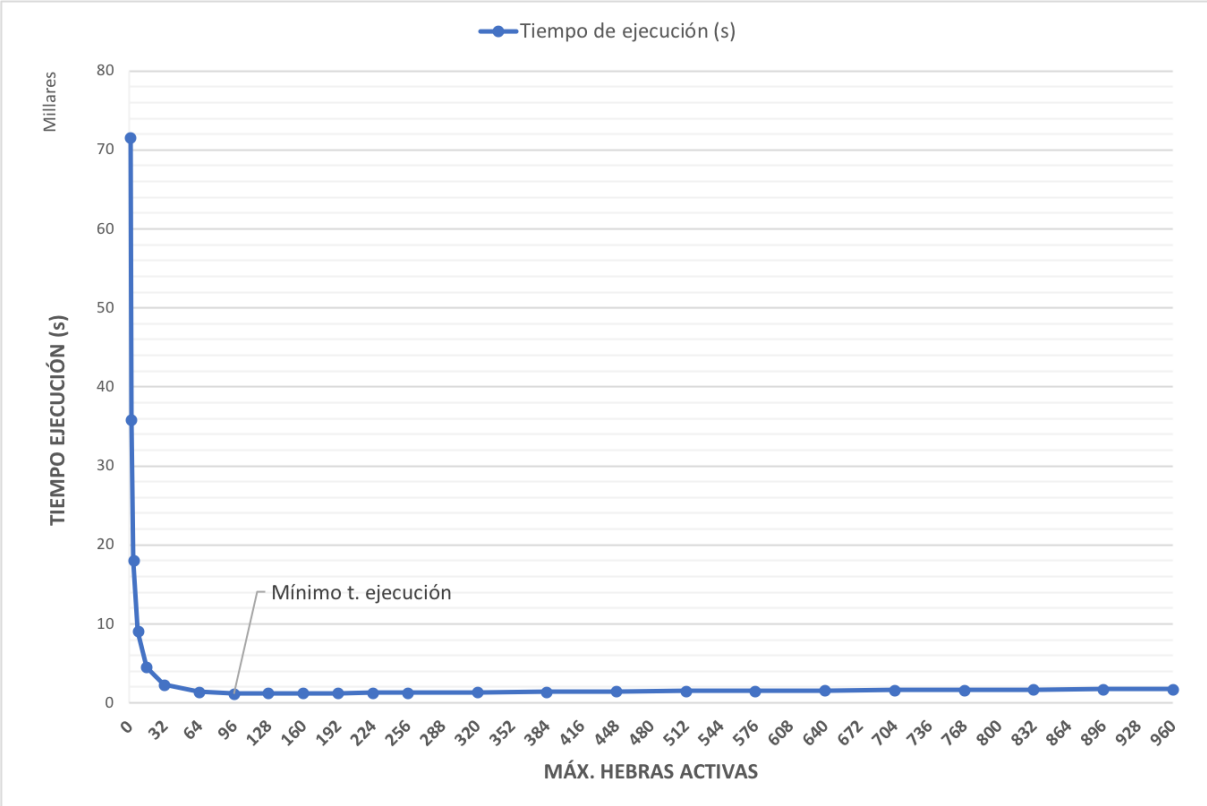


Figura 7.1: Tiempos de ejecución de *N-BodyIGP* con ejecución no adaptativa.

trabajo. En la figura 7.1 se muestra cómo los tiempos de ejecución disminuyen hasta llegar a un número de hebras activas (entorno a las 96 hebras) en el que llega a su valor más bajo. A partir de ese valor, el tiempo de ejecución va aumentando conforme aumentan el número de hebras activas, como se puede observar en figura 7.2. Este aumento en el tiempo de ejecución es debido a la sobrecarga del paralelismo producida por un gran número de hebras activas, sin embargo, este crecimiento es despreciable si se compara con la reducción en el tiempo de ejecución conseguida al ejecutar la aplicación de forma paralela.

