



## MANEJO DE HISTÓRICOS EN CLUSTERS

Memòria del Projecte Fi de Carrera  
d'Enginyeria en Informàtica

realitzat per

**Manuel Méndez Casado**

i dirigit per

**Porfidio Hernández Budé**

Bellaterra,.....de.....de 2008



El sotasignat, **Porfidio Hernández Budé**

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA:**

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en **Manuel Méndez Casado**.

I per tal que consti firma la present.

Signat: .....

Bellaterra, .....de.....de 2008

# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Sistemas Paralelos . . . . .	1
1.2. Planificación de Aplicaciones Paralelas en Entornos NOW . . . . .	3
1.2.1. Planificador de corto alcance: Coplanificación . . . . .	4
1.2.2. Planificador de largo alcance: Space-Sharing . . . . .	5
1.3. Descripción del Entorno de Ejecución CISNE . . . . .	8
1.3.1. El gestor de colas . . . . .	9
1.4. Predicción del Tiempo de Ejecución . . . . .	10
1.5. Estado del Arte . . . . .	11
1.5.1. Métodos basados en históricos . . . . .	11
1.5.2. Modelos y simuladores . . . . .	12
1.5.3. Híbridos . . . . .	13
1.6. Objetivos . . . . .	14
1.7. Metodología . . . . .	14
1.7.1. Distribución de tareas . . . . .	15
1.7.2. Planificación temporal de las tareas . . . . .	15
1.8. Organización de la memoria . . . . .	16
<b>2. Análisis</b>	<b>18</b>
2.1. Análisis de los Datos Disponibles . . . . .	18
2.1.1. Trazas . . . . .	18
2.1.2. Tipos de trabajos, aplicaciones y diferentes políticas . . . . .	21
2.2. Almacenamiento de los Datos . . . . .	23
2.2.1. Elección del sistema gestor de bases de datos . . . . .	24
2.3. Predecir a partir de los datos almacenados . . . . .	26
2.3.1. Árboles de decisión . . . . .	27
2.3.2. Algoritmos genéticos . . . . .	28
2.3.3. Métodos basados en características . . . . .	29
2.4. Solución Propuesta . . . . .	30
2.5. Estudio de Viabilidad . . . . .	31

2.6. Planificación Temporal de la Solución Propuesta . . . . .	34
<b>3. Diseño e Implementación de la Solución Propuesta</b>	<b>36</b>
3.1. Base de Datos: Histórico . . . . .	36
3.1.1. Tabla: trabajos . . . . .	37
3.1.2. Tabla: aplicaciones . . . . .	39
3.1.3. Tabla: cola . . . . .	40
3.1.4. Tabla: ejecución . . . . .	41
3.1.5. Tabla: nodos . . . . .	42
3.1.6. Tabla: carga local prevista . . . . .	42
3.2. Carga de la Base de Datos . . . . .	43
3.3. Herramienta de Predicción . . . . .	47
3.3.1. Conjunto de características . . . . .	48
3.3.2. Extracción de Características . . . . .	50
3.3.3. Elección del trabajo más parecido . . . . .	53
3.3.4. Sintonización . . . . .	54
3.3.5. Aceleración del proceso de cálculo . . . . .	55
<b>4. Experimentación Realizada y Resultados</b>	<b>58</b>
4.1. Entorno de Experimentación . . . . .	58
4.2. Índices de Prestaciones . . . . .	59
4.3. Metodología empleada . . . . .	60
4.4. Resultados obtenidos . . . . .	60
4.5. Análisis de los Resultados . . . . .	65
<b>5. Conclusiones</b>	<b>68</b>
5.1. Objetivos Alcanzados y no alcanzados . . . . .	68
5.2. Posibles Ampliaciones . . . . .	69
5.3. Seguimiento de la Planificación . . . . .	69
<b>Referencias</b>	<b>72</b>
<b>A. Preparación de la BD Histórico</b>	<b>74</b>
<b>B. Utilización de la Herramienta de Predicción</b>	<b>77</b>
<b>C. Sintonización</b>	<b>80</b>

# Capítulo 1

## Introducción

En este capítulo se introduce el tema de los sistemas paralelos y se presenta una clasificación donde podremos situar a los sistemas de interés en este estudio.

Seguidamente se describe lo que significa la planificación de aplicaciones en sistemas paralelos. Posteriormente se describe brevemente el entorno de ejecución CISNE desarrollando en el *Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO)*. Para su descripción en esta introducción, han sido utilizadas algunas de las figuras que aparecen en la tesis doctoral de *Mauricio Hanzich: "Combinando Space y Time-Sharing en una NOW no Dedicada"* [7].

Una vez presentado el entorno *CISNE*, se introduce el problema de la predicción del tiempo de ejecución y se contextualiza este trabajo dentro del estado del arte.

Finalmente se describen los objetivos marcados, la metodología y la organización de esta memoria.

### 1.1. Sistemas Paralelos

A menudo la resolución de problemas científicos de gran escala genera aplicaciones que necesitan más potencia de cómputo de la que puede proporcionar un ordenador convencional. Para conseguir más potencia de cómputo se puede incrementar la potencia de los procesadores utilizados. Sin embargo existen limitaciones tecnológicas que imposibilitan sobrepasar ciertos límites de rendimiento en los procesadores. Por este motivo surgió la alternativa basada en el uso de múltiples procesadores trabajando en conjunto para alcanzar la potencia de cómputo requerida. Los sistemas que resultan de esta alternativa son llamados *paralelos* y permiten realizar el *procesamiento paralelo*, es decir permiten resolver un cómputo a partir de distintos elementos de cálculo que cooperan con el objetivo de mejorar los tiempos o los costes.

Una nueva idea que modificó la forma de resolución de problemas científicos basándose en el procesamiento paralelo es la *computación distribuida*. Este es un método que consiste en que un conjunto de ordenadores conectados mediante una red

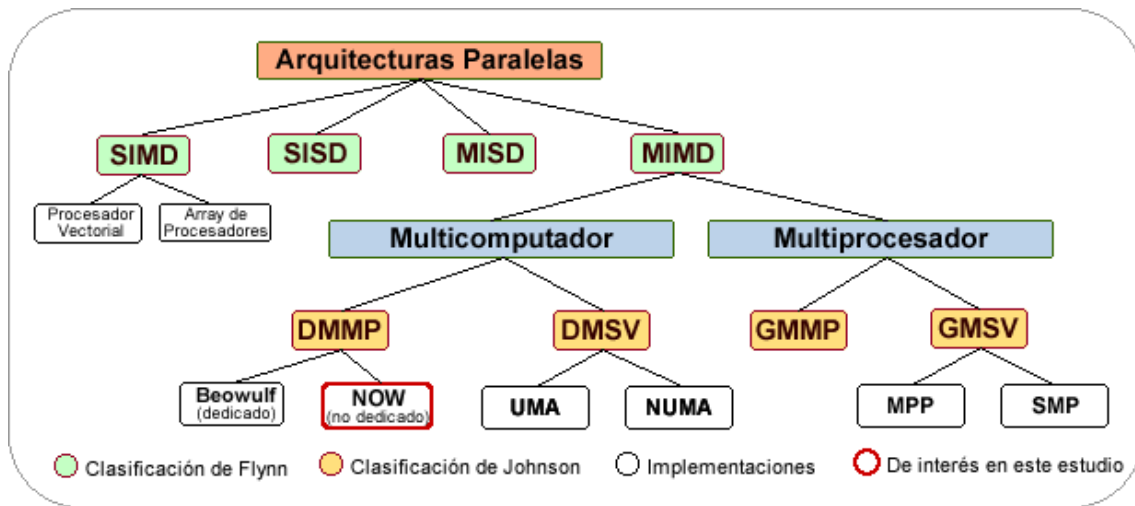


Figura 1.1: Clasificación de sistemas paralelos.

es utilizado para resolver un problema de gran envergadura. El bajo coste, comparado con equipos individuales expresamente diseñados para el computo paralelo, y la escalabilidad hacen de los sistemas distribuidos una opción muy utilizada.

Existe una amplia clasificación de los sistemas paralelos según el tipo de hardware y software empleados. Esta clasificación se muestra en la Figura 1.1 donde se observa el lugar que ocupan los sistemas de interés en este estudio, que son los entornos NOW<sup>1</sup> no dedicados.

En dicha figura se pueden observar los elementos pertenecientes a una clasificación de sistemas paralelos muy difundida que es la *clasificación de Flynn* [4, 5], la cuál divide a los sistemas paralelos en función de la multiplicidad de instrucciones y datos:

- SISD (Single Instruction - Single Data): esta organización representa a la mayoría de los ordenadores secuenciales.
- SIMD (Simple Instruction - Múltiple Data): este esquema responde tal vez más a un modelo de software (SPMD), ya que bien puede ser implementado sobre una máquina MIMD.
- MISD (Multiple Instruction - Simple Data): este esquema no posee una aplicación real.
- MIMD (Multiple Instruction - Múltiple Data): la mayoría de los sistemas multi-computador, construidos con procesadores que poseen cada uno su propio espacio de direcciones de memoria y se comunican a partir de una red, o multiprocesador donde los procesadores comparten el espacio de direcciones, se clasifican dentro de este punto. Un ordenador MIMD puede ser resumido como sigue:
  - Cada procesador ejecuta su propia secuencia de instrucciones.
  - Cada procesador trabaja en una parte distinta del problema.
  - Cada procesador comunica datos a otros procesadores.

<sup>1</sup>NOW, *Network of Workstations*.

- Los procesadores necesitan esperar a otros procesadores para acceder a los datos.

Siguiendo con la Figura 1.1, *Johnson* en [6] define dos criterios que determinan cuatro posibles clasificaciones de los sistemas paralelos. Los criterios son: el método de comunicación provisto por el sistema (paso de mensajes vs. variables compartida) y la distribución de la memoria (memoria compartida vs. memoria distribuida):

- GMSV (Global Memory - Shared Variables): definen sistemas de alto acoplamiento donde normalmente se comparten los recursos de I/O y memoria. En este caso la red de interconexión de recursos debe ser de muy alta eficiencia. Los sistemas SMP (Symmetric Multiprocessors) por su parte incorporan unos pocos procesadores (= 8), mientras que los MPP (Massively Parallel Processors) están constituidos por grandes cantidades (= 32).
- GMMP (Global Memory - Message Passing): este esquema no tiene demasiado sentido práctico, aunque sistemas como la Cray T3E lo permiten ya que la memoria compartida proporciona un medio más eficiente de compartir información que el paso de mensajes.
- DMSV (Distributed Memory - Shared Variables): estos sistemas normalmente utilizan una gran cantidad de hardware o software adicional para presentar al usuario un esquema de memoria compartida a pesar de estar físicamente distribuida.
- DMMP (Distributed Memory - Message Passing): en este caso se engloban los sistemas paralelos generados por la interconexión de ordenadores, como es el caso de los clusters en general y en particular los sistemas Beowulf (dedicados) y los sistemas NOW (no dedicados).

Los sistemas NOW no dedicados son los entornos que interesan en este trabajo. A partir de ahora siempre se hablará sobre este tipo de sistemas. Estos entornos se caracterizan por utilizar equipos cuyos recursos no son exclusivos del sistema distribuido. Pueden estar constituidos por equipos ubicados en salas de informática donde se realizan tareas docentes. Por lo tanto un equipo podrá estar ejecutando aplicaciones propias de las tareas docentes que en esa sala se estén realizando y a la vez estar ejecutando procesos de una aplicación paralela del sistema distribuido. En estos casos se respeta la capacidad de respuesta del ordenador correspondiente al usuario local.

## **1.2. Planificación de Aplicaciones Paralelas en Entornos NOW**

El problema de la planificación ha sido clave en sistemas monoprocesador y en entornos paralelos lo es más aun. Si bien se pueden aplicar algunas soluciones de los sistemas monoprocesador, hay que tener en cuenta otras consideraciones. Por

ejemplo qué tareas de una misma aplicación se ejecutan en qué nodos del conjunto asignado, o la decisión de llevar una tarea de un procesador a otro ante determinadas circunstancias de balanceo de carga. Haciendo un ejercicio de abstracción, se podría decir que la planificación ha de solucionar varios problemas relacionados con *dónde* (qué nodos) y *cuándo*, lanzar las aplicaciones paralelas que se encuentran a la espera de recursos.

Una forma de catalogar los sistemas de planificación paralela es en función de la dimensión en la que se realice el reparto de recursos, que puede ser: *espacial* o *temporal*. En el primer caso, el *espacio de procesadores* (primer nivel de planificación) se divide entre las aplicaciones paralelas en función de una política determinada, esto es conocido como *Space-Sharing*. Otra dimensión desde la que se puede plantear la planificación paralela es el tiempo. En este caso cada procesador divide su tiempo entre los procesos de las distintas aplicaciones a las que debe atender. A estos sistemas se les conoce como de *Time-Sharing*.

En el sistema donde se integra este trabajo el esquema seguido es un esquema de dos niveles. Se dispone de un planificador de largo alcance encargado de la asignación del espacio de procesadores (*Space-Sharing*) a las aplicaciones paralelas que llegan al sistema. En un segundo nivel, se tiene un sistema de planificación local que coordina los distintos procesos de una aplicación paralela que proporciona un esquema de *Time-Sharing* y *coplanificación*.

### **1.2.1. Planificador de corto alcance: Coplanificación**

Una práctica habitual en el reparto de tiempo de una máquina paralela es la *coplanificación*. La idea básica es minimizar los tiempos de espera producidos por eventos de comunicación. Se centra en conseguir que un proceso emisor de un mensaje encuentre al destinatario ejecutándose en el momento que dicho mensaje llegue a su destino.

Para realizar la coplanificación de tareas existen distintas soluciones que se basan en métodos *explícitos*, *implícitos* o *híbridos* de sincronización.

#### ***Explícitos***

En este caso el entorno posee algún método que le permite sincronizar explícitamente los nodos. De esta forma los cambios de contexto de las aplicaciones en cada nodo se realizan al mismo tiempo, generándose un cambio de contexto global en el sistema. Así es posible la coplanificación de aplicaciones paralelas ejecutando las diversas tareas de cada una dentro del mismo período de tiempo en todos los nodos. La ventaja de este tipo de esquemas es que aseguran el progreso uniforme de las aplicaciones y la coplanificación, pero este método dificulta la utilización de *Quantums* de tiempo pequeños y puede perjudicar la escalabilidad del sistema.

#### ***Implícitos***

En estos métodos, en lugar de sincronizar los nodos del sistema, se utiliza la propia comunicación entre las tareas de una aplicación para lograr la coplanificación. El tipo



de coplanificación que se aplica es entre tareas y no sobre la aplicación completa. Suelen ser más escalables que los anteriores y realizan la coplanificación sólo cuando es necesaria. Por esta razón el Quantum utilizado en cada nodo puede hasta ser diferente.

### **Híbridos**

Con estos esquemas se intenta obtener las ventajas de los métodos explícitos e implícitos utilizando la sincronización explícita para ciertos casos y dejando para otros la información obtenible de eventos de comunicación entre algunas de las tareas de la aplicación. Finalmente, decir que este es el esquema utilizado por el sistema donde integraremos nuestro trabajo, aunque basado en una coplanificación de tipo predictiva-implícita.

### **1.2.2. Planificador de largo alcance: Space-Sharing**

El planificador de largo alcance centra sus esfuerzos en dos tareas principales. Por un lado la división del espacio de procesadores, también conocida como *Space-Slicing*, y por otro la asignación a cada subconjunto de procesadores de las distintas aplicaciones de la carga paralela que hay que ejecutar, también conocida como *Planificación de Trabajos*.



Figura 1.2: Planificación de trabajos

### **Space-Slicing**

Trata de encontrar un conjunto de procesadores utilizables para lanzar aplicaciones paralelas. Existen diferentes formas de particionar el espacio de procesadores:

- *Particionamiento Estático*: las particiones son asignadas estáticamente por el administrador del sistema. Este método presenta los problemas de fragmentación interna y externa.
- *Particionamiento Variable*: las particiones se construyen teniendo en cuenta los requerimientos (cantidad de nodos) de las aplicaciones. En este método se podría mejorar el problema de la fragmentación interna pero no el de la externa.
- *Particionamiento Adaptable*: estos sistemas generan las particiones al momento de ejecutar la aplicación considerando los requerimientos de las tareas y la carga actual. Sin embargo la asignación inicial de procesadores no varía durante la ejecución de la aplicación.

- **Particionamiento Dinámico:** los esquemas de particionamiento presentados hasta aquí imponen poca o ninguna restricción sobre las aplicaciones que se ejecutan sobre él, pero esto sin embargo genera ciertas ineficiencias al momento de administrar el sistema, que surgen sobre todo de los problemas de fragmentación y de la utilización de recursos. Los sistemas de particionamiento dinámico adaptan las particiones dinámicamente a las aplicaciones. Este particionamiento se realiza expropiando procesadores o migrando tareas entre ellos.

Independientemente de la flexibilidad con la que se divida el conjunto de procesadores, debe establecerse una política que determine cómo deben seleccionarse los nodos a asignar.

### **Planificación de Trabajos**

Tan importante como determinar cómo dividir el espacio de procesadores, es el hecho de establecer el tratamiento que ha de darse a las aplicaciones que los usuarios desean ejecutar en el sistema. Este tratamiento consta de dos decisiones, la primera es cómo ordenamos los trabajos para su ejecución; y la segunda es que trabajo seleccionamos de entre los que esperan para su ejecución. Estas dos cuestiones pueden resolverse de distintas maneras según los criterios que se empleen. A continuación se describen estas políticas:

- **Políticas de ordenamiento de trabajos:** cuando un trabajo llega al sistema para su ejecución, debe ser colocado en alguna posición relativa a los demás trabajos que esperan para ser ejecutados. Para realizar este objetivo debe establecerse algún criterio de ordenamiento. A continuación se describen los criterios de ordenamiento de algunas políticas:
  - **FCFS (First Come First Served):** en este caso los trabajos son tratados en el orden en que llegan al sistema.
  - **SJF (Shortest Job First):** los trabajos se ordenan, de menos a más, en función del **tiempo de ejecución (estimado)**.
  - **LJF (Largest Job First):** los trabajos se ordenan, de más a menos, en función del **tiempo de ejecución (estimado)**.
  - **SCDF (Smallest Cumulative Demand First):** los trabajos se ordenan, de menos a más, de acuerdo al producto de la cantidad de procesadores solicitados y el **tiempo de ejecución estimado**.
  - **SNPF (Smallest Number of Processes First):** los trabajos se ejecutan en orden creciente según la cantidad de procesadores que solicitan.
  - **XFactor:** los trabajos se ordenan en función de una prioridad asignada por el factor de expansión de un trabajo en la cola. El factor de expansión se determina por el tiempo que un trabajo ha pasado en la cola.

Es posible alternar estas políticas de ordenamiento dinámicamente de manera que, en distintos momentos de tiempo, la política de ordenamiento que gobierna la cola puede variar.

- **Políticas de selección de trabajos:** con la cola de trabajos ordenada, podríamos suponer que bastaría con escoger el trabajo que se encuentra en la primera posición para ejecutarse. Esta decisión sería válida pero podría disminuir mucho el rendimiento del sistema. Esta situación se puede dar en el caso en que tengamos un trabajo de gran tamaño a la cabeza impidiendo la ejecución de otros pequeños que se encuentran esperando detrás. Por lo tanto existen políticas de *selección de trabajos* que intentan mejorar la eficiencia del sistema. Estas políticas se dividen en dos grupos:

- **Políticas bidimensionales:** Estas políticas consideran sólo el estado actual del sistema y buscan un trabajo dentro de la cola en función de este estado para satisfacer su criterio.
  - *FFit (First Fit):* con este tipo de políticas se recorre la cola de trabajos hasta encontrar el primero que quepa en el espacio de procesadores.
  - *WFit (Worst Fit):* se busca en la cola el trabajo más pequeño y por lo tanto aquel que dejará la mayor cantidad de procesadores libres.
  - *BFit (Best Fit):* se busca en la cola el trabajo que se aproxime más al espacio de procesadores libres actualmente.
- **Políticas tridimensionales:** en muchos casos el sistema de planificación exige conocer una estimación del tiempo de ejecución de las aplicaciones que se envían al sistema. Haciendo uso de esta estimación es posible, tomar decisiones en función del estado actual del sistema y tomarlas considerando una carga futura probable. Una de las técnicas que más captan la atención en la literatura es el *backfilling*. Dicho esquema se basa en el siguiente criterio: un trabajo puede ejecutarse, antes que otros que se encuentran en la cola de espera por delante, siempre y cuando no retrase el tiempo de inicio de dichos trabajos. Para implementar esta técnica normalmente cada trabajo se representa como un rectángulo cuya altura está dada por la cantidad de procesadores solicitados y su ancho determinado por el tiempo de ejecución estimado. De esta forma podemos observar en la Figura 1.3 la diferencia de planificación que se produce entre dos políticas donde una no incluye backfilling (Figura 1.3 a) y la otra si (Figura 1.3 b).

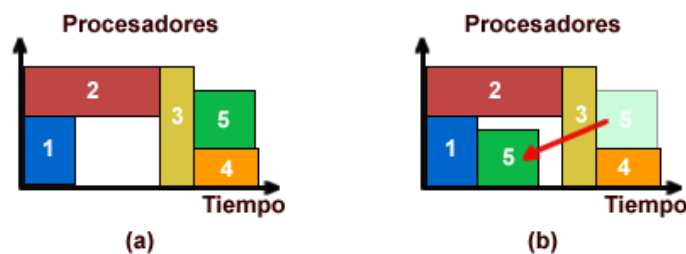


Figura 1.3: Técnica de Backfilling

En la práctica dos variantes del esquema básico son las que normalmente se aplican:

- *Backfilling EASY:* bajo esta política un trabajo siempre puede adelantarse en la cola si esto no retrasa al trabajo que se encuentra en la cabeza.

- *Backfilling Conservativo*: con este criterio un trabajo sólo puede adelantarse en la cola si no retrasa a ninguno de los trabajos que se encuentran por delante.

### 1.3. Descripción del Entorno de Ejecución CISNE

En la siguiente sección se describe los aspectos importantes, para el trabajo que aquí se presenta, del entorno de ejecución CISNE (*Cooperative & Integral Scheduling for Non-dedicated Environments*). Este entorno se encuentra en el *Departamento de Arquitectura de Computadores y Sistemas Operativos* (DACSO), y es el sistema para el que se realiza este trabajo.

Dicho entorno se construye a partir de dos subsistemas. El sistema de más alto nivel es un sistema de Space-Sharing (LoRaS) que permite acomodar la carga paralela en el cluster maximizando la utilización de recursos. El sistema de bajo nivel implementa un entorno de Time-Sharing (CSC) para lograr la coplanificación y balanceo de recursos asignados a las tareas pertenecientes a un mismo trabajo. También es responsable de preservar el rendimiento del usuario local. En la Figura 1.4 podemos apreciar la arquitectura del entorno CISNE<sup>2</sup>.

