Facultad
de
Ciencias

Modelado de la Red de Interconexión en IRMASim

(Interconnection Network Modelling on IRMASim)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Borja Cuevas Cuesta

Director: José Luis Bosque Orero

Co-director: Mario Ibáñez Bolado

Septiembre-2022

Índice general

Índice de Figuras IV

Resumen V

Abstract VII

1. Introducción 1

1.1. Planificación de tareas 1

1.2. Múltiples tipos de trabajos 3

1.3. Objetivos 4

1.4. Metodología y Plan de trabajo 5

1.5. Estructura del Documento 6

2. Background 7

2.1. Conceptos básicos 7

2.1.1. Modelo de programación mediante paso de mensajes 7

2.2. Descripción del estado original y principales actores de la herramienta IRMASim 8

2.3. Trabajos relacionados 11

3. Modelado de la red de interconexión en IRMASim 13

3.1. Motivación para la introducción de tareas paralelas MPI 13
3.2. Creación de *jobs* MPI 14
3.3. Conversión de los *jobs* MPI en tareas ejecutables 17
3.4. Diferenciación de las ejecuciones de *jobs* secuenciales frente a *jobs* MPI . . . 18
3.4.1. Planificación 19
3.4.2. Modelo de Ejecución 22
3.4.3. Implementación del modelo de sincronización 24
3.4.4. Liberación de recursos 26

4. Evaluación de la implementación **30**
4.1. Metodología de las pruebas realizadas 30
4.1.1. Lectura de la información de las ejecuciones en IRMASim 33
4.2. Experimentos 33
4.2.1. Planificación de los *jobs* 33
4.2.2. Comunicación entre tareas MPI 38
4.2.3. Contención por comunicaciones 40
4.2.4. Sincronización entre las tareas MPI en la liberación de recursos . . . 42
4.2.5. Experimento global 43

5. Conclusiones y trabajos futuros **45**
5.1. Objetivos conseguidos 45
5.2. Trabajos futuros 46

Índice de figuras

1.1. Planificador o *scheduler* 1

3.1. Estado de la cola de trabajos ante una determinada traza 17

3.2. Ejemplo de cores disponibles para ejecutar un *job* MPI bajo un estado de la plataforma de ejecución dado 20

3.3. Ejemplo de cores disponibles para ejecutar un *job* secuencial bajo un estado de la plataforma de ejecución dado 21

3.4. Planificacion de tareas MPI lanzadas por un mismo *job* 25

3.5. Liberación de recursos por parte de las tareas de un *job* secuencial 27

3.6. Liberación de recursos por parte de las tareas de un *job* MPI 28

4.1. Modelo de la plataforma de ejecución utilizada para las pruebas experimentales . 31

4.2. *Jobs* que forman la primera traza del experimento de planificación 34

4.3. Tiempo de ejecución de la traza de la figura 4.2 35

4.4. *Jobs* que forman la segunda traza del experimento de planificación 35

4.5. Tiempos de ejecución de la traza de la figura 4.4 36

4.6. *Jobs* que forman la tercera traza del experimento de planificación 37

4.7. Tiempos de ejecución de la traza de la figura 4.6 37

4.8. *Jobs* que forman la primera traza de los experimentos de comunicación y de contención 38

4.9. Tiempos de ejecución de la traza de la figura 4.8 39

4.10. *Jobs* que forman la segunda traza del experimento de comunicación 40

4.11. Tiempos de ejecución 41

4.12. *Job* que forma la primera traza del experimento de sincronización en la liberación de recursos 42

4.13. Tiempos de ejecución 43

Resumen

Hoy en día, los grandes avances tanto en hardware como en software han permitido (y a la vez necesitado) un enorme desarrollo de entornos para la computación de alto rendimiento. Estos entornos específicos están formados por cientos o miles de equipos informáticos almacenados en lo que se conoce como centros de datos o *datacenters*, en los que se ejecutan todas las aplicaciones o programas que requieran los usuarios haciendo uso de los recursos computacionales que ofrezcan dichos equipos informáticos.

Las aplicaciones que los usuarios desean ejecutar en los *datacenters* constan de diferentes tareas lanzadas en forma de procesos. A raíz del desarrollo de las tecnologías mencionado antes, las necesidades de los usuarios y las propias aplicaciones son cada vez más complejas, lo cual ha motivado a los desarrolladores a modificar el modelo de programación para logar que estas aplicaciones, a pesar de ser más pesadas, se ejecuten de una forma más óptima. Dicho modelo ha evolucionado de la programación secuencial a la programación distribuida.

Toda aplicación que se ejecute en un *datacenter*, independientemente del tipo que sea, ha de ser planificada, lo que supone realizar una asignación entre la propia aplicación y los recursos que va a utilizar para ejecutarse. Esta tarea supone tomar decisiones de forma constante en las que influyen diversos factores que condicionan la forma de actuar y, por tanto, también los resultados obtenidos, convirtiéndose así en una labor muy complicada.

Sin embargo, la gran dimensión de los *datacenters* actuales y la inmensa cantidad de recursos que han de ser gestionados hacen inviable realizar estudios en un entorno real sobre la eficiencia de las diferentes políticas de planificación, lo que ha provocado el desarrollo de simuladores que permitan llevar a cabo estos análisis en entornos virtuales para luego realizar las implementaciones pertienentes sobre entornos reales.

Por tanto, el objetivo de este trabajo será la ampliación del simulador de *datacenters* IRMASim para introducir, además de las aplicaciones secuenciales que ya implementa, las aplicaciones de origen distribuido en las cargas de trabajo que es capaz de ejecutar en las simulaciones. Con ello, se realizarán ejecuciones de cargas de trabajo formadas por ambos tipos de aplicaciones, permitiendo evaluar diferentes políticas de planificación de una forma más realista.

Palabras clave: Planificación de tareas, Aplicaciones MPI, IRMASim, Simulador de *datacen-*

ters heterogéneos, Comunicación entre tareas, Sincronización y liberación de recursos

Abstract

Now-a-days, great advances in both, hardware and software have allowed and needed a huge development of specific environments for high-performance computing. These environments are made up of hundreds or thousands of computers stored in what is known as a data center, in which they run all applications or programs that users require making use of computational resources offered by such computer equipment.

The applications that users want to run in data centers consist of different tasks launched through processes execution. As a result of the development of the technologies mentioned before, the needs of users and the applications themselves are increasingly complex, which has motivated developers to modify the programming model to achieve that these applications, despite being heavier, run in a more optimal way. This model has evolved from sequential programming to distributed programming.

Any application that runs in a data center, regardless of its type, must be scheduled, which means allocate the application itself in the resources that will be used. This task involves constantly making decisions in which are influenced by various factors that condition the way of acting and, therefore, also the results obtained, thus becoming a very complicated task.

However, the large size of today's data centers and the immense amount of resources that have to be managed make it unfeasible to carry out studies in a real environment on the efficiency of the different planning policies, which has caused the development of simulators that allow carrying out these analyzes and then carry out the implementations relevant to real environments.

Therefore, the objective of this work would be the expansion of the IRMASim data center simulator to introduce, in addition to the sequential applications that it already contemplates, the applications of distributed origin in the workloads that it is capable of executing in the simulations. With this, executions of workloads formed by both types of applications will be carried out, allowing to evaluate different planning policies in a more realistic way.

Keywords: Task planning, MPI applications, IRMASim, data center simulator. Heterogeneous ters, Communication between tasks, Synchronization and release of resources

Capítulo 1

Introducción

En este primer capítulo se expondrán conceptos generales sobre los que se basa tanto el desarrollo del proyecto como la redacción del documento. En primer lugar, se expondrá la motivación y las razones por las cuales llevar a cabo el desarrollo de un trabajo como este. En segundo lugar, se explicarán los objetivos de este proyecto y se explicará el proceso y la metodología que se ha seguido para su realización, así como una breve descripción de la estructura del documento.

1.1. Planificación de tareas

En sistemas operativos, un planificador o *scheduler* es un componente software esencial que se encarga fundamentalmente de seleccionar, de entre todos los procesos pendientes de ejecutar del sistema, aquel al que se le serán asignados los recursos del procesador necesarios para ejecutarse y durante cuánto tiempo podrá hacer uso de ellos. [1].

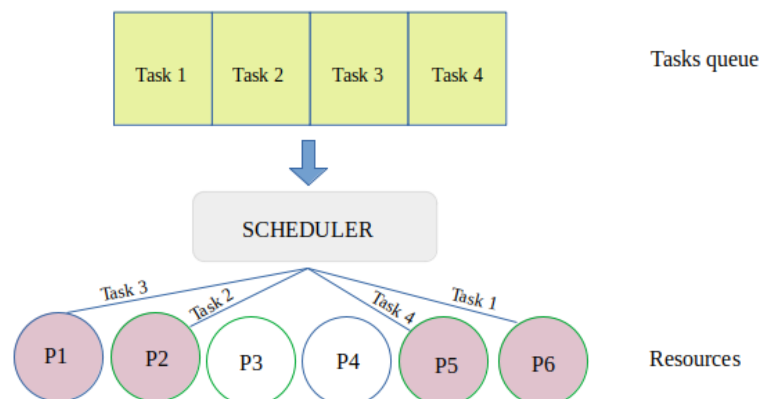

Figura 1.1: Planificador o *scheduler*

Las decisiones que el planificador toma están determinadas por una política que establece unos objetivos a maximizar y que condiciona su comportamiento, convirtiendo este trabajo en un proceso cuyos resultados pueden no llegar a preveerse de forma trivial.

Por tanto, la planificación de tareas consiste en un proceso mediante el cual a las tareas que se deseen ejecutar en un computador le son asignados un conjunto de recursos computacionales en los que poder llevar a cabo sus ejecuciones. En cuanto al sistema operativo, en el caso de un computador convencional, estas tareas podrían ser hilos o procesos y el conjunto de recursos estaría formado, principalmente, por los procesadores, los cores y la memoria del sistema. Sin embargo, la complejidad del proceso de planificación aumenta notoriamente cuando en vez de un computador de propósito general se trabaja en un centro de datos o *datacenter*, grandes instalaciones en las que, de forma simplificada, se interconectan entre sí tantos computadores como se necesite formando un supercomputador con unas prestaciones mucho más altas [2].

La computación de alto rendimiento, cada vez más demandada debido al aumento exponencial de la complejidad de los programas y las aplicaciones actuales, permite mediante la agregación de distintos componentes, entre los cuales se incluyen tanto hardware como software, obtener unas prestaciones computacionales muy superiores a las obtenidas mediante computadores convencionales. Gracias al elevado rendimiento obtenido en los *datacenters*, se consigue ejecutar tareas muy complejas y que requieren gran cantidad de recursos computacionales de una forma más rápida y eficiente.

Adicionalmente, en estos centros de datos se ejecutan cientos o miles de trabajos lanzados por diferentes usuarios que, sumado a la gran cantidad de recursos que hay que gestionar, aumenta considerablemente la complejidad de la tarea de planificación [3]. Además, se debe tener en cuenta que, actualmente, estas plataformas de ejecución suelen tratarse de sistemas heterogéneos [4]. Los *datacenters* están formados por un conjunto de procesadores que, compartiendo memoria entre ellos, forman un nodo de cómputo. Este nodo de cómputo unido a otros cuantos a través de una red de inteconexión forman lo que se denomina un *cluster*, y la unión de varios *clusters* es lo que genera el centro de datos. Esto supone que, dentro de un mismo *datacenter* o incluso dentro de un mismo *cluster*, puedan existir nodos de cómputo con diferentes características y especificaciones: unos que tengan una mayor cantidad de memoria para aquellos procesos que lo necesiten u otros que consten de aceleradores *hardware* como *GPUs* o *FPGAs* [5].

Por lo tanto, es muy común encontrarse un *datacenter* en el que sus nodos de cómputo son diferentes entre sí, lo que, teniendo en cuenta que en estos centros de datos existen diferentes objetivos a la hora de llevar a cabo las ejecuciones, eleva todavía más la dificultad de la planificación de tareas y de la gestión de los recursos computacionales. Estos objetivos suelen estar relacionados con la optimización de determinadas características de la ejecución como el tiempo de ejecución o el consumo de energía [6].

Todas estas variables que influyen en el proceso de planificación tienen que ser gestionadas para realizar las asignaciones entre tareas y recursos de la forma más efciente posible. Para

ello, el planificador deberá tener en cuenta toda esta información que le ayudará a tomar decisiones y escoger en cada momento qué tarea planificar y sobre qué recursos.

1.2. Múltiples tipos de trabajos

Como se ha mencionado en el apartado anterior, el número de trabajos y tareas que pueden estar ejecutándose en un centro de datos estándar en un instante determinado puede ser del orden de cientos o miles. Por tanto, será también elevado el número de usuarios que requieran de la utilización de los recursos disponibles de un *datacenter* para ejecutar sus aplicaciones y programas. Estas aplicaciones estarán adaptadas a las necesidades de cada uno, por lo que pueden presentar diferencias en la forma en la que han sido programadas para ejecutarse según su objetivo concreto.

Las aplicaciones lanzadas para ejecutarse sobre los *datacenters* pueden ser principalmente de dos tipos: secuenciales o distribuidas. En el primer grupo se recogen aquellas aplicaciones cuyas tareas se llevan a cabo de forma aislada, independientemente unas de otras y sin comunicación entre ellas. Cada tarea tiene un único flujo de ejecución a través del cual se ejecutan sus instrucciones una a continuación de otra. En cambio, las aplicaciones distribuidas son aquellas en las que las tareas a efectuar se llevan a cabo a través de diferentes procesos que pueden ser ejecutados en recursos físicamente separados pero conectados a través de una red de interconexión y que trabajan conjuntamente y se comunican entre sí enviando y recibiendo datos para alcanzar un objetivo común [7].

Esta variedad de los tipos de trabajos a ejecutar dentro de un *datacenter* supone otro extra más de dificultad en la tarea de planificación y ejecución de aplicaciones, ya que el comportamiento entre ambos tipos de aplicaciones varía considerablemente: difieren en la forma de planificarse, en sus respectivas fases de ejecución y en los factores que influyen y tienen efecto sobre sus propias ejecuciones.

Las aplicaciones distribuidas están muy ligadas a la paralelización y distribución de trabajos y de datos, modelo de programación que genera situaciones en las que varios procesos lanzados por la misma aplicación están realizando la misma tarea al mismo tiempo y, llegados a un punto de sus respectivas ejecuciones, necesitan comunicarse para intercambiarse datos. Este intercambio provoca que, a la hora de planificar estas tareas, sea necesario tener en cuenta aspectos como la red de interconexión que comunica los distintos recursos del *datacenter*, las latencias de los envíos de datos o incluso la capacidad de los enlaces, complicando todavía más la planificación [8].

Dado que en los últimos años se está llegando al límite de la *Ley de Moore* y no está siendo posible seguir mejorando las prestaciones de los chips [9], la dirección que se está tomando para prolongar el aumento del rendimiento de las aplicaciones pasa por explotar el paralelismo a través de la programación de aplicaciones distribuidas utilizando para ello el modelo de programación de paso de mensajes (*MPI, Message Passing Interface*) [10]. Por lo tanto, son

cada vez más frecuentes las cargas de trabajo lanzadas por los usuarios que se encuentran formadas por aplicaciones de ambos tipos.

La alta complejidad del problema de la planificación y la cantidad de factores que entran en juego y que debe analizar y gestionar el planificador es tan grande que este campo despierta un gran interés para investigadores que desean examinar y realizar pruebas experimentales para determinar las mejores políticas de planificación para una determinada carga de trabajo. Sin embargo, realizar estudios sobre sistemas reales supondría un coste desorbitado debido al tamaño de las infraestructuras utilizadas, por lo que es común recurrir al uso de simuladores que permitan realizar su trabajo a los investigadores de la forma más fiel a la realidad posible. De esta manera, será imprescindible para toda herramienta de simulación que, además de todas las características y factores relacionados con la plataforma de ejecución y el planificador, ofrezca la posibilidad de definir la carga de trabajo de la forma más realista posible, contemplando aplicaciones tanto de tipo secuencial como de tipo MPI.

1.3. Objetivos

El objetivo fundamental que tiene este trabajo es continuar en la mejora de prestaciones de la herramienta de simulación IRMASim, desarrollada en el seno del grupo de investigación de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria [11]. Más concretamente, dado que este objetivo es muy amplio y abarca una gran cantidad de posibilidades para seguir mejorando el simulador, los retos específicos que se pretenden alcanzar con este trabajo son los siguientes:

- Simulación de *jobs* distribuidos basados en el paradigma de programación mediante paso de mensajes (MPI).

