ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

ESTRATEGIAS DE INTERCAMBIO DE INFORMACIÓN EN ALGORITMOS DE EQUILIBRIO DE CARGA

(Strategies for information exchange in load balancing algorithms)

Para acceder al Titulo de

INGENIERO TÉCNICO DE TELECOMUNICACIÓN

Autor: Leticia García Revilla

Octubre- 2012

INGENIERÍA TÉCNICA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL PROYECTO FIN DE CARRERA

Realizado por: Leticia García Revilla.
Director del PFC: José Luis Bosque Orero, Esteban Stafford.
Título: "Estrategias de Intercambio de Información en Algoritmos de Equilibrio de Carga "
Title: "Strategies for information exchange in load balancing algorithms "

Presentado a examen el día:

para acceder al Título de

INGENIERO TÉCNICO DE TELECOMUNICACIÓN, ESPECIALIDAD EN SISTEMAS ELECTRÓNICOS

Composición del Tribunal:
Presidente (Apellidos, Nombre): Menéndez de Llano, Rafael.
Secretario (Apellidos, Nombre): Stafford Fernández, Esteban.
Vocal (Apellidos, Nombre): García López, Carlos.

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del PFC (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Proyecto Fin de Carrera Nº (a asignar por Secretaría)

Índice General

Capítulo 1

Introducción

1.1. Programación paralela

La computación secuencial es el modelo de computador que más éxito ha tenido y su funcionamiento se basa en la ejecución de las tareas una detrás de la otra. Este modelo se puede considerar una buena forma de organizar el procesamiento, y su generalización y adaptación a los computadores actuales ha dado lugar a numerosos lenguajes de alto nivel.

Otra modelo de computación es la computación paralela, la cual se lleva a cabo en computadores, con varios procesadores, donde cada uno puede ejecutar una tarea distinta de un mismo problema, de forma simultánea. Por ello se requiere rediseñar los algoritmos que se utilizan y replantearse las estructuras de datos más adecuadas para algoritmos que van a permitir simultaneidad. En definitiva, se utiliza otro tipo de programación denominada **programación paralela** [1].

Para este tipo de programación cabe destacar los siguientes aspectos: los lenguajes paralelos y los entornos de programación paralela.

- Un **lenguaje paralelo** incorpora en sí mismo las herramientas necesarias para que el programador pueda desarrollar y aplicar las técnicas propias de la programación paralela. Este lenguaje hace transparente al programador la situación real de los datos y le evita las complicaciones inherentes a ello.

- Un **entorno de programación paralela** ofrece la posibilidad de utilizar varias herramientas que corresponden a distintos aspectos implicados

en la programación paralela. Es decir una visión más próxima al nivel físico, pero únicamente en aquellos aspectos que un computador paralelo debe incorporar para poder ser considerado como tal.

Actualmente se tiende a utilizar modelos de computación paralela donde, de forma natural, se superpongan el modelo arquitectónico, el modelo de coste y el modelo de programación. Pero no existe un modelo arquitectónico único, lo que complica notablemente el problema.

En cambio en los modelos de programación paralela, si existen dos alternativas válidas: el **modelo de paso de mensajes** [1] y el **modelo de memoria compartida** [1]. La principal ventaja de disponer de un modelo de programación concreto es que ambos se pueden implementar indistintamente sobre cualquier tipo de computador paralelo de forma absolutamente transparente al usuario, que puede hacer caso omiso de la arquitectura subyacente y concentrarse en el modelo de programación que más le convenga.

El *modelo de paso de mensajes* es uno de los más usados en la programación paralela. Parte de su éxito se debe a los mínimos requerimientos que impone al hardware para poder implementarse. Además se ajusta bien a la mayoría de los supercomputadores. En este modelo toda la comunicación entre procesadores se realiza mediante el envío y recepción explícito de mensajes. Por esto plantean el inconveniente de que el paralelismo es expresado y controlado por el programador, siendo responsabilidad de éste resolver cuestiones como las dependencias de datos, y evitar interbloqueos y condiciones de carrera.

Habitualmente en las implementaciones para paso de mensajes comprenden una librería de subrutinas que se incluyen en el código fuente y que el programador invoca de acuerdo a las necesidades de su programa.

El estándar básico de este tipo de modelo es la librería MPI ([5], [6]) y se va a hacer uso de ella para la realización de las comunicaciones entre los procesos en este proyecto.

El *modelo de memoria compartida* se basa en que un número determinado de procesos comparten un espacio único de direcciones de memoria. Por lo que todos los procesadores ejecutan el mismo código, pero éste puede particularizarse de tal forma que cada procesador ejecute un conjunto diferente de instrucciones.

Debido a limitaciones físicas de la computación secuencial, como son el límite de la velocidad de la luz y el límite físico que representa la integración de cada vez más componentes en menos espacio, con el paso de los años se apuesta más por el paralelismo, como se puede ver claramente en la evolución de los procesadores multi-núcleo (o multi-core). Además se ha entrado en una fase en el que cualquier incremento de las prestaciones a base de tecnología pura, requiere unos costes de inversión muy grandes, difícilmente soportables por fabricantes y usuarios.

1.2. Clúster

Un **clúster** es un grupo de múltiples elementos o nodos de cómputo interconectados entre sí, en el que cada uno tiene su propia memoria local donde almacena sus datos, no existiendo una memoria global común. Una red de interconexión une los distintos elementos, de tal forma que el conjunto es visto como un único ordenador, como se puede observar en la figura 1.1. Cada uno de estos elementos es denominado nodo. En la actualidad cada nodo está compuesto por uno o varios procesadores multicore. De esta forma cada nodo es a su vez un multicomputador de memoria compartida. Por este motivo, se usan cada vez con más frecuencia, modelos de computación híbridos que mezclan tanto el modelo de memoria compartida (dentro de un mismo nodo) como el paso de mensajes (entre distintos nodos del clúster).

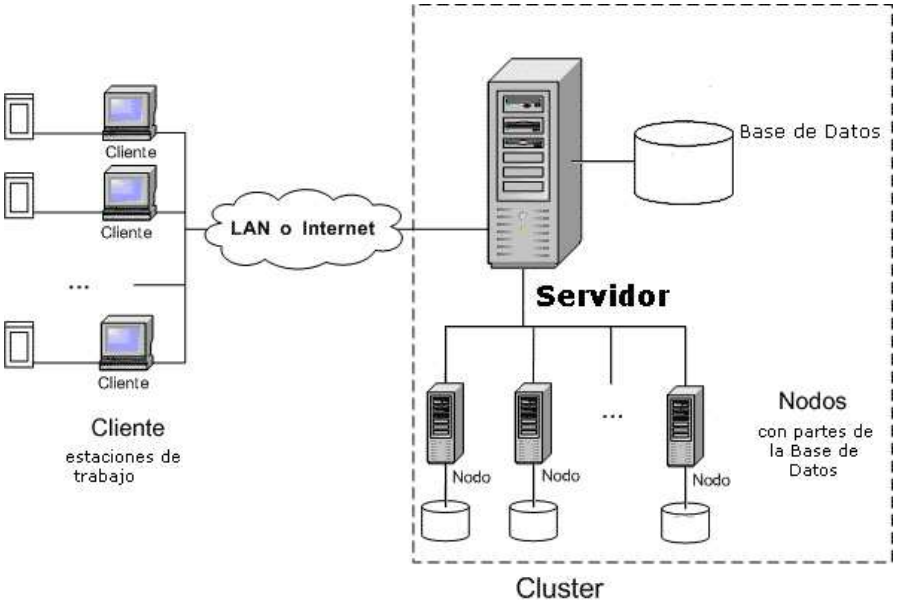


Figura 1.1: Diseño de un clúster.

La construcción de los ordenadores del clúster es más fácil y económica debido a su flexibilidad: pueden todos tener la misma configuración de hardware y sistema operativo (clúster homogéneo), diferente rendimiento pero con arquitecturas y sistemas operativos similares (clúster semihomogéneo), o tener diferente hardware y sistema operativo (clúster heterogéneo), lo que hace más fácil y económica su construcción [2].

Los *clústers* ofrecen las siguientes características a un coste relativamente bajo:

- **Alto rendimiento:** es la capacidad de un equipo para dar altas prestaciones en cuanto a capacidad de cálculo. Los motivos para utilizar un clúster de alto rendimiento son: el tamaño del problema por resolver y el precio de la máquina necesaria para resolverlo.

- **Alta disponibilidad:** es la capacidad de mantener una serie de servicios compartidos y estar constantemente monitorizándose las máquinas entre sí.

- **Alta eficiencia:** es la capacidad de ejecutar la mayor cantidad de tareas en el menor tiempo posible.

- **Escalabilidad:** capacidad de un equipo de hacer frente a volúmenes de trabajo cada vez mayores [3], sin dejar por ello de prestar un nivel de rendimiento aceptable.

El surgimiento de plataformas computacionales de comunicación y procesamiento estándares de bajo coste, les ha brindado la oportunidad a los programadores académicos de crear herramientas computacionales del dominio público o de costo razonable. Estas realidades permiten la implantación de códigos paralelizados sobre este tipo de plataformas, obteniendo un rendimiento competitivo en relación a equipos paralelos especializados, cuyos costes de operación y mantenimiento son elevados.