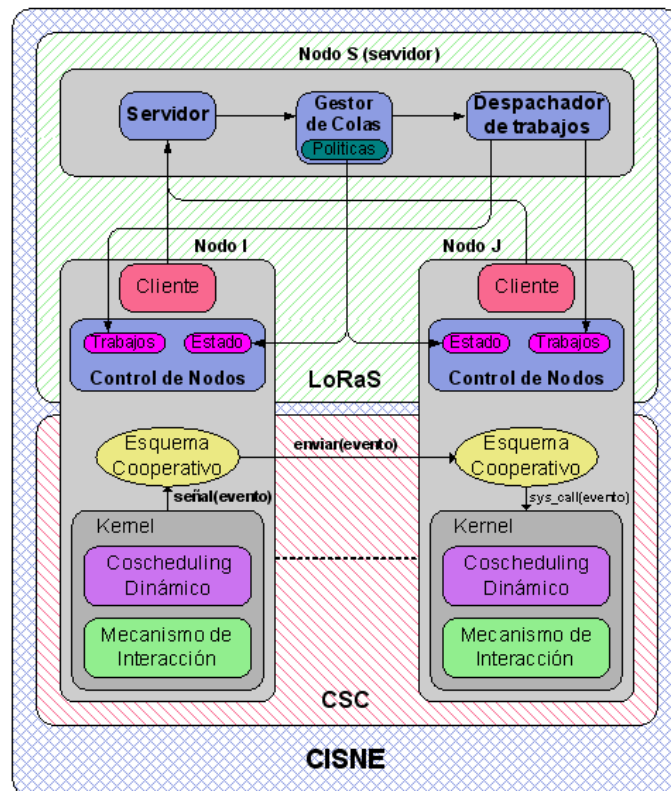


Figura 1.4: Arquitectura del sistema CISNE.

El entorno CSC tiene tres objetivos fundamentales dentro del entorno CISNE:

<sup>2</sup>Figura extraída del trabajo, *Combinando Space y Time-Sharing en una NOW no Dedicada*.

- Minimizar el efecto de las aplicaciones paralelas en la percepción del usuario local de la capacidad de respuesta de su ordenador.
- La coplanificación de las tareas pertenecientes a una misma aplicación paralela que se comunican.
- La asignación uniforme de recursos de CPU y memoria a cada tarea paralela perteneciente a un mismo trabajo a lo largo del cluster.

El sistema de LoRaS es el responsable de proporcionar un entorno de Space-Sharing en el sistema CISNE. Los objetivos de dicho sistema son:

- *Lanzar aplicaciones paralelas en un cluster de ordenadores.* Los trabajos a ejecutar se seleccionan desde una cola de trabajos a ejecutar. En cada caso se seleccionan un conjunto de nodos donde ejecutar una aplicación determinada.
- *Ser un sistema flexible y portable.* De esta forma se podrá adaptar el sistema a nuevos entornos que pueden surgir, por ejemplo, de la renovación de los nodos utilizados.
- *Soportar la utilización de recursos no homogéneos.* El entorno sobre el que se aplica el sistema puede incluir elementos heterogéneos.
- *Predecibilidad.* Para que el sistema sea útil para el usuario de aplicaciones paralelas, es importante que pueda determinar con algún grado de garantía el tiempo de espera y ejecución de una aplicación.

Una de las principales razones por las que el sistema LoRaS ha sido desarrollado es la de evaluar una serie de técnicas existentes de Space-Sharing en el entorno CSC. Por esta razón está implementado de forma que se pueden alterar algunos parámetros o la política completa de manera sencilla, así como añadir políticas nuevas. Por este motivo y porque está directamente relacionado con este trabajo, a continuación se describe el funcionamiento del gestor de colas.

### **1.3.1. El gestor de colas**

El bloque más importante de todo el sistema es, sin lugar a dudas, el gestor de colas. Esto es así debido a que éste es el módulo responsable de administrar la o las colas de trabajos, tomar las decisiones de planificación, entre otras cosas.

Este módulo se encuentra diseñado de manera tal que su objetivo fundamental es el de proporcionar un marco que facilite la toma de decisiones por parte del submódulo de *políticas*. De este modo es factible alterar el módulo de políticas de forma sencilla pudiendo éstas aprovechar las capacidades provistas por el módulo *gestor de colas*. Entre estas capacidades se encuentran: el proceso de recepción de un trabajo, el mantenimiento en su estado de las estructuras de colas, así como un esquema que almacena información sobre los trabajos en ejecución además de un esquema que provee los eventos de scheduling como eventos de tiempo o la finalización de un trabajo.

De esta manera las políticas implementadas simplemente respetan una interfaz definida para las mismas, y utilizando las capacidades del marco provisto por el módulo gestor de colas, sólo deben responsabilizarse de las cuestiones relacionadas con: el orden de los trabajos en las colas, el proceso de selección de un trabajo de la cola para ejecutarlo y la selección del conjunto de nodos más idóneos para hacer esto.

## 1.4. Predicción del Tiempo de Ejecución

Hasta ahora se ha descrito el entorno para el cual se realiza este trabajo. Si se desea más información sobre el sistema CISNE se puede obtener en [7]. Una de las metas propuestas a la finalización de este trabajo es la de encontrar una solución a la *predicción del tiempo de ejecución para aplicaciones paralelas*. Repasando lo escrito anteriormente encontramos diversas citas y referencias al tiempo de ejecución. Seguidamente se listan algunos de los motivos que hacen imprescindible que el sistema LoRaS, y por consecuencia el sistema CISNE, disponga de una herramienta de predicción del tiempo de ejecución. Los motivos son los siguientes:

- En las *Políticas de Ordenamiento de Trabajos*. Para poder implementar las políticas **SJF** (*Shortest Job First*), **LJF** (*Largest Job First*) y **SCDF** (*Smallest Cumulative Demand First*) es necesario disponer del *tiempo de ejecución estimado*, de las aplicaciones que esperan en la cola, para poder establecer un orden.
- Lo mismo ocurre con la política de **Selección de Trabajos**. La técnica de **Backfilling** necesita conocer el *tiempo de ejecución estimado* para poder ser eficiente.
- Para que el sistema sea útil para el **usuario** de aplicaciones paralelas, es importante que pueda determinar con algún grado de garantía el tiempo de espera y *ejecución* de una aplicación. Usualmente, a la suma del tiempo de espera y ejecución se le llama *turnaround*.
- Si se desea emplear un sistema de este tipo con fines económicos, es muy útil conocer anticipadamente el tiempo que el sistema tardará en dar al cliente los resultados de su aplicación.

Por lo tanto se puede decir que predecir con cierta exactitud el tiempo de ejecución es de vital importancia para el *Gestor de Colas* del entorno CISNE y por lo tanto para el propio sistema. Ahora bien, esta predicción no es una tarea fácil ya que existen multitud de factores que determinan este tiempo. Algunos de estos factores son por ejemplo:

- Qué política gestiona la cola de espera.
- En qué nodos se lanza la aplicación.
- El grado de multiprogramación de cada uno de los nodos.
- Que tipo de aplicaciones se ejecutan en cada nodo.
- CPU utilizada en cada nodo.

- Memoria utilizada en cada nodo.
- La carga del usuario local.

En definitiva, un método de predicción que intentara tener en cuenta la totalidad de los factores que afectan al tiempo de ejecución se hace intratable. Esto es debido al gran número de factores que entran en juego y al desconocimiento de algunos de ellos. Estaríamos hablando de un problema np-completo, que por otra parte, para ser útil, necesita ser resuelto en un tiempo reducido. De poco serviría que se tardara más tiempo en predecir que en ejecutar el trabajo.

Por lo tanto se trata de buscar un método que nos proporcione una solución aproximada manteniendo un compromiso aceptable con parámetros como el error cometido y la velocidad de cálculo en la predicción. A continuación se introduce la línea en la que se está trabajando, y se presentan algunos trabajos realizados al respecto.

## **1.5. Estado del Arte**

Existen diferentes maneras de predecir el tiempo de *turnaround* que intentan dar una aproximación reduciendo considerablemente la complejidad de cálculo. Algunos de estos métodos se describen a continuación, analizando las ventajas y los inconvenientes de cada uno de ellos.

### **1.5.1. Métodos basados en históricos**

Estos métodos son los más utilizados en la literatura consultada sobre predicción del tiempo de ejecución en sistemas paralelos. En trabajos como [10, 11, 8, 9] pueden estudiarse varias aproximaciones hacia la predicción del tiempo de ejecución mediante históricos.

Los métodos de predicción basados en históricos intentan imitar a la experiencia humana. Si sabemos el tiempo que tardó en ejecutarse una aplicación similar a la que queremos ejecutar en el presente, y si sabemos que las condiciones del entorno de ejecución eran similares a las actuales, se puede esperar que los tiempos de ejecución entre ambas aplicaciones sean parecidos.

Estos métodos presentan dos dificultades para su realización. Por un lado, requieren almacenar cada una de las ejecuciones que se realicen en el sistema. Por otro lado hace falta un algoritmo capaz de encontrar una aplicación y un estado del cluster parecidos en el histórico para poder ofrecer un tiempo de ejecución estimado.

En cuanto al primer obstáculo, es necesario guardar toda la información de la aplicación y toda la información relevante sobre el estado del cluster para, en un futuro, tener datos objetivos con los que comparar. Estamos hablando de una cantidad importante de información, ya que son múltiples los datos que hay que guardar para cada ejecución: de la aplicación, sobre cada nodo del cluster que interviene en su ejecución, las colas de ejecución, la carga externa, etc.

Diferentes alternativas son propuestas en los trabajos consultados. Desde la utilización de ficheros manejados a modo de base de datos, hasta la utilización de motores de base de datos. En el caso de decantarse por la segunda opción existe la posibilidad de que sea un nodo específico, perteneciente al cluster, el que albergue el servidor de base de datos, o por el contrario dedicar una máquina extra para servir las consultas del histórico.

En cuanto al algoritmo capaz de buscar estados y aplicaciones parecidas dentro del histórico se han utilizado en los trabajos consultados diversas alternativas. Algunas de estas son: árboles de decisión, algoritmos genéticos y métodos basados en características.

Aunque a continuación se describen otros métodos, una de las opciones que contempla el sistema CISNE en su diseño es la predicción mediante un histórico. Existe el esqueleto de una *interface* creada para la comunicación entre el sistema de planificación y el histórico de datos. En la Figura 1.5 se puede ver cómo se integra el histórico dentro del esquema de Space-Sharing (LoRaS)<sup>3</sup>.

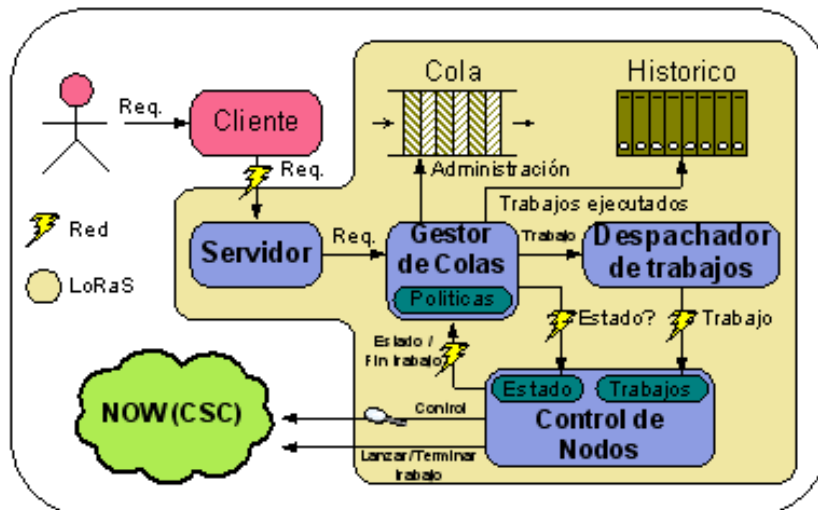


Figura 1.5: Arquitectura del esquema LoRaS.

### 1.5.2. Modelos y simuladores

Se puede decir, que un modelo analítico es un conjunto de relaciones matemáticas entre las variables del sistema, tal que, relaciona la carga con las prestaciones en base a un modelo funcional.

El modelado se basa en un conocimiento profundo del sistema, que nos permite representarlo mediante ecuaciones. En la mayoría de casos estas ecuaciones son diferenciales no lineales y algebraicas que no tienen solución analítica. No obstante, en estos casos, existen métodos que linealizan el sistema y permiten una solución analítica.

El primer paso en el estudio de un sistema es la obtención de una representación

<sup>3</sup>Figura extraída del trabajo, *Combinando Space y Time-Sharing en una NOW no Dedicada*.



adecuada del mismo, es decir, un modelo adecuado. Un buen modelo debe tener dos propiedades:

- Debe reflejar adecuadamente aquellas características del sistema que son de nuestro interés. En nuestro caso el tiempo de ejecución.
- Debe ser suficientemente sencillo para resultar manejable. Recordemos que nos interesa obtener la predicción lo antes posible

En la construcción de modelos hay que tener presente que:

- Un modelo se basa siempre en hipótesis y aproximaciones.
- Se construye con un fin específico y debe formularse para que sea útil para tal fin.
- Es un compromiso entre la sencillez y la necesidad de recoger todos los aspectos relevantes del comportamiento del sistema.

Los modelos suelen realizarse para su acoplamiento en simuladores. Podemos decir que son los "motores" de los simuladores en el sentido que proporcionan los datos que el simulador necesita para progresar.

En la literatura consultada, ver [12] se coincide en la afirmación que estos métodos son apropiados para la predicción del *tiempo de espera*.

### **1.5.3. Híbridos**

Desde el punto de vista de usuario, no sólo interesa conocer el tiempo de ejecución de la aplicación paralela, sino también el tiempo de *turnaround*, diferentes trabajos enumerados en la bibliografía cómo en el caso del trabajo de Smith y Wong [8] nos dejan claro con su experimentación que los modelos basados en históricos son adecuados en el cálculo del *tiempo de ejecución* pero no en el cálculo del *tiempo de espera en cola*. Esto se debe a que la información que el *histórico* puede tener referente al tiempo que un trabajo estuvo en cola, cómo: *posición de entrada, qué trabajos tenía por delante, qué política gobernaba la cola, etc...* no guarda relación con cómo va a evolucionar la planificación de las tareas. Por ese motivo, cuanto más grande sea la cola más error se acumula en estos casos. Por ejemplo, no sabemos cuántos trabajos serán ejecutados antes que el nuestro y estaban en una posición más retrasada en la cola.

Por otro lado un *simulador*, sí puede reproducir estos eventos que al *histórico* se le escapan, ya que se va a comportar como el sistema real.

Por estos motivos los *métodos híbridos* utilizan *simulación* en el cálculo del *tiempo de espera en cola* y los *históricos* en el cálculo del *tiempo de ejecución*.

## 1.6. Objetivos

El objetivo de este trabajo es analizar, diseñar e implementar una herramienta para predecir el tiempo de ejecución de las aplicaciones paralelas que se ejecutan en el sistema CISNE. Para ello se creará un histórico donde almacenar la información de cada ejecución realizada en el sistema. También se creará una herramienta que, utilizando los datos del histórico, sea capaz de predecir el tiempo de ejecución.

Se espera que este trabajo ayude a la implementación de ciertas políticas que necesitan utilizar una estimación del tiempo de ejecución de los trabajos existentes en la cola del sistema. Estas políticas son: **SJF** (*Shortest Job First*) ,**LJF** (*Largest Job First*) y **SCDF** (*Smallest Cumulative Demand First*) para el *ordenamiento de trabajos*; y la implementación del *backfilling* para la *selección de trabajos*.

Así mismo, se espera que la estimación del tiempo de ejecución ayude a predecir el tiempo de *turnaround*, necesario como medida de prestaciones orientada al usuario de aplicaciones paralelas.

## 1.7. Metodología

Una vez existe un objetivo que cumplir, se define en esta sección una metodología a seguir para alcanzarlo. También se introduce una primera distribución de las tareas y su planificación temporal. En el capítulo 2, una vez conocida la propuesta de solución, se presenta con más detalle el estudio de viabilidad y la planificación temporal de la solución propuesta.

Para abordar este trabajo se ha optado por la metodología en espiral. En la primera vuelta de la espiral, se definen los objetivos, el análisis, la implementación del soporte de almacenamiento y un primer prototipo de herramienta de predicción. Por último se analizan los resultados. Llegados a este punto se pueden introducir propuestas de mejora que darán lugar a una nueva vuelta de la espiral. La intención, en este trabajo, es que en cada vuelta de la espiral se mejoren los resultados de predicción hasta obtener unos resultados satisfactorios. En la figura 1.6 se intenta mostrar una imagen esquemática de esta metodología.

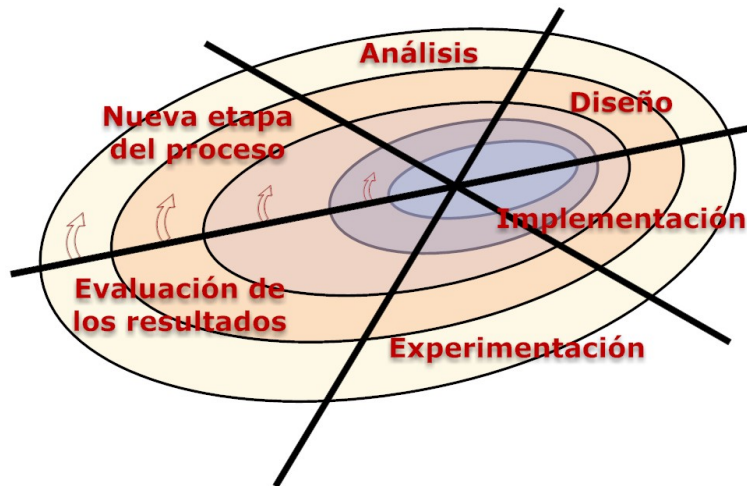


Figura 1.6: Aproximación en espiral para el desarrollo de software.

### 1.7.1. Distribución de tareas

Antes de conocer la solución propuesta, las tareas a realizar se pueden dividir en los siguientes bloques de trabajo:

1. Análisis: se debe analizar la información disponible. A partir de esta información se debe crear el histórico. Es necesario analizar la manera cómo esta información se va a almacenar en el histórico, considerando diferentes posibilidades. De igual forma, para terminar con el análisis, se deben considerar posibles maneras de predecir el tiempo de ejecución.
2. Solución propuesta: una vez analizados todos los aspectos anteriores, se configurará la solución propuesta.
3. Diseño: con la propuesta de solución, se valorarán diferentes posibilidades y se tendrán en cuenta los aspectos necesarios para conseguir minimizar el error de predicción.
4. Implementación: Teniendo en cuenta todos los aspectos del análisis y el diseño se implementará histórico y la herramienta de predicción.
5. Experimentación: Se realizarán diferentes pruebas y se estudiarán los resultados obtenidos.
6. Redacción de la memoria.

### 1.7.2. Planificación temporal de las tareas

A continuación se describe mediante un diagrama de Gantt la planificación temporal de las tareas generales. Esta planificación se verá detallada en el capítulo 2.

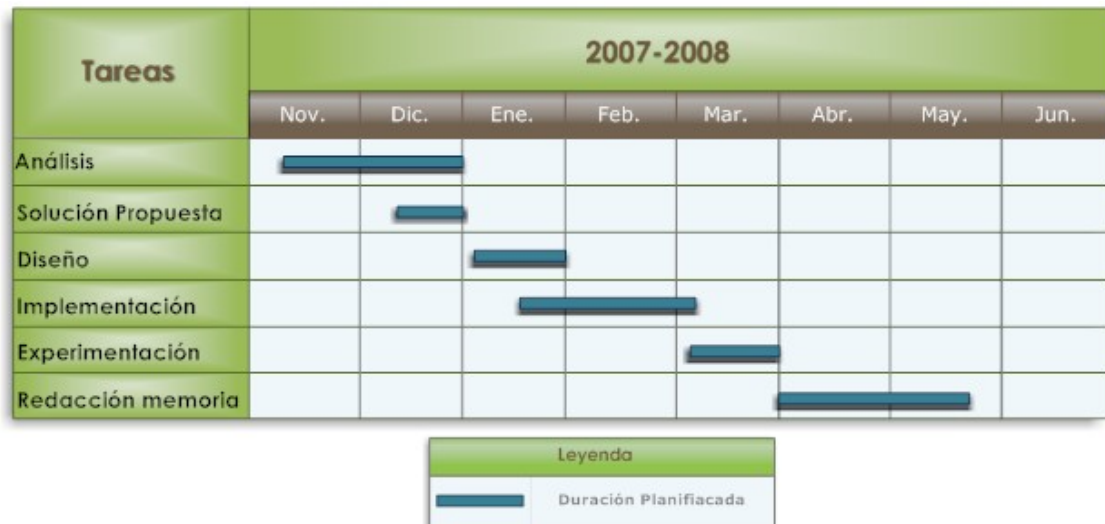


Figura 1.7: Diagrama de Gantt de las tareas generales.

## 1.8. Organización de la memoria

La memoria está organizada en cinco capítulos, más la bibliografía, donde se intenta plasmar el trabajo realizado. El contenido de cada capítulo es el siguiente:

**Capítulo 1: Introducción.** Para empezar se intenta situar al lector en el entorno para el que se realiza este trabajo. Empezando por una clasificación de los sistemas paralelos y terminando en el gestor de colas de dicho sistema. Una vez presentado el entorno se plantea el *problema de la predicción del tiempo de ejecución*, repasando en que puntos del entorno es necesaria esta estimación. Seguidamente se describe el estado del arte en esta técnica y los objetivos marcados para este trabajo. También se describe, de manera general, la metodología a seguir para conseguir los objetivos.

**Capítulo 2: Análisis.** En este capítulo se analizan las necesidades del proyecto, los datos disponibles y las posibles opciones que nos podrían servir para solucionar el almacenamiento y la predicción. A partir de aquí se propone una solución. En base a esta solución propuesta se presenta el estudio de viabilidad y una planificación temporal del trabajo.

**Capítulo 3: Diseño e Implementación de la Solución Propuesta.** En este capítulo se explica cómo hemos diseñado e implementado tanto la base de datos que almacena toda la información referente a los trabajos en el *histórico*, como la herramienta de predicción para el *tiempo de ejecución*.

**Capítulo 4: Experimentación Realizada y Resultados.** Aquí se describe el entorno de Experimentación, los índices de prestaciones utilizados para medir los resultados, la metodología empleada y los resultados obtenidos.

**Capítulo 5: Conclusiones.** En este capítulo se compara la planificación inicial con la real y los resultados obtenidos con los objetivos marcados. También se introducen posibles ampliaciones del trabajo y se valora el trabajo desde un punto de vista personal.

**Apéndices.** En los apéndices se describe cómo preparar la base de datos, cómo utilizar la herramienta de predicción y cómo sintonizarla.

# Capítulo 2

## Análisis

En este capítulo se analizan los datos disponibles sobre ejecuciones en el sistema CISNE. Se abordan los problemas del *almacenamiento* de los datos, el de la *predicción* del tiempo de ejecución de aplicaciones paralelas, y se llega a una propuesta de solución. Una vez analizada la propuesta se presenta el estudio de viabilidad y la planificación temporal del trabajo.

### 2.1. Análisis de los Datos Disponibles

Dado que el entorno CISNE ha estado en desarrollo durante la realización de este trabajo se ha procedido de manera independiente y paralela al sistema. Esto se ha podido hacer gracias a la existencia de información sobre ejecuciones probadas en el sistema. Esta información (trazas) está guardada dentro de unos ficheros que contienen todos los sucesos relativos a la ejecución de una determinada lista de trabajos. A continuación se describen con detalle.

#### 2.1.1. Trazas

Lo primero que se puede decir sobre las trazas es que están físicamente en ficheros y que nos sirven para interpretar lo que ha pasado en el sistema durante la ejecución unos determinados trabajos. Las trazas están compuestas por bloques de datos que corresponden a eventos que suceden en el sistema real. Es decir, se puede determinar cuándo van llegando los trabajos, cuándo empiezan a ejecutarse y cuándo terminan. De esta forma se puede obtener la información necesaria para este estudio sin necesidad de interactuar con el sistema real. Así, se ha podido trabajar de manera paralela a la evolución del sistema CISNE.