  Actualmente, la definición de la carga de trabajo para la realización de simulaciones está muy limitada ya que sólo contempla aplicaciones de tipo secuencial. Sin embargo, cada vez son más las aplicaciones que se programan para que sean ejecutadas de forma distribuida, para las cuales el simulador IRMASim no ofrece ningún mecanismo de representación para poder incluirlas en las ejecuciones de las cargas de trabajo que se simulen.

- Simulación de la comunicación y sincronización entre tareas MPI.

  Las aplicaciones MPI están formadas por diferentes procesos que pueden ejecutarse de manera independiente y en ubicaciones físicamente separadas pero que se comunican entre sí para sincronizarse en determinados puntos de la ejecución. Por tanto, para introducir en el simulador este tipo de aplicaciones de una forma mínimamente realista, deberá desarrollarse una forma de simular la sincronización y la comunicación entre los diferentes procesos lanzados por una aplicación MPI.

- Creación de una primera aproximación de un modelo de simulación de contención por comunicaciones.

Para implementar la comunicación entre los procesos de una aplicación MPI, será imprescindible también implementar el medio utilizado por los procesos para la comunicación. Esto supone la simulación y parametrización de una red de interconexión local que conecte los diferentes nodos de cómputo del datacenter. Esta red deberá poder afectar a la ejecución de una aplicación MPI en la medida en la que influya sobre las comunicaciones que tengan lugar entre los procesos que lance. La red de interconexión de un datacenter puede saturarse y ralentizar los envíos de los mensajes de un proceso a otro, por lo que se tendrá que tener en cuenta para la ejecución de las simulaciones.

1.4. Metodología y Plan de trabajo

Para implementar las funcionalidades descritas en el apartado anterior sobre la versión actual de la herramienta IRMASim, el plan de trabajo que se ha seguido ha sido el siguiente:

- Análisis de la herramienta IRMASim.

  En primer lugar, se procedió a comprender el funcionamiento y la implementación del simulador. Esto posibilitó, por un lado, una primera toma de contacto para familiarizarse con la herramienta a utilizar y, por otro lado, que se fueran viendo las limitaciones de IRMASim para comprender la motivación del trabajo a desarrollar.

- Estudio de las características de las aplicaciones distribuidas de tipo MPI.

  En segundo lugar, antes de comenzar a modificar el estado del simulador, fue necesario analizar los parámetros característicos de las aplicaciones MPI para visualizar qué era exactamente lo que les diferenciaba del tipo de aplicaciones que ya existían dentro de la herramienta y qué modificaciones suponía la introducción de este nuevo tipo de aplicaciones.

- Diseño e implementación de los objetivos establecidos.

  Para cada uno de los objetivos planteados en el apartado anterior, se realizó un estudio de las posibles soluciones y, teniendo en cuenta el estado inicial de la herramienta, se implementaron aquellas más adecuadas para cada problema planteado.

- Comprobación del funcionamiento de las soluciones adoptadas.

  Una vez implementadas las modificaciones necesarias para alcanzar los objetivos propuestos, se realizaron una serie de pruebas experimentales analizando situaciones específicas que permitieran validar la efectividad de las soluciones adoptadas.

- Depuración y optimización de la implementación.

  Tras la realización de las pruebas, en la mayoría de los casos fue necesario volver a modificar el diseño para añadir alguna casuística que no había sido tenido en cuenta o corregir pequeños fallos de implementación.

1.5. Estructura del Documento

Este documento consta de un total de cinco capítulos (el capítulo actual incluido). Los objetivos de cada uno de ellos son los siguientes:

- *Capítulo 1*: capítulo introductorio en el que se presenta el problema principal alrededor del cual gira el objetivo del proyecto y se describe la motivación del trabajo. Además, se ofrece información general sobre el desarrollo del mismo.
- *Capítulo 2*: se describen conocimientos previos, tanto teóricos como relativos a las herramientas utilizadas, básicos para facilitar la comprensión del desarrollo del trabajo.
- *Capítulo 3*: se explican las modificaciones introducidas a la implementación del simulador utilizado y el proceso de trabajo detalladamente.
- *Capítulo 4*: se desarrollan los experimentos llevados a cabo a través de los cuales se verifica que las funcionalidades añadidas a la implementación descritas en el capítulo 3 funcionan correctamente.
- *Capítulo 5*: para finalizar el trabajo, se recopilan los objetivos conseguidos durante el desarrollo. Además, se exponen una serie de frentes de trabajo abiertos con el objetivo de aportar ideas a los desarrolladores a seguir mejorando la versión actual del simulador.

Capítulo 2

Background

En este capítulo se describirán, por un lado, una serie de conceptos muy básicos necesarios para entender el desarrollo del trabajo y, por otro lado, el funcionamiento y estado original del simulador IRMASim, herramienta principal que se ha utilizado como punto de partida. Se expondrán también de forma breve otros trabajos relacionados con este proyecto.

2.1. Conceptos básicos

2.1.1. Modelo de programación mediante paso de mensajes

La programación de sistemas concurrentes o paralelos se ha realizado tradicionalmente bajo el paradigma de memoria compartida, en el que la vía principal de interacción entre las tareas o procesos se realizaba por medio de la lectura y escritura de variables compartidas almacenadas en una región de memoria común y a la cual se accede implementando diferentes mecanismos de sincronización [12].

Sin embargo, cada vez es mayor el número de sistemas concurrentes distribuidos, donde cada proceso que se ejecuta sobre el sistema interactúa con otros procesos sin compartir memoria con ellos. Es decir, cada proceso maneja su propio espacio de variables locales y la comunicación con otros procesos se realiza mediante el envío de copias de sus datos locales a través de una red de interconexión que los comunica [13].

Podemos definir por tanto la programación mediante paso de mensajes como un modelo de programación para aplicaciones ejecutadas sobre sistemas en los que no existe físicamente compartición de memoria entre los procesos que lanza, ya sea porque estos se ejecutan en diferentes nodos de cómputo o porque se trate de una arquitectura que directamente no implemente memoria compartida. Esto supone que la comunicación entre aquellos procesos

que no compartan memoria deberá realizarse mediante mensajes que son transmitidos a través de una red de interconexión. Estos mensajes suelen ser típicamente datos que el proceso receptor necesita para continuar su ejecución. En este caso, el estándar de programación que ha sido utilizado y sobre el que se basa el desarrollo del proyecto es el estándar *MPI* (*Message Passing Interface*).

Bajo este paradigma, se eliminan todos los problemas de gestión y sincronización que suponen los accesos concurrentes a variables compartidas (exclusión mutua, starvation, deadlocks...). Sin embargo, obliga a definir mecanismos de paso de mensajes que resuelvan cómodamente el problema de la comunicación; principalmente, el direccionamiento de los distintos nodos de cómputo y la sincronización para acceder al medio de comunicación [14].

En el entorno de IRMASim, existen cuatro niveles en la jerarquía de recursos: cluster, nodo, procesador y core, y existe compartición de memoria en los dos niveles más bajos de la jerarquía [11]. Es decir, los cores de un mismo procesador comparten memoria entre sí y, a su vez, los procesadores de un mismo nodo comparten memoria también entre sí. Sin embargo, no existe una memoria global de todo el cluster que sea compartida por todos los nodos, por lo que aquellas aplicaciones distribuidas que quisieran ser ejecutadas en el simulador y que requiriésen de comunicación o sincronización entre procesos ejecutados en procesadores de diferentes nodos, deberían ser programadas bajo el paradigma del paso de mensajes [6].

2.2. Descripción del estado original y principales actores de la herramienta IRMASim

IRMASim es un una herramienta de simulación de un cluster heterogéneo diseñada para evaluar políticas de planificación de diferentes cargas de trabajo en el entorno de un *datacenter*, permitiendo para ello la implementación de planificadores tanto clásicos como basados en *Deep Reinforcement Learning* [15]. Incorpora además distintas funcionalidades que dotan a la herramienta de la capacidad de, por un lado, implementar un datacenter heterogéneo con miles de cores de cómputo y, por otro lado, simular la contención que provoca la elevada utilización de recursos compartidos por diferentes procesos que compiten entre sí.

Para comprender su funcionamiento, se comenzará por definir los principales actores que se ven involucrados en cada simulación. Algunos de estos componentes que se definirán son los conceptos de *job*, *tarea* y *recurso* dentro del entorno de IRMASim, el planificador y, de forma más genérica, la parte responsable de construir todo el entorno de ejecución, que denominaremos propiamente el simulador.

- *Job*: la herramienta IRMASim entiende como *job* una abstracción de un programa o aplicación lanzado por un usuario y que ha de ser planificado y ejecutado sobre la plataforma. Estas aplicaciones, a su vez, pueden lanzar cada una de ellas una o varias

tareas o procesos, que serán los que realmente se ejecutarán en los clusters.

Las ejecuciones de los *jobs* sobre la plataforma son totalmente independientes entre sí, por lo que el hecho de tener solamente uno o, por el contrario, varios *jobs* pendientes de ejecutar, no supone ninguna restricción adicional a la hora de tomar decisiones sobre la planificación o la asignación de recursos.

Por otro lado, los *jobs* tienen una serie de parámetros que los definen y caracterizan. Algunos de los más significativos son:

- *ID*: identificador único del *job*. Se utiliza en el simulador a bajo nivel para referenciar y distinguir cada *job* pero, de cara al usuario, es irrelevante.
- *Subtime*: tiempo en segundos que transcurre desde que comienza la simulación hasta que el *job* aparece en el sistema. En un entorno real, correspondería con el instante de tiempo en el que el *job* es lanzado respecto al comienzo de la ejecución del servicio.
- *Res*: representa, por un lado, el número de tareas que lanza el *job* y, por tanto, representa también el número de cores en los que se va a ejecutar ya que, inicialmente, todas las tareas lanzadas por un *job* son copias de la misma que serán planificadas y ejecutadas sobre diferentes cores de trabajo.
- *Profile*: define una lista con los parámetros características de la tarea o tareas lanzadas por el *job*. Como se ha mencionado previamente, todas las tareas lanzadas por un mismo *job* son copias de la misma luego, en estos casos, existirá solamente un valor para este atributo que definirá los mismos parámetros para todas las tareas del *job*.

- *Tarea*: proceso lanzado por un *job* que representa la unidad que será ejecutada realmente sobre los cores del cluster. Los objetos *tarea* no aparecen definidos como tal en el simulador, sino que, según lo indicado anteriormente, son una lista de parámetros recogidos en el atributo *profile* de los *jobs*, y cada uno de ellos lanzará tantas tareas (copias) como indique su parámetro *res*.

  Los atributos de las tareas que define el parámetro *profile* son el número de operaciones que realiza (*req_ops*), el tiempo estimado que el usuario prevé que va a tardar en ejecutarse (*req_time*, lo cual ayudará al planificador a tomar decisiones), instrucciones ejecutadas por cada ciclo del procesador (*ipc*), una estimación de la cantidad de memoria que utilizará la tarea (*mem*) y el volumen de memoria en bytes que se moverá entre memoria y procesador durante la ejecución de la misma (*mem_vol*).

- *Recurso*: es la unidad mínima de cómputo sobre la que se ejecutan las tareas. Representan los cores del sistema, por lo que están definidos dentro de una jerarquía de la cual forman parte siendo el nivel más bajo de ella.

  Esta jerarquía está formado por el cluster, los nodos, los procesadores y los cores. Por tanto, existen dentro de la herramienta una serie de atributos que permiten identificar a cada core y saber el procesador, nodo y cluster (en caso de que hubiera varios) al que pertenece.

Además, otros parámetros importantes relacionados con la jerarquía de recursos son el ancho de banda de cada procesador, la frecuencia de trabajo de cada core, el máximo de GFLOPS que es capaz de desarrollar cada core de un procesador y la disponibilidad del core, que indica si dicho core está disponible para ejecutar una tarea o se encuentra ocupado.

- *Planificador*: es el encargado de decidir a qué tarea o proceso se le asigna un determinado recurso y durante cuánto tiempo. Estas decisiones serán tomadas en base a dos políticas que determinarán por un lado, qué *job* de entre todos los pendientes para planificar será planificado y, por otro lado, qué recurso (o recursos si fuera necesario) entre todos los que estén disponibles será escogido para ejecutar el *job* seleccionado. Vemos así que dos de sus parámetros más importantes son la cola de trabajos, que contiene los *jobs* que están esperando a ser planificados, y la cola de recursos, que contiene los cores del cluster disponibles para computar un trabajo.

  Las políticas que implementa IRMASim para seleccionar el *job* a planificar son:

  - *RANDM, random"*: el *job* a planificar es escogido de forma aleatoria.
  - *FIARR, "first"*: se planifica el *job* que más tiempo lleve en la cola de trabajos.
  - *SHORT, "shortest"*: se escoge el *job* con el menor tiempo de ejecución estimado.
  - *SMALL, "smallest"*: se planifica el *job* que lance menor número de tareas.
  - *LRMEM, "low-memory"*: se elige el *job* que menos cantidad de memoria solicita para su ejecución.
  - *LRMBW, "low-memory-bandwith"*: se planifica el *job* que menos ancho de banda requiere para su ejecución.

  Por otro lado, las políticas implementadas para seleccionar qué recurso de la cola de cores se asignará al *job* son:

  - *RANDOM, random"*: es escoge un core de forma aleatoria.
  - *HICOM, "high-gigaflops"*: se elige el core que tenga la capacidad de cómputo pico máxima de entre todos los cores.
  - *HICOR, "high-cores"*: se escoge un core que pertenezca al procesador que mayor número de cores disponibles tenga. Una vez determinado cuál es este procesador, la elección de uno u otro core es indiferente.
  - *HIMEM, "high-memory"*: se selecciona un core que pertenezca al nodo con la mayor capacidad de memoria.
  - *HIMBW, "high-memory-bandwith"*: se escoge el core con el mayor ancho de banda.
  - *LPOWR, "low-power"*: se selecciona el core con el menor consumo de energía.

- Simulador: es la parte de la herramienta que se encarga de elaborar y construir el entorno de simulación para las ejecuciones y de calcular y retornar los resultados de la simulación. Esta misión se lleva a cabo fundamentalmente mediante la lectura y escritura de datos a través de ficheros en formato JSON. Los más relevantes son dos ficheros de entrada, uno que especifica la carga de trabajo, (*workload*), y otro que

determina la estructura del cluster sobre el que va a tener lugar la ejecución (*platform*), y un fichero de salida, en el que se escriben los resultados de la ejecución (*statistics*).

- *Workload*: fichero que define todas las características y parámetros de la ejecución respectivos a cada uno de los *jobs* del sistema. A través de su lectura, el simulador construye una abstracción de las aplicaciones que deberán ser ejecutadas sobre el cluster.
- *Platform*: fichero que define la estructura del cluster sobre la cual se van a ejecutar los *jobs* definidos en el *workload*. El cluster que ha de definirse debe estar formado por uno o varios nodos, cada uno de los cuales estará formado a su vez por uno o varios procesadores con uno o varios cores cada uno. Como se comentó en el apartado introductorio del documento, el cluster es un sistema heterogéneo, por lo que se podrán indicar especificaciones diferentes para cada uno de sus componentes (nodos, procesadores y cores) si así se desease.

Editando estos dos ficheros, el usuario de IRMASim podrá realizar sus propias simulaciones especificando las características de su carga de trabajo y el diseño de la estructura de su plataforma de ejecución. De esta forma, podrá evaluar varios parámetros de la ejecución (tiempo de ejecución, consumo y EDP) y analizar el rendimiento de una determinada política de planificación para su caso particular.

Esto permitirá al usuario verificar las políticas de planificación más óptimas para sus ejecuciones específicas en función de sus objetivos (optimización del tiempo de ejecución de la carga de trabajo, del consumo de energía, de la utilización de los recursos del cluster o del tiempo de ejecución de cada job de forma individual, entre otros), que influirán en las decisiones adoptadas por el planificador durante la simulación y, por tanto, también en los resultados obtenidos. Otro factor a tener en cuenta es la contención de memoria, que tendrá lugar cuando varias tareas se ejecuten en diferentes cores del mismo procesador y el volumen de información que se mueva entre la memoria del procesador y las tareas sea superior al ancho de banda de la memoria, y que consiste en conflictos en los accesos a memoria (recurso compartido por todos los procesadores de un nodo) que causarán bloqueos o ralentizaciones, ocasionando aumentos en los tiempos de ejecución de las tareas que la sufran.

2.3. Trabajos relacionados

Existen varios experimentos que, al igual que IRMASim, han tratado de aplicar técnicas de inteligencia artificial para la planificación de tareas en centros de datos, tanto en simuladores como en entornos reales. Muchos de estos trabajos se centran más en el aspecto de la planificación, y priorizan el análisis de la aplicación de técnicas de inteligencia artificial como *Deep Learning* para los algoritmos que el estudio de la influencia y el impacto de la red de comunicaciones del *datacenter* sobre la ejecución de las aplicaciones. Sin embargo, sí que existen algunos proyectos en los que es posible la planificación de tareas en diferentes nodos de cómputo, es decir, aplicaciones paralelas de tipo MPI, por lo que implementan la

parte de la red de interconexión a través de la cual se realizan las comunicaciones entre los nodos que forman la plataforma de ejecución.