Esta tecnología ha evolucionado para beneficio de diversas actividades que requieren el poder de cómputo y aplicaciones de misión crítica. Pero existen una serie de problemas que se presentan debido a la red de interconexión y su gestión. Problemas debido a que los mensajes deben cruzar medios físicos (enlaces de la red), debido a la forma en que se gestionan los envíos de los mensajes a través de la red y también dificultades de pérdida o de retraso en la recepción de mensajes debido a un tráfico excesivo en la red.

1.3. Problema equilibrio de carga

La ventaja de los clústers es que se pueden formar con ordenadores con poca potencia pero llegando a buenos resultados, gracias a la división del trabajo entre muchos nodos distintos. Pero también existe la desventaja de que los ordenadores no serán siempre idénticos, si no que tendrán distinto hardware formando sistemas heterogéneos. Debido a esto se produce un problema con el equilibrio de carga de trabajo. Esto quiere decir que al repartir la carga equitativamente entre los distintos nodos, ocurrirá que nodos con menos potencia estarán siendo infrautilizados. En estos casos una distribución

adecuada debe tener en cuenta la capacidad de cómputo de cada nodo, de forma que cada uno reciba una carga de trabajo proporcional a dicha potencia de cómputo.

Para solucionar este problema existen diferentes tipos de algoritmos de carga. El objetivo fundamental de estos algoritmos es que la carga asignada a cada nodo sea tal que todos puedan finalizar su parte de trabajo en el mismo instante de tiempo. Su funcionamiento se basa en el intercambio de mensajes entre los distintos nodos, comprobando cuales están más descargados, para así mandar las tareas a estos nodos.

Los algoritmos de equilibrio de carga, se basan en una serie de reglas o políticas. Una de ellas es la regla de intercambio de información que es la encargada de intercambiar y recoger la información necesaria para tomas decisiones adecuadas sobre cuándo realizar una operación de equilibrio de carga y con qué nodo llevarla a cabo. Existen tres tipos de reglas de información clásicas que se han utilizado en la bibliografía: periódica (realiza la comunicación de información entre los distintos nodos de forma periódica, es decir cada cierto tiempo realiza un envío de esta información), bajo demanda (se produce la comunicación cuando es necesario realizar un equilibrio de carga, siendo el nodo que solicita la información, el que necesita realizar el equilibrio) y por eventos (la comunicación se realiza cuando se produce un evento en un nodo, por lo que enviará la información al resto)[8].

1.4. Objetivos y Plan de trabajo

El objetivo de este proyecto es **estudiar de forma exhaustiva la regla de intercambio de información dentro de un algoritmo de equilibrio de carga de trabajo dinámico, para clúster homogéneos.** Para ello lo primero a realizar, será el estudio de MPI como estándar de programación paralela basada en paso de mensajes, para la realización de la gestión, sincronización y comunicación entre los procesos.

Una vez conocido su funcionamiento, se implementarán los diferentes casos posibles de la regla de información, en las diferentes opciones de diseño. Siendo una de ellas la utilización de estrategias globales, es decir comunicarse entre todos los nodos del sistema, o en el otro caso mediante dominios parciales, siendo en estos la comunicación entre nodos de un mismo dominio. Esta última se plantea como propuesta de solución para los problemas de escalabilidad de estos algoritmos, a medida que crece en número de nodos a controlar.

Por último se realizan una serie de experimentos para validar y evaluar las diferentes propuestas realizadas. Debido a que un algoritmo de equilibrio de carga se basa en diferentes reglas se ha hecho uso de un proyecto de fin de carrera anterior. Por lo que se parte de un algoritmo ya programado pero solo se analiza la regla de información sin realizar ninguna modificación en el resto de reglas. Estos experimentos permiten sacar conclusiones sobre las ventajas y desventajas de cada caso.

1.5. Estructura del documento

Este documento se basa en una serie de capítulos, siendo este el que introduce los contenidos más generales. Los demás serían los siguientes:

- En el **capítulo 2** se explican los fundamentos teóricos que es necesario saber para la comprensión y realización de la regla de intercambio de información.
- En el **capítulo 3** se cuenta como se ha realizado el diseño del algoritmo, entrando en más detalle en la regla de información. Explicando las selecciones que se han realizado en cada caso y el funcionamiento del algoritmo.
- En el **capítulo 4** se explica la forma en que se ha implementado el algoritmo de equilibrio de carga. En este caso se hará mayor hincapié en la regla de información y en sus respectivas reglas.

- En el **capítulo 5** se mostrarán los experimentos que se han realizado, el entorno de trabajo en el que se ha trabajado y las conclusiones correspondientes a los resultados obtenidos. Con todo esto se comprobará si se han cumplido las expectativas de este proyecto.
- En el **capítulo 6** se evaluarán los resultados obtenidos en el apartado anterior, obteniendo unas conclusiones generales. Así como se comentarán las posibles líneas de estudio para próximos trabajos.
- El **capítulo 7** muestra la bibliografía utilizada para realizar el proyecto.

Capítulo 2

Fundamentos teóricos

2.1. Introducción

Para poder realizar el estudio de la regla de intercambio de información de un algoritmo de equilibrio de carga, es necesario conocer previamente el funcionamiento de la comunicación entre procesos (en este caso mediante paso de mensajes, utilizando MPI) y las diferentes reglas que constituyen el conjunto del algoritmo.

2.2. MPI

2.2.1. Introducción

Message Passing Interface (MPI) es un interfaz diseñado para la realización de aplicaciones paralelas basadas en paso de mensajes [4]. También ha sido creado para desarrollar aplicaciones Single Program Multiple Data (SPMD), en el que todos los procesos ejecutan el mismo programa, aunque no necesariamente la misma instrucción al mismo tiempo, sobre un conjunto distinto de datos.

Fue desarrollado por MPI Forum [6], que agrupa unas 40 organizaciones participantes, entre ellos investigadores de universidades, laboratorios y empresas.

Su diseño está inspirado en máquinas con una arquitectura de memoria distribuida, en donde cada procesador tiene acceso a cierta memoria y la única

forma de intercambiar información es a través de mensajes. Sin embargo, hoy en día también encontramos implementaciones de MPI en máquinas de memoria compartida [7].

Sus principales características son:

- La portabilidad de las aplicaciones paralelas.
- La estandarización favorecida por implementación de calidad, de dominio público.
- Las buenas prestaciones.
- La existencia de implementaciones libres (mpich, LAM-MPI,…).
- La amplia funcionalidad.
- Incluye interfaces para Fortran, C y C++.

2.2.2. Funciones básicas MPI

La forma de comunicación en MPI es a través de mensajes que contienen datos. Principalmente existen dos tipos de funciones de comunicación: *punto a punto*, en el cual se comunican entre dos procesos (uno envía y otro recibe), y *colectivas*, entre varios procesos [4]. En este último caso la misma función tiene que ser ejecutada por todo los procesos, aunque cada uno actúe de manera diferente.

Para la comunicación, MPI agrupa los procesos que están implicados en la ejecución paralela, en grupos denominados comunicadores. Estos son una colección de procesos entre los cuales se pueden intercambiar mensajes. Siempre existe, por defecto, el comunicador MPI_COMM_WORLD que engloba todos los procesos y dentro de este cada proceso tiene un identificador denominado rango en MPI, que lo identifica. A partir de un comunicador se pueden crear otros dependiendo del interés que el programador tenga en formar grupos más pequeños, y si un proceso pertenece simultáneamente a varios grupos, tendrá un rango distinto en cada uno de ellos.

Las funciones principales de inicialización que se utilizan en prácticamente todos los programas MPI son:

```
int MPI_Init ( int *argc, char **argv[] );
int MPI_Comm_size (MPI_Comm comm, int *size);
int MPI_Comm_rank (MPI_Comm comm, int *rank);
int MPI_Finalize (void);
```

Siendo MPI_Init() para iniciar la aplicación paralela, MPI_Comm_size() para averiguar el número de procesos de un comunicador, MPI_Comm_rank() para que cada proceso averigüe su identificador dentro del comunicador, y MPI_Finalize() para dar por finalizada la parte paralela de la aplicación.

2.2.3. Tipos de datos

MPI define una colección de tipos de datos primitivos, como se muestra en la figura 1, correspondientes a los tipos de datos existentes en C o Fortran [4], ya que es el tipo de lenguajes que se va a utilizar.

Estos tipos se utilizan cuando se realiza un paso de mensajes, ya que hay que facilitar a MPI un descriptor equivalente de los tipos de datos habituales. MPI permite intercambiar datos entre arquitecturas heterogéneas, los cuales se almacenan internamente de forma distinta.

Tipos MPI	Tipos C equivalentes
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Sin equivalencia
MPI_PACK	Tipo empaquetado

Tabla2.1. Tipos de datos MPI.

2.2.4. Comunicaciones punto a punto

Estas funciones involucran a dos procesos los cuales deben pertenecer al mismo comunicador. Dentro de este tipo puede haber comunicación bloqueante o no bloqueante, y 4 modos distintos de envío con buffer, síncrono, básico y ready.

La comunicación bloqueante mantiene un proceso bloqueado hasta que la operación solicitada finalice. Por otro lado una operación no bloqueante manda al sistema la realización de una operación recuperando el control inmediatamente y más tarde comprueba si la operación ha finalizado o no.

- **Envío con buffer:** Cuando se realiza un envío de este tipo se guarda inmediatamente una copia del mensaje en un buffer proporcionado por el emisor. La operación se da por completa en cuanto se ha efectuado esta copia. Si no hay espacio en el buffer, el envío fracasa.