En la figura 2.1 se puede ver un ejemplo de estas trazas. Se trata de un fragmento de un fichero que contiene diferentes eventos sucedidos durante un flujo de ejecución de 87 trabajos. Concretamente se trata de la llegada al sistema del trabajo con identificador (jid) número 7.

```

ARRIVE: JID:7 / exec:pvmngx.gen / mem:113 / procs:4 / launch:unknown / estim.Start:0
                                           / time:1117117292
Node: aoclp8 / usedMem:377524000 / MPL:1 / JIDs: 1
Node: aoclp7 / usedMem:396784000 / MPL:2 / JIDs: 1 5
Node: aoclp6 / usedMem:387660000 / MPL:2 / JIDs: 1 5
Node: aoclp5 / usedMem:265976000 / MPL:1 / JIDs: 1
Node: aoclp4 / usedMem:257884000 / MPL:1 / JIDs: 1
Node: aoclp3 / usedMem:485688000 / MPL:2 / JIDs: 1 4
Node: aoclp2 / usedMem:469820000 / MPL:2 / JIDs: 1 4
Node: aoclp1 / usedMem:349492000 / MPL:1 / JIDs: 1

Queue: queue1 / jobCount:3 / JIDs: 3 6 7

```

Figura 2.1: Traza correspondiente a la llegada del trabajo nº7.

Con el fin de presentar los datos contenidos en la traza, se han numerado todas las porciones de información para comentarlas seguidamente:

Figura 2.2: Traza correspondiente a la llegada del trabajo nº7.

1. Este campo nos indica el **tipo de evento**, asociado a un trabajo, que se ha producido. En este caso observamos que se trata de la llegada del trabajo número 7 (ver el cuadro 2 de la figura 2.2). Una vez que un trabajo llega, se envía a la cola directamente (ver el cuadro 15 de la figura 2.2). Además del evento ARRIVE, existen otros 3 tipos diferentes de eventos: SCHEDULE, FINISH y ERROR.

**ARRIVE:** indica la llegada de un trabajo.

**SCHEDULE:** indica el momento en que empieza la ejecución de un trabajo.

**FINISH:** este evento se produce al terminar la ejecución de un trabajo.

**ERROR:** si se produce este evento quiere decir que ha habido un error y la ejecución se ha abortado.

2. **JID:** es el identificador del trabajo sobre el que se muestra toda la información de la traza.
3. **EXEC:** contiene el nombre de la aplicación, en este caso *pvmngx.gen*. Para más información sobre el tipo de trabajos posibles véase el capítulo 2.1.2.

4. **mem:** es la memoria, en MB, que necesita la aplicación.
5. **procs:** es el número de procesadores que la aplicación necesita. En este caso necesita 4 procesadores.
6. **launch:** indica el nodo donde se lanzó la ejecución. Este campo sólo tiene sentido en los eventos del tipo *schedule*.
7. **estim.Start:** este campo, aunque aparece en las trazas, siempre vale 0. No aporta ninguna información.
8. **time:** indica el tiempo del sistema en segundos.
9. **Node:** hace referencia al nombre de uno de los nodos que forman el cluster. En este caso *aoclp6*.
10. **usedMem:** indica la memoria usada por el nodo *aoclp6*.
11. **MPL:** es el grado de multiprogramación (número de tareas que se están ejecutando) que tiene ese nodo. En este caso el grado es 2 porque hay dos programas ejecutándose en ese nodo (ver 12).
12. **JIDs:** lista de trabajos que se están ejecutando en ese nodo. En este caso hay dos trabajos ejecutándose, concretamente el trabajo con identificador 1 y el trabajo con identificador 5.
13. **Queue:** nombre de la cola de espera. En este caso *queue1*.
14. **jobCount:** número de trabajos que hay en la cola. En este caso 3.
15. **JIDs:** identificadores de los trabajos que están en la cola. Obsérvese que el trabajo número 7 ya aparece en la última posición de la cola.

Toda esta información que ha sido numerada hasta ahora, se puede dividir en 2 grupos. Un grupo de información de la aplicación: puntos del 1 al 8; y otro grupo de información sobre el estado del sistema: puntos del 9 al 15.

Cabe destacar que la información que estas trazas tienen sobre el estado del sistema, muestra el estado de los nodos del cluster y la cola en los momentos en los que se producen los eventos relacionados con un trabajo. Es decir la cantidad de información que existe sobre el estado del sistema durante la ejecución de un trabajo depende de la cantidad de eventos que se generen entre el inicio y el fin de dicho trabajo. Así pues si durante la ejecución de un trabajo ningún otro trabajo entró en la cola, ni comenzó a ejecutarse, ni terminó de ejecutarse, sólo existirá información sobre el estado del cluster en los momentos de llegada, ejecución y fin de dicho trabajo. Por el contrario si durante la ejecución de ese trabajo entran y terminan de ejecutarse  $n$  trabajos diferentes, entonces habrá información sobre el estado del cluster en  $(n+1)*3$  puntos entre la llegada del trabajo y el final de su ejecución.

Un aspecto clave, en referencia a la información que nos prestan las trazas, es que forman ráfagas de ejecución (*cargas*). Esto quiere decir que muestran todos los eventos sucedidos desde que el primer trabajo de la ráfaga entra en el sistema, hasta



que termina su ejecución. De esta forma, gracias al campo *time* podemos obtener el tiempo de ejecución haciendo la resta del tiempo del evento *finish* y del evento *schedule*

El conjunto de trazas disponible para nuestro trabajo está formado por varios ficheros. Cada uno de estos ficheros contiene las trazas de ejecución de un determinado tipo de trabajos y bajo una determinada política de planificación. A continuación se describen los diferentes tipos de trabajos y los diferentes tipos de políticas relacionados con las trazas disponibles.

### **2.1.2. Tipos de trabajos, aplicaciones y diferentes políticas**

El conjunto de trazas disponibles para este estudio está clasificado según el tipo aplicación que ejecuten los trabajos: PVM o MPI y según la política empleada durante las ejecuciones. A continuación se describen los dos tipos de trabajos citados anteriormente:

**PVM:** es un conjunto de herramientas y librerías que emulan un entorno de propósito general compuesto de nodos conectados de distintas arquitecturas. El objetivo es conseguir que ese conjunto de nodos pueda ser usado de forma colaborativa para el procesamiento paralelo. Decir que en el sistema CISNE se utiliza una versión modificada de PVM en su versión 3.4. Las modificaciones en este caso implementan el módulo del esquema cooperativo, permitiendo que el demonio de PVM se comunique con el kernel mediante señales y llamadas al sistema implementadas al efecto.

El modelo en el que se basa PVM es dividir las aplicaciones en distintas tareas (igual que ocurre con openMosix). Son los procesos los que se dividen por las máquinas para aprovechar todos los recursos. Cada tarea es responsable de una parte de la carga que conlleva esa aplicación. PVM soporta tanto paralelismo de datos, como funcional o una mezcla de ambos.

**MPI:** es una especificación estándar para una librería de funciones de paso de mensajes. MPI fue desarrollado por el *MPI Forum*, un consorcio de vendedores de ordenadores paralelos, escritores de librerías y especialistas en aplicaciones.

Consigue portabilidad proveyendo una librería de paso de mensajes estándar independiente de la plataforma y de dominio público. La especificación de esta librería está en una forma independiente del lenguaje y proporciona funciones para ser usadas con C y Fortran. Abstrae los sistemas operativos y el hardware. Hay implementaciones MPI en casi todas las máquinas y sistemas operativos. Esto significa que un programa paralelo escrito en C o Fortran usando MPI para el paso de mensajes, puede funcionar sin cambios en una gran variedad de hardware y sistemas operativos. Por estas razones MPI ha ganado gran aceptación dentro del mundillo de la computación paralela.

MPI tiene que ser implementado sobre un entorno que se preocupe de el manejo de los procesos y la E/S por ejemplo, puesto que MPI sólo se ocupa de la capa de comunicación por paso de mensajes. Necesita un ambiente de programación paralelo nativo.

Cualquier trabajo utilizado dentro del sistema CISNE, contiene una aplicación que es lo que realmente se ejecuta en el sistema paralelo. El conjunto de las aplicaciones que se utilizan en el sistema está obtenido a partir de los benchmarks del NAS: IS y MG en sus clases T,A y B. En [7] se puede observar que a estas clases se les ha añadido una nueva clase: AB, creada para ofrecer mayor cantidad de opciones. En la Figura 2.3 se pueden ver las aplicaciones posibles junto con su caracterización.

Bench.	Memoria (MB)			Tiempo (seg.)		
	2 nodos	4 nodos	8 nodos	2 nodos	4 nodos	8 nodos
IS.T	6	5	4	1	1	1
IS.A	112	72	44	41	41	27
IS.AB	220	136	88	79	92	58
IS.B	445	260	150	180	205	126
MG.A	220	113	60	49	26	14
MG.AB	220	113	60	129	75	40
MG.B	220	113	60	240	126	-

Figura 2.3: Caracterización de las aplicaciones utilizadas

Hay que decir que el mismo benchmark configurado para 2,4 u 8 nodos proporciona 3 aplicaciones diferentes. Por lo tanto de esta tabla podrían surgir 20 aplicaciones diferentes. Pero si se hacen variaciones con la memoria necesaria se puede obtener un conjunto mucho más amplio. En el trabajo de *Mauricio Hanzich* [7] se puede ver que, utilizando este proceso, los ficheros de trazas disponibles se han obtenido a partir de cargas paralelas que utilizan en total hasta 30 aplicaciones diferentes. Estas 30 aplicaciones han sido determinadas a partir de una relación de compromiso entre el tiempo necesario para cada experimento y la representatividad del mismo.

La manera de nombrar las aplicaciones que aparecen en las trazas es diversa. En ocasiones para saber si se trata de una aplicación bajo PVM o MPI estas siglas están incluidas en el nombre. En otras ocasiones hay que deducirlo por la ubicación del fichero que contiene las trazas. Por otro lado, en el nombre, siempre aparecen las siglas MG o IS con lo que esta característica es más fácil de reconocer. En definitiva, una aplicación queda determinada por la tripleta compuesta por el nombre, el número de procesadores que necesita y la memoria que necesita.

### **Políticas**

Las políticas que se han utilizado durante las pruebas que han originado los ficheros de trazas son 4. Sus nombres son los siguientes: **NORMAL**, **JSTONE**, **3DBF.UNIFORM**, **3DBF.UNIPLS**. Estas políticas tienen diferentes criterios a la hora de elegir los nodos donde se va a ejecutar el trabajo (*Space-Slicing*), y elegir el trabajo a ejecutar. Estos criterios o políticas son los siguientes:

**nodeselect:** es la política de elección de los nodos del cluster que participarán en la ejecución del trabajo. Esta política esta formada por un criterio de ordenación de los nodos y un límite de aceptación:

**nodeorder:** criterio o criterios por los cuales se ordenan los nodos candidatos.

Suele ser el MPL(Multiprogramming Level) de los nodos candidatos. En ocasiones se requiere un criterio extra, que puede ser la cantidad de memoria disponible.

**nodelimit:** este criterio es el límite de aceptación para una característica determinada como podría ser el *MPL* o la *memoria ocupada*.

**jobselect:** es la política de selección del trabajo.

En la siguiente tabla se pueden ver las distintas opciones que implementan las 4 políticas citadas anteriormente:

Política	nodeselect	nodeorder	nodeorder	nodelimit	nodelimit	jobselect
<b>NORMAL</b>	NORMAL	MPL 1	MEM 15	MPL 4	MEM 90	JFIRST
<b>JSTONE</b>	NORMAL	MPL 1	MEM 15	MPL 1	MEM 90	JSTONE
<b>3DBF.UNIFORM</b>	UNIFRM	UNI 1		MPL 4	MEM 90	BF3D
<b>3DBF.UNIPLS</b>	UNIFRM	UNI 1		JBT 1	MEM 90	BF3D

Finalmente decir que las trazas disponibles nos ofrecen información sobre 2400 trabajos lanzados en el sistema. En la Figura 2.4 se puede ver el número de trabajos, de los que se dispone de trazas, según su tipo y la política bajo la que se ejecutaron.

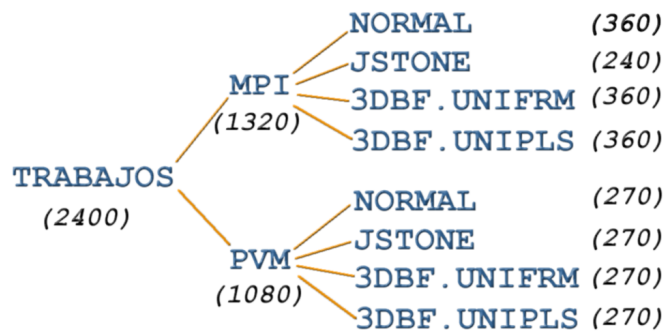


Figura 2.4: Número de trabajos por tipo y política

## 2.2. Almacenamiento de los Datos

La creación del *histórico* supone almacenar una gran cantidad de información sobre las ejecuciones que se realicen en el cluster. La información para almacenar es toda la información contenida en las trazas. Aunque probablemente no se utilice toda esta información para la predicción, si que puede ser útil para crear una herramienta de monitorización que pueda ayudar a entender mejor el comportamiento del sistema.

Teniendo en cuenta que cuando se crea un cluster se intenta tenerlo ocupado la mayor parte del tiempo posible, es de esperar que con el tiempo la cantidad de información a almacenar será considerable. Este hecho, junto con la necesidad de mantener la coherencia de los datos o permitir conexiones múltiples a los datos, hacen

que tome fuerza la utilización de un motor de base de datos. En resumen, estas podrían ser las ventajas:

- Usar un motor ahorra una enorme cantidad de trabajo.
- *Escalabilidad*: es posible manipular bases de datos enormes, del orden de miles de tablas y millones de registros, con hasta 32 índices por tabla.
- *Conectividad*: permite conexiones entre diferentes máquinas con diferentes sistemas operativos.
- *Seguridad*: en forma de permisos y privilegios.
- *Velocidad*: para búsquedas complejas tendremos los datos antes que si tenemos que hacer búsquedas secuenciales en ficheros con un gran número de accesos a disco.

Pero no todo son ventajas, también existen inconvenientes:

- Usar un motor supone tener instalado un servidor con la carga que eso supone en el sistema.
- Según la distribución de los nodos respecto al servidor puede incrementar el tráfico de red.

Está claro que para este trabajo, los inconvenientes no tienen importancia, ya que se trabaja de forma aislada y fuera del cluster real. Por otro lado, la inserción del motor dentro del cluster si que supone una carga extra. Ahora bien, el hecho de ser un cluster no dedicado hace que esta carga sea asumible ya que puede ser semejante a la carga que puede aportar un usuario local. De todas formas existe la opción instalar el servidor de bases de datos en una máquina aparte del cluster.

Si hace un balance de ventajas e inconvenientes, la balanza se inclina del lado de utilizar un motor de bases de datos. En la siguiente sección se busca entre las diferentes posibilidades de motores para intentar encontrar uno que satisfaga nuestra necesidades: respuesta rápida para acelerar los cálculos que necesiten datos del *histórico*, múltiples conexiones, integridad de los datos, y mínimo coste.

### **2.2.1. Elección del sistema gestor de bases de datos**

Hoy día existen diferentes motores de bases de datos. A la hora de elegir uno de ellos hay que tener en cuenta sobre todo que corra bajo *Linux* y que sea un motor de *software libre*, es decir gratuito. Esto reduce sensiblemente el grupo donde elegir, pero es una característica imprescindible para este trabajo, por motivos de economía y filosofía. Sin embargo es necesario que, el motor elegido, sea robusto y eficiente al mismo tiempo.

Existen varios motores que se pueden ajustar a estas necesidades. Entre ellos los más adecuados pueden ser *MySQL*, *PostgreSQL* y *FireBird* (versión liberada de InterBase). Cuando se empezó a escribir esta sección se pensaba incluir tablas detalladas con las características de cada motor, pero la mayoría de esas características

	<b>MySQL 3.23</b>	<b>PostgreSQL 7.1</b>	<b>Interbase 6.2</b>	<b>SAP DB</b>
Inserción 100.000 tuplas	10seg	12min 38seg	42seg	2min 38seg
100.000 variados	17seg	6min 30seg	1min 07seg	8min
1000 selects complejos	55seg	1min 26seg	2min 23seg	1min 14seg

Figura 2.5: Pruebas de acceso a datos.

no nos interesan para nuestro objetivo. Así que, a modo de resumen, se comenta que *MySQL*, *PostgreSQL* y *InterBase* son motores de BD relacionales y orientados a objetos que cuentan con todas las características de un motor de BD comercial: transacciones atómicas, triggers, constrains, replicación y llaves foráneas entre otras. Además de que son rápidos, multi-threaded, multiusuario y robustos.

Con estas características garantizadas hay que centrar la atención en dos aspectos: la velocidad de las consultas y la carga que el motor supone para el sistema. Recordemos que el objetivo de este trabajo será predecir el tiempo que tardará en ejecutarse un trabajo en el cluster. Para calcular este tiempo habrá que buscar los datos de los trabajos anteriores para ver que trabajo se parece y cual no se parece al nuevo, y una vez se tengan estos trabajos parecidos habrá que utilizar sus características para hacer la estimación. La velocidad con la que el sistema ofrezca los datos necesarios para los cálculos es crucial para conseguir una predicción rápida.

En la Figura 2.5 podemos observar el comportamiento, en cuanto a velocidad, de diferentes motores de bases de datos, para la inserción o selección masiva desde la línea de comando.

Si nos fijamos en la velocidad de los *selects* podemos ver que *MySQL* destaca por encima de los demás, siendo casi dos veces más rápido que *PostgreSQL* y casi tres veces más rápido que *Interbase*. A favor de *Interbase* hay que decir que se comporta mejor que el resto de sus competidores sirviendo peticiones concurrentes que impliquen gran cantidad de registros y gran cantidad de clientes. En la Figura 2.6 se pueden observar las peticiones resueltas por segundo que cada motor es capaz de ofrecer. Este test consistía en servir páginas web cada una de las cuales atacaba sobre una base de datos de empleados. Las páginas hacían lo siguiente:

1. Generar un número aleatorio entre 1 y 100.
2. Obtener y mostrar el departamento que corresponde a ese número.
3. Obtener y mostrar todos los empleados (1000) que correspondan a ese departamento, y los del departamento 10 (otros 1000).

	<b>MySQL 3.23</b>	<b>PostgreSQL 7.1</b>	<b>Interbase 6.2</b>	<b>SAP DB</b>
10 clientes	3.17	1.32	4.39	3.24
20 clientes	1.17	1.21	3.77	1.8
30 clientes	0.86	0.61	2.62	1.3

Figura 2.6: Peticiones servidas por segundo.

Teniendo en cuenta que en el sistema CISNE habrá poca concurrencia, se observa que *MySQL* no pierde demasiado terreno respecto *Interbase* para un número de clientes menor que 10. Sin embargo en la Figura 2.5 se puede ver que para un tipo de trabajo parecido al requerido en el sistema CISNE, *MySQL* es hasta 3 veces más rápido que *Interbase*.

Por otra parte también hay que pensar en la carga que supone para el sistema tener instalado un motor. Aquí hay que decir, que si se opta por poner un nodo nuevo que se ocupe de hospedar la base de datos histórico, no hay problema. Pero sí es importante si se opta por la solución de que la base de datos esté instalada en uno de los nodos que participan en la ejecución de trabajos, ya que consumirá recursos de ese nodo. En este aspecto el motor menos pesado para el sistema es *MySQL* por su bajo consumo de recursos.

En principio, tanto *PostgreSQL*, *InterBase* y *MySQL* se adaptan a las necesidades de este trabajo. Sin embargo *MySQL* destaca por ser rápido con un peso bajo, y cuenta con muchas herramientas y documentación que van a facilitar mucho el trabajo. En su contra, decir que no tiene en cuenta la integridad referencial, por lo tanto queda bajo nuestra responsabilidad a la hora de programar la herramienta. Es precisamente esta particularidad la que hace que la elección sea utilizar *MySQL*, ya que el hecho de no tener que hacer comprobaciones de integridad hace que sea más rápido que el resto de motores. Algunas de las razones por las que se ha decidido implementar la base de datos con *MySQL* son:

1. El principal objetivo de *MySQL* es la velocidad y la robustez.
2. Su bajo consumo lo hacen apto para ser ejecutado en una máquina con escasos recursos sin ningún problema.
3. Clientes C, C++, JAVA, Perl, TCL, PHP.
4. Puede trabajar en distintas plataformas y S.O. distintos.
5. Sistema de contraseñas y privilegios muy flexible y segura.
6. Todas las contraseñas viajan cifradas en la red.
7. Los clientes usan TCP o UNIX Socket para conectarse al servidor.
8. Las utilidades de administración de este gestor son envidiables para muchos de los motores comerciales existentes.

### **2.3. Predecir a partir de los datos almacenados**

Ante la llegada de un trabajo al sistema, se tendrán que realizar una serie de acciones con el objetivo de ofrecer un tiempo de ejecución estimado:

1. Obtener la aplicación que este trabajo contiene.

2. Buscar que trabajos hay en el histórico que ejecutaron esa misma aplicación o una parecida.
3. Obtener el estado del sistema en el inicio de ejecución de cada uno de esos trabajos.
4. Clasificar estos trabajos en función del parecido del estado obtenido con el estado actual.
5. Una vez tengamos la lista ordenada se dará como resultado el tiempo del trabajo que se ha considerado más parecido al actual.

En el caso de este trabajo la búsqueda de un trabajo parecido será obvia ya que estamos hablando de un total de 30 aplicaciones posibles. Así que no tendremos que buscar trabajos que ejecutaron aplicaciones similares, si no que tendremos que buscar trabajos que ejecutaron la misma aplicación.

La tarea fundamental es la número 4 a la hora de obtener el *tiempo de ejecución estimado*. Tendremos que comparar estados del sistema para encontrar el más parecido al actual. Será esta búsqueda del estado más parecido la que determinará la eficacia de este trabajo.

A continuación se muestran algunas técnicas dentro del campo de la inteligencia artificial que se podrían emplear para este tipo de búsquedas.

### 2.3.1. Árboles de decisión

En la documentación consultada [3], se describe que un árbol de decisión permite procesar una entidad que ha sido descrita por un conjunto de propiedades y tomar una decisión binaria (Si/No). Este tipo de representación es especialmente útil para procesar sentencias lógicas y funcionan perfectamente para representar funciones booleanas. En la figura 2.7 podemos ver un ejemplo de árbol de decisión para el concepto “buen día para jugar a tenis”

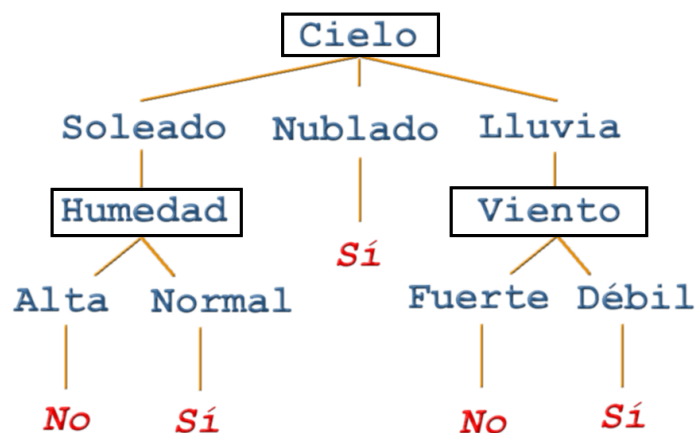


Figura 2.7: Ejemplo de árbol de decisión para jugar a tenis.