Uno de estos ejemplos es el proyecto *BatSim*[16]. En este simulador, las aplicaciones paralelas de tipo MPI se definen mediante un conjunto de cálculos y operaciones que realiza la aplicación en cada host en el que es ejecutada y mediante la cantidad de comunicaciones que tienen lugar a través de la red durante su ejecución por cada par de hosts. Es decir, una aplicación MPI ejecutada en N hosts queda definida mediante N valores que representan el número de operaciones de punto flotante ejecutadas en cada host y mediante N^2 valores que representan la cantidad de bytes enviados por cada host al resto. Para simular sus ejecuciones, se maneja un concepto que es el de la tasa de avance de la ejecución de cada subelemento (tarea) de la aplicación MPI. A pesar de que cada tarea de una aplicación se adapta a su propio contexto de ejecución, en cada paso de la simulación se calcula qué subelemento supone un cuello de botella para la ejecución de la aplicación, es decir, qué tarea tiene la tasa de avance más baja, y se aplica esta tasa al resto de tareas de la aplicación paralela, progresando todas ellas de manera completamente síncrona.

Otro proyecto en el que se tiene en cuenta el impacto de las redes de interconexión en las simulaciones de las cargas de trabajo es en el simulador *CloudNetSim++*[17]. Esta herramienta permite la simulación de entornos en la nube, pudiéndose emplear para evaluar un amplio espectro de sus componentes como los elementos de procesamiento, almacenamiento, consumo de energía, algoritmos de planificación y, entre otras características, las redes de interconexión utilizadas para las comunicaciones. Al tratarse de servicios en la nube, la plataforma de ejecución puede ser físicamente distribuida, por lo que el propio simulador ofrece la posibilidad de implementar diferentes topologías de red para comunicarlas y proporciona herramientas para analizarlas y compararlas.

Por otro lado, el artículo *Simulating MPI Applications: The SMPI Approach* [18] describe otra forma de implementación de aplicaciones MPI y de simular el comportamiento de la red de interconexión del *datacenter*, definiendo para ello diferentes atributos específicos para la plataforma sobre la que se ejecutarán las aplicaciones MPI. En este trabajo, cada unidad de cómputo se define mediante el límite superior del valor de latencia que permite la red (retardo máximo en el envío de un mensaje entre dos máquinas), el *overhead* de la CPU debido a la sobrecarga provocada por el procesamiento de los envíos y las recepciones (tiempo durante el cuál la CPU no podrá realizar ninguna otra operación), el ancho de banda de cada procesador y el número de procesadores del que dispone la máquina paralela.

Por último, se hará mención a la herramienta descrita en el artículo *Simulation-based optimization and sensibility analysis of MPI applications: Variability matters* [19], en el que se describe una herramienta utilizada para simular y analizar diferentes aspectos de aplicaciones MPI. Para realizar la simulación de este tipo de aplicaciones, esta herramienta mapea cada *rank* de la aplicación MPI en un thread simulado. A continuación, estos threads se ejecutan en exclusión mutua y bajo control de la herramienta, que mide el tiempo transcurrido entre dos llamadas MPI y lo toma como un retardo simulado.

Capítulo 3

Modelado de la red de interconexión en IRMASim

En este capítulo se detallará el desarrollo de las contribuciones realizadas a la herramienta IRMASim. Para ello, se irá describiendo el proceso paso a paso explicando las soluciones adoptadas para cada objetivo fijado, planteando en una primera aproximación el problema a resolver para posteriormente explicar cómo se ha implementado dicha funcionalidad.

3.1. Motivación para la introducción de tareas paralelas MPI

En la versión inicial de IRMASim, se había conseguido simular de manera efectiva datacenters utilizando un agente inteligente para la planificación de las tareas [15]. Sin embargo, las aplicaciones que hasta ahora había contemplado el simulador se trataban de aplicaciones que lanzaban un determinado número de tareas que se ejecutaban de manera secuencial, cada una en un core del sistema según determinase la política de planificación, de forma totalmente independiente entre sí y, por lo tanto, sin ningún tipo de comunicación ni sincronización entre ellas.

Esta implementación suponía una limitación a la hora de realizar simulaciones de ejecuciones de cierto tipo de aplicaciones en las que varios procesos trabajan conjuntamente y cooperan para conseguir un objetivo común. En este tipo de aplicaciones, es frecuente que, en función de la disponibilidad de los cores del sistema, la ejecución de sus procesos pueda realizarse en nodos diferentes para mejorar la utilización de los recursos y que no se den casos en los que, teniendo tareas pendientes de ejecutar y cores disponibles repartidos en diferentes nodos, estos no se utilicen.

Además, otra limitación de IRMASim es que, al trabajar únicamente con *jobs* de tipo se-

cuencial, las políticas de planificación no tenían en cuenta el nodo al que pertenecían los cores al realizar las asignaciones entre las tareas y los recursos ya que, al no contemplarse la comunicación entre las tareas de un mismo job, en la práctica resultaba indiferente que se ejecutasen en el mismo o en diferentes nodos. De esta forma, los *jobs* secuenciales adoptaban así un comportamiento distribuido que no debería darse, ya que las tareas que lancen este tipo de *jobs* deberían ejecutarse en procesadores pertenecientes al mismo nodo.

Por consiguiente, la solución implementada para resolver estas situaciones comenzó por la creación de los *jobs* de tipo MPI dentro del entorno del simulador para introducir posteriormente un modelo de simulación muy básico de la red de interconexión para la comunicación y sincronización de tareas MPI ejecutadas en nodos distintos. Para ello, se han seguido los siguientes pasos, que se describen en las siguientes secciones de este capítulo:

- Creación de *jobs* MPI.
- Conversión de los *jobs* MPI en tareas ejecutables
- Diferenciación de las ejecuciones de *jobs* secuenciales a *jobs* MPI.
- Implementación del modelo de sicronización

3.2. Creación de *jobs* MPI

La primera modificación que se realizó sobre la herramienta fue la creación y la introducción de los *jobs* de tipo MPI en el entorno del simulador. Este nuevo tipo de job lanza diferentes tareas o procesos que pueden ser ejecutados en cores de diferentes nodos.

Esto supone, por un lado, la conversión de la clase *Job* en una superclase en la que se definen los atributos comunes a ambos tipos de *jobs* y la creación de otras dos nuevas clases hijas, *Job_Seq* y *Job_MPI*, en las que se definen los atributos específicos de cada tipo de *job*. Además, dado que un *job* de tipo MPI podrá lanzar distintas tareas y estas podrán ser diferentes entre sí, se creará también una clase *Tarea* que definirá las características propias de cada una de ellas y que se detallará más adelante en este mismo apartado.

De esta forma, los atributos más significativos relativos a los *jobs* quedan definidos de la siguiente manera:

- Clase *Job* (atributos comunes a ambos tipos de jobs):
  - *Id*: identificador del *job*.
  - *Job_type*: tipo de *job* (secuencial o MPI).
  - *Subtime*: instante de tiempo durante la ejecución en el que se lanza el *job*.

- *Profile*: a pesar de que ambos tipos de *job* tienen el mismo atributo *profile* definido, en cada uno de ellos se le da un enfoque distinto, por lo que, para un mejor entendimiento del mismo, se explicarán los detalles de cada uno en el apartado correspondiente al *job* que lo defina.

- Clase *Job_Seq* (atributos específicos de los *jobs* de tipo secuencial):

  - *Resources*: indica el número de tareas que lanza y, por tanto, también el número de cores que va a requerir para su ejecución.
  - *Profile*: en este caso, recoge las características de ejecución de las tareas lanzadas por el job.

  Como se explicó en el capítulo 2.2 del documento, las tareas lanzadas por un *job* secuencial siguen siendo copias de la misma tarea, por lo que todas ellas tienen las mismas características de ejecución que define el atributo *profile* (número de operaciones, *ipc*, memoria utilizada, etc.).

- Clase *Job_MPI* (atributos específicos de los *jobs* de tipo MPI):

  - *Num_nodes*: indica el número de nodos en los que se quiere planificar las tareas lanzadas por el *job*. Habitualmente, este atributo será proporcionado por los usuarios cuando se utiliza un gestor de colas.
  - *Comm_vol*: representa el volumen total en bytes de comunicaciones que se transfiere entre las tareas lanzadas por un mismo *job* MPI durante sus respectivas ejecuciones. Se utilizará durante las ejecuciones de este tipo de *jobs* para determinar los casos en los que se produzca *contención por comunicaciones* debido a la sobrecarga de la red de interconexión de los nodos a través de la cual se comunican estas tareas (detallado en el apartado 3.4.3).
  - *T_compute*: representa el factor de tiempo que las tareas lanzadas por un *job* MPI se dedican a realizar cómputo. Es decir, un valor de 0.9 en este atributo signficará que el 90 % del tiempo que sus tareas se están ejecutando se están dedicando a computar operaciones y el tiempo restante, el 10 % de la ejecución, es el que se está dedicando a las comunicaciones con aquellas tareas que hayan sido lanzadas por el mismo *job* y que estén siendo ejecutadas en otro nodo.
  - *Tasks*: lista que contiene las tareas lanzadas por el *job*.
  - *Prof_job*: para los *jobs* MPI, define los mismos parámetros de ejecución que ya incluía el atributo *profile* para los *jobs* secuenciales pero guardando, en cada parámetro de la ejecución, el valor más limitante que exista entre las tareas que lanza. Es decir, para asignar un valor al parámetro *req_ops* (número de operaciones), se analiza dicho parámetro (definido en los *profiles* de todas las tareas del *job*) y se escoge aquel valor que vaya a incrementar el tiempo de ejecución.

Como se ha mencionado previamente, el hecho de que las tareas lanzadas por un *job* de tipo MPI sean idependientes unas de otras implica que no tienen por qué ser todas iguales. De hecho, lo normal es que no ocurra así. Cada una podrá tener sus propias características

(número de operaciones, volumen de memoria, tiempo de ejecución etc.), lo cual motiva la creación de la figura *Tarea* como tal, dentro del entorno del simulador a través de una clase, denominada *Task*, que define todos sus atributos.

Por tanto, de forma similar a los jobs (pues las tareas son en realidad la manera en la que se ejecutaban los *jobs*), los atributos recogidos en esta nueva clase son los siguientes:

- *Id*: identificador unívoco de la tarea.
- *Id_job*: identificador del job al que pertenece la tarea, es decir, que ha lanzado dicha tarea.
- *Profile_task*: al igual que en los *jobs* secuenciales, este atributo define los parámetros de ejecución de las tareas lanzadas por el *job* MPI. La diferencia radica en que estas tareas ya no compartirán este atributo entre ellas, como sucedía con las tareas lanzadas por los *jobs* de tipo secuencial, sino que ahora, al ser diferentes e independientes entre sí, cada una tendrá su propio atributo *profile* que podrá definir diferentes características de ejecución para cada una de ellas.
- *Pending_tasks*: indica, para cada tarea, el número de tareas lanzadas por su mismo *job* que todavía no han finalizado su ejecución. Este atributo se utilizará para gestionar la liberación de recursos utilizados por los *jobs* MPI, lo cual será detallado en la sección 3.4.3.

En el apartado 2.2 se mencionó que el simulador utiliza ficheros de entrada para la creación de la carga de trabajo (fichero *workload.json*) y de la estructura del datacenter (fichero *platform.json*), por lo que, para que las modificaciones introducidas tengan efecto, deberemos modificar el formato de estos ficheros (escritos en formato JSON) para presentarle al simulador la información como él espera recibirla. A través de ellos, proporcionaremos a la parte de la herramienta IRMASim encargada de elaborar el entorno de simulación los datos necesarios para generar las nuevas estructuras de datos con todos estos nuevos atributos incluidos.

La clase de la herramienta IRMASim encargada de la lectura de los ficheros y de la construcción de todo el entorno de ejecución es la clase *util.py*. Para llevar a cabo esta misión, esta clase realiza principalmente dos subtareas:

- Generar la carga de trabajo que debe ser computada.
- Configurar la plataforma sobre la cual va a ser ejecutada dicha carga de trabajo..

Para ello, toma como argumentos de entrada el fichero *workload*, a través del cual generará la lista de jobs que han de ser ejecutados en el sistema, y un *pool* de cores con información sobre la plataforma a través de la cual se generará el sistema de ejecución.

En primer lugar, deberemos modificar la lectura de la información relativa a los *jobs* a través del fichero *workload* para la generación de la carga de trabajo. Ahora, se deberá analizar, para cada *job* del *workload*, su tipo, y se realizará la lectura de sus atributos de una forma u otra en función del tipo de *job* que se trate, ya que cada uno cuenta ahora con su propio método constructor. Además, en caso de que el *job* sea de tipo MPI, deberá leerse también toda la información correspondiente a sus tareas.

Una vez los objetos *jobs* y tareas han sido creados, estos se añaden a la cola de trabajos pendientes de ejecutar. En el caso de los *jobs* MPI, al estar formado por tareas independientes, se ha tomado la decisión de tratar a las tareas que lanza como si fueran cada una un *job* encargado de ejecutar la tarea en cuestión y que puede ser planificado en cualquier core disponible de la plataforma. De esta forma, la cola de trabajos queda formada por *jobs* secuenciales (para los cuales habrá que, posteriormente, planificar y ejecutar las tareas que lancen) y por tareas MPI independientes unas de otras, relacionadas aquellas que hayan sido lanzadas por el mismo *job* mediante el identificador de dicho *job*.

Por ejemplo, ante una traza formada por dos *jobs* secuenciales (*jobs* 1 y 2) que lanzan cuatro y dos tareas respectivamente y dos *jobs* MPI (*jobs* 3 y 4) que lanzan dos y tres tareas, la cola de trabajos quedaría como se indica en la figura 3.1.

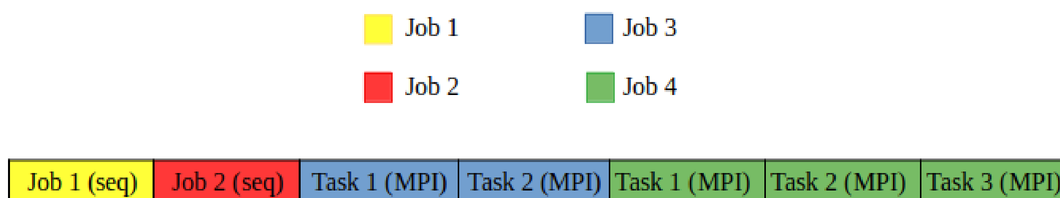

Figura 3.1: Estado de la cola de trabajos ante una determinada traza

Con la cola de trabajos pendientes formada, el siguiente paso es construir, a través de la lectura del fichero *platform.json*, la estructura del datacenter sobre la que se ejecutarán las tareas. Este proceso comienza modelando el nivel más alto de la jerarquía de recursos, el cluster, y continúa por los nodos y los procesadores hasta llegar a los cores, el nivel más bajo. Dado que en este primer paso no se ha modificado ninguna característica de la plataforma de ejecución, este proceso no se ve modificado.

3.3. Conversión de los *jobs* MPI en tareas ejecutables

Como se ha mencionado en el apartado previo, los *jobs* MPI quedan definidos dentro del entorno de ejecución a través de sus tareas, relacionadas entre sí mediante el atributo que registra el identificador del *job* que las ha lanzado. Haciendo una analogía y comparándolas con los *jobs* de tipo secuencial que ya se conocían y obviando la parte de la comunicación

y sincronización, las tareas MPI se tratan como si fueran cada una un *job* secuencial que únicamente lanzase una tarea.

Sin embargo, para que las tareas MPI sean interpretadas dentro del entorno de ejecución como *jobs*, ha sido necesario adaptar algunos atributos de los *jobs* MPI al objeto *Tarea* para que el simulador sea capaz de trabajar con ellas, ya que, dada su configuración previa, la herramienta espera trabajar y realizar la simulación de ejecuciones con objetos de tipo *Job* y no de tareas.

Por lo tanto, se debieron realizar una serie de modificaciones de cara a la parte interna del simulador en busca de que la implementación de la fase de ejecución para las tareas afectara lo mínimo posible, por un lado, a las propias tareas ya implementadas y, por otro lado, a la lectura de sus parámetros. Todas estas modificaciones están relacionadas con atributos de los *jobs* MPI que las tareas han debido incorporar para adaptar el proceso de ejecución tanto a los *jobs* secuenciales como a las tareas MPI. Además, guardan también relación con el método constructor de este tipo de tareas, el cual ha habido que ampliar para contemplar estos nuevos atributos heredados del objeto *Job* para que puedan ser manejadas por el simulador como *jobs*.