- **Envío síncrono:** En este caso la operación se da por terminada sólo cuando el mensaje ha sido recibido en destino.

- **Envío ready:** Solo se realiza el envío si el otro extremo está preparado para la recepción inmediata.

- **Envío básico:** En este caso no especifica la forma en la que se completa la operación, es algo dependiente de la implementación. Normalmente equivale a un envío con buffer para mensajes cortos y a un envío síncrono para mensajes largos. Para este proyecto se va a utilizar este tipo de envío, ya que todos los mensajes que será necesario enviar son cortos.

    - Bloqueantes:

      int MPI_Send(void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm);*
      int MPI_Recv(void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM comm, MPI_Status *status);*

    - No bloqueantes:

      int MPI_Isend(void buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_COMM comm, MPI_Request *request);*
      int MPI_Irecv(void buf, int count, MPI_Datatype datatype, int source, int tag, MPI_COMM comm, MPI_Request *request);*

Existen unas funciones de recepción que permiten saber si existe un mensaje recibido pero sin leerlo. Con ellas se puede averiguar la identidad del emisor del mensaje, la etiqueta del mismo y su tamaño.

*MPI_Probe (int source, int tag, MPI_Comm comm, MPI_Status *status);*

*MPI_Iprobe (int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status);*

2.2.5. Comunicaciones colectivas

Este tipo de operaciones permite la transferencia de datos entre todos los procesos que pertenecen a un grupo específico. La operación colectiva tiene que ser invocada por todos los participantes, aunque los roles que jueguen no sean los mismos, el buffer tiene que ser del tamaño exacto y son de tipo bloqueantes. Algunos tipos son:

- **Barreras de sincronización:** Es una operación que bloquea a los procesos de un comunicador hasta que todos ellos han pasado por la barrera. Pero no implica que exista comunicación de datos, sino solamente sincronización.

  int MPI_Barrier (MPI_Comm comm);

- **Broadcast:** Este tipo sirve para que el proceso, raíz, envíe un mensaje a todos los miembros del comunicador. Siendo el proceso raíz, indicado en el parámetro 'root'.

  int MPI_Bcast (void buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);*

- **Gather:** Realiza una recolección de datos en el proceso raíz. Este proceso recopila un vector de datos, al que contribuyen todos los procesos del comunicador con la misma cantidad de datos. El proceso raíz almacena las contribuciones de forma consecutiva, en función del rango de los procesadores (no del orden de llegada).

  MPI_Gather (void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

  Existe también una versión de esta función "MPI_Gatherv()" que permite almacenar los datos recogidos de forma no consecutiva y que cada proceso aporte bloques de datos de diferentes tamaños.

Las funciónes MPI_Allgather() y MPI_Allgatherv() permiten distribuir a todos los procesos el resultado de una recolección previa siendo los tamaños de los datos fijos o variables y almacenándose de forma consecutiva o no.

- **Scatter:** Envía a cada proceso del comunicador una parte del vector total que tiene el proceso raíz.

  MPI_Scatter(void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

  Si los datos a enviar no se encuentran de forma consecutiva en memoria, o los bloques a enviar a cada proceso no son todos del mismo tamaño se utiliza la función MPI_Scatterv();

- **Reduce:** Consiste en una operación realizada de forma cooperativa entre todos los procesos de un comunicador, de tal forma que se obtiene un resultado final que se almacena en el proceso raíz.

  *MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);*

  Las operaciones que define MPI del tipo MPI_Op son máximo, mínimo, suma, producto, and lógico, etc. También se puede crear un tipo de operación mediante MPI_Op_create() y eliminar el mismo mediante MPI_Op_free().

  Para realizar un reduce y que deje el resultado en todos los procesos del comunicador existe la función *MPI_Allreduce (void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);*

2.2.6. Otras funciones de MPI

Existen diferentes funciones para crear comunicadores, *MPI_Comm_split (MPI_Comm comm, int color, int key, MPI_Comm *newcomm)* es una de ellas, la cual se va a utilizar en este proyecto. Consiste en crear, a partir de un comunicador inicial, varios comunicadores diferentes, cada uno con un conjunto disjunto de procesos. El "color" determina en qué comunicador queda cada uno de los procesos. Para borrar los comunicadores creados se utiliza la función MPI_Comm_free().

Si se quiere enviar un conjunto de datos juntos existen las funciones MPI_Pack que empaqueta y MPI_Unpack que desempaqueta. Respectivamente estas funciones permiten copiar en un buffer continuo de memoria, datos almacenados en el buffer de entrada y copiar en un buffer, los datos almacenados en el buffer continuo de entrada.

> *int MPI_Pack (void* inbuf, int incount, MPI_Datatype datatype, void* outbuf, int outsize, int *position, MPI_Comm comm);*
>
> *int MPI_Unpack (void* inbuf, int insize, int *position, void* outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm);*

2.3. Equilibrio de carga

2.3.1. Introducción

El estudio del equilibrio de carga es uno de los problemas clave que afectan al rendimiento de aplicaciones. Su objetivo es distribuir de una forma equitativa la carga computacional entre todos los procesadores disponibles y con ello conseguir la máxima velocidad de ejecución.

Lo ideal del equilibrio de carga en un sistema heterogéneo es que cada procesador realice la misma cantidad de trabajo, donde además se espera que los procesadores trabajen al mismo tiempo.

Sin embargo una mala distribución puede llevar a una situación en la que existan nodos inactivos mientras otros están sobrecargados. Por lo tanto se

produce un desequilibrio de la carga, bien sea porque las tareas no están bien distribuidas o porque algunos procesadores sean más eficientes que otros. Por ello ha sido un tema muy estudiado y se han propuesto múltiples soluciones [8] De las cuales algunas se explican a continuación.

Algoritmos estáticos

Esta solución puede ser buena cuando la carga se conoce antes de la ejecución, ya que el reparto de las tareas se realiza basándose en las estimaciones de los requerimientos de recursos y al comienzo de la computación, es decir, en el tiempo de compilación, eliminando la sobrecarga en el tiempo de ejecución. Pero existen algunos inconvenientes serios que lo sitúan en desventaja respecto a los algoritmos dinámicos, algunos de estos serían:

- La dificultad de saber de forma estimada el tiempo de ejecución de todas las partes en las que se divide el programa.
- La necesidad, a veces, de un número indeterminado de pasos computacionales para alcanzar la solución de un problema.
- La posibilidad de tener retardos en las comunicaciones en algunos sistemas, que pueden variar bajo diferentes circunstancias.

Algoritmos dinámicos

Con este algoritmo es posible tener en cuenta todos los inconvenientes del algoritmo estático, debido a la división de la carga computacional que depende de las tareas que se están ejecutando y no de la estimación del tiempo que pueden tardar en ejecutarse. Aunque el algoritmo dinámico lleva consigo una cierta sobrecarga durante la ejecución del programa, resulta una alternativa mucho más eficiente que el algoritmo estático.

Por lo tanto, la principal ventaja de los algoritmos dinámicos respecto a los estáticos es la inherente flexibilidad de los primeros para adaptarse a los requisitos del sistema en ejecución, a costa de la sobrecarga de comunicaciones

entre los nodos. Consecuentemente, las desventajas vienen producidas principalmente por los retardos en las comunicaciones, las transferencias de la información sobre la carga de los nodos y las sobrecargas en cada nodo producidas por el tiempo de procesamiento correspondiente a la toma de decisiones.

En el algoritmo dinámico, las tareas se reparten entre los procesadores durante la ejecución del programa. Dependiendo de dónde y cómo se almacenen y repartan las tareas, estos algoritmos se dividen en:

- **Algoritmos centralizados:** los cuales se basan en un nodo raíz o central al que cada nodo le tiene que comunicar los cambios que se produzcan en él, considerando dicho nodo raíz realizar el equilibrio de carga o no.
- **Algoritmos distribuidos o descentralizados:** en este caso el equilibrio de carga puede ser iniciado por el nodo que necesite realizarle.

Una gran desventaja del algoritmo centralizado es que el nodo raíz únicamente puede repartir una tarea cada vez, y una vez enviadas las tareas iniciales sólo podrá responder a nuevas peticiones de una en una. Por tanto, se pueden producir saturaciones si varios nodos solicitan peticiones de tareas de manera simultánea. La estructura centralizada no posee una gran tolerancia a fallos y únicamente será recomendable para clústers pequeños.

Si se tiene en cuenta la escalabilidad puede haber dos soluciones distintas basadas en lo anterior. Siendo la primera una solución global, es decir que las comunicaciones se producen entre todos los nodos del clúster. Y la segunda una solución local, en la que se forman unos grupos (dominios) y solo se producirá comunicación entre nodos del mismo dominio.

2.3.2. Reglas para el equilibrio de carga

Como se ha explicado anteriormente existen distintos tipos de algoritmos, por lo que se va a escoger trabajar con una estructura distribuida que es una de las más utilizadas. Este tipo de algoritmo se basa en cuatro componentes: la regla de medida de estado, la regla de intercambio de información, la regla de iniciación y la operación de equilibrio de carga; las cuales se detallan a continuación [8].

2.3.2.1. Regla de medida de estado

Para la medida del estado de cada nodo es necesario hacer uso de un parámetro, el índice de carga. Este se encarga de medir el estado de cada nodo para saber si este puede recibir nuevas tareas o no y en función de esto tomar una decisión, que se discretiza para determinados comportamientos. En este caso, se van a definir tres posibles estados: receptor, neutro y emisor.