A continuación, se muestra en la figura 7.3 los valores de rendimiento (o *speed-up*) obtenidos al ejecutar *N-BodyIGP* de manera *no adaptable*. Se puede apreciar como a partir del valor de tiempo mínimo indicado anteriormente, el *speed-up* deja de crecer debido a que los tiempos de ejecución son muy similares entorno a 96 hebras simultáneas, e incluso desciende debido a la sobrecarga del paralelismo.

La figura 7.4 muestra los valores de rendimiento de algunos criterios adaptativos frente a la ejecución no adaptativa de *N-BodyIGP* para la misma carga de trabajo dividida en 960 bloques. Al ejecutar *N-BodyIGP* con cada criterio adaptativo, este establece un número máximo de hebras activas inferior al número de bloques propuesto (en este caso, 960 bloques). Se puede observar cómo las ejecuciones adaptativas, con criterios *NO_IBT*, *SLEEPTIMEvsINTERVAL* y *MNT_IBTvsINTERVAL*, obtienen valores de rendimiento inferiores al óptimo al establecido por 96 hebras con *KERNEL_NONADAPTABLE*, pero mejorando la ejecución no adaptativa, ya que esteblece menos cantidad de hebras al especificar 960 bloques de trabajo que en el criterio *KERNEL_NONADAPTABLE*, que