Los atributos que han sido incorporados a los objetos *Tarea* son *subtime*, *job_type*, *T_compute*, *res* y *comm_vol*. Todos ellos se tratan de atributos propios del *job* que las tareas necesitan acoplarse para que puedan ser ejecutadas. A la hora de darlos un valor, tienen lugar dos escenarios:

- En el caso de los atributos *subtime*, *T_compute* y *comm_vol*, se trata de atributos en los que puede variar su valor (y, de hecho, en la mayoría de los casos, variará) entre unas tareas y otras en función de quién sea el *job* que las lance. Sin embargo, por simplicidad, todas las tareas lanzadas por el mismo *job* van a tener siempre el mismo valor para estos tres atributos, por lo que estos tres valores se tomarán directamente del atributo que los defina del *job* que las lanza.
- En el caso de los atributos *job_type* y *Res*, van a tener siempre el mismo valor para todas las tareas indpendientemente del *job* que las lance y de la tarea en cuestión de la que se trate. El atributo *job_type* siempre indicará que se tratan de tareas de tipo MPI y, por su parte, el atributo *res* tendrá siempre un valor de 1 para indicar que dicha tarea sólo requiere un core. Por tanto, a estos atributos se les asigna directamente su valor sin necesidad de realizar ninguna lectura.

3.4. Diferenciación de las ejecuciones de *jobs* secuenciales frente a *jobs* MPI

En esta sección, se implementará la funcionalidad específica de los *jobs* de tipo MPI, detallando para ello las diferencias de comportamiento que deben tener lugar entre ambos tipos

de *jobs*

3.4.1. Planificación

Con el nuevo tipo de *job* introducido, tenemos ahora una herramienta en la que se contemplan cargas de trabajo formadas bien por *jobs* secuenciales, MPI (a través de sus tareas) o una mezcla de ellos pero que, sin embargo, a la hora de la ejecución, ambos se comportan todavía de la misma manera.

Por tanto, antes de entrar en la fase de ejecución de los trabajos, lo primero en lo que deben diferir ambos tipos de *jobs* es en la planificación y en la asignación de recursos para sus tareas. En una primera instancia, el planificador de la herramienta IRMASim realizaba una planificación a nivel de core. Para ello, se tenían dos políticas: en la primera de ellas, se realizaba una ordenación en función de una política concreta de aquellos cores del sistema disponibles para ejecutar una tarea y el primer core de esta cola era asignado a la primera tarea pendiente de la cola de tareas. En la segunda, se ordenaban también los trabajos según cierto atributo (operaciones, memoria, etc.) y se realizaba posteriormente la elección del primer core y primer trabajo de cada ordenación para llevar a cabo la asignación.

De esta forma, la única comprobación que se realiza a nivel de core a la hora de asignarlo a una tarea es que esté disponible, pudiendo ocurrir que dos tareas lanzadas por el mismo *job* secuencial fuesen planificadas en cores pertenecientes a procesadores de distintos nodos, adoptando de esta forma un comportamiento MPI que no deberían tener los *jobs* secuenciales.

La primera diferencia en la asignación de recursos entre los *jobs* secuenciales y los *jobs* de tipo MPI, por tanto, está en establecer el *pool* de cores disponibles para ejecutar las tareas lanzadas por los *jobs*. En el caso de los *jobs* secuenciales, sus tareas deberán ser planificadas y ejecutadas en cores de procesadores pertencientes al mismo nodo. En cambio, las tareas lanzadas por un *job* de tipo MPI, podrán planificarse en cualquier core de la plataforma siempre que esté disponible, pudiendo caer distintas tareas del mismo *job* en diferentes nodos. Se ve entonces que, para los *jobs* de tipo secuencial, la planificación pasará a ser por nodo (y no por core como se hacía antes) mientras que, para los *jobs* MPI, la planificación se realizará por core.

La clase de la herramienta IRMASim encargada de realizar las asignaciones entre tareas y recursos es la clase *manager.py*. Para llevar a cabo estas nuevas funcionalidades, se ha modificado el método *get_resources*, encargado de obtener un conjunto de recursos donde poder ejecutar el siguiente *job* de la cola de trabajos y de cambiar el estado de los recursos a medida que se seleccionan y se utilizan. Para ello, en este método se maneja, entre otras estructuras, una lista que contiene todos los cores disponibles del sistema en los que poder ejecutar el *job* a planificar (lista *available*). Cuando en el simulador únicamente existía un sólo tipo de *job* (los secuenciales), la lista *available* se formaba recorriendo todos los cores del cluster y guardando los que cumplieran solamente la condición de que no estuvieran ejecutando ningún trabajo, es decir, aquellos cores que estuvieran disponibles. Los cores en

los que finalmente acababa ejecutándose el *job* eran seleccionados de esta lista, por lo que el hecho de no comprobar el nodo al que pertenecían antes de añadirlos podía ocasionar, como se ha comentado previamente, que las tareas de un *job* secuencial se ejecutaran en cores pertenecientes a diferentes nodos, adoptando así un comportamiento MPI que habría que evitar.

De esta forma, la lista *available* será completada de formas distintas en función de si lo que se quiere planificar es un *job* de tipo secuencial o es una tarea MPI:

- Si el *job* a planificar se trata en realidad de una tarea de tipo MPI, la lista *available* deberá ser creada como se ha hecho hasta ahora, ya que no es necesario que estas tareas se ejecuten en el mismo nodo que el resto de tareas que haya lanzado su *job*. Por tanto, no se comprobará el nodo al que pertenezcan los cores y la única condición que debe satisfacerse es que el core donde vaya a ejecutarse dicha tarea debe estar libre, condición que ya se contemplaba desde un principio.

  Dado un estado de la plataforma de ejecución como el de la figura 3.2, en el que los cores pintados de rojo representan los cores ocupados, y un *job* MPI que lanza un determinado número de tareas, observamos que dichas tareas podrán planificarse en cualquiera de los cores libres independientemente del nodo al que pertenezcan.

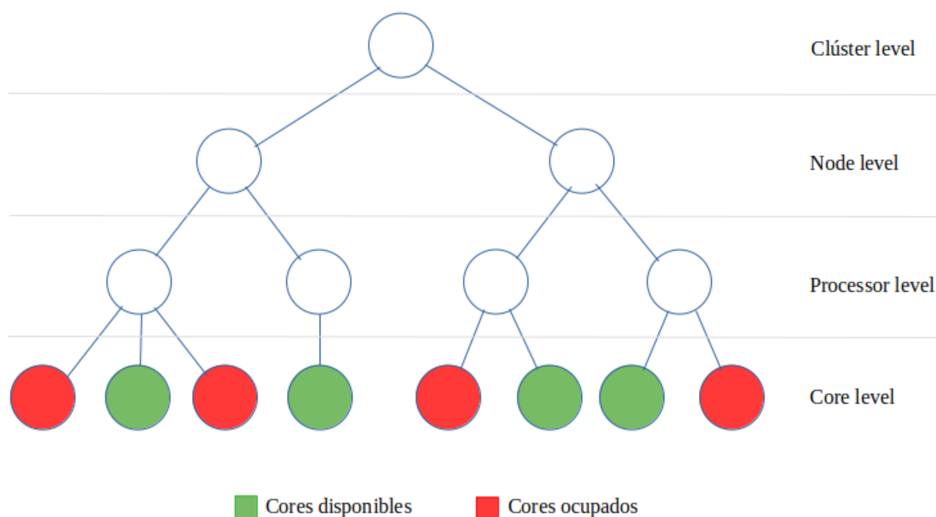

Figura 3.2: Ejemplo de cores disponibles para ejecutar un *job* MPI bajo un estado de la plataforma de ejecución dado

- En cambio, si el *job* a planificar es un *job* de tipo secuencial, debe garantizarse que todas las tareas que lance se ejecuten en el mismo nodo, pudiendo caer en procesadores diferentes pero siempre que se cumpla la condición anterior. Por tanto, para crear ahora la lista *available* lo que hay que hacer es analizar, para cada nodo de la plataforma, cuántos cores tiene disponibles para ejecutar el *job*. Para ello, se recorren los cores y se comprueba cuántos cores libres hay en su mismo nodo. En el momento en el que se encuentre un core cuyo nodo contenga un número de cores disponibles suficiente

como para ejecutar el *job* en cuestión, finaliza la búsqueda y se obtendrán los cores disponibles de dicho nodo donde poder realizar la planificación del *job*.

Se ve, por tanto, que la búsqueda del nodo donde realizar la planificación del *job* es una búsqueda *first_fit*: la solución al problema se obtiene cuando se encuentra el primer nodo que satisfaga las necesidades del *job*, a pesar de que puedieran existir soluciones más óptimas [1].

Siguiendo el mismo ejemplo que se ponía para la planificación del *job* MPI, en la figura 3.3 se puede ver ahora que dado el mismo estado de la plataforma de ejecución y un *job* que lanza un número de tareas concreto:

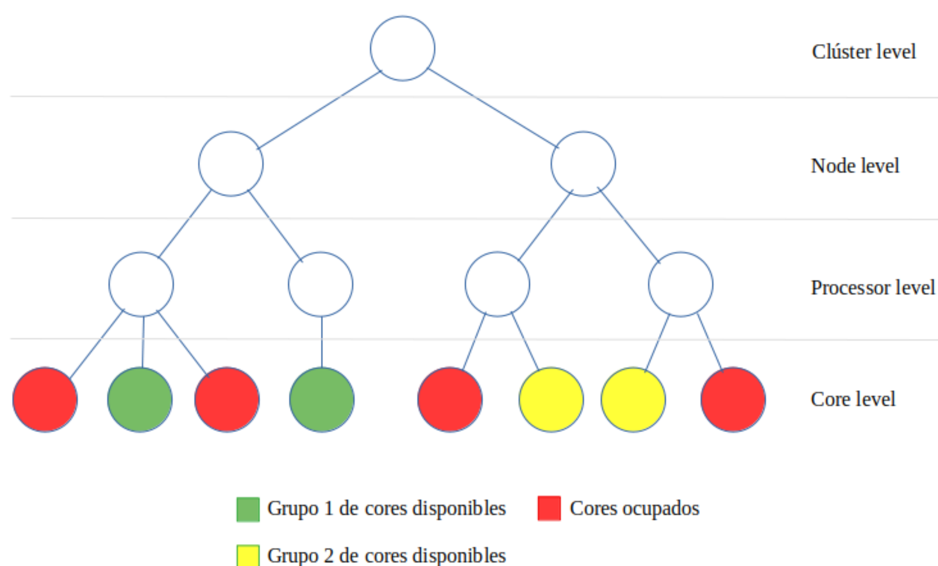

Figura 3.3: Ejemplo de cores disponibles para ejecutar un *job* secuencial bajo un estado de la plataforma de ejecución dado

Existen dos conjuntos de cores disponibles donde ejecutar las tareas del *job* secuencial: uno está formado por los cores disponibles del primer nodo y el otro por los del segundo nodo. El *job* deberá ser planificado utilizando los cores de uno u otro grupo, nunca en una mezcla de ambos.

Por otro lado, independientemente del tipo de *job* que se planifique, se debe garantizar que el número de cores disponibles para ejecutar dicho *job* es lo suficientemente grande. En otras palabras, debe verificarse que la longitud de la lista *available* es mayor o igual que el número de tareas lanzadas por el *job*, tanto si es de tipo secuencial como si es de tipo MPI. Podrían ser adoptados otros comportamientos pero, dada la implementación actual de IRMASim, sólo se contempla que un *job* pueda ser ejecutado si existe un número de cores libres igual o superior al número de sus tareas. En el caso de que esta condición no se cumpla, la ejecución del *job* se verá bloqueada a la espera de que otros recursos ocupados por otros *jobs* sean liberados y pueda de esta forma comenzar su ejecución.

Esta decisión de implementación puede generar situaciones en las que un *job* que no puede ser planificado por falta de recursos bloquee la ejecución de otros *jobs* que vienen por detrás de él en la cola de trabajos y que sí disponen de recursos suficientes para ejecutarse pero que, sin embargo, están a la espera de que ese primer *job* sea planificado.

3.4.2. Modelo de Ejecución

El siguiente paso en el ciclo de vida de los *jobs* y tareas dentro de la simulación es la fase de ejecución. La ejecución consiste básicamente en la lectura e interpretación de los parámetros relativos a las operaciones y del contexto de ejecución de ambos objetos. La diferencia, como se ha introducido en apartados anteriores, radica principalmente en la variedad de los valores de los atributos *resources* y *profile* de cada *job* y tarea.

Los *jobs* de tipo secuencial ocuparán tantos cores del mismo nodo como indique su atributo *resources*, y en cada uno de ellos lanzará una tarea con las características de ejecución que aparezcan definidas en el atributo *profile* del *job*. Por tanto, todas las tareas lanzadas por un *job* de tipo secuencial son, al fin y al cabo, distintas instancias de la misma. En cambio, las tareas MPI ocupan cada una de ellas únicamente un core ubicado en cualquier nodo de la plataforma, en el cual se lanza siempre sólamente una copia de dicha tarea (atributo *resources* a valor 1) y se ejecutan cada una atendiendo a las características de ejecución recogidas en los parámetros de sus respectivos atributos *profile_task*.

Ambos atributos *profile*, tanto el de los *jobs* secuenciales como el de las tareas MPI, definen los mismos parámetros de ejecución (los parámetros *req_ops*, *ipc*, *req_time*, *mem* y *mem_vol*, explicados todos ellos en el apartado 2.2). A través de los parámetros *req_ops* e *ipc*, es posible calcular, para cada *job* y tarea, el número de ciclos de ejecución que requieren. Por lo tanto, conociendo también la frecuencia de trabajo de cada core a través del tiempo de ciclo indicado para cada uno de ellos en el fichero *platform.json*, se calcula el tiempo de ejecución de cada *job* y tarea.

Sin embargo, la implementación de los *jobs* de tipo MPI en el simulador originan la aparición de una nueva diferencia sustancial entre las ejecuciones de ambos tipos de *jobs* ocasionada por las comunicaciones provocadas por las tareas lanzadas por los *jobs* MPI. Las ejecuciones de este tipo de *jobs* se verán afectadas en los casos en los que sufran contención por comunicaciones, lo que provocará un aumento en el tiempo de ejecución de algunas o todas sus tareas. Por tanto, a los factores que influían en los tiempos de ejecución de los *jobs* hay que añadirle, para los de tipo MPI, la posibilidad de sufrir contención por comunicaciones. De esta forma, los factores que influyen en el tiempo de ejecución de las tareas y, por tanto, también de los *jobs*, son los siguientes:

- En primer lugar, influye el tiempo de ciclo del core en el que sea planificada la tarea. A pesar de que es un valor conocido en el momento en el que se complete la asignación de recursos para la tarea y que no va a variar durante la ejecución de la misma, es

importante tener en cuenta que, al tratarse de una plataforma de ejecución heterogénea, los cores del cluster pueden contar con diferentes características unos de otros, que provocarían que dos tareas con el mismo número de ciclos de ejecución tengan diferentes tiempos de ejecución.

- En segundo lugar, hay que tener en cuenta la congestión en el acceso a memoria del procesador en el que se está ejecutando una tarea, es decir, el número de tareas que hay ejecutándose en el resto de cores del procesador de forma simultánea paralelamente a la tarea en cuestión.

  Cuando una tarea se ejecuta en un determinado core de un procesador, existe un volumen de datos que se transfiere entre la memoria del procesador y el core para que la tarea pueda ejecutarse. Por otro lado, los procesadores cuentan con un ancho de banda que detemina la capacida máxima de transmisión del canal de comunicación entre la memoria principal y el procesador, es decir, el volumen máximo (generalmente expresado en bits) que es capaz de transferir el enlace por segundo. Por lo tanto, si una tarea se ejecuta en un procesador en el que el volumen de transferencias de información con su memoria agregada supera el ancho de banda de su canal, los accesos a memoria se verán ralentizados y repercutirá negativamente en el tiempo de ejecución de la tarea que, lógicamente, se verá incrementado. Estas situaciones dependen del número de tareas que se están ejecutando en un procesador de forma simultánea y del ancho de banda consumido por cada tarea (o volumen de transferencia de datos por tarea). Estos conflictos entre las tareas por el uso de la memoria, un recurso compartido entre ellas, es lo que se conoce como contención de memoria, lo cual provoca, a través de un retardo en los accesos debido a la saturación del canal, una penalización en el tiempo de ejecución de las tareas.