Se denomina que un nodo está en estado *receptor*, cuando este puede recibir tareas locales o remotas debidas al equilibrio de carga. Esto quiere decir que la máquina esta casi o completamente libre con lo que ejecutara el tipo de tareas antes nombradas.

Como es difícil saber cuándo se satura el nodo y cuando se debe ayudar al clúster, se crea un estado intermedio para evitar sobrepasar el punto de saturación, cuando llegan un número excesivo de tareas. Este estado se denomina *neutro* y solo puede recibir tareas locales, rechaza cualquier tarea remota. Con esto se da tiempo a que vayan terminando las tareas pero no garantiza que no se produzca la saturación.

El último estado es el *emisor* que es el caso en el que tiene carga extrema. Debido a que la carga sea demasiada para la potencia del nodo y sobrepase el estado neutro, por lo que necesitará ayuda de nodos que estén en estado receptor.

El índice de carga debe reflejar la carga local y facilitar una aproximación al posible tiempo de respuesta de las nuevas tareas. Esto se puede hacer sin

necesidad de ejecutar las tareas ya que el tiempo de respuesta depende de la CPU, la memoria, el disco y de sus requisitos de comunicación con otras tareas. Por lo que un buen índice debe tener las siguientes características:

- **Dinámico**: refleja los cambios de estado que se producen en un nodo y la posibilidad de calcularlos en tiempo real.
- **Reflejo del estado actual**: ser lo más fiel posible a la situación actual del nodo.
- **Sencillo y rápido**: la información debe ser lo más precisa y actual posible por lo que es importante obtener el índice de forma rápida y sin muchas complicaciones.
- **Estable**: debe ser capaz de mantener su valor aunque existan cambios bruscos de estado.
- **Adaptable**: funcionar en un gran número de clústers distintos.

Un índice de carga debe generar la menor sobrecarga posible para no ralentizar el equilibrio de carga por lo que debe ser sencillo, fiel y rápido.

2.3.2.2. Regla de Intercambio de información

Esta regla define los mecanismos de la recogida e intercambio de información entre los distintos nodos del clúster. Idealmente todos los nodos deberían tener información actualizada del resto de los nodos pero esto implicaría una gran cantidad de mensajes a través de la red de interconexión del clúster. Por lo que en la práctica, se intenta que los nodos estén lo más actualizados posibles pero sin llegar a saturar la red.

Como se ha dicho anteriormente se va a utilizar un algoritmo distribuido, por lo que la regla de información también lo será. Esto quiere decir que cada nodo tendrá información del estado del resto de los nodos o parte de ellos. Si existiese una gran cantidad de nodos, surgiría un problema de escalabilidad, debido a que volvería a existir un gran tráfico en la red. Por eso existen dos tipos dentro de las soluciones distribuidas, las **globales** y las **locales**. En este

último el conjunto de nodos se divide en una serie de grupos o dominios. Por lo que cada nodo tendrá información solo de los nodos pertenecientes a dicho dominio. Y en el primer caso todos los nodos tienen información sobre el estado de todos los demás. Para la creación de estos dominios existen varias alternativas, en dominios aleatorios, dominios solapados, dominios fijos, según similitud... Además los dominios pueden ser permanentes o variar a lo largo del tiempo.

Con los dominios se crea menos carga en la red pero esto puede implicar que surjan desequilibrios entre estos dominios, debido a que solo se producen equilibrios de carga con los nodos pertenecientes a su dominio.

También existen otras soluciones como por ejemplo las reglas jerárquicas que se organizan en dos niveles. Esto quiere decir que los nodos se comunican con los que pertenecen a su dominio pero a su vez el nodo raíz se comunica con el resto de dominios. Con esto el raíz tiene información de forma local y de los demás dominios.

Desde el punto de vista temporal, existen tres reglas de recogida e intercambio de información que cumplen con el compromiso de no sobrecargar la red:

Bajo demanda

En este caso solo cuando un nodo se plantea la posibilidad de hacer una operación de equilibrio de carga, solicita información al resto de nodos. Con esto la información siempre estaría actualizada, por lo que no se trabajaría con información obsoleta. También minimizaría el número de mensajes pero en el momento del equilibrio de carga se produciría una sobrecarga en la red al tener que sincronizar y recibir de todos los nodos, provocando un retardo en la operación.

Periódica

Esta regla se basa en que cada nodo informa de su estado a los demás *periódicamente* sin importar si va a ser necesario o no.

Debe escogerse correctamente el intervalo de envío para que cada cierto tiempo todos los nodos estén informados y no haya una gran saturación en

la red. Gracias a esto se puede tomar la decisión de realizar la operación de equilibrio de carga de forma local y sin necesidad de sincronizar todas las máquinas.

Con esta regla existe la posibilidad de que se realicen los equilibrios con información obsoleta, siendo muy sensible al intervalo de actualización elegido. Ya que puede haberse realizado un cambio de estado y no haberse comunicado.

Por eventos

Con esta regla los nodos únicamente informan de su estado cuando se produce un evento, en este caso cuando existe un cambio de estado. Con ello se realizan menos comunicaciones entre los nodos con lo que el número de mensajes es menor y no se produciría una saturación en la red. Además el equilibrio de carga se realizaría con información actualizada.

Por último cabe destacar que en algunas ocasiones se ha trabajado con soluciones mixtas, normalmente entre la regla periódica o por eventos y la regla bajo demanda. La primeras se denominan voluntarias donde cada nodo envía su estado a los demás dependiendo si se cumple o no una condición local. Y en la segunda cada nodo envía su estado cuando se lo pide otro.

2.3.2.3. Regla de iniciación

Esta regla determina cuando comenzará la operación de equilibrio de carga. Debido a que una operación de equilibrio de carga introduce mucha sobrecarga, se debe considerar si se compensa ejecutar las tareas remotamente o no. Lo ideal sería conocer cómo va a comportarse la máquina remota pero esto no es posible ya que solo se conoce el estado actual.

Las operaciones de equilibrio de carga, generalmente son iniciadas por un nodo emisor o receptor, dependiendo en qué estado se encuentre dicho nodo. En el caso que se inicie desde un nodo en estado emisor, se necesitará una lista con las máquinas que se encuentren, en ese momento, en estado receptor y que

sirvan para realizar el equilibrio de carga con ellas. Cuando se inicia a través de nodos que están en estado receptor sería al contrarío, ya que en este caso se necesitaría una lista con máquinas emisoras y cuando se produjese un cambio al estado receptor buscaría dentro de esta lista un nodo con el que compartir los procesos.

Existe otro tipo de iniciación que no depende del estado en el que se encuentre un nodo. Este se denomina iniciación simétrica, en este caso la operación de equilibrio de carga se iniciará cuando el índice se encuentre por encima o por debajo de un porcentaje dependiente de la última operación de equilibrio realizada.

2.3.2.4. Operación de equilibrio de carga

Para realizar una operación de equilibrio de carga se necesitan por lo menos dos nodos, uno que envíe la carga y otro que la reciba. Esta operación se basa en tres reglas:

- La regla de localización.
- La regla de distribución.
- La regla de selección.

Existe un conjunto de nodos candidatos para la operación llamado dominio de equilibrio. Con la regla de localización se escoge al nodo, perteneciente a este dominio, con el que se va a realizar el equilibrio. Por otro lado la regla de distribución, como su nombre indica, distribuye la carga entre los nodos participantes en la operación de equilibrio. Y por último la regla de selección escoge los procesos que se van a mandar para ejecutar en el nodo elegido para la realización del equilibrio de carga.

Debido a esta última regla el algoritmo se puede clasificar como con interrupción o sin interrupción. Para el primer caso los procesos se cambian de un nodo a otro, suspendiéndolo, transmitiendo todo su contexto y reiniciando el nuevo nodo que lo aloja. En el segundo caso los procesos se mueven antes de haberse empezado a ejecutar con lo que es mucho más ligera la operación de equilibrio.

Capítulo 3

Diseño del algoritmo de equilibrio de carga

3.1. Introducción

Para el diseño del algoritmo de equilibrio de carga se ha escogido un algoritmo dinámico y distribuido. Debido a que es dinámico, las tareas van a ser repartidas durante la ejecución del programa, no en el tiempo de compilación. Esto permite adaptarse a posibles cambios en la situación de carga global del sistema. Además cada nodo va a poder realizar la operación de equilibrio de carga si le es necesario ya que también es un algoritmo distribuido.

Este algoritmo se basa en una serie de reglas comentadas anteriormente y de las cuales se va a explicar su diseño a continuación.

3.2. Regla de medida de estado

La regla de medida de estado obtiene información de cada nodo para determinar el estado en el que se encuentra y poder tomar decisiones. Como se explico en el apartado 2.2.2.1, se utiliza un índice de carga para obtener dicha información.

Índice de carga

El índice de carga es una variable que refleja la carga de trabajo que tiene un nodo en un instante de tiempo. Su valor varía entre cero y un valor mucho mayor que la unidad. El índice de carga implementado en este algoritmo se basará en dos parámetros estáticos y uno dinámico. Los estáticos son el número de cores y los bogomips, con los cuales se mide la potencia de los procesadores. Y el dinámico es el número de tareas que se estén ejecutando y que variará dependiendo de la carga a la que esté sometido el nodo.