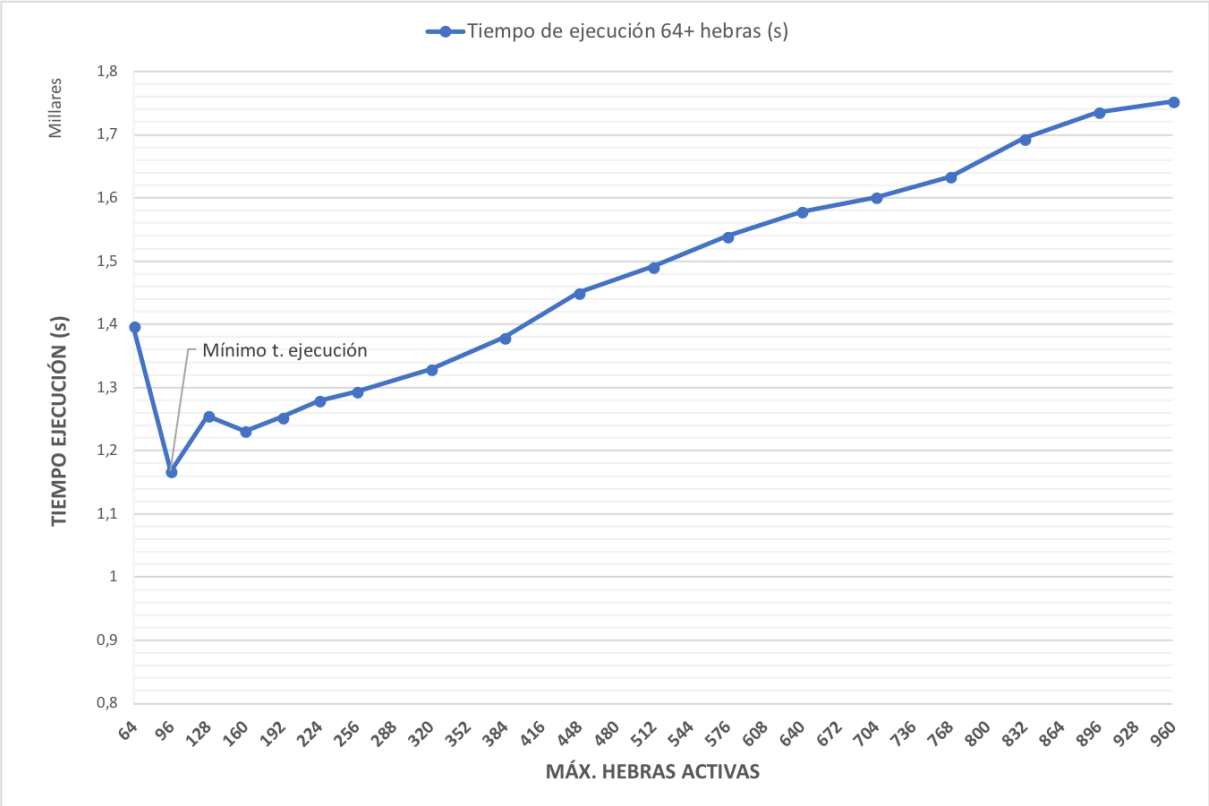


Figura 7.2: Tiempos de ejecución de *N-BodyIGP* con ejecución no adaptativa a partir de 64 hebras activas.

trabaja con las 960 hebras. Además, se consiguen unos tiempos de ejecución similares al óptimo (ver tabla 7.1). Sin embargo, también se ha detectado que la ejecución adaptativa con el criterio *NO_SLEEPTHREAD* tiene un comportamiento similar a la ejecución no adaptativa. El análisis de este comportamiento deberá ser estudiado en un trabajo futuro a través de otros experimentos con diferentes aplicaciones.

Criterio	**Máx. hebras activas**	**Tiempo de ejecución (s)**	**Speed-Up**	**Eficiencia**	**Incremento**
No Adaptable (960 bloques)	960	1.752,59	40,84	0,04	50,08 %
No Adaptable (96 bloques)	**96**	**1.167,71**	**61,29**	**0,64**	—
NO_SLEEP THREAD	960	1.760,82	40,65	0,04	50,08 %
NO_IBT	516	1.360,03	52,62	0,10	16,47 %
SLEEPTIMEvs INTERVAL	554	1.381,14	51,82	0,09	18,28 %
MNT_IBTvs INTERVAL	480	1.389,95	51,49	0,11	19,03 %

Tabla 7.1: Comparación de ejecución no adaptativa y adaptativa de *N-BodyIGP*.