- Finalmente, se debe tener en cuenta un factor que antes no existía en el simulador y que se ha implementado con la introducción de los *jobs* de tipo MPI en el sistema, que es la contención por comunicaciones. Este último factor penalizará, de forma análoga a como sucede con la memoria, a aquellas tareas que se ejecuten en nodos cuyo volumen de comunicaciones con otros nodos supere, en un instante dado de la ejecución de las tareas, su ancho de banda máximo para las comunicaciones. Este tipo de contención afectará únicamente a las tareas de tipo MPI ya que son las que utilizan la red de interconexión para comunicarse con el resto de tareas lanzadas por el mismo *job* para sincronizarse o enviarse datos. Las tareas lanzadas por los *jobs* secuenciales no hacen ningún tipo de uso de la red de interconexión para comunicarse entre ellas, dado que la comunicación en estos casos se realiza bajo el paradigma de memoria compartida.

  En resumen, las comunicaciones entre dos tareas MPI tendrán lugar cuando hayan sido lanzadas por el mismo *job* y sean planificadas en diferentes nodos, y la contención por comunicaciones aparecerá cuando el volumen de comunicaciones de uno de los nodos sea superior a su ancho de banda, ralentizando el proceso de comunicación de las tareas y provocando que sufran una penalización en sus tiempos de ejecución.

3.4.3. Implementación del modelo de sincronización

Como se ha explicado en el apartado anterior, la implementación de los *jobs* de tipo MPI en el simulador ha provocado la aparición del fenómeno de la contención de comunicación, que tiene lugar cuando el volumen de comunicaciones entre dos tareas o conjuntos de tareas de dos nodos diferentes supera el ancho de banda de comunicaciones de dichos nodos, y provoca un retardo en la velocidad de las comunicaciones que penaliza el tiempo de ejecución de las tareas.

Para implementar este nuevo modelo de contención por comunicaciones, en primer lugar se modificaron las estructuras de datos relativas a la plataforma de ejecución y a los *jobs* MPI añadiéndoles los siguientes atributos:

- Nodos de cómputo:
  - Atributo *comm_bw*: representa el ancho de banda máximo (expresado en bytes por segundo) para las comunicaciones con otros nodos que es capaz de soportar. En el instante de la simulación en el que el volumen de comunicaciones que esté teniendo lugar en el nodo sea mayor que el valor de este atributo, las tareas MPI involucradas sufrirán contención.
- *Jobs* MPI:
  - Atributo *comm_vol*: representa el volumen de comunicaciones (expresado en bytes) que tienen lugar durante la ejecución de cada tarea lanzada por el *job*. Al ser un atributo relativo al *job*, todas las tareas lanzadas por el mismo *job* tendrán el mismo volumen de comunicaciones. Un valor de, por ejemplo, 10 Mbytes en este atributo, significará que se transferirán 10 Mbytes de datos entre cada tarea del *job* y el resto de tareas del mismo *job* que estén ejecutándose en un nodo diferente.
  - Atributo *T_compute*: es un valor entre 0 y 1 que indica el factor de tiempo de su ejecución que las tareas del *job* dedicarán a realizar operaciones. El factor restante (1 - T_compute), será el factor de tiempo que las tareas dedicarán a comunicarse con otras. Este atributo será utilizado para realizar el cálculo de la contención por comunicaciones para cada tarea.

Por lo tanto, si para cada tarea que esté ejecutándose en un nodo existe otra tarea lanzada por el mismo *job* ejecutándose en otro nodo con la que se comunique, se acumulará su valor de volumen de comunicaciones. Si este valor agregado supera el ancho de banda de comunicaciones de dicho nodo, todas las tareas MPI que estén siendo ejecutadas en ese nodo y que se comuniquen a través de la red de interconexión con tareas de otros nodos sufrirán contención. Al contrario que como sucede con la contención de memoria, no se ha implementado un modelo de cálculo práctico para las tareas que sufren contención adopten un comportamiento realista, sino que la contención por comunicaciones se representa mediante un factor arbitrario que se utilizará para penalizar los tiempos de ejecución de las tareas que la sufran. Una implementación realista tendría en cuenta el volumen de congestión del

nodo en los casos en los que se den contención, ya que no sería lo mismo superar ligeramente el límite de ancho de banda del nodo que sobrepasarlo ampliamente (la penalización en los tiempos de ejecución de las tareas sería mucho mayor en este último caso).

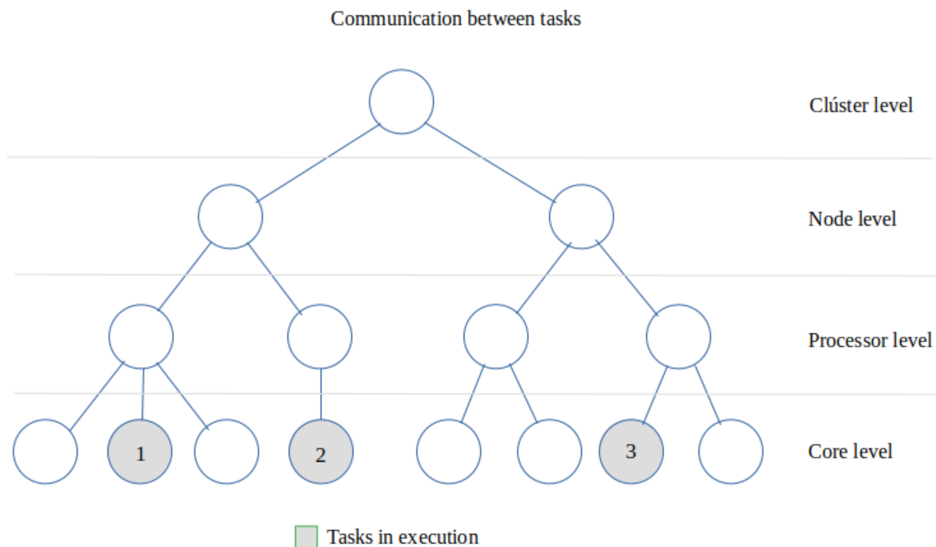

Figura 3.4: Planificacion de tareas MPI lanzadas por un mismo *job*

Según el escenario de ejecución planteado en la figura 3.4 en el que un *job* MPI lanza tres tareas de las cuales dos se planifican en un nodo y una en el otro nodo, se obtendría que solamente existe comunicación a través de la red de interconexión que conecta ambos nodos entre la tarea 3 y las tareas 1 y 2. Entre las dos primeras, al haber sido planificadas en el mismo nodo, no existiría comunicación explícita por la red, sino que ésta se llevará a cabo mediante la memoria compartida.

Sin embargo, sí que se ha tenido en cuenta para la simulación de la contención que las comunicaciones suelen representar en la ejecución de las tareas un porcentaje de instrucciones muy pequeño en comparación con la cantidad de operaciones que realiza. Para ello es para lo que se utiliza el atributo *T_compute*. Dada la situación actual, la contención que sufra una tarea MPI puede deberse al uso de la memoria, a las comunicaciones o a ambas, por lo que ha de ser calculada teniendo en cuenta ambos tipos de contención. De esta forma, el atributo *T_compute* y su complementario, que podría entenderse como *T_comm*, se utilizan para ponderar la influencia de cada tipo de contención sobre la contención final que sufren las tareas MPI, quedando el cálculo de la siguiente forma:

$$S_f = (T_compute \cdot S_c) + (T_communication \cdot S_m)$$

- S_f: valor final del *speedup* a aplicar sobre el tiempo de ejecución de la tarea.
- *T_compute*: representa el factor de tiempo que la tarea dedica a ejecutar operaciones.

Aparece definido como parámetro de las tareas y es indicado a través del fichero de entrada relativo a la carga de trabajo (*workload.json*).

- S_c: *speedup* que representa el factor de penalización sufrido por la contención por comunicaciones. Es un valor arbitrario entre 0 y 1.

- *T_communication*: representa el factor de tiempo que la tarea dedica a las comunicaciones. Es el valor complementario a *T_compute* que se obtiene mediante la resta $1 - T_compute$.

- S_m: valor de *speedup* que indica el factor de penalización sufrido por la contención de memoria. Es calculado a través de un cálculo realista.

Este modelo de cálculo es válido para ambos tipos de *jobs*, tengan comunicación entre sus tareas o no, ya que al estar ponderada la influencia de cada *speedup* sobre el *speedup* final, podría anularse, para los *jobs* secuenciales, la parte de la operación correspondiente a las comunicaciones incorporándoles el atributo *T_compute* e indicando que el porcentaje de tiempo que dedican a realizar operaciones es del 100 % (*T_compute* a valor uno).

3.4.4. Liberación de recursos

Todavía falta de añadir un matiz a la implementación del simulador, que tiene que ver con la liberación de recursos: el momento en el que una tarea finaliza su ejecución y está lista para liberar los recursos que ha utilizado para que otra tarea nueva pueda comenzar a ejecutarse después de ella. En el caso de las tareas lanzadas por *jobs* secuenciales, estas liberarán los recursos empleados en el momento en el que finalicen sus respectivas ejecuciones, independientemente del estado en el que se encuentren el resto de tareas lanzadas por el mismo *job*.

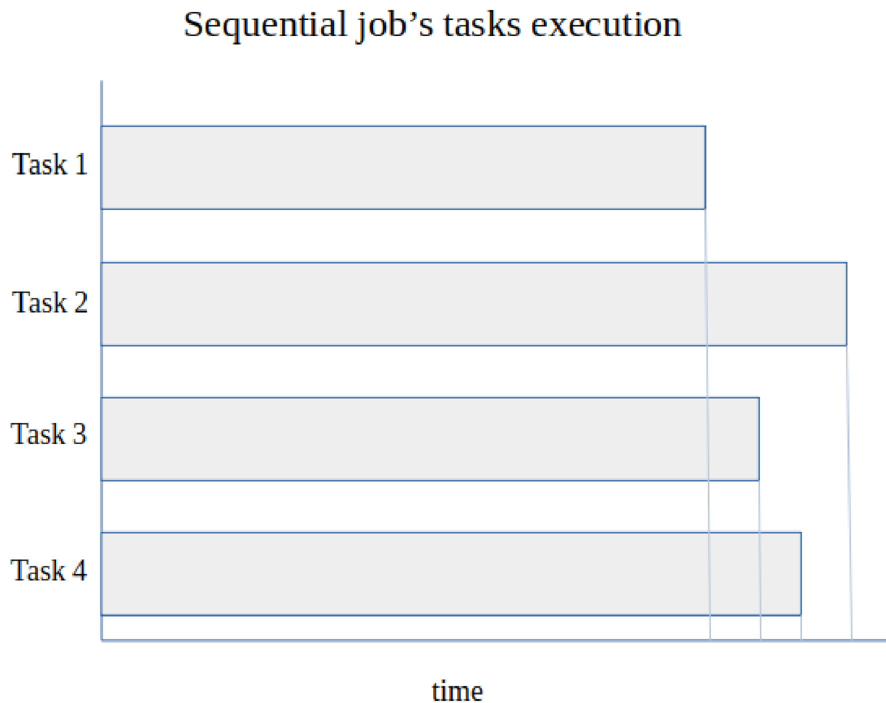

Figura 3.5: Liberación de recursos por parte de las tareas de un *job* secuencial

A pesar de que las tareas lanzadas por el mismo *job* tengan diferentes tiempos de ejecución y finalicen en diferentes instantes de tiempo, la liberación de los recursos utilizados la llevan a cabo en cuanto completen su ejecución (como se observa en la figura 3.5) sin importar el estado del resto de tareas del *job*.

En cambio, esto varía cuando las tareas que finalizan su ejecución y deben liberar los recursos son de tipo MPI. En estos casos, no se liberarán los recursos utilizados hasta que todas las tareas del mismo *job* hayan finalizado su ejecución. Es decir, los recursos utilizados por las tareas lanzadas por un *job* MPI se quedarán bloqueados hasta que todas las tareas hayan finalizado sus respectivas ejecuciones, momento en el cual todos los recursos utilizados por dicho *job* serán liberados de golpe.

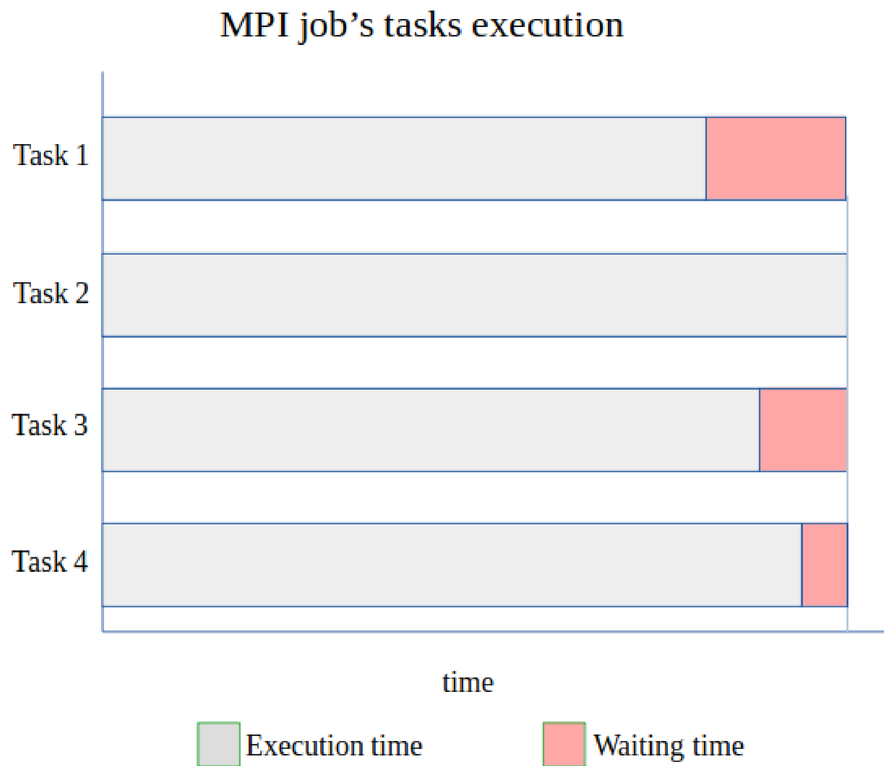

Figura 3.6: Liberación de recursos por parte de las tareas de un *job* MPI

Como se observa en la figura 3.6, las tareas no liberan recursos hasta que la más lenta de las que haya lanzado el *job* finaliza su ejecución. Para ello, será necesario llevar un registro de la tarea más lenta de cada *job* MPI. Con los valores iniciales de los parámetros de ejecución de cada tarea no sería posible saberlo dado que, como se ha mencionado en el apartado anterior, existen varios factores que pueden provocar que la tarea más lenta de un *job* MPI vaya variando durante la propia ejecución del job.

Para llevar a cabo esta sincronización a la hora de liberar los recursos utilizados, se emplea el atributo *pending_tasks* de las tareas MPI descrito en el apartado 3.2. Este atributo define el número de tareas de un *job* MPI que todavía no han finalizado su ejecución.

Por tanto, cuando finalice una tarea MPI y quiera liberar los recursos utilizados, tendrá que comprobar si es la última tarea pendiente del *job* que la ha lanzado analizando su atributo *pending_tasks*.

- Si *pending_tasks* es mayor que uno, significa que, además de la propia tarea que acaba de finalizar, existen otras tareas lanzadas por el mismo *job* que todavía no han finalizado, por lo que la tarea no llevará a cabo ninguna operación y no será liberado ningún recurso.
- Si *pending_tasks* es igual a uno, significa que la tarea que acaba de completar su ejecu-

ción es la última tarea del *job* que quedaba pendiente de finalizar. Por tanto, tendrán que liberarse todos los recursos utilizados por el *job*. Para ello, se recorrerán todos los cores de la plataforma en busca de aquellos que estén siendo ocupados por tareas del mismo *job* y liberando los recursos utilizados por cada tarea una a una. Para saber qué cores estaban siendo ocupados por tareas MPI del mismo *job*, se ha utilizado el atributo *job_id* de las tareas, que indica el *job* que las ha lanzado.

Capítulo 4

Evaluación de la implementación

En este capítulo se describirán los distintos experimentos que se han realizado con la herramienta IRMASim para probar su funcionamiento bajo todas las posibles situaciones que puedan darse durante la planificación y ejecución de todo tipo de *jobs*. Estos experimentos tienen como objetivo verificar el comportamiento de las funcionalidades implementadas que se han descrito en el capítulo 3.

4.1. Metodología de las pruebas realizadas

La definición y ejecución de experimentos es fundamental para comprobar si las aportaciones realizadas en el capítulo previo contribuyen a mejorar el software para su uso en entornos más realistas. Para confirmar el correcto comportamiento de cada una de las aportaciones descritas, se han dividido los experimentos en dos partes: una primera parte en la que se han creado distintos entornos de simulación específicos para centrar el análisis en aspectos concretos del simulador y facilitar así su estudio; y una segunda parte en la que se lleva a cabo un experimento más complejo que nos permite observar el comportamiento de la herramienta en una situación más realista.

En el entorno de simulación y, por tanto, también en las pruebas realizadas, influyen principalmente dos factores que serán modificados para adaptarlos a las necesidades de cada experimento: la carga de trabajo y la plataforma de ejecución.