El cálculo del índice se hace teniendo en cuenta dos casos diferentes dependiendo de la cantidad de procesos en cada núcleo del nodo. El primer caso es si el número de tareas es inferior al número de núcleos, por lo que automáticamente el nodo se convertirá en estado receptor, con lo que podrá recibir cualquier tipo de tarea (local o remota). Y el segundo caso es lo contrario, si el número de tareas es superior al número de núcleos. En este caso habrá que mirar la potencia de los núcleos para ver si sigue estando en estado receptor o habría que pasar a otro estado. Esto se comprueba obteniendo la carga local a partir de la fórmula de la figura 3.1.

$$Load_{index} = \frac{Bogomips_{local\_average}}{Bogomips_{cluster\_average}} \cdot \frac{Cores\_Number}{Tasks\_Number}$$

Figura 3.1. Fórmula para la obtención de la carga local.

El índice de carga se actualiza periódicamente, por lo que el periodo para que este índice no trabaje con datos obsoletos en la adquisición de la información y no se genere una sobrecarga, se realiza con un intervalo establecido. Por lo que se busca un equilibrio entre la velocidad de actualización y la carga generada por el proceso.

Como se explicó en los fundamentos teóricos, dependiendo del índice de carga se crean tres estados: emisor, neutral, receptor.

- **Emisor:** el nodo estará en este estado cuando el índice se encuentre por debajo del umbral neutral-emisor. Esto quiere decir que el nodo se estará saturando.

- **Neutral:** si el índice tiene un valor medio, sin llegar a saturarse el nodo se encontrará en estado neutral.

- **Receptor:** cuando el índice se encuentre por encima del umbral neutral-receptor o el número de tareas sea inferior al número de núcleos, el nodo estará en estado receptor.

3.3. Regla de la información

Mediante la regla de intercambio de información se recoge e intercambia la información entre los distintos nodos del clúster. En el caso ideal la regla de información debe proporcionar una visión actualizada y precisa de la situación de todos los nodos del clúster. Esto es muy costoso en términos del número de mensajes que hay que intercambiar, por lo que se busca un equilibrio entre información y sobrecarga en la red. En este proyecto se utilizan tres reglas clásicas y bien conocidas: periódica, por eventos y bajo demanda, para no sobrecargar la red a través de los mensajes producidos para el intercambio de información. Por otro lado, en función del ámbito de comunicación la regla puede ser global, que el intercambio se produzca con cualquier nodo, o local, que solo se realice entre nodos que pertenezcan al mismo dominio.

Como las operaciones de carga van a ser realizadas por un nodo que se encuentre en estado emisor, solo van a ser necesarios, en el momento de la operación, nodos que estén en estado receptor. Por esto se crea una lista con los receptores que estén en ese estado y se actualiza según se realizan los intercambios de información. Esta lista se denomina lista de receptores y es similar par todas las reglas.

Dicha lista se va actualizando, por lo que unas veces se añadirán nodos y otras serán eliminados. El momento y motivo por el que se realiza la actualización varía en función del tipo de regla. Cuando se da el caso de añadir un nodo a la

lista, se colocará al final de ella. Y en el caso de que haya que eliminar un nodo, se buscará dentro de dicha lista, se eliminará y pasará ocupar su lugar el último nodo que hubiese en esa lisa. Esto se ha hecho así para optimizar el proceso, ya que no tiene importancia el orden de los nodos dentro de la lista.

3.3.1. De información global

Como se ha dicho antes en este tipo de comunicación todos los nodos envían su estado al resto por lo que dependiendo del tipo de regla se envían periódicamente, debido a un evento o si se necesitan.

Periódica

Esta regla envía periódicamente a intervalos de tiempo prefijados, el estado en el que se encuentra actualmente el nodo y el estado anterior en el que se encontraba al resto de nodos. Con lo que cada nodo tiene información sobre el resto en el momento de realizar el equilibrio de carga.

En este caso es fundamental la selección del periodo o intervalo de tiempo, por lo que de forma empírica se ha escogido un tiempo de envío que mantenga a todos los nodos informados pero sin que exista una saturación de mensajes por la red. Debido a esto a veces ocurre que al realizarse la operación de equilibrio, los nodos están actualizados pero con información obsoleta ya que cuando se realiza el equilibrio algún nodo puede haber modificado su estado y no haber llegado el tiempo de envío.

Cuando un nodo recibe la información de otro, se comprueba si el nodo del que se recibe la información está en la lista de receptores o no. Esto se hace para no añadir a dicha lista un nodo más de una vez o para no intentar borrar un nodo que no está en ella. Esto puede ocurrir cuando en un nodo se ha producido más de un cambio dentro de un intervalo de tiempo y no se ha comunicado. Por ejemplo un nodo que era receptor y se había añadido a la lista de otro nodo, antes de volver a comunicarle su estado, este nodo deja de ser receptor y a continuación vuelve a ser lo. Por lo que cuando se produce el intercambio, el nodo que ha sufrido dos cambios, en el otro continúa en la lista por lo que si no

se comprueba si pertenecía a la lista se hubiese añadido doblemente. Esto mismo puede ocurrir en el caso contrario en el que un nodo no esté en la lista y se vaya a eliminar algo que no existe. Este ejemplo se puede ver en la figura 3.2.

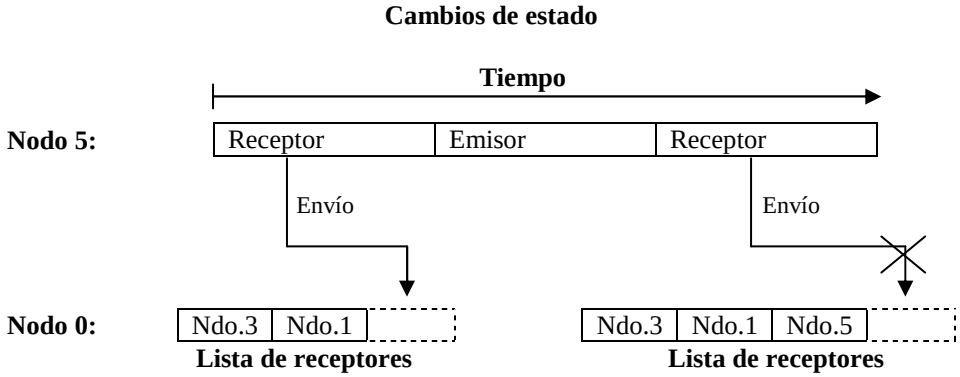


Figura 3.2. Ejemplo de cómo añadir un nodo a lista de receptores en regla periódica.

Como con esta regla se recibe el estado nuevo y el viejo de cada nodo, se tiene que comprobar si se ha modificado su estado convenientemente o continúa siendo el mismo para añadirlo o no a la lista de receptores. Esto se hace comprobando si el nuevo estado o el viejo eran receptores o no, como se expresa en la tabla 3.1. Ya que esta regla envía la información aunque no se haya producido ningún cambio de estado.

Estado actual	**Estado anterior**	**Añadir**	**Borrar**
Receptor	Neutral o emisor	X	
Neutral o emisor	Receptor		X
Para el resto de los casos no se haría nada en la lista de receptores.			

Tabla 3.1. Tabla que muestra cuando se modifica la lista de receptores.

Por eventos

Mediante esta regla se envía información sólo al resto de nodos cuando se produce un evento, definiendo evento como un cambio de estado en un proceso. Esta información es solo el estado actual del nodo que envía, ya que siempre que exista un cambio de estado va a ser comunicado.

Como es un algoritmo iniciado por el emisor solo interesa conocer los nodos receptores por lo que solo se producirá un evento cuando el nodo sea receptor o lo haya dejado de ser, como se puede observar en la tabla 3.2.

Cambio de estado	**Realiza evento**
Receptor a Neutral	Sí, se borra de todas las colas.
Neutral a Emisor	No.
Receptor a Emisor	Sí, se borra de todas las colas.
Emisor a Neutral	No.
Neutral a Receptor	Sí, se añade en todas las colas.
Emisor a Receptor	Sí, se añade en todas las colas.

Tabla 3.2. Realización del evento dependiendo del cambio de estado.

Cuando un nodo recibe un mensaje indicando un cambio de estado actualiza la lista de receptores. Si el cambio de estado que se ha producido es a receptor lo añade; en caso contrario será un nodo que deja de ser receptor y se borra de dicha lista.

Con todo esto los nodos tienen información actualizada para la realización del equilibrio de carga.

Bajo demanda

Con esta regla sólo se producen comunicaciones cuando es necesario realizar un equilibrio de carga. Por lo que cuando esto ocurre, se pide a los demás nodos su estado actual para comprobar si se puede realizar la operación o no con ellos.

La información que se recibe de cada nodo, se comprueba para saber si el estado recibido de un nodo es receptor y si es así añadirle como se ha dicho antes. Si fuese el caso contrario no se realizaría nada simplemente se pasaría a comprobar el siguiente.