El árbol de decisión de la figura 2.7, asigna una clasificación binaria, en este caso *Si* o *No* a cada día que se analice. Un árbol de decisión puede ser extendido fácilmente, para representar funciones objetivo con más de dos valores posibles. Una extensión menos simple y menos común consiste en considerar funciones objetivo de valores discretos.

De cara a este trabajo se tendrían que considerar funciones objetivos de intervalos de tiempo. En función de cómo se elijan esos intervalos habría dos escenarios posibles. Si se eligen intervalos grandes de tiempo el error medio de predicción sería demasiado grande. Por otro lado, si se eligen intervalos pequeños de tiempo el árbol resultante sería muy complejo de crear.

Podemos concluir que este tipo de representación no se ajusta a nuestro escenario, ya que la respuesta que nosotros necesitamos es un tiempo y por lo tanto no tiene nada que ver con una respuesta binaria del tipo *si/no*. Aun así se ha considerado la posibilidad de crear un conjunto discreto de tiempos, siendo esta posibilidad desestimada por la pérdida de precisión que representa.

### **2.3.2. Algoritmos genéticos**

Los algoritmos genéticos se basan en la idea de la *selección natural* introducida en 1859 por *Charles Darwin* dentro de su libro "El Origen de las Especies". Esta idea se utiliza para construir métodos de búsqueda en problemas de optimización combinatoria y métodos de aprendizaje que tienen en cuenta ideas cómo las siguientes:

- Cada individuo tiende a pasar sus características a su descendencia.
- Aún así, la naturaleza produce individuos con características diferentes.
- Los individuos más adaptados tienden a tener más descendencia, y a la larga, la población tiende a ser "mejor".
- A grandes periodos, la acumulación de cambios puede producir especies totalmente nuevas, adaptadas a su entorno.
- Además, la naturaleza dispone de una serie de mecanismos reguladores externos a este proceso pero igualmente interesante: el mecanismo de diversidad, los parásitos, las organizaciones sociales, etc.

Los mecanismos biológicos que hacen posible la evolución son conocidos hoy en día, y son los que estos algoritmos utilizan:

- Genoma.
- Cromosomas.
- Cruces y mutaciones.
- Funciones de adaptación.
- Mecanismos correctores o moduladores: diversidad, parasitismo, etc.



Los algoritmos genéticos son un modelo de aprendizaje automático basado en la creación y manipulación de un conjunto de individuos (soluciones posibles), representados mediante cromosomas que entran dentro de un proceso evolutivo. Este proceso evolutivo asegura que todos los “nuevos” individuos generados también formarán parte del conjunto de soluciones hipotéticas del problema. La calidad del individuo dentro de ese entorno determinará su posibilidad de supervivencia, y por tanto la probabilidad de pasar sus “genes” a las generaciones futuras.

Dentro de todos los elementos que forman parte del método, el que contiene el objetivo del problema es la función de adaptación. Hay que remarcar que éste no es un proceso dirigido a solucionar un problema sino únicamente un modelo de competición entre individuos.

### 2.3.3. Métodos basados en características

Estos métodos son conocidos como métodos de *reconocimiento de patrones*. En este caso, un patrón sería la descripción de un objeto. Para describir el objeto se utilizan las *características*. Una característica es una propiedad importante que permite distinguir total o parcialmente un objeto de otro. El conjunto de características utilizadas para describir el objeto se llama *espacio de características*. La composición de varias características en un vector se llama *vector de características* tal y como nos muestran la Figura 2.8.



Figura 2.8: Vector de características.

Cada objeto está descrito por un vector de n-características. En la Figura 2.9 se puede ver como, con respecto a las características de peso y altura, los niños menores de 10 años forman un grupo de puntos cercanos entre sí, mientras que los niños que juegan a baloncesto forman otro.

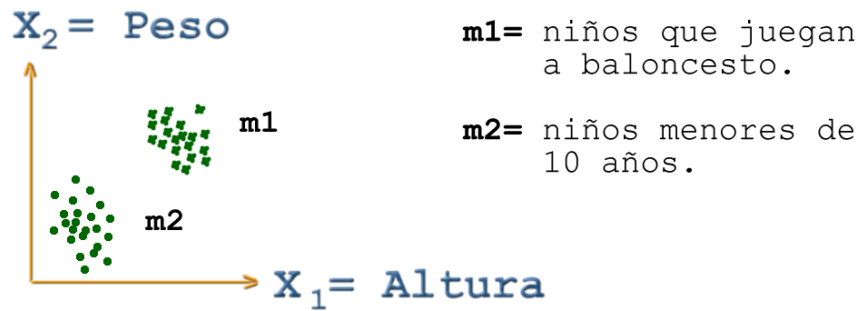


Figura 2.9: Ejemplo: Espacio de características de 2 dimensiones.

En este ejemplo, las características peso y altura son suficientes para caracterizar los dos grupos de niños posibles. Cuando son necesarias más características, o los grupos no están tan bien delimitados, el proceso se complica. En los estudios consultados sobre predicción, como son: [8, 9, 10, 11, 12], se puede ver que la predicción a partir de históricos basada en búsqueda por características obtiene unos resultados notables en cuanto al tiempo de ejecución.

## 2.4. Solución Propuesta

A partir del análisis realizado hasta ahora, junto con los requerimientos del sistema CISNE, se propone la solución que a continuación se describe teniendo en cuenta que todo el trabajo se hará en un equipo independiente:

### **Preparación del equipo**

Se utilizará un PC doméstico donde se instalará una distribución Linux, un servidor MySQL y las librerías necesarias para la utilización de MySQL en C. En este ordenador se llevaran a cabo todas las implementaciones y pruebas.

### **Histórico**

La solución propuesta para crear el histórico de datos está compuesta de las siguientes acciones:

1. Diseño e implementación de la base de datos *histórico* teniendo en cuenta la información contenida en las trazas y posibles utilidades.
2. Carga de la base de datos con parte de las trazas disponibles. La otra parte se utilizará como conjunto de test para obtener los resultados. En este punto se propone la implementación de una herramienta, a modo de escáner, que recorra los ficheros de trazas e inserte automáticamente la información en la base de datos.

### ***Herramienta de predicción***

Se propone crear una herramienta en lenguaje C que dado un trabajo de un fichero de trazas haga las siguientes acciones:

1. Extraer la aplicación contenida en ese trabajo y con esa información se buscarán en el histórico todos los trabajos que ejecutaron esa aplicación.
2. Una vez que se tenga la lista de trabajos, se ordenarán según el grado de similitud del estado del cluster en el momento que esos trabajos empezaron su ejecución y el estado actual.
3. Esa ordenación se efectuará mediante un método basado en las características de los estados. Mediante una función se asignará un valor a cada trabajo. Ese valor será un índice que reflejará la diferencia (o similitud) del estado actual del cluster, con el estado que había cuando se ejecutó cada uno de los trabajos obtenidos del histórico.
4. De la lista de trabajos, elegiremos el trabajo que haya obtenido una *distancia* mínima y daremos como resultado el tiempo que ese trabajo tardó en ejecutarse.

## **2.5. Estudio de Viabilidad**

Se pretende crear un sistema independiente al sistema CISNE para el estudio y experimentación. De esta forma no se interfiere en su desarrollo y se puede trabajar de forma independiente. Una vez finalizado el trabajo, el sistema será susceptible de ser integrado en CISNE en función de los resultados. La integración en CISNE no forma parte de los objetivos de este proyecto.

Este trabajo se puede dividir en varios bloques. Por un lado hay que disponer de información sobre ejecuciones de trabajos en el sistema para extraer toda la información posible. Toda esta información hay que almacenarla en un histórico para que esté disponible y bien organizada. Por último habrá que crear una herramienta que, dado un trabajo, sea capaz de predecir el tiempo de ejecución. A continuación se expone la viabilidad por bloques planteando si se dispone de los recursos necesarios o describiendo cómo solucionar cada punto:

### ***Información sobre ejecuciones en el sistema.***

En el apartado *Análisis de los Datos Disponibles*, se ha descrito con detalle toda la información de la que se dispone para este trabajo. A modo de recordatorio decir que existen unos ficheros con información sobre ejecuciones de trabajos en el sistema real. Se dispone de ficheros de este tipo para cada una de las posibles políticas que pueden gestionar la cola de espera. Estos ficheros sirven para interpretar que ha pasado en el sistema durante la ejecución de una serie de trabajos según la política de la cola. En ellos se pueden observar bloques de datos que representan eventos que suceden en el sistema real. Un ejemplo de uno de estos eventos se puede ver a continuación, se trata del evento de llegada del trabajo con identificador número 7. Este evento pertenece a un flujo de ejecución de 87 trabajos.

```

ARRIVE: JID:7 / exec:pvmngx.gen / mem:113 / procs:4 / launch:unknown / estim.Start:0
/ time:1117117292
Node: aoclp8 / usedMem:377524000 / MPL:1 / JIDs: 1
Node: aoclp7 / usedMem:396784000 / MPL:2 / JIDs: 1 5
Node: aoclp6 / usedMem:387660000 / MPL:2 / JIDs: 1 5
Node: aoclp5 / usedMem:265976000 / MPL:1 / JIDs: 1
Node: aoclp4 / usedMem:257884000 / MPL:1 / JIDs: 1
Node: aoclp3 / usedMem:485688000 / MPL:2 / JIDs: 1 4
Node: aoclp2 / usedMem:469820000 / MPL:2 / JIDs: 1 4
Node: aoclp1 / usedMem:349492000 / MPL:1 / JIDs: 1
Queue: queue1 / jobCount:3 / JIDs: 3 6 7

```

Figura 2.10: Evento de llegada del trabajo número 7.

En la figura 2.10 se puede observar, en azul la información disponible sobre el trabajo que ha llegado, en rojo la información sobre el estado de los nodos del sistema (estado del cluster), y en verde la información sobre la cola de espera. Para cada trabajo que entre en el sistema tendremos eventos de llegada, inicio de ejecución, fin de ejecución y en ocasiones de error. Esta es la información disponible y la que almacenaremos para cada trabajo en el histórico.

Por lo tanto respecto a este punto, se considera viable ya que se dispone de información sobre 2400 trabajos ejecutados en el sistema.

### ***Histórico de datos.***

Se deberá diseñar una base de datos para el almacenamiento de la información de cada trabajo. En la implementación de la base de datos se tendrán que tener en cuenta una serie de requisitos. Por un lado necesitamos que el motor de base de datos elegido sea rápido y capaz de crecer en cuanto a volumen de datos y por otro lado necesitamos que la carga que este motor introduzca en el sistema sea lo más pequeña posible y, además, sin coste económico. Todos estos parámetros son críticos para la consecución del objetivo final y por ese han sido estudiados en la elección del motor de base de datos. En el capítulo *Elección del sistema gestor de bases de datos* se han estudiado las diferentes posibilidades siendo *MySQL* el motor elegido.

Dado lo ampliamente extendido de este gestor gratuito de bases de datos y la gran cantidad de información existente: manuales, herramientas, foros, etc. Se considera viable la implementación de una base de datos histórico mediante *MySQL*.

### ***Carga de la base de datos.***

Una vez creada la base de datos se deberá inicializar con los datos disponibles. Existen varios ficheros de trazas para cada una de las políticas de gestión de cola, por tanto se cargará la base de datos con algunos de esos ficheros y se reservarán otros como conjunto de test para poder llevar a cabo la experimentación.

La carga de la base de datos se podrá llevar a cabo mediante la construcción de un programa que, a modo de escáner, recorra los ficheros de trazas extrayendo la información y la inserte en el lugar que le corresponda dentro de la base de datos. Para implementar esta tarea se dispone del compilador y librerías necesarias en lenguaje de programación C.

### **Crear herramienta de predicción.**

Una vez cargada la base de datos se creará una herramienta que para un trabajo del conjunto de test, busque en el histórico los trabajos parecidos (en cuanto a la aplicación ejecutada y estado del sistema) y nos devuelva el tiempo de ejecución predicho para ese trabajo. Tanto para este programa como para el de la carga de la base de datos se utilizará el lenguaje de programación C con SQL empotrado compilado con gcc y utilizando las librerías de MySQL.

### **Detalle de viabilidad.**

En la siguiente figura se muestra el detalle de las tareas a seguir con su viabilidad y coste asociado.

	<b>Viabilidad</b>	<b>Coste</b>
<b>Preparación del equipo</b>		
Instalación de:		
- Sistema operativo Linux.	✓	0
- Servidor y cliente MySQL.	✓	0
- GCC.	✓	0
- Librerías MySQL.	✓	0
<b>Información sobre ejecuciones en el sistema</b>		
- Trazas	✓	0
<b>Diseño e implementación de la base de datos</b>		
- Creación del diagrama ER.	✓	0
- Implementación.	✓	0
<b>Carga de la Base de Datos.</b>		
- Creación de la herramienta en C para leer datos de las trazas e insertarlos en la BD.	✓	0
<b>Crear herramienta de predicción.</b>		
- Diseño del método de predicción.	✓	0
- Implementación.	✓	0
- Experimentación, resultados y mejoras.	✓	0
<b>Redacción de la memoria.</b>	✓	0

Figura 2.11: Detalle de la viabilidad de las tareas

Como se puede ver en la figura 2.11 todas las tareas se consideran viables y con coste económico 0, por lo tanto el conjunto del trabajo se considera viable.

## 2.6. Planificación Temporal de la Solución Propuesta

Una vez conocida la solución propuesta y su viabilidad se debe planificar respecto al tiempo su implementación. En la figura 2.12 se describe en que meses está previsto que se realicen las diferentes tareas que componen la solución propuesta.

	Fecha ejecución
<b>Preparación del equipo</b>	
Instalación de:	
- Sistema operativo Linux.	
- Servidor y cliente MySQL.	Dic 2007
- GCC.	
- Librerías MySQL.	
<b>Información sobre ejecuciones en el sistema</b>	
- Trazas	Dic 2007
<b>Diseño e implementación de la base de datos</b>	
- Creación del diagrama ER.	Dic 2007
- Implementación.	Ene 2008
<b>Carga de la Base de Datos.</b>	
- Creación de la herramienta en C para leer datos de las trazas e insertarlos en la BD.	Ene 2008
<b>Crear herramienta de predicción.</b>	
- Diseño del método de predicción.	Feb 2008
- Implementación.	Feb 2008
- Experimentación, resultados y mejoras.	Mar 2008
<b>Redacción de la memoria.</b>	Abr-May 2008

Figura 2.12: Planificación temporal del trabajo

En el capítulo 1 se presentaba un diagrama de Gantt de las tareas a realizar sin entrar en detalle. Ahora se muestra un diagrama más detallado que se puede ver en la figura 2.13. El reparto temporal de las tareas se realiza de forma estimada. Este reparto puede diferir respecto al tiempo real empleado para cada tarea. Una vez finalizado el trabajo se comparará este diagrama de la planificación temporal con el coste real en tiempo de cada tarea.



Figura 2.13: Diagrama de Gantt. Planificación de la solución propuesta.

## Capítulo 3

# Diseño e Implementación de la Solución Propuesta

En esta sección se describe la base de datos con todas las tablas y atributos que la forman. Se muestran diagramas que ayudarán a entender mejor la relación entre las tablas. También se habla sobre cada uno de los atributos de las diferentes tablas y se comenta su utilidad en la predicción. A continuación se muestra cómo se ha inicializado la base de datos con toda la información contenida en el conjunto de trazas.

Finalmente se presenta el diseño y la evolución de la herramienta de predicción remarcando que aspectos se han contemplado en su implementación.

### 3.1. Base de Datos: Histórico

Antes de abordar los detalles de la base de datos convendría ver qué factores intervienen en la ejecución de una aplicación en el sistema CISNE. Conforme se avance en la explicación, surgirán entidades y acciones que darán lugar a las tablas y relaciones de la base de datos. En las siguientes secciones se describirán las tablas y los atributos con más grado de detalle.

Lo primero que ocurrirá cuando se lance una aplicación en el sistema es que se generará un trabajo. Así pues, cada vez que se lance esa misma aplicación, se generará un trabajo diferente con unas características diferentes. Esto es así porque el cluster no tendrá, salvo coincidencia, nada que ver en una u otra ejecución en cuanto a recursos libres se refiere. Probablemente en lo único que se parecerán estos trabajos será en la aplicación que ejecutan. Por lo tanto, de esta abstracción se pueden obtener dos entidades que darán lugar a dos tablas en la base de datos. Estas entidades son: *aplicaciones y trabajos*.

En la tabla aplicaciones se guardará toda la información de la aplicación de la que disponemos: tipo(pvm,mpi), nombre, número de parámetros, memoria necesaria y número de procesadores que necesaria para su ejecución.



En la tabla trabajos se guardarán: la aplicación a la que hace referencia, el instante en el que el trabajo llegó al sistema, el instante en el que se terminó de ejecutar el trabajo, el tiempo que tardó en ejecutarse, y si se produjo algún error durante su ejecución.

Volviendo a cómo transcurren los acontecimientos en el sistema, después de la aparición de un trabajo que tendrá una aplicación asociada, este trabajo entrará en la cola de espera. En esta cola será ordenado según una política de gestión. Así que *cola* será otra tabla de la base de datos donde se podrá percibir el progreso de cada trabajo durante su espera.

Posteriormente el trabajo empezará su ejecución y dependiendo del planificador seleccionado del sistema asignado a diferentes nodos donde se ejecutará en paralelo. Durante su ejecución coincidirá con otros trabajos que compartirán los recursos de cada nodo. Este hecho interesa que quede reflejado también en la base de datos, por lo tanto se necesitarán generar dos tablas más: *ejecución y nodos*.

El sistema CISNE está pensado de forma que el cluster podría estar instalado en cualquier aula de informática con el fin de aprovechar los ordenadores desocupados o con poca carga y, también, las horas durante las que las aulas están cerradas. Este hecho hace que durante la ejecución de un trabajo en paralelo pudiera haber presencia de carga local inducida por un usuario del aula. Para reflejar la presencia o no de esta carga se implementa una tabla donde podría caracterizarse esta carga si fuera necesario. La tabla se llamará *carga local prevista*.

Una vez finalizada la ejecución debe calcularse y guardar el tiempo transcurrido entre el inicio y el final de la ejecución, ya que esta información será necesaria para la predicción. También se guardará el tiempo desde que el trabajo entró en el sistema hasta que comenzó a ejecutarse, de esta forma se sabrá el tiempo de espera en cola.

En la Figura 3.1 se observa el diseño general de la base de datos con las *tablas y relaciones* que se han nombrado anteriormente. También se pueden ver los *atributos* que las forman y que se describen a continuación.

### **3.1.1. Tabla: trabajos**

La tabla *trabajos* es la tabla principal de la base de datos. Cuando una aplicación llega al cluster se genera un trabajo que la contiene. Si más tarde se vuelve a ejecutar la misma aplicación existirán dos trabajos que habrán ejecutado la misma aplicación pero serán trabajos diferentes. Esto es así porque en cada una de las ejecuciones la aplicación habrá entrado en la cola en una posición diferente, habrá coincidido con aplicaciones diferentes en nodos diferentes y, y porque su tiempo de ejecución seguramente habrá sido diferente. En la Figura 3.2 se observan los atributos de la tabla *trabajos*. Estos atributos son descritos a continuación:

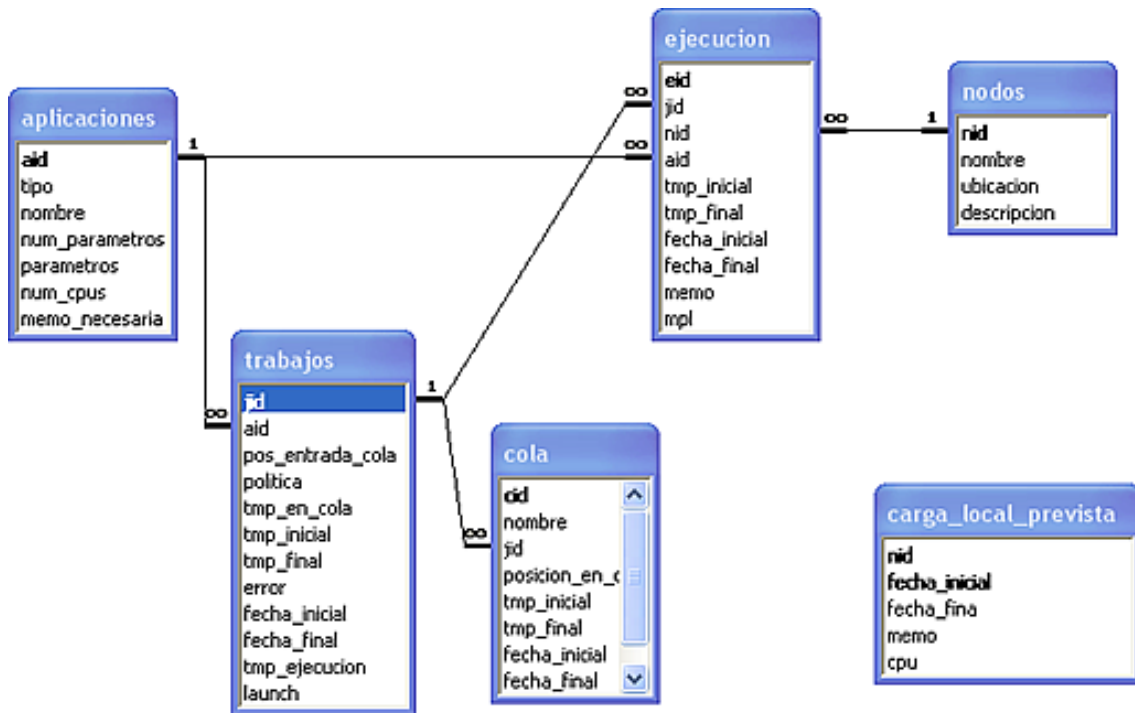


Figura 3.1: Diagrama E-R.

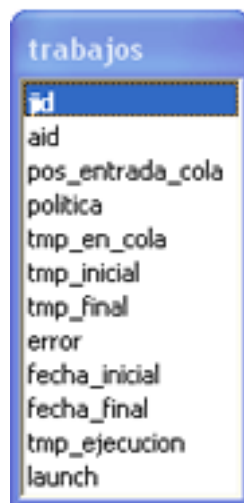


Figura 3.2: Tabla trabajos.

- **jid**: el *identificador de trabajo*. Cada trabajo que se ejecute tendrá un jid diferente. Este atributo será utilizado en las demás tablas para referenciar a este trabajo.
- **aid**: aquí se guarda el *identificador de la aplicación* contenida en este trabajo. Así es posible acceder a toda la información de la aplicación que está almacenada en la tabla *aplicaciones*.
- **política**: en este campo se guarda la política que regía la cola de espera en el momento en que el trabajo entró en ella.

- **tmp en cola:** es el tiempo que el trabajo estuvo en la cola de espera, desde que entró en ella hasta que comenzó la ejecución.
- **tmp inicial:** aquí guardaremos el tiempo en el que el trabajo empezó a ejecutarse.
- **tmp final:** el tiempo en el que el trabajo terminó su ejecución.
- **error:** este campo indica si la ejecución del trabajo ha sido abortada.
- **fecha inicial:** aquí quedará almacenada la fecha del comienzo de la ejecución. Esto será útil porque, en principio, puede haber trabajos que duren más de un día.
- **fecha final:** contendrá la fecha de finalización del trabajo.
- **tmp ejecución:** una vez terminada la ejecución de un trabajo, se almacena el tiempo de ejecución real.
- **launch:** en este campo se guarda el nodo *padre* donde es lanzado el trabajo.