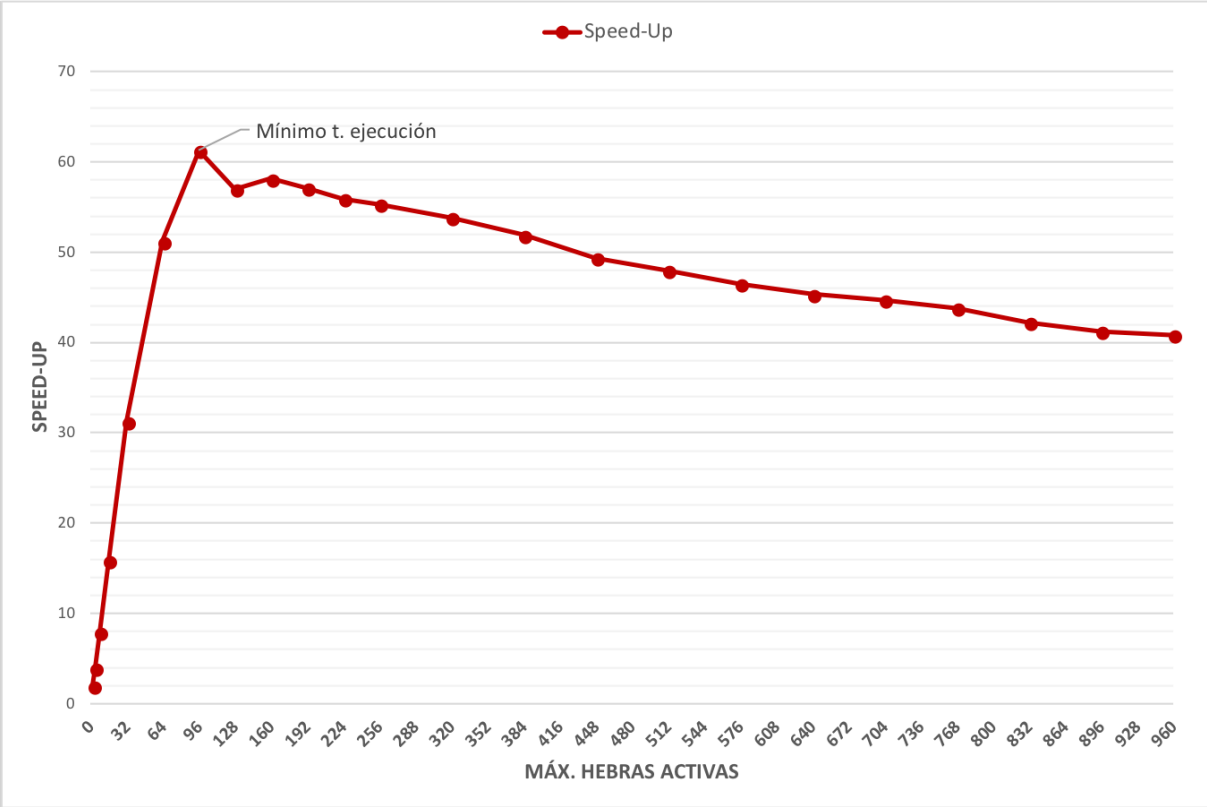


Figura 7.3: *Speed-Up* de *N-BodyIGP* con ejecución no adaptativa.