- La carga de trabajo será distinta para cada experimento según el objetivo concreto del mismo. Como se ha explicado previamente, las características de los *jobs* que se deban ejecutar serán diferentes de acuerdo a la situación específica que se busque en cada prueba. De esta forma, se podrán ir analizando las funcionalidades una a una facilitando así su observación.

En cuanto a los parámetros de ejecución, todos los *jobs* del primer grupo de pruebas serán lanzados en el sistema en el instante cero de la simulación y, por simplicidad, todas las tareas lanzadas se ejecutarán en el mismo número de ciclos, de forma que se facilite controlar qué es lo que sucede en cada instante de la ejecución. Sin embargo, sí que variará, en función del objetivo de cada experimento, el volumen de comunicaciones de las tareas MPI, que será indicado en la explicación del desarrollo de cada prueba cuando sea relevante.

En el caso del experimento más complejo, los parámetros de ejecución tendrán unos valores pseudoaleatorios ya que su objetivo es el análisis de los resultados obtenidos a nivel global y no el estudio detallado de lo que sucede durante la ejecución.

- La plataforma de ejecución (figura 4.1), por su parte, será común para la primera parte las pruebas que se realicen en las que, como se ha explicado anteriormente, se analizará cada funcionalidad añadida de una en una. De esta forma, resultará más sencillo familiarizarse con ella y favorecerá el análisis de cada experimento. La plataforma utilizada consta de un cluster con dos nodos, con dos procesadores por nodo y cuatro cores por procesador.

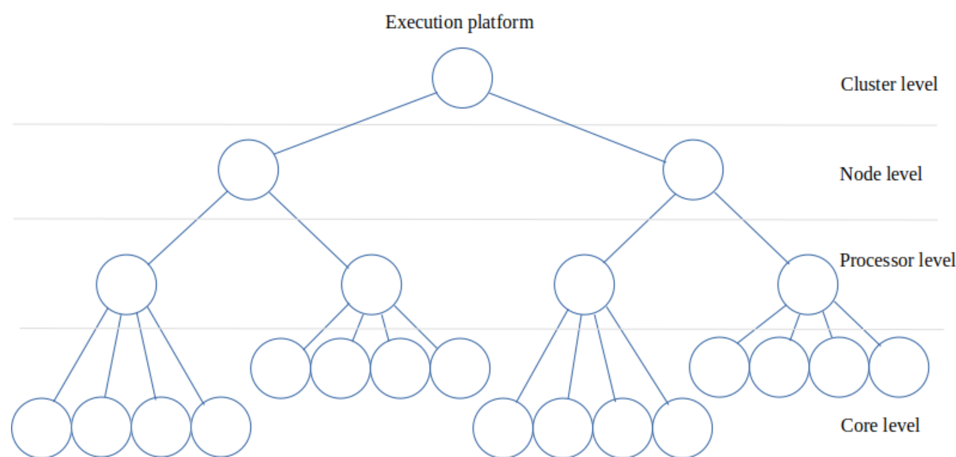

Figura 4.1: Modelo de la plataforma de ejecución utilizada para las pruebas experimentales

Para la segunda parte de las pruebas, la del experimento más complejo, se ha diseñado una plataforma de ejecución formada por treinta nodos y el objetivo de este experimento será comparar la ejecución de una misma carga de trabajo aplicando y sin aplicar el modelo de comunicación desarrollado y observar las diferencias en los resultados obtenidos.

Entre las características más relevantes de la plataforma de ejecución, cabe destacar el ancho de banda de los nodos para las comunicaciones, que será de 125 MBytes/s (lo cual equivaldría a una red Ethernet a 1 Gbps), la frecuencia de trabajo de los cores de los procesadores, que tendrá un valor de 2 GHz, y los dos paramétros relacionados con la memoria de los procesadores, el ancho de banda y la capacidad de la memoria, que serán ambos lo suficientemente

grandes como para que no influyan en las simulaciones, ya que no es objeto de estudio de este trabajo.

Todas las pruebas realizadas se han llevado a cabo siguiendo el mismo método de desarrollo que incluye los siguiente pasos:

- En primer lugar, se debe definir el objetivo concreto del experimento, es decir, el comportamiento específico del simulador que se desea analizar y comprobar. Teniendo claro cuál es la funcionalidad que se desea estudiar, el diseño de la carga de trabajo a ejecutar para observar dicho comportamiento se hace más sencillo ya que se tienen claras de antemano las características que deben tener los *jobs* para que se den esas situaciones.

  De manera introductoria, ya que serán explicados y detallados en el siguiente apartado, los objetivos concretos que se van a analizar en este trabajo a través de los experimentos son:

  - Correcta planificación de los *jobs*.
  - Correcta simulación de la comunicación entre tareas MPI.
  - Simulación de la contención por comunicaciones.
  - Sincronización entre tareas MPI a la hora de liberar recursos.
  - Correcto comportamiento en una situación más realista.

- Con el objetivo del experimento fijado, el siguiente paso es definir la traza, es decir, la carga de trabajo formada por los *jobs* a ejecutar, que van a generar el escenario que se desea analizar. Se utilizará una traza diferente en cada caso ya que cada objetivo requiere de unas características de la carga de trabajo específicas y, de esta forma, se conseguirá que en cada experimento sólo se observe como suceso relevante el comportamiento en cuestión que se esté analizando.

  En todos los casos, la traza estará formado por un número de *jobs* reducido que permitirá tener un control total sobre el estado de la simulación y sobre todo lo que ocurra durante las fases de planificación, ejecución y liberación de recursos por parte de las tareas lanzadas.

- Antes de llevar a cabo la ejecución de la traza, se realizará un breve análisis teórico del comportamiento que debería adoptar el simulador ante la carga de trabajo indicada para, una vez se lleve a cabo la simulación, contrastar los datos experimentales con el estudio previo que se había realizado y verificar así si el comportamiento adoptado por el simulador es el esperado.

- Después de realizar la ejecución de la traza, se trasladarán los datos obtenidos a una gráfica que refleje el comportamiento del simulador y se estudiará si su funcionamiento real se corresponde con el esperado.

4.1.1. Lectura de la información de las ejecuciones en IRMASim

Para tomar los datos de las ejecuciones de los *workload*, se ha configurado la herramienta IRMASim para presentar la información al usuario de la siguiente manera:

- Plataforma de ejecución:

  Ofrece información sobre el número total de *clusters*, nodos, procesadores y *cores* del sistema.

- *Workload*:

  Proporciona información sobre la cola de trabajos inicial. Cada elemento de esta cola podrá ser un *job* de tipo secuencial, para los cuales se indica el número de tareas que lanza, o una tarea MPI, para la cual se indica el identificador del *job* que la ha lanzado.

- Planificación:

  Para cada unidad planificable, es decir, un *job* secuencial o una tarea MPI, se indica el core o cores en los que ha sido planificada y, por tanto, los recursos que ocupará durante su ejecución.

- Comienzo y finalización de la ejecución de cada *job*:

  Para cada *job* se indica el instante de tiempo en el que comienza a ejecutarse y el instante en el que se completa su ejecución acompañado de los recursos que libera.

- Resultados de la simulación:

  Se escribe en el fichero *Statistics.py* la información relacionada con el consumo de energía, el *EDP*, el tiempo de la ejecución del *workload* y el volumen de información transferido a través de la red de interconexión.

4.2. Experimentos

En esta sección se describen los experimentos desarrollados para cada uno de los objetivos mencionados en el apartado anterior.

4.2.1. Planificación de los *jobs*

El objetivo de esta prueba experimental es comprobar, por un lado, que todas las tareas lanzadas por un mismo *job* secuencial se planifican y ejecutan en el mismo nodo y, por otro lado, que aquellas tareas que hayan sido lanzadas por *jobs* de tipo MPI pueden ser planificadas en cualquier core disponible de la plataforma. Además, se comprobará también

que los *jobs* solamente se planifican cuando existen recursos disponibles suficientes para que puedan ejecutarse y que, en caso de no ser así, se bloquearán esperando a que otros recursos sean liberados para que puedan comenzar su ejecución.

Para ello, se han diseñado tres trazas diferentes. Al ser el proceso de planificación el objeto de estudio de este apartado, el volumen de comunicaciones entre las tareas MPI es indiferente, por lo que se le ha asignado en todas las trazas un valor pequeño a dicho parámetro que no genere contención para que no influya en la simulación.

Además, para este tipo de pruebas en específico, se ha configurado la herramienta IRMASim para que, una vez se obtengan los cores disponibles en los que poder planificar una determinada tarea, el core en el que se vaya a ejecutar dicha tarea se escoja de forma aleatoria para confirmar que la planificación se hace de forma correcta bajo cualquier situación. Para otras pruebas, el hecho de que los cores en los que ejecutar las tareas se vayan seleccionando en orden nos ayudará a tener un mayor control sobre el entorno de ejecución y nos facilitará forzar determinadas situaciones de interés pero, para esta prueba en concreto, este punto de aleatoriedad ayudará a verificar la robustez del proceso de planificación.

La primera de las trazas (figura 4.2) está formada por dos *jobs* secueciales que lanzan seis y cuatro tareas respectivamente y un *job* MPI que lanza otras seis tareas.

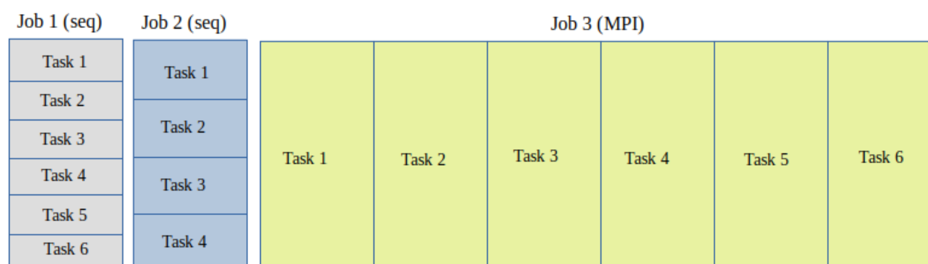

Figura 4.2: *Jobs* que forman la primera traza del experimento de planificación

El resultado que se debería obtener de la planificación de dicho conjunto de *jobs* es que todos pueden planificarse en el instante en el que llegan ya que existen recursos suficientes para ejecutarlos todos y, además la distribución de los cores permite ejecutarlos de forma paralela. Con esta traza, deberíamos observar que todas las tareas lanzadas por el *job* 1 se planifican en el mismo nodo, al igual que todas las tareas del *job* 2 se planificarán en el otro nodo disponible. Finalmente, las tareas lanzadas por el *job* 3 serán planificadas en cualquier core siempre que esté disponible indistintamente del nodo al que pertenezca.

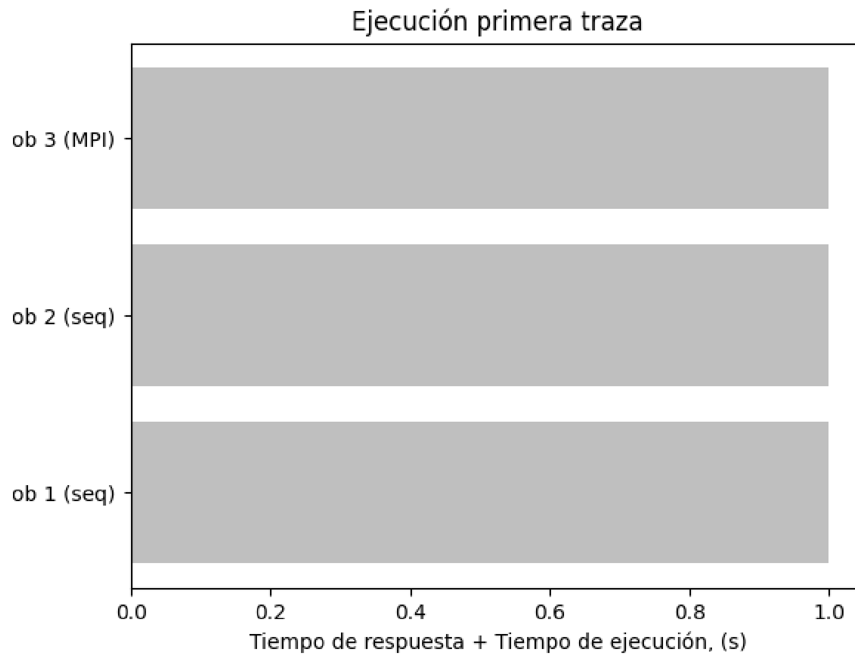

Figura 4.3: Tiempo de ejecución de la traza de la figura 4.2

Como vemos en la gráfica 4.3, todos los *jobs* son planificados en el instante cero de la ejecución. Las tareas lanzadas por el *job* 1, de tipo secuencial, son planificadas y se ejecutan todas en cores del primer nodo, ocupando en total seis de ellos. Esto fuerza que las cuatro tareas lanzadas por el *job* 2, también secuencial, se ejecuten en el segundo nodo, ya que el primero no dispone de recursos libres suficientes. En cambio, el último *job*, de tipo MPI, es planificado en cores pertenecientes a ambos nodos, pudiéndose de esta forma ejecutar los tres jobs paralelamente y consiguiendo un *makespan* de 1 segundo.

La segunda traza (figura 4.4) cuya ejecución se va a analizar en esta primera fase de las pruebas experimentales, está formada solamente por tres jobs de tipo secuencial de los cuales dos de ellos lanzan seis tareas y el tercero lanza cuatro..

Job 1 (seq)	Job 2 (seq)	Job 3 (seq)
Task 1	Task 1	Task 1
Task 2	Task 2	Task 2
Task 3	Task 3	Task 3
Task 4	Task 4	Task 4
Task 5	Task 5	
Task 6	Task 6	

Figura 4.4: *Jobs* que forman la segunda traza del experimento de planificación

Esta traza está diseñada para verificar que las tareas lanzadas por el mismo *job* de tipo secuencial se ejecutan siempre en el mismo nodo. Para ello, se pretende conseguir una situación en la que un *job* secuencial tenga recursos suficientes para ejecutarse pero estos estén repartidos en ambos nodos, con lo que se comprobará si el *job* espera a que se liberen los recursos necesarios en uno de los dos nodos, adoptando de esta forma un comportamiento correcto, o se ejecuta utilizando distintos recursos de cada nodo.

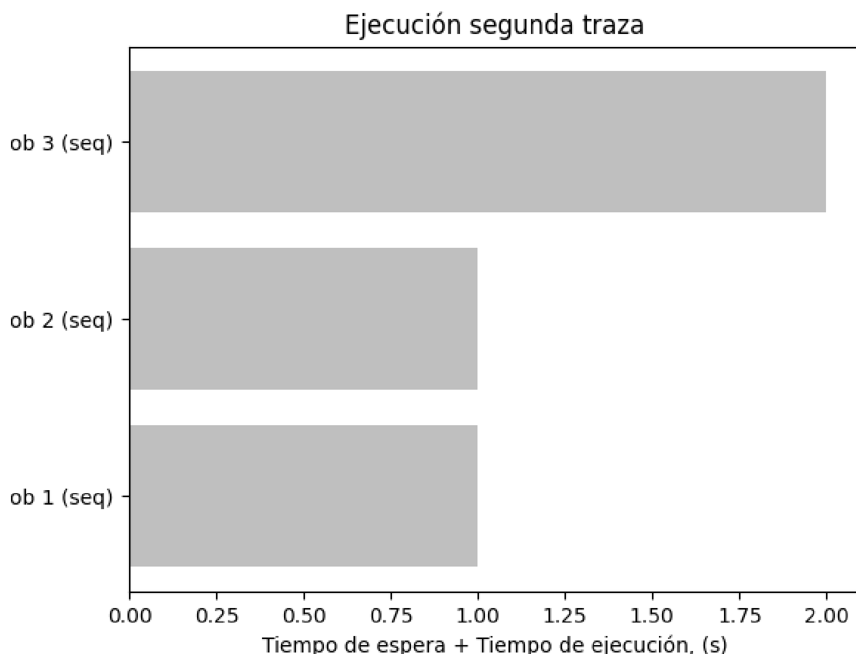

Figura 4.5: Tiempos de ejecución de la traza de la figura 4.4

El *job* 1 se planifica en el segundo nodo, utilizando seis de los ocho cores disponibles. El *job* 2, por su parte, se planifica en el primer nodo, utilizando también seis de los ocho cores libres. Quedan disponibles en el cluster un número de cores igual al número de tareas que lanza el último *job* (cuatro). Sin embargo, estos cuatro cores se encuentran dos en cada nodo, por lo que, como refleja la gráfica 4.5, el tercer *job* esperará a que uno de los dos primeros *jobs* finalice su ejecución para que libere los recursos que haya utilizado y este último pueda comenzar a ejecutarse, planificándose únicamente en cores del mismo nodo y comenzando su ejecución en el instante 1.