### 3.1.2. Tabla: aplicaciones

La tabla *aplicaciones* contiene la información de la aplicación, como su nombre, su identificador y otros parámetros que se muestran a continuación.

aid
tipo
nombre
num_parametros
parametros
num_cpus
memo_necesaria

Figura 3.3: Tabla aplicaciones.

- **aid:** el *identificador de aplicación*. Cada aplicación tendrá un aid diferente. Este atributo será utilizado en las demás tablas para referenciar a esta aplicación.
- **tipo:** en este campo se guarda si la aplicación es pvm, mpi otro tipo de aplicación.
- **nombre:** el nombre de la aplicación será especialmente útil cuando llegue una nueva aplicación y busquemos si se ha ejecutado ya en el sistema.
- **num parametros:** aquí se guarda el número de parámetros que tiene la aplicación.
- **parametros:** aquí se guarda el valor de los parámetros.
- **num cpus:** en este campo se almacena el número de cpus que necesita la aplicación para ejecutarse.
- **memo necesaria:** la memoria que necesita la aplicación.

A la hora de utilizar esta información para predecir el tiempo de ejecución, estos datos serán de gran ayuda. Ante la ejecución de una nueva aplicación, se buscarán trabajos, ya ejecutados, en el histórico con la misma aplicación. Si se encuentra la misma aplicación, se estará buscando el tiempo de ejecución entre aplicaciones idénticas, por lo que sólo habrá que preocuparse de buscar el estado del cluster más parecido de entre esas ejecuciones. Si por el contrario no se encuentra la misma aplicación en el histórico, se deberá buscar entre las aplicaciones que sean del mismo tipo, que tengan el mismo número de parámetros, que necesiten el mismo número de procesadores y que consuman una cantidad de memoria semejante a la de la aplicación con la que se van a comparar.

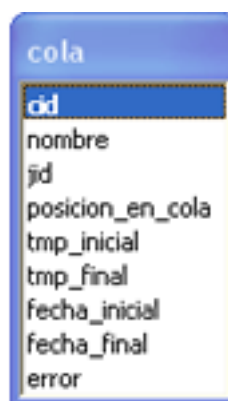
### 3.1.3. Tabla: cola

En la tabla cola quedará registrado el avance de los trabajos en la cola del sistema durante su espera para ser ejecutados. De esta manera se podrá saber, para cualquier trabajo que haya sido ejecutado con anterioridad, junto a que trabajos coincidió en la cola. También podremos saber cuanto tiempo estuvo en cola. Ya que el sistema contempla la posibilidad de gestionar varias colas, esta tabla tendrá que guardar el nombre de la cola.

El funcionamiento sería el siguiente, cada vez que llega un trabajo y es enviado a la cola de espera se almacenarán los siguientes datos: cola en la que ha entrado, posición que ocupa, el instante de tiempo. Una vez que el trabajo empieza su ejecución se guardará el instante de tiempo en que terminó su espera y se comparará este tiempo con el tiempo inicial para obtener el *tiempo de espera en cola*. Este tiempo se almacenará en la tabla *trabajos*.

No sólo se almacena la información del instante en que un trabajo llega a la cola y la del instante en que la deja. Cada vez que sucede uno de estos eventos a cualquier trabajo: *arrive*, *schedule*, *finish* o *error* se almacena el estado de la cola. De esa forma se tiene almacenado el progreso de los trabajos en la cola de espera.

Estos son los atributos que constituyen la tabla *cola*.



cola
cid
nombre
jid
posicion_en_cola
tmp_inicial
tmp_final
fecha_inicial
fecha_final
error

Figura 3.4: Tabla cola.

- **cid:** atributo que se utiliza para guardar el identificador de la cola. En la actualidad sólo existe una cola en el sistema pero en un futuro es posible que existan varias.
- **nombre:** nombre descriptivo de la cola.
- **jid:** identificador del trabajo del que se está guardando la información.
- **posicion en cola:** posición que está ocupando en la cola.
- **tmp inicial:** tiempo inicial para esta posición.
- **tmp final:** tiempo final para esta posición.
- **fecha inicial:** fecha inicial para esta posición.
- **fecha final:** fecha final para esta posición.
- **error:** en este atributo se guarda si hubo o no un error durante su estancia en esta posición.

### 3.1.4. Tabla: ejecución

En la Figura 3.5 se observan los atributos que forman esta tabla. Esta tabla contendrá, como mínimo, una fila por cada nodo que participe en la ejecución de un trabajo . En ese registro estará guardado el tiempo inicial y el final en el que ese nodo participó en la ejecución del trabajo, así como la memoria utilizada por el nodo. También estará disponible el grado de multiprogramación del nodo. Es decir, el número de trabajos que se estaban ejecutando en ese nodo en el momento en que se inició la ejecución del nuevo trabajo.

ejecucion	
eid	
jid	
nid	
aid	
tmp_inicial	
tmp_final	
fecha_inicial	
fecha_final	
memo	
mpl	

Figura 3.5: Tabla ejecuciones.

- **eid:** identificador de ejecución.
- **jid:** identificador del trabajo que se está ejecutando.
- **nid:** identificador del nodo que participa en la ejecución del trabajo.
- **aid:** identificador de la aplicación relacionada con el trabajo.

- **tmp inicial:** tiempo de inicio de la ejecución para este nodo.
- **tmp final:** tiempo en el que finaliza la ejecución para este nodo.
- **fecha inicial:** fecha en la que se inicia la ejecución.
- **fecha final:** fecha en la que finaliza la ejecución.
- **memo:** memoria usada por el nodo.
- **mpl:** grado de multiprogramación del nodo.

### 3.1.5. Tabla: nodos

Esta tabla contiene la información sobre los nodos que forman el cluster. Los datos son: el nombre de los nodos, pero en un futuro puede ser interesante guardar su ubicación y una breve descripción del equipo para tareas de mantenimiento.

nodos			
nid	nombre	ubicacion	descripcion

Figura 3.6: Tabla nodos.

- **nid:** identificador del nodo.
- **nombre:** nombre del nodo.
- **ubicacion:** ubicación del nodo.
- **descripcion:** pequeña descripción del equipo.

Durante la realización de este proyecto, el cluster constaba de 8 nodos: aoclp1,aoclp2,....,aoclp8.

### 3.1.6. Tabla: carga local prevista

En esta tabla se puede almacenar la información referente a la carga que los nodos pueden tener no proveniente del sistema CISNE. Dependiendo de la utilización del cluster, los procesadores estarán parcialmente ocupados por las aplicaciones de los usuarios. En esos casos, ese uso consumiría recursos de los nodos e influiría negativamente en el tiempo de ejecución de las aplicaciones paralelas. Es útil saber si existió esta carga en la ejecución de un determinado trabajo.

Se da el hecho de que en las clases de prácticas de una determinada asignatura se podría caracterizar, aproximadamente, la carga de las aplicaciones que se utilizan.

Normalmente se utilizan las mismas aplicaciones y bastaría con monitorizar una sesión para ver el consumo de recursos en cuanto a memoria y cpu. La descripción de la tabla sería la siguiente:

carga_local_prevista	
nid	
fecha_inicial	
fecha_fina	
memo	
cpu	

Figura 3.7: Tabla carga local prevista.

- **nid:** identificador del nodo al que hacemos referencia.
- **fecha inicial:** día y hora a la que asociamos una carga.
- **fecha final:** día y hora a la que finaliza la carga.
- **memo:** memoria ocupada durante el periodo que dura la carga.
- **cpu:** CPU ocupada durante el periodo que dura la carga.

### 3.2. Carga de la Base de Datos

Como se describe en el capítulo de análisis, se dispone de información sobre 2400 trabajos. Ahora bien, no todos estos trabajos se han introducido en la base de datos. La información de 1560 trabajos se han introducido en el histórico mientras que la de 840 se han reservado para que hagan de conjunto de test. Con ese conjunto se medirá el resultado del método de predicción.

En la Figura 3.8 se puede ver la cantidad de trabajos de cada grupo que se han destinado a formar parte del histórico o a formar parte del grupo de test.

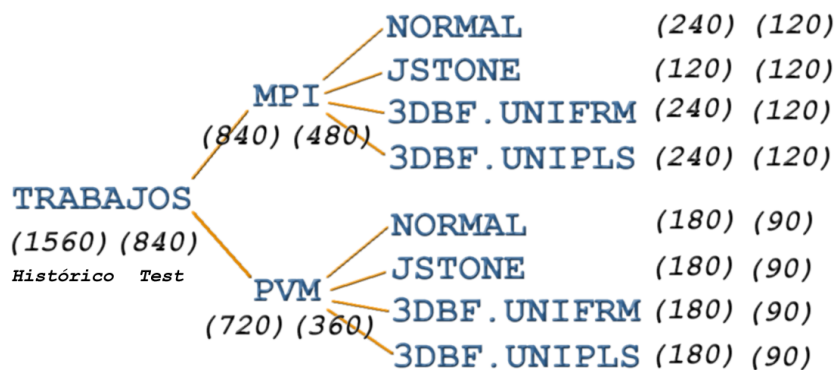


Figura 3.8: Número de trabajos para el histórico y para experimentación

Las trazas de estos trabajos están en unos ficheros llamados “*output*” que se encuentran en un árbol de directorios que podría ser perfectamente el árbol representado en la Figura 3.8. Así pues se ha creado un programa, implementado en C, que escanea los ficheros que contienen las trazas e inserta todos los datos en el histórico. Se lee traza por traza y se insertan los datos pertinentes en las distintas tablas del histórico. La manera de utilizar este escáner desde la línea de comando sería la siguiente:

```
./scanner <ruta al fichero> <tipo> <política> <núm. secuencia trabajo>
```

**ruta al fichero:** especifica la ruta absoluta hasta el fichero de trazas.

**tipo:** especifica si las trazas pertenecen a trabajos de tipo MPI o PVM.

**política:** especifica una de las cuatro políticas: NORMAL, JSTONE, 3DBF.UNIFRM, 3DBF.UNIPLS.

**núm. secuencia trabajo** especifica el número de trabajo que se asigna al primer trabajo de la traza. Todos los ficheros de trazas comienzan con el trabajo número 1 pero son trabajos diferentes por lo tanto tendrán números identificativos diferentes en la base de datos. En la Figura 3.9 se puede observar como cada vez que se escanea un fichero de trazas se incrementa el número de secuencia.

Una ejecución de este comando para un fichero de trazas cargaría toda la información de las trazas en la base de datos. Como las trazas se encuentran en diferentes ficheros, se ha creado un *script* para cargar todos los ficheros. El *script* se llama *cargardb* y carga uno por uno todos los ficheros de trazas que se le indiquen. El contenido de ese *script* para cargar las trazas en el histórico es el siguiente:

```
./scanner /home/man/CISNE/trazas/mpi/3dbf.uniform/output1 mpi 3dbf.uniform 0
./scanner /home/man/CISNE/trazas/mpi/3dbf.uniform/output2 mpi 3dbf.uniform 120
./scanner /home/man/CISNE/trazas/mpi/3dbf.unipls/output1 mpi 3dbf.unipls 240
./scanner /home/man/CISNE/trazas/mpi/3dbf.unipls/output2 mpi 3dbf.unipls 360
./scanner /home/man/CISNE/trazas/mpi/jstone/output1 mpi jstone 480
./scanner /home/man/CISNE/trazas/mpi/normal/output1 mpi normal 600
./scanner /home/man/CISNE/trazas/mpi/normal/output2 mpi normal 720
./scanner /home/man/CISNE/trazas/pvm/3dbf.uniform/output1 pvm 3dbf.uniform 840
./scanner /home/man/CISNE/trazas/pvm/3dbf.uniform/output2 pvm 3dbf.uniform 930
./scanner /home/man/CISNE/trazas/pvm/3dbf.unipls/output1 pvm 3dbf.unipls 1020
./scanner /home/man/CISNE/trazas/pvm/3dbf.unipls/output2 pvm 3dbf.unipls 1110
./scanner /home/man/CISNE/trazas/pvm/jstone/output1 pvm jstone 1200
./scanner /home/man/CISNE/trazas/pvm/jstone/output2 pvm jstone 1290
./scanner /home/man/CISNE/trazas/pvm/normal/output1 pvm normal 1380
./scanner /home/man/CISNE/trazas/pvm/normal/output2 pvm normal 1470
```

Figura 3.9: Script para cargar el Histórico

Toda esta información sobre los trabajos cargada en la base de datos forma la base de experiencias sobre la que trabajar. Ahora se puede crear un método de predicción para estimar el tiempo que un trabajo nuevo tardará en ejecutarse.

Antes de describir el método de predicción, es importante notar que el histórico también puede ser utilizado para tener información sobre lo que ha pasado durante



las ejecuciones de los trabajos. Una buena idea sería poder navegar por esta base de datos con un sistema de visualización agradable. Una página web o un programa cliente implementado con cualquier lenguaje compatible con *MySQL* que mostrara la información de manera adecuada sería de gran utilidad para los administradores del sistema CISNE.

A continuación se presentan algunos ejemplos de la información que se puede obtener mediante consultas en SQL. En la Figura 3.10 se observa un trabajo que ejecutó la aplicación llamada “*is.A.4*”, que se identifica por su *aid=12*. En la Figura 3.11 se pueden ver otros trabajos que ejecutaron esta aplicación mientras la política de gestión de cola era la “*3dbf.uniform*”. Obsérvese en esta figura, en relación a lo comentado sobre la dificultad de predecir el tiempo de ejecución, que la misma aplicación ejecutada cuando gobernaba la misma política de gestión tardó entre 30 y 150 unidades de tiempo.

```
man@localhost: /home/man - Terminal - Konsole
mysql> select aid,nombre,num_cpus,memo_necesaria from aplicaciones where aid='12';
+-----+-----+-----+-----+
| aid | nombre | num_cpus | memo_necesaria |
+-----+-----+-----+-----+
| 12 | is.A.4 | 4 | 32 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
```

Figura 3.10: Información sobre la aplicación *is.A.4*

```
man@localhost: /home/man - Terminal - Konsole
mysql> select jid,aid,politica,tmp_ejecucion from trabajos where aid='12' and
politica='3dbf.uniform';
+-----+-----+-----+-----+
| jid | aid | politica | tmp_ejecucion |
+-----+-----+-----+-----+
| 18 | 12 | 3dbf.uniform | 30 |
| 25 | 12 | 3dbf.uniform | 30 |
| 54 | 12 | 3dbf.uniform | 150 |
| 61 | 12 | 3dbf.uniform | 121 |
| 75 | 12 | 3dbf.uniform | 60 |
| 98 | 12 | 3dbf.uniform | 80 |
| 138 | 12 | 3dbf.uniform | 90 |
| 145 | 12 | 3dbf.uniform | 40 |
| 174 | 12 | 3dbf.uniform | 120 |
| 181 | 12 | 3dbf.uniform | 40 |
| 195 | 12 | 3dbf.uniform | 60 |
| 218 | 12 | 3dbf.uniform | 60 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)

mysql>
```

Figura 3.11: Otros trabajos que ejecutaron la aplicación *is.A.4*

Otro tipo de información que da una idea de hasta dónde se puede llegar a través de la base de datos histórico es, la posición en la que entró un trabajo en la cola, y cuanto tiempo estuvo en cada posición hasta que empezó su ejecución. Esta información se muestra en la Figura 3.12 referente al trabajo 315. Otro tipo de información se presenta en la Figura 3.13, por medio del campo *mpl(multiprogram level*, que muestra con cuantos trabajos coincidió en los diferentes nodos del cluster.

```
man@localhost: /home/man - Terminal - Konsole <2>
mysql> select nombre,jid,posicion_enCola,tmp_inicial,tmp_final from cola where jid='315';
+-----+-----+-----+-----+-----+
| nombre | jid | posicion_enCola | tmp_inicial | tmp_final |
+-----+-----+-----+-----+-----+
| queue1 | 315 | 25 | 1117785209 | 1117785259 |
| queue1 | 315 | 24 | 1117785259 | 1117785260 |
| queue1 | 315 | 23 | 1117785260 | 1117785291 |
| queue1 | 315 | 22 | 1117785291 | 1117785292 |
| queue1 | 315 | 21 | 1117785292 | 1117785332 |
| queue1 | 315 | 20 | 1117785332 | 1117785334 |
| queue1 | 315 | 19 | 1117785334 | 1117785375 |
| queue1 | 315 | 18 | 1117785375 | 1117785441 |
| queue1 | 315 | 17 | 1117785441 | 1117785473 |
| queue1 | 315 | 16 | 1117785473 | 1117785487 |
| queue1 | 315 | 15 | 1117785487 | 1117785518 |
| queue1 | 315 | 14 | 1117785518 | 1117785660 |
| queue1 | 315 | 13 | 1117785660 | 1117785661 |
| queue1 | 315 | 12 | 1117785661 | 1117785681 |
| queue1 | 315 | 11 | 1117785681 | 1117785692 |
| queue1 | 315 | 10 | 1117785692 | 1117785693 |
| queue1 | 315 | 9 | 1117785693 | 1117785765 |
| queue1 | 315 | 8 | 1117785765 | 1117785874 |
| queue1 | 315 | 7 | 1117785874 | 1117786097 |
| queue1 | 315 | 6 | 1117786097 | 1117786129 |
| queue1 | 315 | 5 | 1117786129 | 1117786241 |
| queue1 | 315 | 4 | 1117786241 | 1117786243 |
| queue1 | 315 | 3 | 1117786243 | 1117786263 |
| queue1 | 315 | 2 | 1117786263 | 1117786314 |
| queue1 | 315 | 1 | 1117786314 | 1117786316 |
+-----+-----+-----+-----+-----+
25 rows in set (0.00 sec)
```

Figura 3.12: Posición de entrada en la cola del trabajo 315

```
man@localhost: /home/man - Terminal - Konsole
mysql> select jid,nid,aid,tmp_inicial,tmp_final,mpl from ejecucion
where jid='315';
+-----+-----+-----+-----+-----+-----+
| jid | nid | aid | tmp_inicial | tmp_final | mpl |
+-----+-----+-----+-----+-----+-----+
| 315 | 5 | 12 | 1117786316 | 1117786356 | 2 |
| 315 | 6 | 12 | 1117786316 | 1117786356 | 2 |
| 315 | 7 | 12 | 1117786316 | 1117786356 | 2 |
| 315 | 8 | 12 | 1117786316 | 1117786356 | 2 |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>
```

Figura 3.13: Información de los nodos donde se ejecutó el trabajo 315

A partir de la información de esta consulta se podría llegar a saber, qué trabajos se estaban ejecutando en esos nodos en los instantes considerados.

Aunque no se disponga de una herramienta de visualización de los datos a alto nivel, mediante consultas como las que se han mostrado en esta sección es posible obtener toda la información de una forma clara y concisa. Es importante notar que si se quisiera obtener esta información directamente de las trazas sería una tarea desalentadora. Por ese motivo, se puede decir que, con el histórico se ha añadido una mejora a nivel de poder consultar, de manera rápida y eficaz, información sobre las ejecuciones del sistema.

### 3.3. Herramienta de Predicción

El método de predicción implementado se basa en comparar trabajos utilizando la técnica de las diferencias entre características para hacer la predicción. El punto de partida será encontrar las características que definan los aspectos relevantes, para la predicción, de un *trabajo*. Esos aspectos serán los que hacen referencia a la *aplicación a ejecutar* y al *estado del cluster*. Con ese conjunto de características se podrá comparar un nuevo trabajo con la base de experiencias. Para ello se utilizará la idea de distancia entre características. La suma de las distancias de cada característica nos dará como resultado la distancia total. Esta distancia total será la referencia para saber el grado de similitud de un nuevo trabajo con cualquier otro ejecutado en el pasado.

En la Figura 3.14 se pueden ver las tareas que realizará la herramienta de predicción ante la llegada de un nuevo trabajo al sistema:

1. Obtener la información de la aplicación que forma el trabajo y la del estado del cluster en la actualidad.
2. Insertar la información en el histórico
3. Consultar la base de datos *histórico* para obtener la lista de todos los trabajos anteriores que ejecutaron esa aplicación. En caso de no existir ejecuciones de la misma aplicación, se buscan ejecuciones de aplicaciones similares.
4. Comparar el nuevo trabajo con los de la lista del histórico. Se compara el estado del cluster a la llegada del nuevo trabajo con el estado del cluster que había cuando llegó el trabajo guardado en el histórico.
5. Elegir el trabajo más parecido utilizando una función de distancia.
6. Dar como resultado el tiempo que tardó en ejecutarse el trabajo con más grado de similitud.

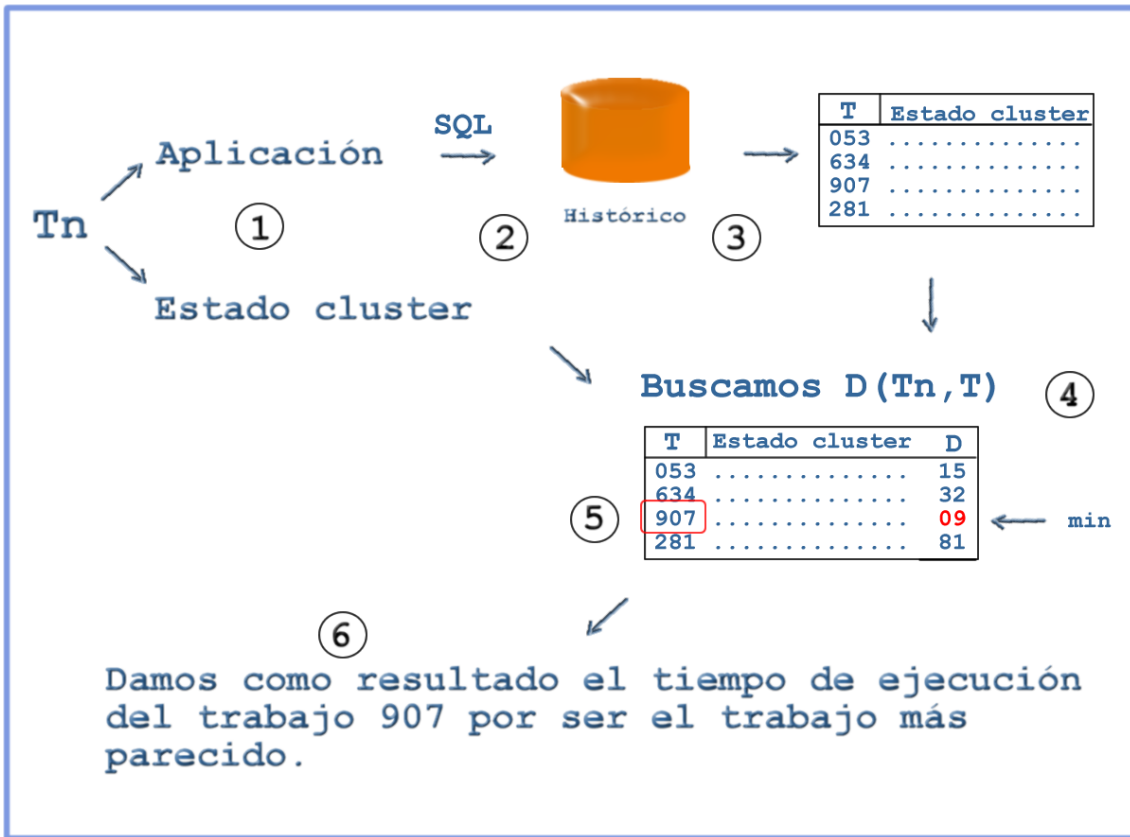


Figura 3.14: Sistema inicial de predicción.