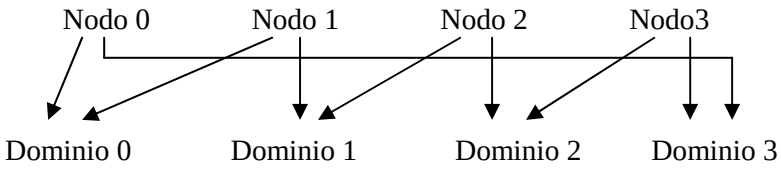
En este caso cada vez que se quiera realizar un equilibrio de carga y se soliciten los estados al resto de nodos se creará una nueva lista de receptores. No pasa como en los casos anteriores que se iba actualizando sino que se sobrescribe eliminando la anterior. Se ha diseñado así ya que es en el momento de la operación de equilibrio cuando se recibe la información, con lo que está totalmente actualizada y es más rápido realizar una nueva lista que modificar la anterior.

3.3.2. De información local

En este caso se divide el conjunto de nodos en una serie de grupos que se denominarán **dominios**. Esta manera de intercambiar la información solo entre nodos de un dominio se ha diseñado mediante dos tipos de dominios distintos: solapados y aleatorios. Esto permite mejorar la escalabilidad de la regla de información. Pues una regla global es inabordable cuando crece mucho el número de nodos del sistema

Dominios solapados

Están compuestos por una serie de nodos, definidos por el tamaño del dominio, que se escogen en función de su identificador o rango (ver figura 3.3). Además existe un solapamiento, esto quiere decir que todos los nodos pertenecen a más de un dominio. Esto se puede ver en la figura 3.3.



Dominio 0	Nodo 0	Nodo 1
Dominio 1	Nodo 1	Nodo 2
Dominio 2	Nodo 2	Nodo 3
Dominio 3	Nodo 3	Nodo 0

Figura 3.3. Ejemplo de reparto de nodos en dominios con tamaño 2 y solape 1.

El solapamiento se ha diseñado para que siempre sea la mitad del tamaño del dominio, con lo que todos los nodos pertenecen a dos dominios. Esto se ha hecho así debido a que surgían muchos problemas al determinar a cuántos y a que dominios pertenecía cada nodo.

Las tres reglas explicadas en el apartado 3.3.1, se han diseñado basándose en la misma estructura, pero con las diferencias propias de cada regla.

Los dominios se han formado creando las listas a las que corresponde cada nodo y comprobando a qué dominio pertenece. Para realizar el envío del estado, se recorren las dos listas y se envía al nodo correspondiente la información, comprobando que no se envía a algún nodo repetido. Cada tipo de regla se comunicará periódicamente, por necesidad o cada vez que se produzca un evento como se ha explicado detalladamente en el apartado anterior.

Dominios aleatorios

En este caso los dominios se forman mediante nodos aleatorios como su mismo nombre indica. Por lo que no es necesario crear ninguna lista como ocurría en los dominios solapados. Esto es debido a que cada vez que sea necesario realizar una comunicación con los nodos, se elegirán en ese momento

aleatoriamente a cuales debe ser comunicado el estado. Se espera que a lo largo del tiempo todos los nodos dispongan de información para realizar sus operaciones.

El número de nodos a los que se les debe enviar la información viene determinado por el tamaño del dominio, como ocurría en los dominios solapados.

3.4. Regla de iniciación

Como se ha dicho anteriormente el equilibrio de carga va a ser iniciado por un nodo que se encuentre en estado emisor. Por lo que si un nodo se encuentra en estado receptor seguirá realizando las tareas que tenga de forma local, pero no podrá iniciar la operación. Cuando algún nodo pase al estado emisor debido a que está desbordado de tareas, iniciará una operación de equilibrio pidiendo ayuda a los nodos que estén en estado receptor. Si no encontrase ningún nodo receptor la tarea será encolada hasta que se encuentre un nodo disponible o el mismo cambie de estado.

En el caso de que un nodo se encuentre en estado neutral, se ha optado porque no inicie la operación. Debido a que está en un punto intermedio y no se sabe cómo va a reaccionar ya que es un estado inestable.

3.5. Operación de equilibrio de carga

Una vez decidido que se va a realizar una operación de equilibrio de carga, lo primero es escoger el nodo con el que se va a realizar dicha operación. Esto se hace con la regla de localización. Seguidamente hay que saber cuánta carga se le envía a cada nodo escogido, para ello está la regla de distribución. Y por último teniendo todo lo anterior se realiza el equilibrio mandando a los nodos escogidos el número de tareas correspondientes.

Regla de localización

Para poder ejecutar una operación de equilibrio de carga es necesario encontrar el nodo con el que se va a realizar. Para ello se busca en la lista de receptores que tiene cada nodo, la cual no tiene porque ser igual ya que depende del orden en el que lleguen los mensajes o de los dominios. En nuestro caso se escogen tres nodos, como máximo, de manera aleatoria para evitar colapsos y se guardan en una lista provisional. Para seleccionar el nodo al que se le van a enviar las tareas se hace mirando cual es el más descargado. Si el nodo elegido cuando se le pide la petición la rechaza, se probaría con el siguiente nodo menos descargado dentro de la lista provisional.

Cuando a un nodo le llega la petición de equilibrio tiene la opción de rechazarla o aceptarla. Si la regla de medida de estado le informa que puede aceptar tareas remotas, informará al nodo que solicito la petición y también le comunicará el número de tareas máximas que puede aceptar. Pero si por el contrario se rechaza la petición, hará lo mismo con el resto que se lo pida hasta que realice una nueva medida de estado.

Regla de distribución

La regla de distribución decide qué cantidad de carga se reparte a cada nodo, es decir que número de procesos debe enviarle el nodo emisor al receptor. Permitiendo mandar tareas hasta que el índice de carga pasa a estado neutro.

Esta regla se ha diseñado repartiendo las tareas al primer nodo de la lista provisional, la cual está ordenada en sentido ascendente, en función de la carga de los nodos. Este reparto continúa hasta que dicho nodo llega al estado neutral. En ese momento las tareas pasarán a ser repartidas al segundo nodo de la lista, y así hasta que se acaben las tareas o los nodos receptores. En este último caso las tareas pasarán a ser encoladas hasta encontrar otro nodo receptor.

Capítulo 4

Implementación del algoritmo

4.1. Introducción

En este proyecto se ha implementado la regla de intercambio de información, por ello se va a hacer más hincapié en ella. Pero en primer lugar se va a realizar una descripción detallada de todas las partes que forman el algoritmo de equilibrio de carga, para que se pueda comprender completamente su funcionamiento.

La implementación del algoritmo de equilibrio de carga se ha dividido en tres procesos, que están replicados en todos los nodos del sistema: load, global y balance. Estas funciones forman el algoritmo de equilibrio de carga funcionando de la siguiente manera.

- El proceso *load* es el encargado de comprobar el estado del nodo y comunicárselo al *global* y *balance* del mismo nodo.
- El proceso *global* se encarga de enviar al resto de globals su estado y de elegir los receptores con los que el nodo va a realizar el equilibrio de carga.
- Por último el proceso *balance* es el que inicia la operación de equilibrio de carga solicitando los receptores y preguntando a dichos receptores si pueden realizar la operación, si es así envía las tareas al correspondiente nodo.

Todas las comunicaciones locales y remotas se realizan gracias a la librería MPI. Estas se pueden ver resumidas en la figura 4.1.

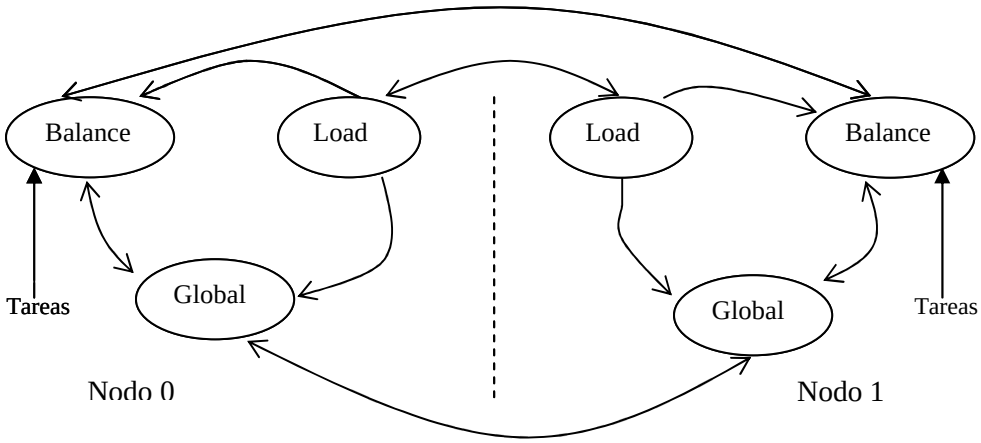


Figura 4.1 Resumen de las comunicaciones.

4.2. Implementación con MPI

Para poder usar MPI lo primero que hay que hacer es incluir en el código la librería mpi.h, ya que sin ella ninguna función que se utilice, de MPI, será encontrada. Además es necesario compilar el programa con el comando 'mpicc', que compila y enlaza programas escritos en C que hacen uso de MPI. Este comando puede utilizarse con las mismas opciones que el compilador de C usual.

Este interfaz lo que hace es crear una serie de procesos y ejecutar el código en todos ellos paralelamente. Para ello es necesario ejecutar el programa mediante el comando 'mpirun'. La forma más común de utilizar este comando es: 'mpirun -np 4 ejemplo'. Siendo '-np' el parámetro que indica que el número que va seguido de él es el número de procesos. En este proyecto, para tener un mayor control sobre qué proceso se lanza en cada máquina, se utiliza el parámetro '-machinefile' seguido de un fichero, en el cual se indican los nodos donde se quieren ejecutar los procesos. Por lo que MPI al primer proceso le ejecuta en el primer nodo del fichero, el segundo en el segundo y así sucesivamente. Si ocurriese el caso en el que hubiese menos nodos que

procesos a ejecutar, MPI vuelve al principio del fichero y sigue mandando los procesos. En este caso habría más de un proceso ejecutándose en algunos nodos. Si en caso contrario hubiese menos procesos que nodos, estos se mandan hasta que se acaban, dejando el resto de nodos sin ejecutar nada en ellos.