Para finalizar, la última traza (figura 4.6) que se ejecutará para el estudio de este primer objetivo está formada por dos *jobs* MPI que lanzan respectivamente doce y cuatro tareas y un *job* de tipo secuencial que lanza dos tareas.

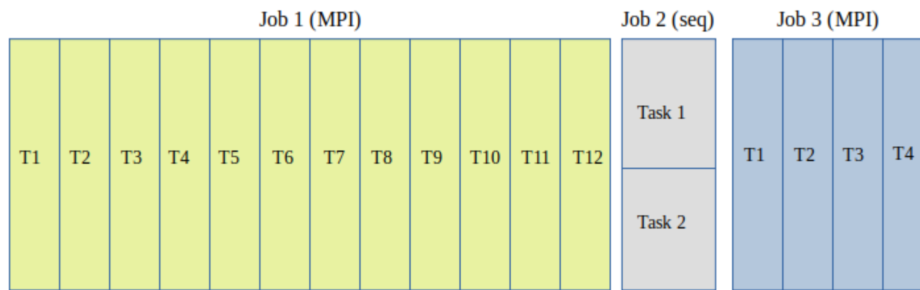

Figura 4.6: *Jobs* que forman la tercera traza del experimento de planificación

Con esta traza lo que se pretende es forzar una situación similar a la anterior, en la que esta vez será un *job* de tipo MPI el que se tendría que ver obligado a bloquear su ejecución a la espera de que otro *job* en ejecución libere los recursos empleados. Para ello, se ha diseñado la traza para que haya un momento durante la ejecución en el que se quiera planificar el *job* 3 (de tipo MPI) pero sólo estén disponibles dos cores en todo el cluster. Ante esta situación, este *job* debería esperar, al igual que en el caso anterior, a que uno de los *jobs* en ejecución libere los recursos utilizados para poder ser planificado, dado que la decisión que se ha tomado durante la implementación es que en el instante en el que se vaya a planificar cualquier tipo de *job*, deben exisitir los recursos necesarios para ejecutar las tareas que lance.

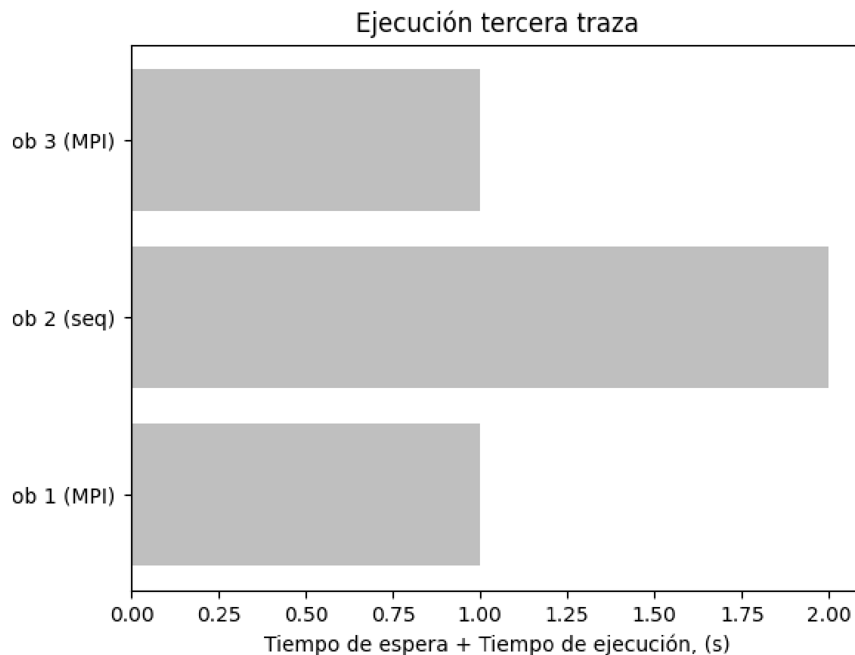

Figura 4.7: Tiempos de ejecución de la traza de la figura 4.6

En este caso, el comprtamiento ha sido el siguiente: los *jobs* 1 y 3 se han planificado en el

instante de su llegada, ocupando entre ambos un total de seis cores, pero el *job* 2 (de tipo MPI), que lanzaba doce tareas, no se ha podido planificar hasta que uno de los dos anteriores (en este caso, los dos) haya finalizado, ya que el número de cores que había libres (diez) era menor que el número de tareas lanzadas (doce).

4.2.2. Comunicación entre tareas MPI

Con este experimento se pretende comprobar que la simulación de la comunicación entre tareas lanzadas por un mismo *job* MPI se lleva a cabo correctamente. Para ello se deberá comprobar que cuando distintas tareas MPI lanzadas por el mismo *job* se planifican en diferentes nodos existe un volumen de información que se transfieren entre sí, mediante mensajes enviados a través de la red de interconexión que comunica ambos nodos. Además, se tendrá que garantizar también que la comunicación sólo existe dada la situación anterior y no en cualquier otro contexto de ejecución.

Para ello, se han diseñado dos trazas muy sencillas para representar dos situaciones diferentes.

La primera traza (figura 4.8) está formada por un *job* secuencial que lanza cuatro tareas y dos *jobs* MPI que lanzan cuatro y dos tareas respectivamente. Entre cada pareja de tareas que se ejecutan en nodos diferentes, existe un intercambio de 200 MBytes de información a través de la red de interconexión.

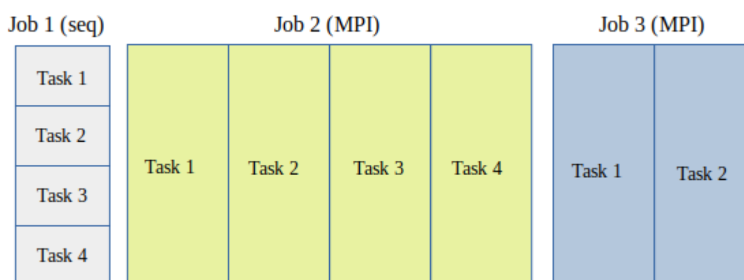

Figura 4.8: *Jobs* que forman la primera traza de los experimentos de comunicación y de contención

Esta traza ha sido diseñada para forzar una situación en la que distintas tareas lanzadas por el mismo *job* MPI sean planificadas y se ejecuten en distintos nodos, permitiendo que pueda comprobarse si se comunican entre sí. Por tanto, lo que se pretende es que uno de los dos *jobs* de tipo MPI que contiene la traza sea planificado en ambos nodos, y este será el que usaremos para estudiar si existe comunicación entre sus tareas.

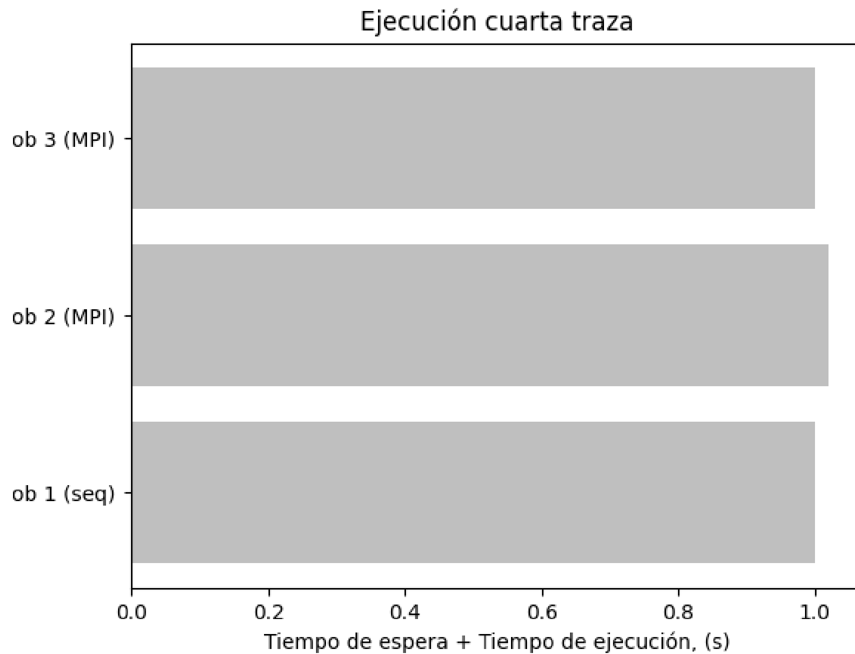

Figura 4.9: Tiempos de ejecución de la traza de la figura 4.8

Por otro lado, los resultados mostrados por el simulador son los siguientes:

Resultados obtenidos	
Energía consumida (J):	923.1
EDP (J*s):	941.56
Makespan (s):	1.02
Volumen total de comunicaciones (B):	800000000

Dado que el volumen total de comunicaciones es de 800 MBytes, se concluye que únicamente se comunican las tareas del *job* 2 que se han ejecutado en los cores cinco y siete (primer nodo) con las que se han ejecutado en los cores ocho y nueve (segundo nodo): 4 parejas por 200 MByes de información que se intercambia cada pareja obtenemos un total de 800 MBytes. Se pueden apreciar, por tanto, otros dos detalles: por un lado, una tarea MPI que ha sido lanzada por un *job* y que está ejecutándose en un nodo no se comunica con otra tarea MPI que esté ejecutándose en otro nodo pero que haya sido lanzada por un *job* diferente. Por otro lado, se observa que distintas tareas MPI que hayan sido lanzadas por el mismo *job* pero que estén ejecutándose en el mismo nodo se comunican a través de la memoria compartida y no de la red de interconexión.

En esta línea se ha diseñado la segunda traza de este experimento (figura 4.10), con la que no se va a probar nada nuevo que no se haya hecho con la traza anterior, pero se van a observar estos dos últimos comportamientos de una forma más clara. Para ello, la traza está

formada por un *job* secuencial que lanza dos tareas y dos *jobs* MPI que lanzan cuatro y dos tareas respectivamente.

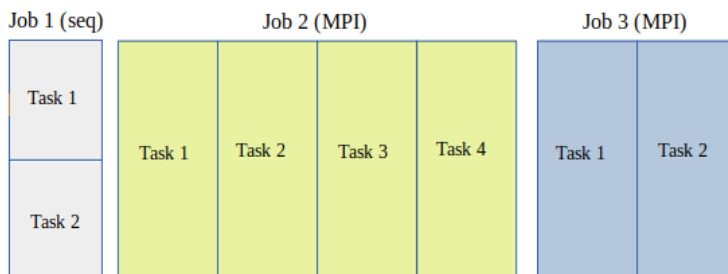

Figura 4.10: *Jobs* que forman la segunda traza del experimento de comunicación

En este caso, debería ocurrir que la planificación de todas las tareas se realizase sobre cores del primer nodo, con lo que no tendría que darse ningún tipo de comunicación entre tareas.

Resultados obtenidos	
Energía consumida (J):	905.0
EDP (J*s):	905.0
Makespan (s):	1.0
Volumen total de comunicaciones (B):	0

En efecto, los resultados arrojados por el simulador que el volumen de información transferida entre tareas a través de la red de interconexión es igual a cero. Además, cabe mencionar también que tareas MPI lanzadas por el mismo *job* se han ejecutado en procesadores diferentes pero, como se ha repetido varias veces, al pertenecer al mismo nodo, no se deben comunicar mediante paso de mensajes. Se verifica así que la simulación de la comunicación haciendo uso de la red se realiza de forma correcta.

4.2.3. Contención por comunicaciones

En este experimento se evaluará el comportamiento adoptado por tareas MPI que se comunican, es decir, lanzadas por el mismo *job* y planificadas en diferentes nodos, cuando el volumen de información que se transfieren supera el ancho de banda para las comunicaciones de alguno de los nodos.

En este experimento no se profundizará todavía sobre cuáles y cuántas tareas son las que han excedido el ancho de banda máximo (que se abordará en el siguiente, relativo a la liberación de recursos), sino que esta primera prueba ha sido diseñada para que todas las tareas excedan dicho límite y poder observar de momento el efecto de la contención sobre todas ellas.

Para ello, se ha diseñado una sola traza con la que se pretende que distintas tareas MPI lanzadas por el mismo *job* y con un volumen de comunicaciones muy alto (superior a 125 MBytes) sean planificadas en diferentes nodos para asegurar que se vaya a producir el fenómeno de contención por comunicaciones y poder observar así su efecto en los tiempos de ejecución.

Dicha traza (figura 4.8) está formada por un *job* secuencial que lanza cuatro tareas y dos *jobs* de tipo MPI que lanzan cada uno cuatro y dos tareas.

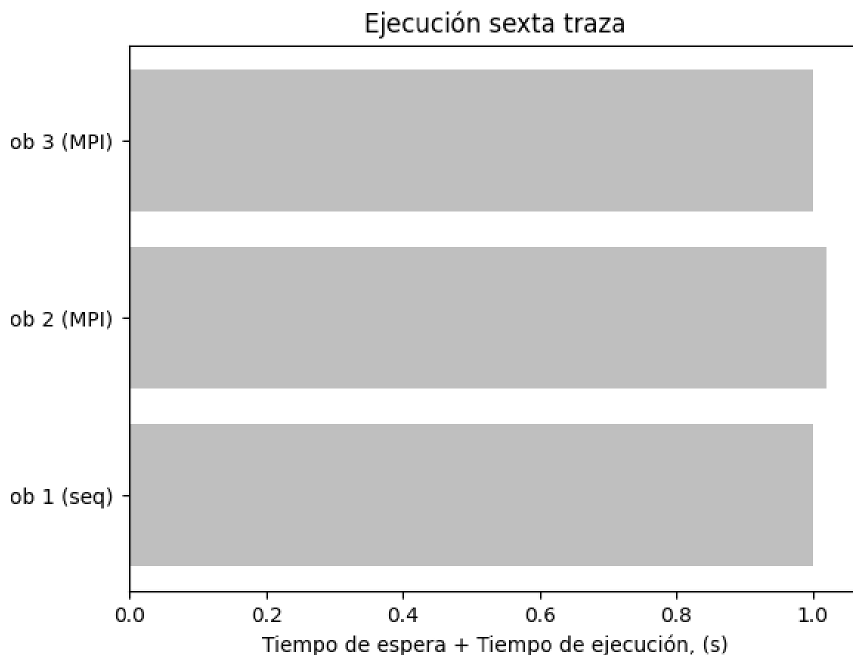

Figura 4.11: Tiempos de ejecución

Como se había previsto, existe un *job* (observando los tiempos de ejecución se deduce que es el 2) cuyas tareas han sido planificadas en diferentes nodos. Como se ha indicado antes, dos se han planificado en el primer nodo y las otras dos en el segundo. El volumen de comunicaciones, dado el valor fijado para sus correspondientes tareas, supera el ancho de banda de comunicaciones de los nodos en ambos casos, por lo que se puede apreciar el efecto de la contención (figura 4.11) reflejado en su tiempo de ejecución, el cual aumenta. Esto repercute también en el *makespan* de la simulación.

Conociendo el parámetro *T_compute* de las tareas MPI que generan la contención, podemos además comprobar si el cálculo de la contención está bien realizado (teniendo en cuenta que no se basa en un modelo realista). Sustituyendo todos los datos de la fórmula descrita en el apartado 3.4.3:

$$S_f = (T_compute \cdot S_c) + (T_communication \cdot S_m)$$

Obtenemos que:

$$S_f = (0{,}90 \cdot 1) + ((1 - 0{,}90) \cdot 0{,}8) = \frac{49}{50}$$

Por tanto, el valor de *speedup* final es de $\frac{49}{50}$, por lo que el tiempo de ejecución total del *job* número 2 teniendo en cuenta la contención por comunicaciones sería de 1.020408164 segundos. Se verifica así que el cálculo y la aplicación del *speedup* causado por la contención por comunicaciones se realiza correctamente.

4.2.4. Sincronización entre las tareas MPI en la liberación de recursos

Con esta prueba se pretende evaluar la sincronización que debiera tener lugar entre las tareas MPI al completar sus respectivas ejecuciones para liberar los recursos utilizados. La intención de este experimento es provocar un contexto de ejecución en el que haya una tarea de un *job* MPI cuyo tiempo de ejecución sea superior al resto de tareas y observar el comportamiento que adoptan.

Esta situación será provocada de la siguiente manera: se realizará una simulación en la que la traza estará formada solamente por un *job* de tipo MPI que lanzará un número de tareas suficiente como para ser planificado en diferentes nodos. Además, se modificarán los valores del parámetro relativo al ancho de banda de comunicaciones de cada nodo, provocando que sólo se genere contención en uno de los nodos. En concreto, se buscará que todas las tareas del *job* se planifiquen en el mismo nodo excepto una, que será la que sufrirá contención.

Para forzar esta situación, la traza que se ha diseñado está formada por un sólo *job* de tipo MPI que lanza nueve tareas (figura 4.12).