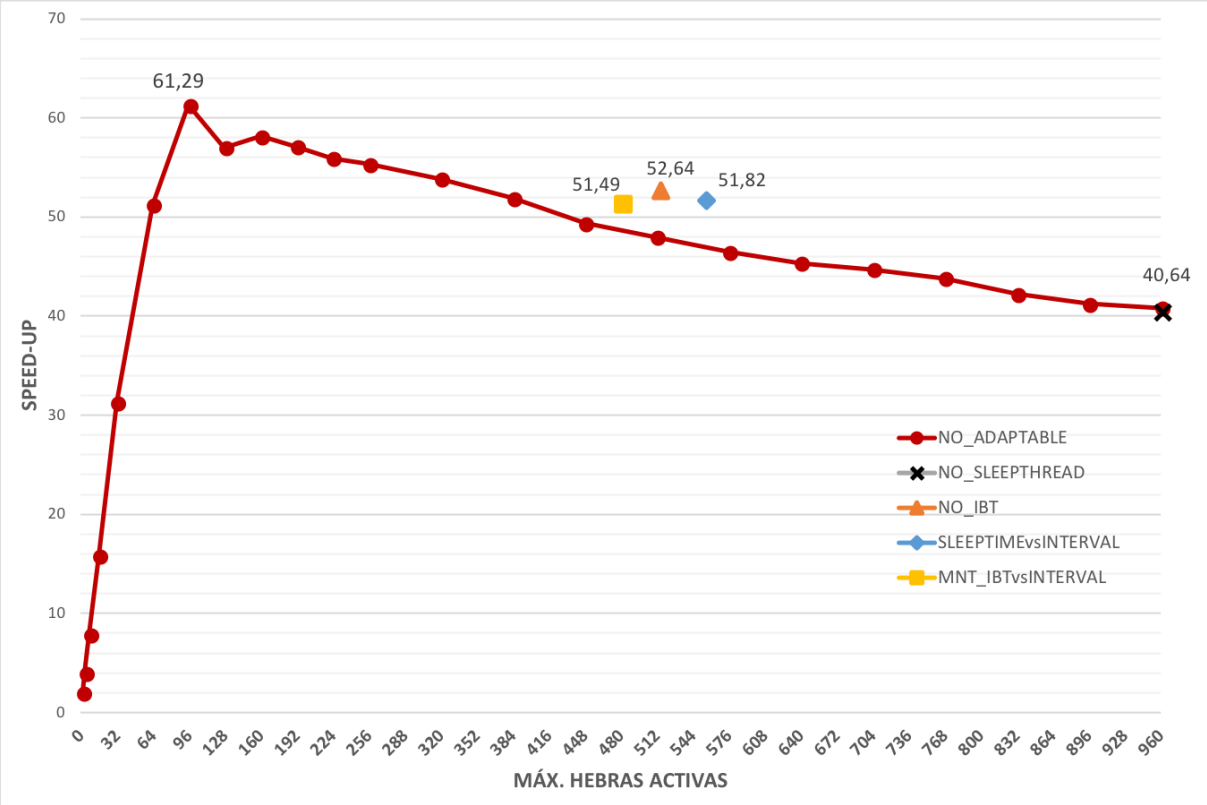


Figura 7.4: *Speed-Up* de *N-BodyIGP* con ejecución no adaptativa.

Capítulo 8

Fase V: Difusión del trabajo

Para la difusión de este TFG se ha creado un repositorio en *GitHub*[1] donde están alojados esta memoria y el código utilizado para el trabajo: el código fuente del kernel final, los scripts creados, el código fuente del programa *N-BodyIGP* y la librería *IGP*, todo comentado y documentado.

También, se proporciona un manual, elaborado con material de la memoria, para la modificación y compilación del kernel, así como la integración de *IGP* a las aplicaciones multihebradas.

Finalmente, en el repositorio se encuentra una lista con objetivos a realizar que forman parte de las mejoras posibles descritas en el capítulo 9, para que la comunidad pueda colaborar en este proyecto. Para facilitar la comunicación se facilita un correo electrónico[2] para dudas y propuestas.

Se debe recordar que todo el material disponible estará protegido por una licencia *GNU General Public License v3.0*. El material que no es propio, como *Parallel-N-Body-Simulation* o todo el material relacionado con el GP, está indicado como tal, respetando y señalando la propiedad de su autor.

[1] https://github.com/rodrigoespeso/gp
[2] rec790@inlumine.ual.es

Capítulo 9

Conclusión y posibles mejoras

En este TFG se ha continuado el estudio sobre los gestores de nivel de paralelismo desarrollados en [8], enfocándose en el GP a nivel de kernel. Para ello, se ha utilizado un máquina con arquitectura diferente a la utilizada en los experimentos realizados en [8] y con otra versión del kernel de Linux. El presente trabajo se ha elaborado sobre el coprocesador Intel Xeon Phi, perteneciente a la arquitectura *Intel MIC*, que ejecuta una pila de software (MPSS) sobre un kernel versión *linux-2.6.38+mpss3.8.3*. Tras un previo proceso de familiarización con el equipo y actualización del MPSS, se ha modificado el código fuente del núcleo Linux para poder hacer uso del GP a nivel de kernel. Para realizar las medidas de rendimiento y eficiencia del equipo Xeon Phi se ha elaborado, a partir de [17], la aplicación *N-BodyIGP*, integrando la librería *IGP* para poder ejecutar la aplicación de forma adaptativa.

9.1. Trabajo futuro y mejoras

A continuación se proponen diversas áreas de mejora para este trabajo, así como el enfoque de posibles implementaciones para mejorar la funcionalidad del GP en el futuro.

9.1.1. Exploración y experimentación

Para obtener una mejor base de conocimiento acerca del uso del GP, se propone experimentar con otras aplicaciones que implementen algoritmos distintos[1], diferentes condiciones en cuanto a memoria, secciones críticas, etc. Un ejemplo sería integrar *IGP* en una aplicación que utilice el algoritmo *Branch and Bound*, ya que se ha estudiado en [8] que estas aplicaciones obtienen un buen incremento en el nivel de rendimiento y eficiencia al utilizar el GP. También se propone la utilización del GP junto a otras tecnologías de programación multihebrada, como por ejemplo *OpenMP*[2] o *TBB (Intel Threading Building Blocks)*.