### 3.3.1. Conjunto de características

En la base de datos existe gran cantidad de información sobre los trabajos ejecutados, por lo tanto es necesario seleccionar las características que participarán en la predicción. En principio, las características posibles son las siguientes (divididas en tres grupos):

1. Características de la aplicación:

**Nombre aplicación.** Nombre que describe a la aplicación.

**Tipo aplicación.** El tipo de aplicación: PVM, MPI.

**Número de procesadores necesarios.** Cantidad de procesadores que la aplicación necesita para su ejecución.

**Memoria Necesaria.** Cantidad de memoria que la aplicación necesita para su ejecución.

2. Características del estado de la cola:

**Nombre de la cola.** Identifica diferentes colas.

**Política.** El tipo de política de gestión de colas.

a) A la llegada del trabajo:

**Posición de entrada.** Informa de la cantidad de trabajos que hay por delante.

**Trabajos.** Identificadores de los trabajos que hay en la cola.

b) Al inicio de la ejecución del trabajo:

**Trabajos.** Identificadores de los trabajos que quedan en la cola.

3. Características del estado de los nodos:

**Nodos.** Nodos donde se ejecutó la aplicación.

**Nodo padre.** Nodo donde se lanza la aplicación.

**MPL.** Grado de multiprogramación que hay en cada nodo.

**Trabajos.** Identificadores de los trabajos que se están ejecutando en cada nodo.

**Memoria.** Memoria utilizada en cada nodo.

El número de características es elevado, teniendo en cuenta que algunas características, como el grado de multiprogramación de cada nodo, la memoria utilizada o los identificadores de los trabajos que coinciden ejecutándose en un mismo nodo, van multiplicadas por el número de nodos que participan en la ejecución. En el peor de los casos estaríamos hablando de unas 35 características, y en el mejor de 17. Un número elevado de características conlleva inconvenientes, entre los que se encuentran los siguientes:

- Riesgo de utilizar características inadecuadas.
- Alta correlación entre características. La aportación que algunas características ofrecen para agrupar trabajos se compensa con la de otras características y el resultado final no sirve para diferenciar grupos.
- Fronteras de decisión no lineales. Se hace extremadamente difícil decidir a partir de que valor hay que decantarse por un trabajo o por otro.

Por otra parte, algunas de estas características son de difícil comparación. Por ejemplo, a la hora de comparar dos listas de distintos trabajos y de distintos tamaños es difícil tomar un criterio adecuado para dar un índice de similitud o diferencia.

Por estos motivos es conveniente realizar un preprocesado de los datos. El preprocesado suele contemplar:

**Codificación de los datos.** Se normalizan las características dentro de un rango establecido.

**Datos aislados.** Los datos aislados provocan un error en la predicción. Ignorar estos datos mejora el resultado pero podemos estar eliminando un dato que mejore el resultado.

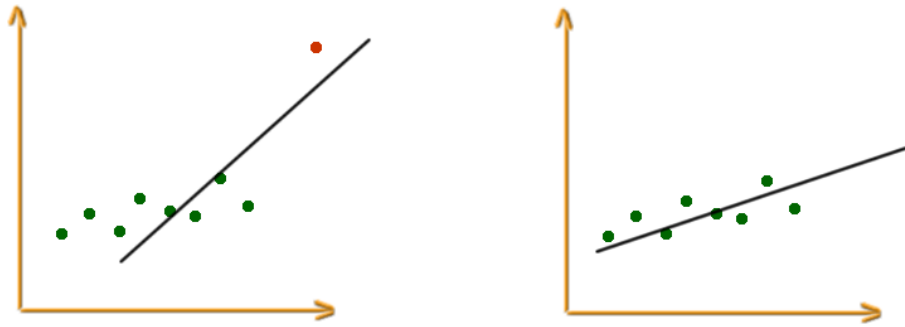


Figura 3.15: Eliminación de los datos aislados.

**Tratamiento de los datos incompletos o incorrectos.** Se intenta modificar los datos incorrectos o incompletos de la base de datos o incluso se eliminan.

**Transformación de las características en otras más eficientes.** A este proceso se le llama *extracción de características* porque suele conllevar una reducción en la dimensionalidad del espacio de características.

Este preprocesado de los datos ayuda a manejar la información que se utilizará para las comparaciones. Entre los puntos implementados se destaca la extracción de características. El hecho de transformar la información contenida en varias características, agrupándola en una sola característica reduce considerablemente el total de características a valorar. A continuación se profundiza más en este aspecto.

### 3.3.2. Extracción de Características

La obtención de una nueva información (característica) sobre las aplicaciones, ha permitido mejorar tanto el error como la velocidad del método implementado. Se trata del termino *jiffy (jiffies)* que intentamos describir a continuación.

**Jiffies.** Unidades (de tiempo) necesarias para la ejecución de una tarea. No es una medida absoluta. El tiempo asociado a un jiffy está relacionado con la frecuencia de reloj del sistema operativo.

Otra definición del termino Jiffies puede ser: lapso de tiempo utilizado como mínima unidad de asignación en algunos sistemas Linux.

A partir de ahora se utilizarán los jiffies como otra característica de las aplicaciones. En realidad es una medida equivalente, si sabemos el factor de equivalencia, al tiempo de CPU pero independiente del procesador del sistema. Esta información ha sido obtenida de los estudios de estabilidad realizados en el sistema, donde se han obtenido los consumos medios en jiffies para cada una de las aplicaciones. Además el consumo en jiffies de cada aplicación, es diferente según la política del sistema.

Para poder utilizar esta información se añade una nueva tabla a la base de datos que estará relacionada con la tabla aplicaciones. Esta tabla se llama superficie (por la idea de superficie computacional que ofrecen los jiffies).

superficie
aid
politica
jiffies
cpu

Figura 3.16: Nueva tabla con los jiffies y tiempo de cpu.

Gracias a los jiffies se pueden cambiar dos de las características analizadas en el apartado anterior. Tanto en las características del estado de la cola como en las del estado del cluster, existía una característica que hacía referencia a qué trabajos había en la cola o que trabajos había en los nodos. Ahora se sustituye la lista de trabajos por la suma de los jiffies de los mismos. Con este cambio se simplifica la comparación, ya que comparar una lista de 30 o 40 trabajos resulta costoso en tiempo y dificulta la predicción. Gracias a esta modificación sólo se necesita comparar un número. Algunas de las ventajas de esta representación son:

- Es mucho más fácil comparar dos números que una lista de  $n$  trabajos con las consultas anidadas que eso significa.
- Es independiente del número de trabajos. Cobra más importancia el tamaño total de los trabajos que el número, ya que un trabajo grande puede equivaler en tiempo a  $n$  trabajos pequeños.

Siguiendo con el propósito de reducir la dimensionalidad del conjunto de características hay que tener en cuenta un aspecto en cuanto a la búsqueda de trabajos en la base de datos. Ante la búsqueda de todos los trabajos que ejecutaron una determinada aplicación, esta búsqueda es rápida en el caso de que la aplicación se haya ejecutado anteriormente. En el caso de que la aplicación no se haya ejecutado anteriormente el sistema busca la aplicación más parecida según tres características:

**Tipo.** El tipo de aplicación: PVM,MPI.

**Número de CPUs.** Número de CPUs que necesita para ejecutarse.

**Memoria.** Memoria que necesita para ejecutarse.

En cualquiera de los dos casos, e independientemente de que exista alguna aplicación similar en el histórico o no, se utiliza el nombre de la característica (si aplica) y la política en la propia consulta SQL. Esta manera de filtrar los trabajos por nombre de la aplicación y política de gestión reduce el número de comparaciones a realizar y produce un error menor. Así pues, estas 2 características de la aplicación no se utilizan en el cálculo de las distancias porque ya se han utilizado en la pre-selección de trabajos. Esto significa que el número de características a utilizar se reduce en 2. Hay que hacer notar que a partir de ahora, en todos los conjuntos de características que se describan, existirán dos características implícitas. Estas características serán el nombre de la aplicación y la política del sistema.

Respecto a la información sobre el estado de la cola, aunque no forma parte de este trabajo predecir el tiempo de espera en cola, si que interesa conocer cómo estaba la cola cuando el trabajo empezó a ejecutarse. Es importante porque este estado puede dar una idea de lo saturado que estaba el sistema durante la ejecución del trabajo. Es decir, si una cola con 50 trabajos puede significar que un trabajo, que se está ejecutando, compartirá recursos durante toda su ejecución. Por el contrario una cola vacía significará, que el trabajo que se está ejecutando, dispondrá de todos los recursos durante la mayor parte de su ejecución. Este hecho influye de manera muy importante en el tiempo que el trabajo tardará en ejecutarse. Teniendo en cuenta lo citado hasta el momento, las características que interesan principalmente son:

**Política.** Política de gestión del sistema.

**TrabjosCola.** Número de trabajos que había en la cola en el momento del inicio de la ejecución.

**JiffiesCola.** Suma de los Jiffies de los trabajos que había en la cola en el momento que el trabajo empezó la ejecución.

A destacar que con estos cambios, ahora la atención sobre el estado de la cola se centra en cómo estaba la cola en el inicio de la ejecución de un trabajo, mientras que antes se centraba en cómo estaba la cola cuando llegó el trabajo.

En relación a las características del estado del cluster, un cambio realizado ha sido eliminar la característica que almacena los nodos donde se ejecutó el trabajo. Otro característica que se ha eliminado ha sido la que contenía la memoria consumida en cada nodo participante en la ejecución de un trabajo. Esa característica se ha sustituido por la suma de memoria utilizada de todos los nodos que participan en la ejecución del trabajo. El grado de multiprogramación de cada nodo, se ha sustituido por la suma de los grados de multiprogramación de todos los nodos participantes en la ejecución.

Por lo tanto, después de todas las modificaciones que se han introducido, el conjunto de características resultante es:

**TrabjosCola.** Número de trabajos que había en la cola en el momento del inicio de la ejecución.

**JiffiesCola.** Suma de los Jiffies de los trabajos que había en la cola en el momento que el trabajo empezó la ejecución.

**Nodo Padre.** Es el nodo principal de los nodos que participan en la ejecución del trabajo. Es el nodo donde se lanza el trabajo.

**MPL.** Suma de los grados de multiprogramación de todos los nodos.

**JiffiesNodos.** Suma de los jiffies de todos los trabajos que había en los nodos.

**Memoria.** Suma de la memoria ocupada de todos los nodos.

Estas serán las característica que se utilizarán para calcular las distancias entre los trabajos. Se realizarán combinaciones para comprobar que características minimizan el



error. El resultado de estas combinaciones se podrá ver en el capítulo de resultados. Algunas de las ventajas que aporta esta reducción de características son:

1. Cuanto menor es la dimensión del espacio de entrada menor es el número de parámetros a determinar.
2. Mayor velocidad de cálculo y de aprendizaje.
3. Ambos parámetros tienen una dependencia cuadrática de la dimensión por lo tanto su reducción es relevante.

Pero también implica un gran inconveniente:

- Se puede perder mucha información.

Se ha intentado que la pérdida de información en el proceso de reducción de características haya sido la mínima posible. En todo caso se ha perdido información de detalle que ha sido englobada en nuevas características. Ya que esta encapsulación se ha realizado teniendo en cuenta el objetivo de este trabajo, lo que se ha logrado es mantener la información relevante de una manera más ágil de manejar para las comparaciones.

Una vez creado el conjunto de características apropiado el siguiente paso será calcular la distancia del conjunto de características del trabajos al que haya que estimar el tiempo de ejecución y de los trabajos del histórico. El trabajo que obtenga la distancia mínima será el trabajo que se considerará más parecido. Ese proceso se describe a continuación.

### 3.3.3. Elección del trabajo más parecido

Existen diferentes maneras para calcular la distancia. Entre ellas está la distancia Euclídea y la distancia de Manhattan, cuyas fórmulas se expresan a continuación:

$$D_{Euclidea}(T_n, T_h) = \sqrt{\sum_{i=1}^n (C_i T_n - C_i T_h)^2}$$

$$D_{Manhattan}(T_n, T_h) = \sum_{i=1}^n |C_i T_n - C_i T_h|$$

Donde  $T_n$  es el trabajo nuevo del que queremos predecir el tiempo de ejecución,  $T_h$  es un trabajo del histórico con el que se está comparando.  $C_i T_n$  es la característica  $i$  del trabajo nuevo, y  $C_i T_h$  es la característica  $i$  del trabajo del histórico.

En la Figura 3.17 se puede ver la representación gráfica de estas dos maneras diferentes de calcular la distancia.

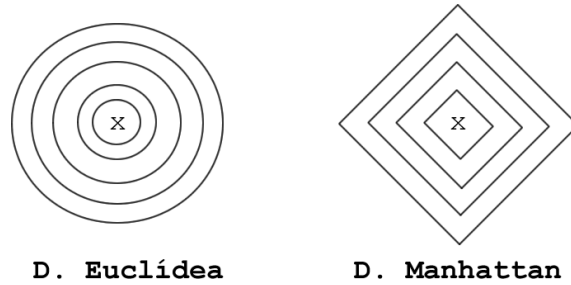


Figura 3.17: Distancia Euclídea y de Manhattan.

La primera intención fue utilizar la distancia de Manhattan ya que las operaciones que realiza son sumas y su cálculo es rápido. Pero las diferentes pruebas realizadas han decantado la balanza por la distancia Euclídea, por obtener un error menor. Por lo tanto, la función implementada para calcular la distancia de la suma de características es la distancia Euclídea.

### 3.3.4. Sintonización

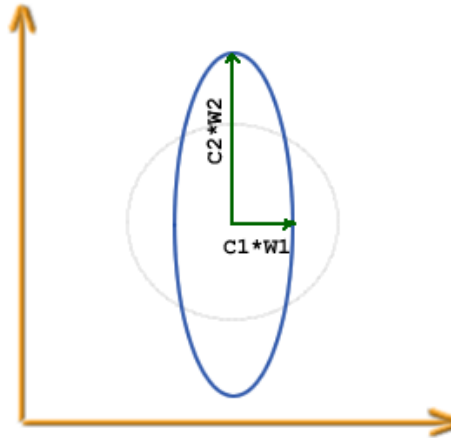
Con este proceso se intenta mejorar la función de la distancia. Se ha descrito anteriormente que la función utilizada para calcular la distancia entre trabajos es la suma de las distancias entre características:

$$D_{Euclídea}(T_n, T_h) = \sqrt{\sum_{i=1}^n (C_i T_n - C_i T_h)^2} \quad (3.1)$$

Ahora bien, una de las mejoras que se pueden introducir consiste en asociar peso a cada una de las características. De esta forma, habrá características que serán más importantes que otras en el cálculo de la distancia total. Esa importancia estará representada por un coeficiente que multiplicará la característica. Modificando ese coeficiente se puede dar más o menos importancia a la característica. Por lo tanto, existe una segunda versión de la fórmula para calcular la distancia:

$$D_{Euclídea}(T_n, T_h) = \sqrt{\sum_{i=1}^n W_i (C_i T_n - C_i T_h)^2} \quad (3.2)$$

El efecto que esto produciría en un espacio de 2 características se muestra a continuación:



$W_i$  es el peso asociado a la característica  $C_i$ . Elegir el peso adecuado para cada característica no es tarea fácil, no siempre lo intuitivo se refleja en los resultados, sobre todo en sistemas tan complejos. Para una cantidad pequeña de características se puede hacer una búsqueda, más o menos, exhaustiva de los pesos sin demasiado coste en tiempo. Los pesos, en este trabajo, varían entre 0 y 100 en incrementos o decrementos de 5 unidades, con lo que buscar las posibles combinaciones de 2 o 3 características es viable. La búsqueda de los pesos que hacen que el error sea mínimo es lo que se llama sintonización.

En un marco con una cantidad de características grande, la búsqueda de los pesos de forma exhaustiva supondría días o semanas de computo. Una manera alternativa que requiere emplear tiempo en la implementación, sería los algoritmos genéticos de los que se hablaba en el inicio de este trabajo. Habría que encontrar un individuo (conjunto de pesos definidos por cromosomas) que minimice el error de la predicción. La idea básica es que se empezara con un conjunto de individuos (vectores de pesos) que irían evolucionando en cada iteración en base a una función de adaptación. Esta función de adaptación tendría en cuenta el resultado obtenido en la predicción y cuanto menor sea el error medio obtenido por el individuo, mayor será su posibilidad de existir en la siguiente iteración. De esta manera se conseguiría una sintonización que aunque no tendría por que ser la mejor si sería una solución aceptable. Sin embargo, en este trabajo se ha optado por la reducción al mínimo de características y no se cree oportuno implementar la sintonización con algoritmos genéticos dado el gasto de tiempo que supondría.

### 3.3.5. Aceleración del proceso de cálculo

Otro aspecto que se ha tenido presente en el diseño del método de predicción ha sido minimizar el tiempo que la herramienta necesita para dar un resultado. En este sentido se han realizado diferentes esfuerzos.

La obtención de las características necesarias para la comparación puede ser más o menos costosa en cuanto a tiempo. Se deben obtener las características del trabajo nuevo y las características de los trabajos del histórico que se hayan seleccionado. Las características del trabajo nuevo son accesibles en el sistema real a través de las clase

*job* con lo que el coste en tiempo es mínimo. Ahora bien, en este trabajo los datos se obtienen de la traza utilizada para el test. Esto significa que hay que acceder a disco con la consecuente pérdida de tiempo. Independientemente de esto, donde se emplea más tiempo es en la obtención de los datos de todos los trabajos del histórico que van a ser comparados, y en el cálculo de las diferencias. Recordemos las 6 características:

**TrabjosCola.** Número de trabajos que había en la cola en el momento del inicio de la ejecución.

**JiffiesCola.** Suma de los Jiffies de los trabajos que había en la cola en el momento que el trabajo empezó la ejecución.

**Nodo Padre.** Es el nodo principal de los nodos que participan en la ejecución del trabajo. Es el nodo donde se lanza el trabajo.

**MPL.** Suma de los grados de multiprogramación de todos los nodos.

**JiffiesNodos.** Suma de los jiffies de todos los trabajos que había en los nodos.

**Memoria.** Suma de la memoria ocupada de todos los nodos.

La obtención de las características TrabajosCola y MPL supone únicamente la consulta a un campo de la base de datos. La característica MPL se consigue accediendo a cada nodo donde se ejecutó el trabajo. Este dato, también se puede conseguir con una consulta que sume el dato de cada nodo. Sin embargo, con el diseño inicial de la base de datos, obtener las características JiffiesCola y JiffiesNodos supone un importante gasto de tiempo. Hay que obtener la lista de los trabajos, buscar la aplicación que ejecutó cada trabajo, obtener los jiffies de cada aplicación de otra tabla, y finalmente hacer la suma de todos los jiffies. En los nodos el número de trabajos en ejecución no será muy elevado, pero en la cola el número de trabajos puede ser del orden de 50 o 100.

Una manera de solucionar este problema, es hacer los cálculos en el momento que un trabajo inicia la ejecución. En el sistema real se disponen de todos los datos necesarios. Se calcularía una sola vez y se guardaría en unas nuevas tablas. De esta manera cuando en el futuro se quiera obtener los jiffies que había en la cola, sólo habrá que hacer una consulta a una tabla que contendrá el trabajo y la suma de los jiffies que había en ese momento. Evitamos así las consultas múltiples y las operaciones posteriores.

Para poder hacerlo, se han introducido dos tablas nuevas en nuestra base de datos *histórico*.

La tabla **estado inicial nodos** permite obtener de manera directa tanto los jiffies de los trabajos que se estaban ejecutando como la suma de los MPLs de los nodos.

estado_inicial_nodos	
<b>id</b>	
jid	
nid	
mpl	
memo	
jids_nodo	
jiffies	
cpu	

Por otro lado la tabla **estado schedule cola** permite obtener de forma directa el número de trabajos que había en la cola y la suma de sus jiffies.

estado_schedule cola	
<b>id</b>	
jid	
cid	
jids_cola	
jiffies	
cpu	

Con la introducción de estas tablas se obtienen los datos de manera directa con lo que la ganancia en tiempo es notable.

Otro aspecto interesante donde ganar tiempo es evitando las operaciones con decimales. Si, por ejemplo, la normalización de las distancias entre características se había hecho entre (0,1) se cambia a (0,100) para evitar operaciones con decimales.

Con todos estos cambios de diseño e implementación se consigue un sistema más eficiente en la predicción.

## Capítulo 4

# Experimentación Realizada y Resultados

### 4.1. Entorno de Experimentación

Toda la experimentación se ha llevado a cabo en un equipo independiente al sistema LoRaS con el fin de trabajar de manera paralela y sin interferir en su desarrollo. Se ha utilizado un ordenador personal donde se han instalado el sistema operativo y recursos necesarios para la implementación del histórico y herramienta de predicción. A continuación se describen las características del equipo de pruebas:

#### Características del Equipo

AMD Sempron 3000+  
Memoria 512 MB  
313,8 MegaFLOPS

#### S.O. y Software Instalado

Mandriva Linux 2005 (Kernel 2.6.11.6)  
MySQL-Max-4.1.11.  
gcc 3.4.3, gdb 6.3

Observando las características del equipo se puede notar que estamos hablando de un ordenador de bajas prestaciones. Uno de los índices de prestaciones que utilizaremos para medir el resultado de la herramienta de predicción es el tiempo promedio que se tarda en dar el resultado. En este sentido se puede decir que, con toda probabilidad, el equipo donde se instale finalmente la herramienta será superior en prestaciones. Hoy día un ordenador personal de gama baja dobla la capacidad en MegaFLOPS del equipo de pruebas.

En la figura 4.1 se esquematiza su funcionamiento que ha sido descrito con más detalle en el capítulo de *Diseño e Implementación*.



Figura 4.1: Esquema(argumentos/resultado) de la herramienta.

A la herramienta de predicción se le pasan como argumentos el fichero de trazas de una determinada política y el número de trabajo al que queremos predecir el tiempo de ejecución. La herramienta se encarga de extraer los datos necesarios sobre el trabajo y el estado del cluster del fichero de trazas. Estos datos son, principalmente, la aplicación contenida en el trabajo, el estado del cluster y el estado de la cola. Como resultado esta herramienta nos ofrece el tiempo estimado de ejecución, el tiempo real que tardó el trabajo en ejecutarse y el error en la estimación.

Para obtener el dato promedio se predice la totalidad de trabajos contenidos en todos los ficheros de trazas. De esa manera se obtiene el error promedio para todas las trazas disponibles sea cual sea su política. También se puede medir el tiempo en realizar la totalidad de las predicciones y dividirlo por el número de trabajos que hay en el fichero de trazas. De esta forma se obtiene el tiempo promedio consumido en realizar una predicción.

## 4.2. Índices de Prestaciones

Para poder medir y comparar los resultados obtenidos es conveniente establecer los índices de prestaciones. En este caso los índices son dos:

1. **Error medio:** error promedio cometido al predecir el tiempo de ejecución los trabajos contenidos en un fichero de trazas.
2. **Tiempo promedio:** el promedio del tiempo gastado en la predicción de un conjunto de trabajos contenidos en un fichero de trazas.

Con estas dos medidas se dispone de dos índices que permiten evaluar los resultados. De esta forma se puede saber de manera objetiva si las modificaciones, realizadas en la herramienta de predicción, mejoran el sistema.

Cuanto menor sea el error medio y menor el tiempo promedio de predicción, mejor será la herramienta que se ha desarrollado en este trabajo.