Para este proyecto como se quiere que cada nodo ejecute tres procesos seguidos, se repite cada nodo tres veces en el fichero correspondiente. Por lo que los procesos 0, 1 y 2 se ejecutarán en el primer nodo, 3, 4 y 5 en el segundo y así sucesivamente.

Como los tres procesos a ejecutar en cada nodo son *load*, *global* y *balance*, cada uno ejecutará un código distinto. Para ello se ha realizado como aparece en la figura 4.3.

Esto quiere decir que al dividir el rank entre el número de procesos por nodo (en este proyecto 3) el resto quede 0, se ejecutará el código del load, cuando sea 1 el del global y cuando sea 2 el del balance. En el caso del global existirá otro nivel dependiendo de si la regla de intercambio de información es por eventos (3), periódica (4) o bajo demanda (5).

Es importante incluir los parámetros argv y argc, debido a que se va a utilizar MPI en las funciones y son necesarios los valores que contienen para su utilización. El tercer parámetro que se le pasa a las funciones es un comunicador, que se ha creado como aparece en la figura 4.2. El comunicador Load_Comm se utiliza para recopilar datos de los demás loads para poder realizar cálculos. Los otros dos comunicadores son creados para la realización de los experimentos.

```
if( (World_i.rank%Num_of_proc == Load) && (World_i.rank != (World_i.size-1)))
        color=0;
else
        color=1;
MPI_Comm_split( MPI_COMM_WORLD, color, World_i.rank/Num_of_proc, &LOAD_COMM);
```

Figura 4.2. Ejemplo de implementación de un comunicador load.

```
switch( World_i.rank % Num_of_proc ){

   case Load:

        Load_Process( argc, argv, LOAD_COMM);
        break;

  case Global:

        switch(atoi(argv[3])){

            case 3:

                Global_Process( argc, argv,GLOBAL_COMM);
                break;

           case 4:

                Periodic_Global( argc, argv,GLOBAL_COMM);
                break;

            case 5:

                Demand_Global( argc, argv,GLOBAL_COMM);
                break;
           }
           break;

        case Balance:

                Balance_Process( argc, argv,BALANCE_COMM);
                break;
}
```

Figura 4.3. Distribución de los procesos según el tipo.

Como aparece en la figura 4.2 el comunicador se crea gracias a la función MPI_Comm_split. Esta función lo que hace es añadir todos los procesos que tengan el mismo valor en “color” al mismo comunicador, llamado en este caso “LOAD_COMM”. Como sólo se necesitan incluir los procesos loads, a estos se les dará un color con valor cero y al resto uno.

Para crear los comunicadores para los procesos globals o balance bastaría con que el color fuese cero para los globals o balance en cada caso.

En el caso de realizar comunicaciones entre los procesos locales o remotos son todas prácticamente realizadas mediante las funciones MPI_Send() y MPI_Recv().

4.3. Proceso Load

La función de este algoritmo es calcular la carga del nodo cada cierto tiempo fijo. El índice de carga es el que guarda esta carga y con él se sabe en qué estado se encuentra el nodo. A continuación se puede ver un pseudocódigo de este algoritmo.

```
Inicio
cores = numero de cores
local = media de los bogomips en el nodo
cluster = media de los bogomips del cluster
Mientras terminar sea igual a 0 realiza:
        tasks = obtiene el número de tareas
        index = índice de carga = (local / cluster ) * cores / tasks
        Si tasks es menor o igual que cores entonces:
                max = máximo de tareas = (1.33 * cores) – tasks
        si no entonces:
                max = ((local / cluster) * cores /  valor neutral medio) – tasks
        fin
        Si tasks es menor que 0 entonces:
                max = 0
        fin
        nuevo_estado = obtiene el estado actual
        comunica estado a los procesos correspondientes
        viejo_ estado = nuevo_ estado
        comprueba si hay que acabar
fin
```

Figura 4.4. Pseudocódigo del proceso load.

4.3.1. Funcionamiento

Para calcular el índice de carga son necesarios una serie de parámetros correspondientes a los nodos donde se ejecuta el programa. Estos datos son leídos por el sistema operativo linux, y son almacenados en la carpeta /proc/. Su actualización la realiza el kernel.

El primer archivo que se lee es *cpuinfo*, el cual proporciona al usuario mucha información estática del sistema (estructura, potencia, número de núcleos, …). Para este proyecto solo va a ser necesario utilizar el número de núcleos y la potencia del sistema, expresada en bogomips.

El segundo que se utiliza es el archivo *loadavg*. Del cual se va a utilizar la información que da sobre la cantidad de procesos que se están ejecutando en el momento de lectura de este archivo.

Con estos datos se puede calcular el índice de carga según la expresión de la figura 4.5.

$$Load_{index} = \frac{Bogomips_{local\_average}}{Bogomips_{cluster\_average}} \cdot \frac{Cores\_Number}{Tasks\_Number}$$

Figura 4.5. Expresión para el cálculo del índice de carga.

Siendo:

- $Bogomips_{local\_average}$ el valor medio de los bogomips de los procesadores de la máquina. Esto se calcula sumando la potencia de cada núcleo, y dividiendo esta suma entre el número de cores. La potencia está expresada en bogomips y es leída del fichero cpuinfo.
- $Bogomips_{cluster\_average}$ la media del parámetro anterior que tiene todo el sistema al principio de la ejecución. Este valor no cambia posteriormente, y sirve para saber si se van a poder aceptar más tareas dependiendo de si es más potente que la media.
- *Cores_Number* el número de procesadores que tiene el nodo en el que se calcula el índice. Este valor se lee del archivo cpuinfo y no es

necesario volverlo a leer, ya que este número no cambia (número de cores estático).

- *Task_Number* el número de tareas que se están ejecutando. Este valor se lee del archivo loadagv y se refresca cada cierto tiempo. El tiempo de refresco es importante ya que determinará el buen funcionamiento de la regla de medida de estado.

Una vez calculado el índice de carga hay que determinar en qué estado se encuentra el nodo. Los posibles estados que se consideran en el proyecto son, emisor, neutral o receptor. Para ello se tiene en cuenta la carga de cada nodo. De esta forma si un nodo no está utilizando todos los procesadores, automáticamente pasa a estado receptor. Pero por otro lado, si esto no ocurre se definen una serie de umbrales de cambio, dependiendo de la relación procesos/cores (Tabla 4.1).

Estado	**Relación procesos/cores**
Receptor	< 1
Neutral	1 > x > 1.5
Emisor	> 1.5

Tabla 4.1. Relación de procesos/cores para cada estado

Debido a que pueden surgir cambios de estados innecesarios e inestabilidad en los nodos se ha añadido una histéresis para evitarlo. Quedando los umbrales como aparecen en la tabla 4.2.

Cambio de estado que se genera	Valor del índice
Receptor a Neutral	< 0.800
Cualquier estado a Receptor	> 1.000
Emisor a Neutral	> 0.727
Cualquier estado a Emisor	< 0.667

Tabla 4.2. Umbrales para los cambios de estado.

Cada nodo debe conocer si es capaz de aceptar nuevas tareas o en su lugar rechazarlas, por lo que se calcula el número máximo de procesos que puede ejecutar. Este cálculo depende del número de procesos y del número de núcleos, por lo que existen dos casos:

- *Número de procesos menor o igual al número de núcleos*: en este caso se calcula un número máximo para que haya un proceso y un tercio por cada core (1.33 procesos por cada núcleo).
- *Número de procesos mayor que el número de núcleos:* este caso es el más sensible ya que está más cerca del cambio de estado. Por ello se ha hará en función del índice y no de la carga de procesos, mediante la expresión siguiente:

$$Num_{procesos} = \frac{Bogomips_{local}}{Bogomips_{cluster}} \quad \frac{Num_{Cores}}{Valor_{Nuevo\_Indice}} \quad - Num\_tareas_{Actual}$$

Figura 4.6. Expresión para un número mayor de núcleos.

Una vez calculado el estado actual del nodo se envía dicho estado al proceso global para poder realizar o no equilibrios de carga, y al proceso balance para la regla de iniciación. Si el estado actual es receptor, además al balance se le envían el número máximo de tareas que va a poder ejecutar, para realizar la operación de equilibrio de carga.

4.4. Proceso Global

Este algoritmo consiste básicamente en la regla de información ya que aunque realiza otras funciones como encargarse de la regla de localización del receptor, la principal es informar al resto de nodos de su estado. En este proyecto se van a implementar tres versiones del global que se ejecutan dependiendo del caso que se quiera tratar.