En este trabajo se ha experimentado con un sistema no dedicado pero en condiciones de un sistema dedicado, es decir, cuando se ha ejecutado *N-BodyIGP* para realizar las métricas, ha sido de forma aislada, solo una instancia de la aplicación y sin más programas ejecutándose. Se deja como trabajo futuro la experimentación del GP en sistemas no

[1] El código fuente de *Parallel-N-Body-Simulation* [17] contiene una versión del mismo programa implementando el algoritmo *Barnes-Hut*.

[2] El código fuente de *Parallel-N-Body-Simulation* [17] contiene una versión utilizando *OpenMP*.

dedicados, de manera que conviva la ejecución del programa de benchmark junto con otros programas diferentes o varias instancias del benchmar, para comprobar cómo la compartición de recursos afecta al rendimiento del GP.

9.1.2. Implementación de otros GPs a nivel de kernel y otros criterios adaptativos

Durante este trabajo se ha utilizado el GP a nivel *KST* basado en llamadas al sistema, pero en [8] se desarrollaron dos tipos de gestores más:

- *Scheduler decides based on Sleeping Threads (SST).* En este caso, el GP realiza las mismas funciones que el modelo KST, pero está integrado en el núcleo del SO, de tal forma que el manejador de interrupciones del reloj ejecuta periódicamente las fases de *Información* y *Evaluación*.

- *Kernel Idle Thread decides based on Sleeping Threads (KITST).* Este gestor realiza las mismas funciones que los modelos anteriores, pero ejecuta la fase de *Evaluación* en la hebra ociosa del kernel, mientras que mantiene la ejecución de la fase de *Información* en el manejador de interrupciones del reloj.

Además, se propone experimentar con el resto de criterios adaptativos del GP a nivel de kernel, además del desarrollo de otros nuevos, por ejemplo, un criterio adaptativo que decidiera en función a la eficiencia energética.

9.1.3. Detención de hebras

Durante la ejecución de una aplicación adaptativa, el GP decide simplemente la creación o no creación de hebras, de forma que el número de hebras simultáneas una vez creadas se mantiene constante hasta la finalización de cada una. Decidir crear hebras o mantener un número elevado de hebras activas puede ser un inconveniente, sobre todo en sistemas no dedicados en los que pueden ejecutarse otras aplicaciones concurrentemente y el número óptimo de hebras puede ser diferente en determinados periodos de su ejecución. Por ello, es muy interesante incorporar al GP el mecanismo de detención de hebras para poder adaptarse dinámicamente, tanto a las variaciones intrínsecas de la aplicación, como a los recursos disponibles. De esta manera el GP actúa de forma más versátil ampliando las opciones de decisión: crear nuevas hebras, dormir hebras para reducir la sobrecarga, despertar hebras dormidas o no crear hebras y continuar con la ejecución.

Bibliografía

[1] J. F. Sanjuan Estrada, L. G. Casado, I. García and E. M. T. Hendrix. Performance driven cooperation between kernel and auto-tuning multi-threaded interval B&B applications. In *Proceedings of ICCSA'12*, volume 7333, pages 57-70. Salvador de Bahia, June 2012. LNCS, Springer.

[2] J. F. Sanjuan Estrada, L.G. Casado, J.A. Martínez, M. Soler and I. García. Concurrencia-paralelismo auto-gestionada por procesos multihebrados en sistemas multicore. In *XX Jornadas de Paralelismo*, pages 7-11, A Coruña, 2009.

[3] E. Perla and M. Oldani. A guide to Kernel Exploitation, attacking the core. Ch. 1. Syngress, 2011.

[4] W. Stallings. Operating Systems, Internal and Design Principles (Seventh edition). Chs. 3, 4. Prentince Hall, 2012.

[5] D. P. Bovet, M. Cesati. Understanding the Linux Kernel (Third Edition). Ch. 7. O'Reilly, 2005.

[6] G. Díaz. Planificador de Linux (Scheduler). *Universidad de Los Andes (Venezuela).* 2013.

[7] A. Morante Bailón. Librería para aplicaciones multihebradas adaptativas. *Trabajo de Fin de Grado en la Universidad de Almería*. Junio, 2018.