### 4.3. Metodología empleada

Antes de continuar se hace preciso notar que previo a la aplicación de la metodología para obtener los resultados finales, se han realizado multitud de pruebas. Estas pruebas se han realizado con el objetivo de implementar y probar las mejoras descritas en el apartado de *Diseño e Implementación*.

Una vez que el sistema contempla las mejoras, es necesario proceder según una metodología que ayude a concretar posibilidades y a llegar a conseguir un buen resultado.

Llegados a este punto se tienen 6 características sobre los trabajos y el estado del cluster. Se trata de buscar las combinaciones entre estas características que minimicen el error medio cometido. Hay varias metodologías para intentar buscar la combinación de características que mejor resultado ofrezca. Algunas de estas metodologías son:

**Métodos exhaustivos** Consiste en probar todas las posibles combinaciones entre las características.

**Selección por pasos** Consiste en seleccionar la característica con la que se comete el menor error e ir añadiendo la siguiente que mejor combine con ella, así sucesivamente hasta que no haya ninguna mejora.

De las dos metodologías enumeradas se ha utilizado la segunda. Se estudiarán las características de una en una y se obtendrá cual minimizan el error de predicción. A partir de esa característica se irán añadiendo otras y se volverá a estudiar el resultado.

Por lo tanto se va a utilizar una metodología de selección por pasos. En una primera iteración no se usarán pesos para ponderar las características. En la segunda iteración se utilizarán pesos que multiplicarán cada una de las características combinadas con el fin de ver si esta modificación mejora los resultados obtenidos.

### 4.4. Resultados obtenidos

En el apartado anterior se ha descrito la metodología a emplear. Aplicando esta metodología, en la figura 4.2 se puede ver el resultado para cada una de las 6 características que forman el conjunto.

Característica	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(1) TrabajosCola	41,0	102,4	2,3	62,8	66,0	30,5	9,6	25,1	44,4
(2) JifflesCola	61,0	95,9	1,7	63,3	45,1	34,4	10,7	27,1	44,4
(3) NodoPadre	51,0	100,6	1,8	123,7	46,1	39,8	37,4	49,2	56,7
(4) MPL	41,9	78,1	0,9	55,7	51,4	33,2	30,8	37,0	42,4
(5) JifflesNodos	33,3	52,3	0,9	86,8	45,9	41,7	30,8	35,0	41,0
(6) Memoria	43,7	167,8	0,6	103,4	50,0	37,5	31,5	46,5	62,1

Figura 4.2: Error medio de predicción con 1 característica



Se observa que la mejor de estas características es la característica *JiffiesNodos*. Y que, con sólo esta característica se consigue un error del 41 % en un tiempo de 0,18 segundos. En este punto se puede observar el beneficio de las mejoras introducidas en el sistema. Se ha pasado de un error medio del 69 % a un error medio del 41 %.

Siguiendo con la metodología, a continuación se muestran las combinaciones de esta característica con las demás:

Características	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(1)	42,5	97,7	1,5	53,4	63,6	38,5	29,7	48,0	<b>48,7</b>
(5)+(2)	42,5	97,7	1,5	53,1	63,6	38,5	29,7	47,7	<b>48,6</b>
(5)+(3)	51,0	100,6	1,8	123,7	46,1	39,8	37,4	49,2	<b>56,71</b>
(5)+(4)	44,6	96,9	1,9	69,5	69,0	38,8	30,8	48,8	<b>51,65</b>
(5)+(6)	44,6	96,9	1,9	69,5	69,0	38,8	30,8	48,8	<b>51,65</b>

Figura 4.3: Error medio de predicción con 2 característica

Ninguna de las combinaciones mejora el resultado obtenido utilizando sólo la característica *JiffiesNodos*. Seguidamente mostramos los mejores resultados de todas las combinaciones realizadas para 3,4,5 y 6 características.

Características	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(4)	32,0	154,9	1,3	88,4	62,2	28,1	10,4	23,9	<b>52,22</b>
(5)+(2)+(1)	30,8	155,7	2,1	96,0	55,9	27,3	5,8	20,4	<b>51,2</b>

Figura 4.4: Error medio de predicción con 3 característica

Características	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(1)+(4)	35,0	70,2	0,9	89,4	61,6	33,3	30,8	36,2	<b>45,2</b>

Figura 4.5: Error medio de predicción con 4 característica

Características	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(1)+(4)+(3)	30,8	159,5	1,3	91,9	88,5	25,9	18,1	29,2	<b>57,8</b>

Figura 4.6: Error medio de predicción con 5 característica

Características	MPI				PVM				Error%
	3dbf.unifrom	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(1)+(4)+(3)+(6)	58,2	148,9	2,2	140,0	40,9	38,1	30,7	44,6	<b>64,1</b>

Figura 4.7: Error medio de predicción con 6 característica

Los datos de estas tablas muestran que el aumento en el número de características empeora los resultados. La figura 4.8 muestra una gráfica donde se puede ver esta relación.

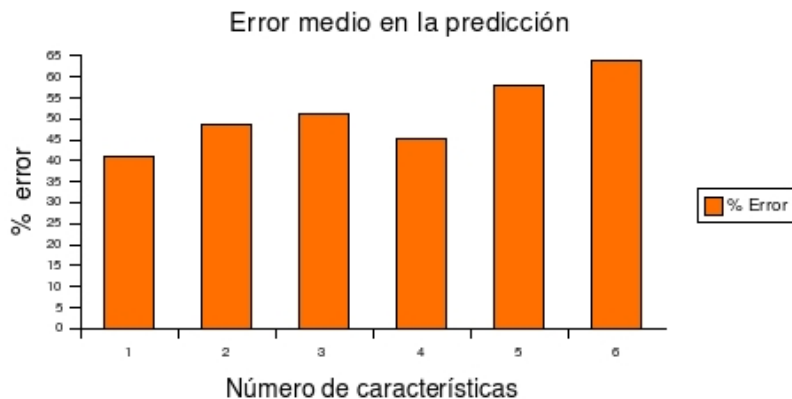


Figura 4.8: Error medio según número de características

Estos resultados son un poco sorprendentes, ya que se podía esperar que, cuantas más características se tuvieran en cuenta, menos error se cometería. Pero cómo ya se ha ido introduciendo en otras secciones, esto no es así. En este caso el aumento en número de características introduce incertidumbre.

Hasta ahora el mejor resultado es de un *error medio* del 41% en la predicción del tiempo de ejecución. Siguiendo con la metodología, seguidamente se muestran los resultados teniendo en cuenta los pesos.

Si se vuelve a mirar la tabla de la figura 4.9 se puede observar que hay características que funcionan mejor para unas políticas que para otras. Esto depende de la naturaleza de cada política. Por este motivo, se utilizarán combinaciones diferentes de pesos según el la política que gobierne el sistema.

Característica	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(1) TrabajosCola	41,0	102,4	2,3	62,8	66,0	<b>30,5</b>	<b>9,6</b>	<b>25,1</b>	44,4
(2) JiffiesCola	61,0	95,9	1,7	63,3	<b>45,1</b>	34,4	10,7	27,1	44,4
(3) NodoPadre	51,0	100,6	1,8	123,7	46,1	39,8	37,4	49,2	56,7
(4) MPL	41,9	78,1	0,9	<b>55,7</b>	51,4	33,2	30,8	37,0	42,4
(5) JiffiesNodos	<b>33,3</b>	<b>52,3</b>	0,9	86,8	45,9	41,7	30,8	35,0	<b>41,0</b>
(6) Memoria	43,7	167,8	<b>0,6</b>	103,4	50,0	37,5	31,5	46,5	62,1

Tiempo medio de predicción= 0.176 segs.

Figura 4.9: Error medio con 1 característica

El hecho de tener las trazas de test separadas por tipo de trabajo y por política facilita esta tarea. Tiene sentido realizar sintonizaciones de los pesos diferentes para cada conjunto de test, dependiendo del tipo de trabajo y de la política utilizada. Así se obtienen sintonizaciones más ajustadas para cada tipo de trabajo y de política. Esta distinción permitirá, ante la llegada de un trabajo nuevo, seleccionar la sintonización dependiendo de su tipo y de que política se esté llevando a cabo en el sistema.

En la figura 4.10 se pueden ver los resultados del predictor sintonizado para 2 características:

Características	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(1)	26,7	52,6	0,9	65,2	48,8	27,8	8,5	21,5	32,21
(5)+(2)	29,0	52,4	1,7	68,1	40,7	26,9	10,4	21,2	31,9
(5)+(3)	52,3	95,4	1,8	86,8	38,8	36,9	30,8	34,9	48,26
(5)+(4)	34,0	55,2	0,9	67,1	54,0	33,0	30,8	34,9	39,31
(5)+(6)	33,3	52,3	0,6	63,9	45,9	37,5	29,5	33,4	37,61

Tiempo medio de predicción= 0.187 segs.

Figura 4.10: Error medio con 2 características sintonizadas

La mejor de las combinaciones entre características obtiene un resultado por debajo del 32% de *error medio*. Se trata de la combinación de la característica *JiffiesNodos* y la característica *JiffiesCola*. En la figura 4.11 se pueden ver los pesos, según el tipo de trabajo y política, que hacen posible este resultado. El peso W1 es el que multiplica a la característica *JiffiesNodos* y el peso W2 a la característica *JiffiesCola*. Esto supone un salto de calidad importante en lo que se refiere a la capacidad de predicción del sistema. En cuanto al *tiempo promedio* que se tarda en hacer la predicción, sólo se emplea una centésima de segundo más.

		W1	W2
MPI	3dbf.uniform	50	50
	3dbf.unipls	70	30
	Jstone	0	100
	Normal	0	100
PVM	3dbf.unifrom	100	0
	3dbf.unipls	50	50
	Jstone	0	100
	Normal	100	0

Figura 4.11: Pesos asociados a *JiffiesNodos* y *JiffiesCola*

En la figura 4.10 se observa que la segunda combinación con mejor resultado es la combinación con la característica *TrabajosCola*. A continuación se muestra las combinaciones de las dos mejores características con la característica *TrabajosCola* con la característica (4) *MPL*:

Características	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(4)	28,8	51,6	0,9	56,1	40,7	25,0	10,2	21,2	30,1
(5)+(2)+(1)	25,7	52,4	0,9	62,3	40,7	25,0	5,4	18,5	29,57

Tiempo medio de predicción= 0.188 segs.

Figura 4.12: Resultados de tres características sintonizado

En esta ocasión se reduce el error. Se obtiene un *error medio* del 30%, lo que supone una pequeña mejora respecto al resultado anterior. Siguiendo con la metodología, veamos que ocurre si combinamos las cuatro características:

Características	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(1)+(4)	29,3	51,6	0,9	56,1	40,7	29,8	15,3	23,4	31,6

Tiempo medio de predicción= 0.188 segs.

Figura 4.13: Resultados de cuatro características sintonizado

Se observa que no hay mejora en el *error medio*. A partir de aquí, en las diferentes combinaciones que se han probado, no se ha encontrado ninguna combinación de 4,5 o 6 características que mejore el error de predicción. En la siguiente tabla puede verse un resumen de los mejores resultados obtenidos:

Opción	Características	Error medio	Tiempo medio (seg.)
1	JiffiesNodos	41,0	0,176
2	JiffiesNodos+JiffiesCola	31,9	0,187
3	JiffiesNodos+JiffiesCola+TrabajosCola	29,6	0,188
4	JiffiesNodos+JiffiesCola+MPL	30,1	0,190
5	JiffiesNodos+JiffiesCola+TrabajosCola+MPL	31,6	0,200

Figura 4.14: Mejores Resultados.

Las opciones 3 y 4 son las que menos *error medio* cometen. A continuación se muestran los errores según la política:

Características	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(4)	28,8	51,6	0,9	56,1	40,7	25,0	10,2	21,2	30,1
(5)+(2)+(1)	25,7	52,4	0,9	62,3	40,7	25,0	5,4	18,5	29,57

Figura 4.15: Mejores combinaciones

Se puede ver que existe una diferencia entre los resultados de las dos opciones para los trabajos de tipo *MPI* con política *normal*. Mientras que en la opción con menor error total se comete un 62.3% de error en este apartado, en la otra opción se comete un 56.1% de error. A pesar de un error medio global un 0.53% mayor, la 4 opción es preferible a la 3 ya que reduce en un 6.2% el error para los trabajos de tipo *MPI* y política *normal*. De esta forma el error para este tipo de trabajos bajaría del 60%. Pensamos que es mejor bajar este error un 6.2% y subir un 0.53% el error global.

La elección como mejor resultado es la opción 4. Las tres características que se han utilizado para el cálculo de las distancias, y por tanto para encontrar los trabajos más parecidos son las siguientes:

**JiffiesNodos:** Suma de los jiffies de los trabajos que se estaban ejecutando en los nodos en el momento del inicio de la ejecución del trabajo.

**JiffiesCola:** Suma de los jiffies de los trabajos que estaban esperando en la cola justo cuando el trabajo empezó a ejecutarse.

**MPL:** Suma de los grados de multiprogramación que había en los nodos en el momento del inicio de la ejecución del trabajo.

La fórmula de la distancia que se ha utilizado para buscar el trabajo más parecido es la siguiente:

$$Distancia(Tn, Th) = \sqrt{W1 * d(JiffiesNodos)^2 + W2 * d(JiffiesCola)^2 + W3 * d(MPL)^2}$$

Para conseguirlo se ha sintonizado el método para cada tipo de trabajo y política. Los pesos asociados a estas características que consiguen este error son:

		W1	W2	W3
MPI	3dbf.uniform	70	25	5
	3dbf.unipls	65	5	30
	Jstone	0	0	100
	Normal	5	10	85
PVM	3dbf.unifrom	50	50	0
	3dbf.unipls	20	60	20
	Jstone	0	90	10
	Normal	10	90	0

Para concluir con este capítulo, resumir que se ha conseguido un método que utilizando tres características comete un *error medio* del **30.1%**. Para ello el método consume menos de **0.2 segundos** de *tiempo promedio* por predicción.

## 4.5. Análisis de los Resultados

Durante todas las pruebas se ha observado que los resultados de algunas políticas has sido siempre peores que los resultados de otras. Si observamos la figura 4.16 los resultados en función del binomio tipo de trabajo política.

Características	MPI				PVM				TOTAL
	3dbf.uniform	3dbf.unipls	Jstone	Normal	3dbf.unifrom	3dbf.unipls	Jstone	Normal	
(5)+(2)+(4)	28,8	51,6	0,9	56,1	40,7	25,0	10,2	21,2	30,1
(5)+(2)+(1)	25,7	52,4	0,9	62,3	40,7	25,0	5,4	18,5	29,57

  Mejores resultados      Peores resultados

Figura 4.16: Comparación de resultados por políticas

Mientras que para la política *Jstone* se obtienen unos excelentes resultados, para otras políticas el resultado supera incluso el 50% de error medio.

Mientras que hay resultados que son lógicos, otros requieren un estudio más detallado para su comprensión. Respecto a los primeros, es lógico que los mejores resultados sean para la política *Jstone* ya que esta política se caracteriza por ejecutar un único trabajo en todo el cluster. Hasta que el trabajo no termina no entra el siguiente a ejecutarse. En estas condiciones los tiempos de ejecución siempre son más parecidos que si la carga del cluster fuera variable.

Por otro lado, la política *Normal* para los tipos de trabajo *MPI* es la que presenta los peores resultados. Si se estudia el resultado del test para esta política se observan algunas particularidades. Se han predicho 72 trabajos que han ejecutado 16 aplicaciones diferentes. De esos 72 trabajos, 22 han sufrido un error de predicción mayor al 100% del tiempo predicho. Lo particular de este caso es que esos 22 han ejecutado sólo 3 aplicaciones. Es decir, el método funciona mal para la predicción de 3 de las 16 aplicaciones. Si no tuviéramos en cuenta esas 3 aplicaciones, el resultado pasaría de un error del 56.1% al 31.1%. En la figura 4.17 se pueden observar las 16 aplicaciones con el número de trabajos que las han ejecutado en el test y el error medio cometido.

Aplicación	Ejecuciones	Error medio
cg.A.2	3	43,3
cg.A.4	6	25,5
ep.A.2	4	24,3
ep.A.4	3	12,7
ep.B.2	4	12,3
ep.B.4	3	9,0
is.A.2	15	103,3
is.A.4	6	32,2
is.B.2	9	47,7
is.B.4	5	21,8
lu.A.2	2	71,5
lu.A.4	3	44,0
mg.A.2	4	129,3
mg.A.4	1	1,0
mg.B.2	1	56,0
mg.B.4	3	137,7
<b>Total general</b>	<b>72</b>	<b>56,1</b>

Figura 4.17: Error medio por aplicación ejecutada.

Se observa que la aplicación *is.A.2* es la aplicación que más veces se ha ejecutado y precisamente es la que peor funciona para la predicción. Tanto es así que si no contáramos con esta aplicación, el error medio quedaría en un 43.6%.

Hay una explicación para estos casos. Este alto porcentaje de error en la predicción del tiempo de ejecución de algunas aplicaciones se debe al estado del cluster. Un estado del cluster que viene definido para esta política según las 3 características utilizadas para definirlo y sus pesos. Según esta configuración, la herramienta de predicción no encuentra en el histórico ejecuciones parecidas. En 14 de las 15 ocasiones sólo encuentra ejecuciones un 15% parecidas en el histórico. El error cometido en estos

casos es muy grande. Sin embargo, en una ocasión, encuentra en el histórico una ejecución idéntica en cuanto a la similitud de las 3 características. En esa ocasión el error cometido es del 0%.

Una apreciación que hay que tener en cuenta es que durante los test no se actualiza el histórico. Esto se hace para no perder el conjunto de test, ya que si se guardaran los datos en el histórico no quedarían trazas inéditas para probar. Este funcionamiento es diferente de como funcionaría el sistema real. En el sistema real, con cada ejecución se actualizaría el histórico. De esta forma, el error se iría reduciendo a medida que el histórico se fuera actualizando ya que las últimas ejecuciones de una aplicación suelen ser las más parecidas a la siguiente.

Por lo tanto cabe esperar que en el sistema real, el error en estos casos aislados se suavice contribuyendo a una disminución del error medio total.

# Capítulo 5

## Conclusiones

### 5.1. Objetivos Alcanzados y no alcanzados

A continuación se enumeran los objetivos iniciales del proyecto valorando si se han alcanzado:

**Creación del histórico:** se ha creado una base de datos que contiene más información de la necesaria para la predicción. De esta forma se puede utilizar el histórico para otras tareas como el estudio del sistema y su administración. Por estas razones se considera alcanzado este objetivo.

**Herramienta de predicción** se ha creado una herramienta off-line de predicción que consigue predecir el tiempo de ejecución en menos de 0.2 segundos y con un error medio del 30 %. Cuando se marcó este objetivo se tomó como referencia el trabajo de [8] que conseguía un error medio del 37 %. Por lo tanto se considera este objetivo cumplido.

**Soporte para políticas de ordenación de cola:** se considera que la herramienta de predicción puede utilizarse para la implementación de las políticas: *Shortest Job First, Largest Job First*, y otras que necesitan hacer estimaciones del tiempo de ejecución de los trabajos.

**Soporte para políticas de selección de trabajos:** se considera cumplido el objetivo de dar soporte para la implementación de la técnica del *backfilling* que, también, necesita estimaciones del tiempo de ejecución.

**Soporte para el turnaround:** se considera cumplido el objetivo de dar soporte para el cálculo del turn-around ya que la herramienta de predicción puede ayudar al simulador del sistema CISNE a obtener el tiempo de espera en cola. De esta forma sumando los dos tiempos se obtendría el turn-around.

Aunque los objetivos han sido conseguidos, la utilidad de este trabajo depende del grado de integración y utilización que esta herramienta alcance en el sistema CISNE. Esto depende de que CISNE siga desarrollándose y de la utilización que se haga de esta herramienta de predicción.



## 5.2. Posibles Ampliaciones

Durante la realización de este trabajo han surgido algunas ideas que podrían dar lugar a interesantes aplicaciones. Estas ampliaciones pueden dotar de más funcionalidad a la herramienta de predicción y el histórico o simplemente mejorar sus prestaciones. Se citan a continuación:

1. Buscar una alternativa de predicción para los casos con mayor porcentaje de error. Ciertas aplicaciones con determinadas políticas aumentan el error de predicción debido. Esto se sucede porque los estados del cluster guardados en el histórico son muy diferentes del que hay en momento en el que hay que hacer la predicción. Para este tipo de casos se puede utilizar la capacidad que tiene la herramienta de conocer la coincidencia entre los estados del cluster. De esta manera si se detecta que la coincidencia es muy baja, se podría optar por buscar la aplicación más parecida de entre las que tuviesen el estado del cluster más parecido. Esta es una posibilidad que no ha dado tiempo de probar y que puede ser interesante.
2. La integración en el sistema CISNE implementado las políticas de ordenación de cola y selección de nodos.
3. Una buena herramienta para la administración y el estudio del sistema CISNE sería una GUI (*Graphic User Interface*) para hacer consultas a la base de datos histórico. Por ejemplo una página web, de manera que mediante conexión con la base de datos se pudiera navegar de forma amigable por los datos del histórico.

## 5.3. Seguimiento de la Planificación

En los capítulos 1 y 2 se presentó la planificación temporal de las tareas a realizar en este trabajo. En algunas de las tareas se han producido desviaciones respecto a esa planificación. En la figura 5.1 se observa la planificación temporal de las tareas y el tiempo real que cada tarea ha necesitado.

En la figura 5.1 se han marcado, en rojo, puntos donde existen diferencias respecto a la planificación inicial. A continuación se enumeran estos puntos marcados en rojo justificando el porqué de estas diferencias respecto a lo planificado:

1. El análisis y documentación se ha extendido considerablemente en el tiempo ya que esta tarea no sólo ha sido analizar el estado del arte y llegar a una solución propuesta. Se han añadido tareas de manera que el proceso de documentación se ha realizado paralelamente con otras tareas, llegando hasta la etapa implementación de la herramienta de predicción. Las principales tareas del análisis y documentación han sido:
  - Estudio del trabajo donde se enmarca este proyecto, para su comprensión y situación dentro del marco adecuado.
  - Estudio de diferentes trabajos ([8, 9, 10, 11, 12, 15, 14]) sobre predicción del tiempo de ejecución para contemplar el estado del arte.