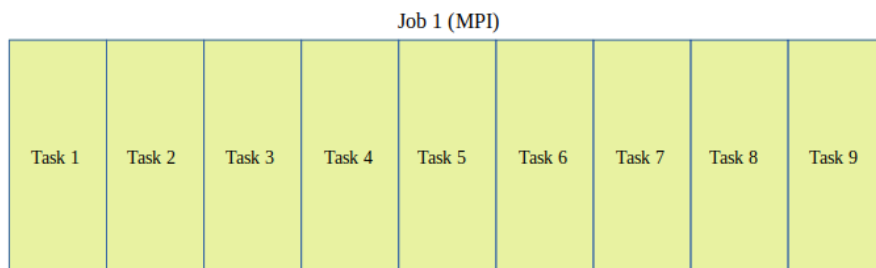

Figura 4.12: *Job* que forma la primera traza del experimento de sincronización en la liberación de recursos

El comportamiento que se espera observar es que, a pesar de que el resto de tareas no sufren contención, su tiempo de ejecución se ve incrementado de la misma forma en la que le sucede

a aquella que sí la sufre, debido a la sincronización entre las tareas a la hora de realizar la liberación de los recursos utilizados durante sus respectivas ejecuciones.

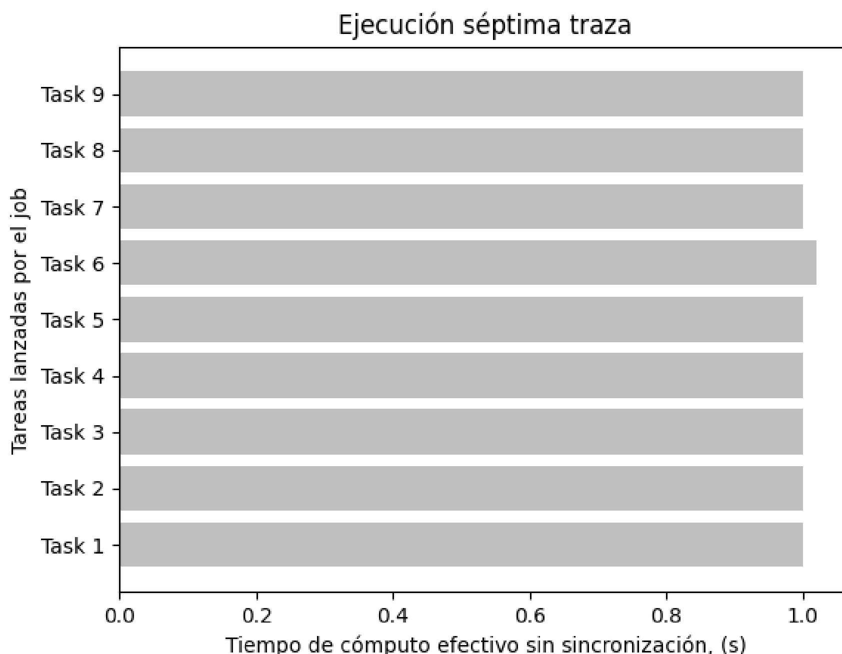

Figura 4.13: Tiempos de ejecución

Todas las tareas del *job* se planifican en el primer nodo excepto una, que se ejecutará en el segundo. Esta última sufre contención ya que el volumen de comunicaciones que soporta es muy superior que el ancho de banda de comunicaciones del nodo en el que se ejecuta (125 MBytes), por lo que su tiempo de ejecución se ve incrementado sufriendo un speedup de $\frac{49}{50}$.

Por otro lado, el volumen de transferencias del resto de tareas ejecutadas en el primer nodo no superan el ancho de banda para las comunicaciones de dicho nodo, por lo que se observa que sus tiempos de ejecución son inferiores al tiempo de ejecución de la tarea que sí que sufre contención (figura 4.13).

Sin embargo, lo que ocurre es que, debido a que cuando finalizan todavía existe otra tarea del *job* que no ha completado su ejecución, no pueden liberar los recursos y se bloquean con los recursos tomados. En el momento en el que la tarea más lenta (la que sufre contención) finalice su ejecución, todas las tareas del *job* liberarán los recursos que se hayan utilizado.

4.2.5. Experimento global

Por último, se ha desarrollado un experimento más complejo en el que se aumenta el tamaño tanto de la plataforma de ejecución como de la carga de trabajo. Los objetivos de esta

última prueba consisten, por un lado, en verificar que el simulador se comporta de manera adecuada ante situaciones más realistas y, por otro lado, evaluar el impacto del modelo de comunicación implementado en los resultados obtenidos.

Se ha diseñado una plataforma de ejecución heterogénea formada por treinta nodos de cuatro procesadores cada uno, cada uno de los cuales cuenta a la vez con cuatro cores. Los valores de los parámetros más característicos de esta plataforma son de 125 MB/s para el ancho de banda de los nodos para las comunicaciones y una frecuencia de trabajo para los *cores* que oscila entre los 2,4 y los 4,4 GHz.

Por su parte, la carga de trabajo consta de *jobs* de ambos tipos, los cuales se planificarán y se ejecutarán sin un orden establecido. En el caso de los *jobs* MPI, tanto el volumen de información como el procentaje de ejecución dedicado a las comunicaciones será variable, por lo que el impacto del modelo de comunicación será distinto en cada caso en función del volumen de información que se transmita y del procentaje de tiempo de ejecución que el *job* dedique a las comunicaciones.

Tras realizar una ejecución aplicando el modelo de comunicación y otra sin aplicarlo, los resultados obtenidos son los siguientes:

Parámetro	Con comunicaciones	Sin comunicaciones
Energía consumida (J)	25691.69	1158401.59
EDP (J*s):	627138.61	30736649.77
Makespan (s):	24.41	26.53
Volumen total de comunicaciones (B):	0	3348000000

Se observa que la contención ocasionada por las comunicaciones repercute en el tiempo de las ejecuciones. En este caso en concreto aumenta el tiempo de ejecución en un 8,7 %, lo cual provoca al mismo tiempo que aumente también el consumo de energía.

Por tanto, queda demostrada a través de estos experimentos la funcionalidad del simulador y se verifica de esta forma que se cumplen todos los objetivos planteados al inicio del proyecto.

Capítulo 5

Conclusiones y trabajos futuros

En este último capítulo se describirán los objetivos conseguidos y las conclusiones extraídas del trabajo desarrollado y se propondrán una serie de frentes de trabajo abiertos sobre el desarrollo de la herramienta IRMASim que ayudarían a mejorar el simulador.

5.1. Objetivos conseguidos

Los objetivos propuestos en este proyecto son los descritos en la sección 1.3 del documento, los cuales se han alcanzado satisfactoriamente. Estas metas conseguidas son las siguientes:

- Creación de los *jobs* de tipo MPI.

  Actualmente, el simulador es capaz de trabajar con un *workload* formado tanto por *jobs* de tipo secuencial como de tipo MPI, mientras que antes solamente se contemplaban los del primer tipo. Este avance es fundamental para la simulación de cargas de trabajo más realistas, ya que el aumento en la demanda de rendimiento unido a la evolución de las tecnologías han contribuido a la popularización de las aplicaciones distribuidas [7]. Por tanto, incluir este tipo de aplicaciones en el simulador posibilita la simulación de una carga de trabajo más real y fiel a la actualidad, en la que las aplicaciones lanzan numerosos procesos que se ejecutan de forma independiente pero que se comunican y sincronizan entre sí para trabajar de forma conjunta y logar un objetivo común.

  Además, la implementación de las fases de planificación y ejecución de este nuevo tipo de *jobs* ha servido también para corregir pequeños detalles de la versión anterior de IRMASim relacionados, principalmente, con la planificación de *jobs* secuenciales. Al no existir una red de interconexión que afectase a la ejecución de los *jobs*, se cometían ciertos errores que no afectaban a las simulaciones de las cargas de trabajo pero que hacía que los *jobs* adoptaran un comportamiento que teóricamente no deberían adoptar dada su naturaleza (explicados en el apartado 3.1).

- Comunicación y sincronización entre las tareas de un *job* MPI.

  A raíz de la creación de los *jobs* MPI en la herramienta IRMASim, se ha implementado también la simulación de la comunicación entre distintas tareas de un mismo *job* MPI. A pesar de ser un modelo de simulación muy básico, ya que simplemente se considera que la comunicación entre ellas puede producir contención y que esa contención tienen un impacto sobre el tiempo de ejecución de las tareas, es un paso muy importante. Este avance supone un comienzo en los aspectos de la comunicación entre procesos y la red de interconexión local que servirá para sentar las bases de futuras implementaciones del modelo de comunicación y que otorga la importancia que le corresponde a una parte fundamental de los *datacenters* como lo es la red de interconexión.

- Simulación de la contención por comunicaciones sufrida por tareas MPI.

  Por último, se ha conseguido implementar una primera versión de un modelo que, de la misma manera que ya se hacía en la versión anterior del simulador con la memoria de los procesadores, simula la contención sufrida por las tareas ejecutadas por un *job* MPI, esta vez debido a la saturación de la red de interconexión por un volumen de comunicaciones entre tareas superior al ancho de banda de los nodos. Aunque el cálculo de este nuevo de tipo de contención es muy básico, supone una primera aproximación a un modelo de cálculo más realista que pueda implementarse en un futuro.

Con todas estas metas alcanzadas, se consigue el objetivo final del trabajo, que es continuar mejorando las capacidades de planificación y simulación de la herramienta IRMASim para poder realizar simulaciones cada vez más fieles al mundo real. Gracias a estas aportaciones, el simulador será capaz de ejecutar cargas de trabajo ahora formadas también por aplicaciones distribuidas para las cuales se podrán simular aspectos básicos como son la comunicación entre sus procesos y la red de interconexión local del datacenter.

5.2. Trabajos futuros

Durante el desarrollo del trabajo descrito, han ido apareciendo una serie de puntos débiles que no se han considerado mejorar dados los objetivos del proyecto pero que podrían utilizarse como puntos de partida de nuevos proyectos para mejorar el simulador y hacerlo más realista. Algunas de estas ideas que podrían desarrollarse son las siguientes:

- Modificar el cálculo del valor del *speedup* originado por la contención por comunicaciones utilizando para ello un modelo realista.

  Actualmente, el valor del *speedup* que representa la contención sufrida por las tareas MPI debido a las comunicaciones es asignado de forma arbitraria, teniendo en cuenta para ello solamente que el volumen de comunicaciones entre tareas MPI lanzadas por el mismo *job* supere el ancho de banda de comunicaciones de los nodos en los que se estén ejecutando dichas tareas. Sin embargo, no se tiene en consideración si el ancho de banda

es superado por un margen muy pequeño o muy amplio, puesto que el valor asignado al *speedup* será el mismo en ambos casos. Una posible mejora sería implementar un modelo de regresión realista que tuviese en cuenta el volumen de comunicaciones de cada nodo para determinar el valor del *speedup*.

- Implementación de un algoritmo de planificación que permita que un *job* de la cola de trabajos pendientes que se vaya a planificar pero que no tenga recursos suficientes para ejecutarse no actúe de tapón para el resto de *jobs* que vienen detrás de él en la cola y evitar así un efecto convoy.

  Por decisión de implementación, los *jobs* serán planificados si existen recursos disponibles suficientes en el cluster para ejecutarlos. Esto puede provocar que un *job* que no pueda ser planificado por falta de recursos se bloquee a la espera de que se liberen recursos y tapone otros *jobs* que vengan detrás de él en la cola de trabajos que sí dispongan de recursos suficientes para sus ejecuciones.

  Existen políticas de planificación basados en algoritmos *backfilling*, los cuales permiten mejorar la utilización de los recursos disponibles de la plataforma de ejecución mediante una ejecución fuera de orden de los *jobs* de la cola de trabajos pendientes [1].

- Modificar la búsqueda del nodo en el que planificar los *jobs* de tipo secuencial para convertirla en una búsqueda *best-fit* en vez de *first-fit* como es ahora.

  En el estado actual del simulador, los *jobs* secuenciales son planificados en el primer nodo del cluster donde puedan ser ejecutados. La búsqueda *first-fit* tiene la ventaja de que es la más rápida ya que no requiere analizar todos los nodos de la plataforma, pero puede aumentar la fragmentación de la lista de cores disponibles donde planificar los *jobs*, lo cual podría empeorar la utilización de los recursos del cluster. En cambio, una búsqueda *best-fit* requerriría, en primer lugar, una selección de aquellos nodos donde poder ejecutar un determinado *job* secuencial y, en segundo lugar, escoger, de entre todos esos nodos, la opción más pequeña, es decir, el nodo con el menor número de cores disponibles. Esta opción de búsqueda mejoraría la utilización de los recursos del cluster a costa de empeorar el rendimiento del simulador, ya que el tiempo de búsqueda de la mejor opción es más elevado [2].

- Ampliar las funcionalidades del simulador para que un *job* pueda ejecutarse aún cuando el número de cores disponibles sea menor que el número de tareas que lanza, añadiendo para ello planificaciones *Round-Robin* o *CFS*, por ejemplo.

  Actualmente, los *jobs* del *workload* no son ejecutados si existen un mínimo de cores disponibles en el cluster disponibles que cumplan unas determinadas condiciones que varían en función del tipo de *job*. Este comportamiento podría ampliarse y comenzar a contemplar escenarios de ejecución en los que el número de cores disponibles para ejecutar un *job* es menor que el número de tareas que lanza y, por ejemplo, su ejecución se lleva a cabo rotando la ejecución de las distintas tareas del *job* entre los diferentes cores que haya disponibles. Esto aumentaría la complejidad del simulador pero aportaría más realismo y versatilidad.

Bibliografía

[1] Andrea C. Arpaci-Dusseau Remzi H. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2015.

[2] Parthasarathy Ranganathan Luiz André Barroso, Urs Hölzle. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Morgan and Claypool, 2018.

[3] E. Stafford and J. L. Bosque. Improving utilization of heterogeneous clusters. *J. Supercomput.*, 76(11):8787–8800, 2020.

[4] José Luis Bosque and L. P. Perez. Theoretical scalability analysis for heterogeneous clusters. In *4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004), April 19-22, 2004, Chicago, Illinois, USA*, pages 285–292. IEEE Computer Society, 2004.

[5] José Luis Bosque, Oscar David Robles, Pablo Toharia, and Luis Pastor. Evaluating scalability in heterogeneous systems. *J. Supercomput.*, 58(3):367–375, 2011.

[6] E. Stafford and J. L. Bosque. Performance and energy task migration model for heterogeneous clusters. *J. Supercomput.*, 77(9):10053–10064, 2021.

[7] T. Kindberg G. Coulouris, J. Dollimore. *Distributed Systems: Concepts and Design.* Addison-Wesley Longman, 2005.

[8] Pablo Fuentes, José Luis Bosque, Ramón Beivide, Mateo Valero, and Cyriel Minkenberg. Characterizing the communication demands of the graph500 benchmark on a commodity cluster. In *1st IEEE/ACM International Symposium on Big Data Computing, BDC 2014, London, UK, December 8-11, 2014*, pages 83–89. IEEE Computer Society, 2014.

[9] The future of computing beyond moore's law, author=John Shalf, journal=The Royal Society, volume=378, year=2020, publisher=Royal Society.

[10] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 2011.

[11] A. Herrera, M. Ibáñez, E. Stafford, and J. L. Bosque. A simulator for intelligent workload managers in heterogeneous clusters. In *21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2021, Melbourne, Australia, May 10-13, 2021*, pages 196–205.

[12] V. Luchangco M. Spear M. Herlihy, N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2020.

[13] M. Malensek P. Pacheco. *An Introduction to Parallel Programming.* Elsevier, 2021.

[14] O. D. Robles, J. L. Bosque, L. Pastor, and A. Rodríguez. Performance analysis of a CBIR system on shared-memory systems and heterogeneous clusters. In *Seventh International Workshop on Computer Architectures for Machine Perception (CAMP 2005), 4-6 July 2005, Palermo, Italy*, pages 309–314. IEEE Computer Society, 2005.

[15] Adrián Herrera Arcila et al. Hdeeprm: Deep reinforcement learning for workload management in heterogeneous clusters. 2019.

[16] Batsim Team. Batsim github repository.

[17] CloudNetSim Team. Cloudnetsim++.

[18] George S. Markomanolis Martin Quinson Mark Stillwell Frédéric Suter Augustin Degomme, Arnaud Legrand. Simulating mpi applications: The smpi approach. *IEEE Transactions on Parallel and Distributed Systems*, 28(8):2387 – 2400, 2017.

[19] Arnaud Legrand Tom Cornebize. Simulation-based optimization and sensibility analysis of mpi applications: Variability matters. *Journal of Parallel and Distributed Computing*, 166(8):111–125, 2022.