[8] J. F. Sanjuan Estrada. Algoritmos multihebrados adaptativos en sistemas no dedicados. Prologo, Cap. 2. *Tesis Doctoral en la Universidad de Almería*. Enero, 2015.

[9] R. Love. Linux Kernel Development (Third edition). A thorough guide to the design and implementation of the Linux Kernel. Chs. 1, 3, 4. Addison-Wesley, 2010.

[10] H. Pérez Montiel. Física 2. Enseñanza Media Superior. Pags. 160 y 161. México D.F., 1994. Publicaciones Cultural

[11] Intel. Intel® Xeon Phi Coprocessor Performance Monitoring Units. 2012. Link: *https://software.intel.com/sites/default/files/forum/278102/intelr-xeon-phitm-pmu-rev1.01.pdf*

[12] Intel®. Intel® Xeon Phi coprocessor system software developers guide. 2017. Link: *https://software.intel.com/sites/default/files/managed/09/07/xeon-phi-coprocessor-system-software-developers-guide.pdf*

[13] Intel. Intel Xeon Phi coprocessor developers quick start guide. 2014. Link: *https://software.intel.com/sites/default/files/managed/26/d6/Intel_Xeon_Phi_Quick_Start_Developers_Guide-MPSS-3.4.pdf*

[14] Intel. Intel Manycore Platform Software Stack (Intel MPSS). 2017. Link: *https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss#lx38rel*

[15] Intel. Intel Manycore Platform Software Stack (MPSS) user's guide. 2017. Link: *http://registrationcenter-download.intel.com/akdlm/irc_nas/12828/mpss_users_guide.pdf*

[16] Intel. Intel Manycore Platform Software Stack MPSS 3.8 README (Intel(R) MPSS README). 2017. Link: *http://registrationcenter-download.intel.com/akdlm/irc_nas/12828/readme.txt*

[17] leVirve (Salas). Aplicación Parallel-N-Body-Simulation. Link: *https://github.com/leVirve/Parallel-N-Body-Simulation*

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

Gracias al auge de las arquitecturas paralelas, las aplicaciones multihebradas se han visto muy desarrolladas en estos últimos años y, por lo tanto, han sido necesarias herramientas de ayuda al desarrollador en la programación paralela para aprovechar la potencia de los multiprocesadores y procesadores multihebrados. En el presente trabajo se estudia como herramienta el Gestor del nivel de Paralelismo (GP) a nivel de kernel, que ayuda a las aplicaciones multihebradas a controlar el número de hebras en ejecución y, de esta manera, mejorar su rendimiento. Durante el desarrollo del trabajo se realiza la integración del GP en el núcleo Linux de la pila de software Intel® Manycore Platform Software Stack (Intel® MPSS) del coprocesador Intel® Xeon Phi. Además, se ha realizado la integración de la librería Interfaz para el Gestor de Paralelismo (IGP) en una aplicación existente para obtener una aplicación de benchmark, mediante la cual se han realizado pruebas de rendimiento sobre este equipo para demostrar el funcionamiento del GP a nivel de kernel.

Due to the growth of the parallel architectures, also the multithreading applications have been developed during the last years, so it was needed tools which support the parallel programing in order to take advantage of the potential of multiprocessors and multithreaded processors. In this project, the "Gestor del nivel de Paralelismo " or "GP" (paralelism level manager) is estudied at kernel level as a tool which assist multithreaded applications to manage the number of threads in execution and therefore, to develop its performance. In this project the GP is integrated in the Linux kernel of the Intel® Manycore Platform Software Stack (Intel® MPSS) installed on the coprocessor Intel® Xeon Phi. Besides, the "Interfaz para el Gestor de Paralelismo" or "IGP" library (paralelism manager interface) was integrated into an existent application, in order to obtain a benchmark through which tests have been executed on the coprocessor to demonstrate the performance of the GP at kernel level.