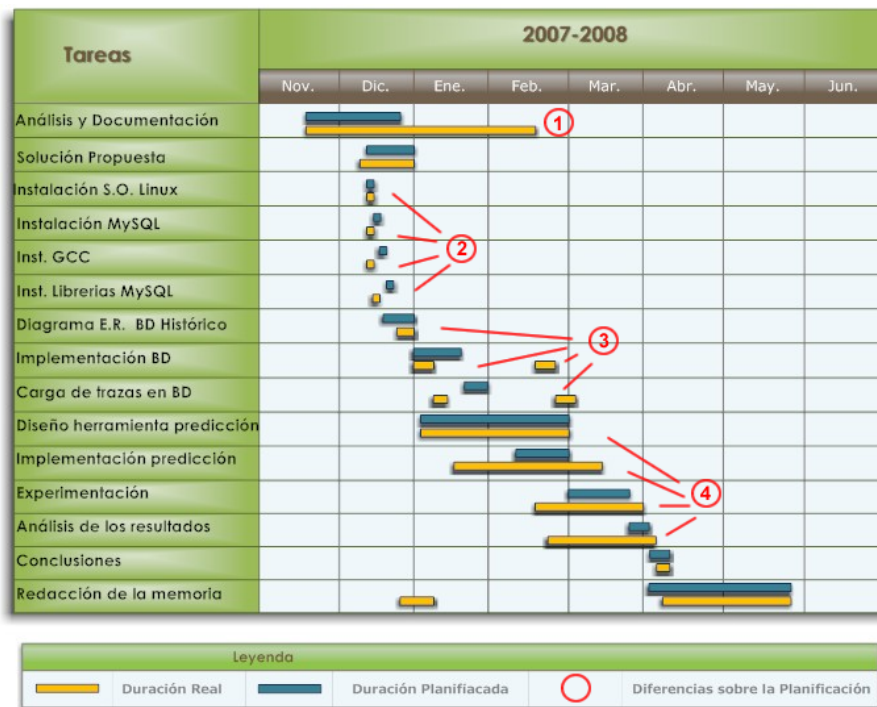
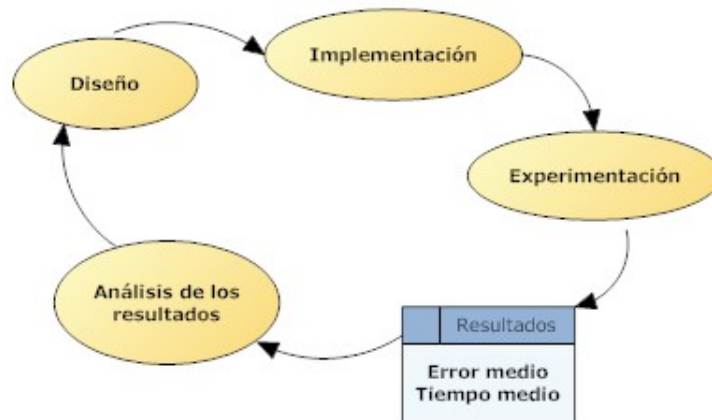


Figura 5.1: Comparativa entre la planificación y el tiempo real de las tareas.

- Configuración de una solución para abordar el problema.
  - Documentación de MySQL.
  - Documentación sobre cómo mejorar la predicción en cuanto a resultados y velocidad.
2. Las tareas de preparación del equipo se han realizado antes de lo planificado. Nada más tomar la decisión de que se utilizaría un equipo off-line con MySQL, como motor de base de datos, se instaló el software necesario de manera conjunta y de manera paralela a otras tareas.
  3. Respecto al diseño, creación y carga de la base de datos, el trabajo también se realizó en menos tiempo del esperado. Sin embargo las mejoras introducidas en la herramienta de predicción para reducir el tiempo de respuesta y el error cometido provocaron que se tuviera que modificar la base de datos y ser nuevamente cargada con datos adicionales.
  4. La herramienta de predicción ha sido lo que más tiempo ha necesitado y donde más diferencias ha habido respecto a la planificación. La explicación para esta diferencia es que conforme se han ido obteniendo resultados y analizando se han ido buscando e introduciendo funcionalidades para mejorar. Esta manera de proceder según la metodología en espiral ha permitido la introducción de muchas mejoras, pero también a provocado que esta parte del trabajo consumiera más tiempo del previsto inicialmente. En la siguiente figura se pueden los pasos a seguir en cada una de las vueltas de la espiral:

## Desarrollo de la herramienta de predicción



# Referencias

- [1] E. Martí, ***“Apunts de l’Assignatura Bases de Dades I.”*** Universitat Autònoma de Barcelona.
- [2] María Vanrell, ***“Apunts de l’Assignatura Intel·ligència artificial I.”*** Universitat Autònoma de Barcelona.
- [3] Jordi Vitrià, ***“Apunts de l’Assignatura Intel·ligència artificial II.”*** Universitat Autònoma de Barcelona.
- [4] M. J. Flynn, ***“Very High Speed Computing Systems”*** En Proceedings of the IEEE, Volume 54(12), pp. 1901-1909, Diciembre 1966
- [5] M. J. Flynn, ***“Some Computer Organizations and Their Effectiveness”*** n IEEE Transactions on Computing, Vol. C-21, No. 9, pp. 948-960 - Septiembre 1972.
- [6] E. E. Johnson, ***“Completing an MIMD multiprocessor taxonomy”*** En Computer Architecture News, Volume 16(3), pp. 44-47 - Junio 1988.
- [7] Referencias Mauricio Hanzich, ***“Combinando Space y Time-Sharing en una NOW no Dedicada”*** Tesis Doctoral. Universitat Autònoma de Barcelona.
- [8] Referencias W. Smith and P. Wong, ***“Resource Selection Using Execution and Queue Wait Time.”*** NAS Technical report Number: NAS-02-003, 2002.
- [9] P. Dinda, ***“Inline Prediction of the Running Time of Tasks.”*** In Proceedings of the The 10th IEEE International Symposium on High Performance Distributed Computing, 2001.
- [10] W. Smith, Ian Foster and V. Taylor, ***“Predicting application Run Times Using Historical Information.”*** Argonne National Laboratory and northwestern University, 1998.
- [11] Richard Gibbons, ***“A Historical Application Profiler for Use by Parallel Schedulers.”*** University of British Columbia, 1997.
- [12] A. Downey, ***“Predicting Queue Times on Space-Sharing Parallel Computers.”*** In Proceedings of the 11th international Parallel Processing Symposium, 1997.
- [13] C. McCann y J. Zahorjan, ***“Scheduling Memory Constrained Jobs on Distributed Memory Parallel Computers.”*** In Proceedings of the 1995 ACM

SIGMETRICS joint international conference on Measurement and modeling of computer systems, 1995.

- [14] M. Hanzich, F. Giné, P. Hernández, F. Solsona y E. Luque, ***“Effects and Limits of the Multiprogramming Level on a non-dedicated Cluster.”*** In Proceedings of VecPar 2004, Volume 3, 2004.
- [15] K. Aida, ***“Effect of Job Size Characteristics on Job Scheduling Performance.”*** In 6th Workshop on Job Scheduling Strategies for Parallel Processing, 2000.
- [16] Y. Zhang, H. Franke, J. E. Moreira y A. Sivasubramaniam, ***“A Comparative Analysis of Space- and Time-Sharing Techniques for Parallel Job Scheduling in Large Scale Parallel Systems.”*** In IEEE Transactions on Parallel and Distributed Systems, 2002.

## Apéndice A

# Preparación de la BD Histórico

En este apéndice se detallan los pasos a seguir para la creación de la base de datos *histórico* en MySQL. También se describe cómo inicializarla con las trazas disponibles.

El primer paso que hay que dar es instalar el servidor de MySQL. Dependiendo de sistema operativo que se utilice esta operación puede variar. En este caso, al tratarse del sistema operativo Mandriva Linux, la instalación se realiza mediante el administrador de paquetes URPMI o su interfaz gráfica RPMDRAKE. La versión instalada ha sido MySQL-Max-4.1.11, pero se pueden instalar las nuevas versiones sin problema. Si se dispone de conexión a Internet y los repositorios están actualizados, simplemente con ejecutar como *root* el comando `urpmi mysql-max` el sistema nos ofrece la posibilidad de elegir los paquetes necesarios.

Una vez que el servidor de MySQL está instalado, se puede crear la base de datos. El código que crea la base de datos está contenido en el fichero `crear_db.sql`. Este fichero crea las tablas y relaciones descritas en esta memoria. Además inserta en la tabla *nodos* sus nombres, y en la tabla *superficie* los *jiffies* de cada aplicación según el tipo de política.

```
[man@local ~]# mysql < crear_db.sql
```

Para que la herramienta de predicción acceda a la base de datos es necesario crear un usuario para la conexión. A este usuario se le ha llamado "loras". Los comandos necesarios para crear este usuario están guardados en el fichero `crear_usuario.loras.sql`. El contenido de este fichero se muestra a continuación:

```
GRANT USAGE ON *.* TO loras@localhost IDENTIFIED BY 'loras';
GRANT ALL PRIVILEGES ON historico.* TO loras@localhost;
```

El siguiente paso a realizar es cargar la base de datos con la información de los ficheros de trazas. Para ello se ha creado una utilidad en lenguaje C que escanea los ficheros de trazas para introducir los datos en el histórico. El fichero con el código fuente se llama `rellenar.c` y para compilarlo hay que ejecutar el siguiente comando:

```
gcc -o escaner rellenar.c -I/usr/include/mysql -L/usr/lib64 -lmysqlclient -lz
```

Una vez compilada se puede ejecutar la herramienta. Al compilarla se le ha llamado *escaner*. La sintaxis es la siguiente:

```
./scaner <ruta al fichero> <tipo> <política> <núm. secuencia trabajo>
```

**ruta al fichero:** especifica el directorio donde se encuentra el fichero.

**tipo:** especifica si el fichero contiene trazas de trabajos de tipo MPI o PVM.

**política:** especifica la política del fichero de trazas: NORMAL, JSTONE, 3DBF.UNIFORM, 3DBF.UNIPLS.

**núm. secuencia trabajo** especifica el número de trabajo que se asignará al primer trabajo del fichero. Todos los ficheros de trazas comienzan con el trabajo número 1 pero son trabajos diferentes por lo tanto tendrán números identificativos diferentes en la base de datos.

Con el fin de automatizar el escaneo de los ficheros de trazas, se ha creado un *script* para cargar la información de todos los ficheros. El *script* se llama *cargardb* y escanea uno por uno todos los ficheros de trazas que se le indiquen. Para el caso particular en el que las trazas se encuentre ubicadas en un árbol de directorios como el de la Figura A.1, el contenido del *script* para escanear todos los ficheros de trazas sería el mostrado en la Figura A.2.

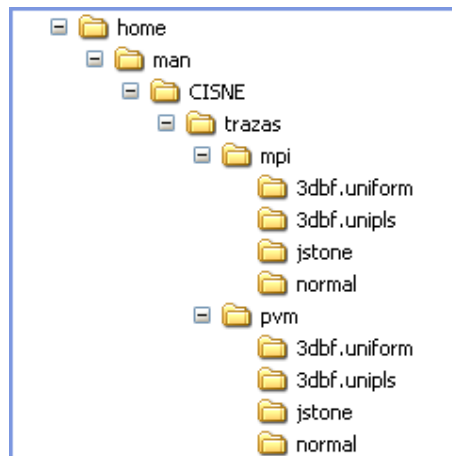


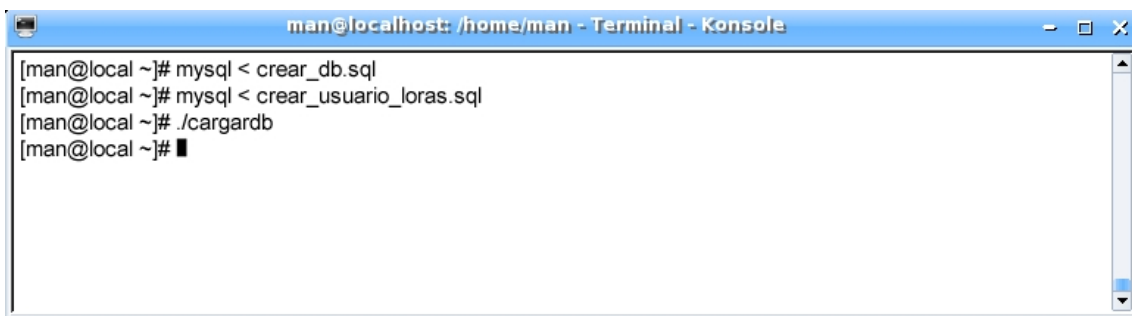
Figura A.1: Ubicación de las trazas.

```
./scanner /home/man/CISNE/trazas/mpi/3dbf.uniform/output1 mpi 3dbf.uniform 0
./scanner /home/man/CISNE/trazas/mpi/3dbf.uniform/output2 mpi 3dbf.uniform 120
./scanner /home/man/CISNE/trazas/mpi/3dbf.unipls/output1 mpi 3dbf.unipls 240
./scanner /home/man/CISNE/trazas/mpi/3dbf.unipls/output2 mpi 3dbf.unipls 360
./scanner /home/man/CISNE/trazas/mpi/jstone/output1 mpi jstone 480
./scanner /home/man/CISNE/trazas/mpi/normal/output1 mpi normal 600
./scanner /home/man/CISNE/trazas/mpi/normal/output2 mpi normal 720
./scanner /home/man/CISNE/trazas/pvm/3dbf.uniform/output1 pvm 3dbf.uniform 840
./scanner /home/man/CISNE/trazas/pvm/3dbf.uniform/output2 pvm 3dbf.uniform 930
./scanner /home/man/CISNE/trazas/pvm/3dbf.unipls/output1 pvm 3dbf.unipls 1020
./scanner /home/man/CISNE/trazas/pvm/3dbf.unipls/output2 pvm 3dbf.unipls 1110
./scanner /home/man/CISNE/trazas/pvm/jstone/output1 pvm jstone 1200
./scanner /home/man/CISNE/trazas/pvm/jstone/output2 pvm jstone 1290
./scanner /home/man/CISNE/trazas/pvm/normal/output1 pvm normal 1380
./scanner /home/man/CISNE/trazas/pvm/normal/output2 pvm normal 1470
```

Figura A.2: Contenido del script *cargadb*.

Con esta carga, el histórico queda inicializado y preparado para servir consultas relacionadas con los trabajos almacenados.

En resumen, la base de datos se crea insertando estas tres instrucciones en la consola del sistema:



```
man@localhost: /home/man - Terminal - Konsole
[man@local ~]# mysql < crear_db.sql
[man@local ~]# mysql < crear_usuario_loras.sql
[man@local ~]# ./cargadb
[man@local ~]# █
```



## Apéndice B

# Utilización de la Herramienta de Predicción

El código fuente de la herramienta de predicción se encuentra en un fichero llamado *predictor.c*. La forma de compilar el fichero se muestra a continuación:

```
gcc -o test predictor.c -I/usr/include/mysql -L/usr/lib/mysql -lmysqlclient -lz
```

Al programa ejecutable se le ha llamado *test*. La sintaxis del ejecutable es la siguiente:

```
./test <fichero trazas test> <tipo trabajo> <tipo politica> <# trabajo test>
```

Por ejemplo si se quiere predecir el tiempo de ejecución del trabajo número 7 del fichero de trazas *output* que contiene las trazas de ejecuciones de trabajos de tipo *mpi* bajo política *3dbf.uniform*, se debería escribir:

```
./test output mpi 3dbf.uniform 7
```

La herramienta nos muestra como resultados: la aplicación que forma parte del trabajo, el tiempo real que tardó en ejecutarse, el tiempo predicho, el porcentaje de error cometido y el porcentaje de error estimado por la propia herramienta de predicción. Para el ejemplo anterior, el resultado sería el siguiente:

```
JID:7 APLI:is.B.4 tmp en ejec:211 tmp predicho:190 Error:9 Error predicho:7%
```

Con el objetivo de realizar este test para todas las trazas disponibles se han creado una serie de *scripts*. Con la ayuda de estos scripts y las herramientas que se describen a continuación, se han podido calcular los resultados cada vez que se ha modificado la herramienta de predicción. En la tabla de la Figura B.1 se muestran los nombres y una breve descripción de cada uno de ellos.

Nombre del Script		Descripción
<code>ejecuta1</code>	Para el fichero de trazas de trabajos: mpi	política: 3dbf.uniform
<code>ejecuta2</code>	Para el fichero de trazas de trabajos: mpi	política: 3dbf.unipls
<code>ejecuta3</code>	Para el fichero de trazas de trabajos: mpi	política: jstone
<code>ejecuta4</code>	Para el fichero test de trabajos: mpi	política: normal
<code>ejecuta5</code>	Para el fichero de trazas de trabajos: pvm	política: 3dbf.uniform
<code>ejecuta6</code>	Para el fichero de trazas de trabajos: pvm	política: 3dbf.unipls
<code>ejecuta7</code>	Para el fichero de trazas de trabajos: pvm	política: jstone
<code>ejecuta8</code>	Para el fichero de trazas de trabajos: pvm	política: normal
<code>test_mpi</code>	Llama a los scripts: <code>ejecuta1</code> , <code>ejecuta2</code> , <code>ejecuta3</code> y <code>ejecuta4</code> . El resultado de esas ejecuciones lo guarda en los ficheros: <code>resultados_mpi_3dbf.uniform.txt</code> <code>resultados_mpi_3dbf.unipls.txt</code> <code>resultados_mpi_jstone.txt</code> <code>resultados_mpi_normal.txt</code>	
<code>test_pvm</code>	Llama a los scripts: <code>ejecuta5</code> , <code>ejecuta6</code> , <code>ejecuta7</code> y <code>ejecuta8</code> . El resultado de esas ejecuciones lo guarda en los ficheros: <code>resultados_pvm_3dbf.uniform.txt</code> <code>resultados_pvm_3dbf.unipls.txt</code> <code>resultados_pvm_jstone.txt</code> <code>resultados_pvm_normal.txt</code>	
<code>test_total</code>	Ejecuta los scripts: <code>test_mpi</code> y <code>test_pvm</code>	

Figura B.1: Listado de scripts

resultados\_mpi\_3dbf.uniform  
Documento de texto  
10 KB

resultados\_mpi\_3dbf.unipls  
Documento de texto  
10 KB

resultados\_mpi\_jstone  
Documento de texto  
8 KB

resultados\_mpi\_normal  
Documento de texto  
8 KB

resultados\_pvm\_3dbf.uniform  
Documento de texto  
10 KB

resultados\_pvm\_3dbf.unipls  
Documento de texto  
10 KB

resultados\_pvm\_jstone  
Documento de texto  
8 KB

resultados\_pvm\_normal  
Documento de texto  
10 KB

```

JID: 1  APLI: pvmisx.gen    tmp en ejec: 190    tmp predicho: 190    Error: 0
JID: 2  APLI: pvmisx.gen    tmp en ejec: 60    tmp predicho: 60    Error: 0
JID: 3  APLI: pvmisx.gen    tmp en ejec: 263   tmp predicho: 230   Error: 12
JID: 4  APLI: pvmngx.gen    tmp en ejec: 280   tmp predicho: 271   Error: 3
JID: 5  APLI: pvmisx.gen    tmp en ejec: 70    tmp predicho: 80    Error: 14
JID: 6  APLI: pvmisx.gen    tmp en ejec: 190   tmp predicho: 191   Error: 0
JID: 7  APLI: pvmngx.gen    tmp en ejec: 185   tmp predicho: 180   Error: 2

```

Figura B.2: Fragmento de uno de los ficheros de resultados.

En la Figura B.1, se muestran todos los scripts disponibles, pero basta con llamar al script `test_total` para predecir los tiempos de ejecución de todas las trazas disponibles.

Este *script* se encarga de ejecutar todos los demás. El resultado son 8 ficheros que contienen la predicción completa de cada uno de los ficheros de test. En la Figura B.2 se puede observar un fragmento de uno de estos ficheros de resultados.

Una vez se tienen estos 8 ficheros de resultados se puede calcular el error medio de predicción cometido para cada uno. Para ello existe una utilidad llamada *calcular\_error*. Este programa lee un fichero resultado y muestra por pantalla el error medio.

## Apéndice C

# Sintonización

En este trabajo se ha explicado que cada una de las tres características utilizadas para calcular la similitud entre trabajos, está multiplicada por un factor a los que se les ha llamado *pesos*. Para encontrar los pesos que minimizan el error de predicción se ha utilizado, también, una herramienta creada en C. El código fuente de esta herramienta se encuentra en el fichero *pesos.c*.

Si ejecutamos este programa pasándole como argumentos el tipo de trabajo y la política de un fichero de trazas de test, este programa se encarga de probar distintas combinaciones de pesos y calcular el error para cada una de ellas. De esta manera se pueden obtener los pesos que minimizan el error para cada una de las políticas de las que disponemos de trazas.

En la figura C.1 se puede ver cómo ejecutaríamos el programa para obtener los pesos que minimizan el error para cada una de las políticas. Un buena práctica es guardar estos resultados en ficheros de texto. En la Figura C.2 se puede ver el contenido del fichero *mpi\_uniform.txt* que contiene la sintonización para los trabajos de tipo *mpi* y política *3dbf.uniform*.

```
./pesos mpi 3dbf.uniform output1 > mpi_uniform.txt
./pesos mpi 3dbf.unipls output2 > mpi_unipls.txt
./pesos mpi jstone output3 > mpi_jstone.txt
./pesos mpi normal output4 > mpi_normal.txt
./pesos pvm 3dbf.uniform output5 > pvm_uniform.txt
./pesos pvm 3dbf.unipls output6 > pvm_unipls.txt
./pesos pvm jstone output7 > pvm_jstone.txt
./pesos pvm normal output8 > pvm_normal.txt
```

Figura C.1: Ejecución para buscar los pesos

```

-> w1:0   w2:0   w3:100  error:41.884209  i:1  .
-> w1:0   w2:5   w3:95   error:40.315788  i:2  ..
-> w1:0   w2:15  w3:85   error:38.431580  i:4  .
-> w1:0   w2:20  w3:80   error:37.926315  i:5  .....
-> w1:0   w2:70  w3:30   error:37.915791  i:15 ...
-> w1:0   w2:85  w3:15   error:37.778946  i:18 ....
-> w1:15  w2:20  w3:65   error:35.610527  i:65  .
-> w1:15  w2:25  w3:60   error:33.473682  i:66  .
-> w1:15  w2:30  w3:55   error:32.789474  i:67 ....
-> w1:20  w2:20  w3:60   error:32.578949  i:83 ....
-> w1:30  w2:25  w3:45   error:32.336842  i:117 ...
-> w1:35  w2:25  w3:40   error:31.757895  i:132 ..
-> w1:35  w2:35  w3:30   error:31.305264  i:134 .
-> w1:35  w2:40  w3:25   error:31.157894  i:135 ...
-> w1:40  w2:35  w3:25   error:31.115789  i:148 .
-> w1:40  w2:40  w3:20   error:29.863157  i:149 ...
-> w1:55  w2:25  w3:20   error:29.810526  i:182 ...
-> w1:60  w2:25  w3:15   error:29.484211  i:192 ..
-> w1:60  w2:35  w3:5   error:29.336842  i:194 ...
-> w1:65  w2:20  w3:15   error:29.052631  i:200 ...
-> w1:70  w2:25  w3:5   error:28.810526  i:209 ...

```

Figura C.2: Sintonización para los trabajos *mpi* y política *3dbf.uniform*

Firmado,

a..... de ..... de 2008

## **Resumen**

### **Català**

CISNE és un sistema de còmput en paral·lel del *Departament d'Arquitectura de Computadors i Sistemes Operatius (DACSO)*. Per poder implementar polítiques d'ordenació de cues i selecció de treballs, aquest sistema necessita predir el temps d'execució de les aplicacions. Amb aquest treball es pretén proveir al sistema CISNE d'un mètode per predir el temps d'execució basat en un històric on s'emmagatzemaran totes les dades sobre les execucions.

### **Castellano**

CISNE es un sistema de cómputo en paralelo del *Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO)*. Para poder implementar políticas de ordenación de colas y selección de trabajos, este sistema necesita predecir el tiempo de ejecución de las aplicaciones. Con este trabajo se pretende proveer al sistema CISNE de un método para predecir el tiempo de ejecución basado en un histórico donde se almacenarán todos los datos sobre las ejecuciones.

### **English**

CISNE is a parallel computing system of the *Department of Architecture of Computers and Operating Systems (DACSO)*. For being able to implement politics of queues ordering and job selection, this system needs to predict the execution time of the applications. With this work, it is intended to supply to the system CISNE of a method to predict the execution time based on one historical where all the data about executions will be stored.