A continuación se puede observar un pseudocódigo del funcionamiento de este proceso sin hacer hincapié en las distintas reglas, ni en el envío de información.

```
Inicio
nuevo_estado = se inicializa a estado neutral
list_receptores = genero el array para almacenar los receptores

Mientras terminar sea igual a 0 realiza:

        Espera hasta que recibe un mensaje

        Si recibe:
                Mensaje con etiqueta "Local_Request":

                        Proceso balance pregunta por receptores
                        Selecciona receptores de list_receptores

                Mensaje con etiqueta "Local_Change":

                        nuevo_estado = recibe estado actual del proceso load

                Mensaje con etiqueta "Remote_Change":

                        estado = recibe estado de nodo remoto

                        Si estado == Receptor entonces:
                                Añado estado a list_receptores

                        si no entonces:
                                Borro estado de list_receptores
                        fin

                Mensaje con etiqueta "Close":

                        Recibe acabar la ejecución
                        terminar = 1
        fin
fin
```

Figura 4.7. Pseudocódigo en general del proceso global.

4.4.1. Funcionamiento

Sin hacer distinción entre las reglas, este algoritmo se queda expectante a la llegada de envíos del resto de procesos. Por lo que no realiza ningún cálculo, solo hace de intermediario entre los procesos y almacena la información.

Los envíos que recibe este algoritmo pueden ser a nivel local, es decir entre procesos del mismo nodo, o a nivel remoto, entre procesos de otros nodos.

Envíos a nivel local

A nivel local existen envíos de los dos procesos restantes del nodo: load y balance.

Cuando el envío es realizado por el proceso *load*, se conoce gracias a la etiqueta “Local_Change”. Con esto se sabe que el proceso global va a recibir el estado actual del nodo. Como se ha indicado antes, el proceso load ejecutará el envío cuando se haya realizado un cambio a receptor o haya dejado de ser lo.

Si el envío se produce del *balance*, esto quiere decir que se ha iniciado una operación de equilibrio y que se necesita saber el número de receptores disponibles. En este caso la etiqueta “Local_Request” es la encargada de indicar que es el proceso balance el que está realizando la petición. Con esto el proceso global está cumpliendo la regla de localización, ya que se encarga de seleccionar el número de receptores. Esta selección se reduce a un número máximo de receptores, en este proyecto de tres. Esto se hace así, debido a que si se le devolviese al proceso balance la lista completa de receptores y en esta hubiese un número muy alto de ellos, sería muy costoso elegir el más descargado. Por ello se eligen aleatoriamente, excepto cuando el número de receptores que se tiene es menor al máximo posible. En este último caso se enviarían todos los receptores que hubiese en la lista.

El proceso global después de determinar el número de receptores que va haber para el equilibrio de carga, envía al proceso balance el número de receptores con el que va a poder trabajar y los identificadores de los receptores seleccionados.

Envíos a nivel remoto

A nivel remoto solo se producen los envíos del proceso global de otros nodos. Con estos envíos se comunica la información del estado del nodo que envía, por lo que dependiendo de la regla de información habrá distintos casos en el momento del envío.

Con la etiqueta “Remote_Change” se sabrá que es un proceso global el que comunica la información y se recibirá su estado. Debido a que se pueden recibir estados de varios nodos, se comprueba si el estado recibido es receptor para añadirle a la lista de receptores o si ha dejado de ser lo, para que pueda ser eliminado.

Lista de receptores

Para implementar la regla de intercambio de información es importante guardar los estados que llegan de nodos remotos, para el momento que se realice la regla de localización. Por eso se ha implementado un lista, llamada *lista de receptores*, que almacena los nodos que envían la información pero sólo si su estado es receptor.

Para generar dicha lista hay que tener en cuenta que el número de receptores puede llegar a ser, el máximo de nodos que existan. Por ello se guarda espacio para poder almacenar el número total de nodos multiplicado por el tamaño de un entero. Esto se realiza mediante la función malloc como se puede observar a continuación.

```
st_info.List_Receptors = (int *) malloc ((sizeof (int))*(World_i.Size/Num_of_proc));
```

El número de nodos máximo se haya dividiendo el número total de procesos entre el número de procesos por nodo, que en este proyecto son tres.

Una vez que esta generada la lista vacía, habría que ir rellenándola con los nodos receptores que van a llegando a cada nodo. Para añadir un nodo a la lista primero se comprueba si es receptor y si es así se añade el identificador de ese nodo en la última posición de la lista y se incrementa uno el contador que lleva el número de receptores (Num_of_Receptors). En el caso contrario de que se

quiera eliminar un nodo de la lista debido a que ha dejado de ser receptor, se buscará dicho nodo en la lista y se elimina. Con esto quedaría un espacio libre pero esto no ocurre ya que el último nodo de la lista pasaría a ocupar esa posición vacía. También habría que decrementar el contador Num_of_Receptors.

4.4.2. Tipos de algoritmos

En este proyecto se implementan los tres tipos distintos de reglas de intercambio de información que analizamos en el apartado 3, esto es periódica, bajo demanda y por eventos. A continuación se explican los tres casos, detallando el paso de mensajes a nivel remoto, es decir el intercambio de información entre procesos globals.

4.4.2.1. Periódica

Esta regla como se ha explicado anteriormente, se diferencia de las demás debido a que la información se envía periódicamente.

Para realizar el envío periódicamente se han utilizado las señales propias del sistema operativo Linux, las cuales son un mecanismo de comunicación síncrono o asíncrono entre procesos.

Para este caso se utiliza la señal 'SIGALRM', que se produce cuando finaliza un temporizador. Cuando esto ocurre se espera a que finalice la instrucción de código que se esté ejecutando, se ejecuta una excepción, guardando el estado del proceso para poder reanudar su ejecución, y se salta a la función de tratamiento de señal adecuada, definida por el proceso receptor. Al acabar esta función se continúa con la ejecución en la instrucción que se había dejado antes de que se produjese la señal. A continuación se puede ver cómo hay que declarar la señal y cómo inicializar el temporizador.

```
signal(SIGALRM, envio);
alarm(x);
```

Con la primera función se declara qué tipo de señal será y cuál va a ser el nombre de la función de tratamiento de señal, en este caso envio. La segunda línea inicializa el temporizador con el tiempo ‘x’.

Dentro de la función de tratamiento de señal se realiza el envío de la información. Esta información sería el estado actual del nodo y el estado viejo, por lo que para no realizar dos envíos a cada nodo, se empaquetan en una sola variable mediante la función MPI_PACK, explicada en el apartado 2.2.6. Con lo que solo se realiza un envío por cada nodo.

Como se quiere que continuamente este sonando la alarma cada 'x' tiempo, es necesario volver a inicializar el temporizador. Esta nueva inicialización se realizará al final de la función de tratamiento y cuando acabe el tiempo elegido se volverá a repetir el mismo proceso.

La función de tratamiento de señal no permite el paso de variables a partir de ellas. Por lo que se optó por utilizar variables globales en el caso de que fuese necesario. Es decir, para las variables que fuese a ser necesaria su utilización en dicha función. Se tomó esta decisión porque no supone ningún riesgo de sobre escritura y permite que dichas variables estén siempre actualizadas y que se puedan utilizar dentro de la función de tratamiento.

Cuando se produzca el intercambio de información, el nodo remoto que la recibe mediante la etiqueta “Remote_Change”, tendrá que desempaquetar la información y comprobar si ese nodo está o no, en su lista de receptores. Por lo que si un nodo no está en la lista de receptores es que en la comunicación anterior no era receptor, con lo que también hay que comprobar si ahora lo es. Si es así se añade a la lista. En el caso en el que el nodo si esté en la lista porque en el envío anterior era receptor, se comprueba si sigue siendo receptor y si no es así se borra de la lista d receptores. Esto se puede observar en la figura 4.8.

```
Se desempaqueta en las variables nuevo y viejo

Mientras exista receptor en la lista  realiza:
        Si receptor de la lista = nodo que envía entonces:
                pertenece = 1 = ya existe el nodo en la lista
        fin
fin

Si nuevo = Receptor y viejo distinto Receptor y pertenece = 0 entonces:
        Añade nodo que envía a la lista de receptores
Si nuevo distinto Receptor y viejo = Receptor y pertenece = 1 entonces:
        Borra nodo que envía de la lista de receptores
fin
```

Figura 4.8. Pseudocódigo del recibo de información de un nodo remoto

4.4.2.2. Bajo demanda

El intercambio de información para este tipo de algoritmo, se produce cuando el proceso balance solicita los receptores. Esto quiere decir que sólo cuando se produzca una operación de equilibrio de carga, se solicitarán los estados del resto de nodos. Por lo que la regla de información y de localización se producen en el momento de la operación de equilibrio.

En este caso el proceso global, cuando recibe la solicitud del balance con etiqueta “Local_Request”, solicita al resto de nodos su estado actual. A continuación recibe todos los estados de cada nodo comprobando si el estado recibido es receptor y si es así se añade a la lista de receptores. El siguiente paso es seleccionar los receptores a enviar al proceso balance, pero esto es igual en los distintos casos de la regla de intercambio de información y ha sido explicado anteriormente (apartado 4.4.1).

La operación de comunicación etiquetada con “Remote_Change” será la encargada de solicitar el estado actual de un nodo remoto. En este caso no se

cumpliría lo explicado en el apartado 4.4.1, ya que la etiqueta se utiliza para enviar el propio estado a otro global que lo solicitó.

4.4.2.3. Por eventos

En este caso la información del estado se envía cuando se produce un cambio de estado local, es decir en el propio nodo. Esto se produce cuando al proceso global le llega la etiqueta “Local_Change”, del proceso local. Entonces se comprueba si el estado al que ha cambiado es receptor o si ha dejado de ser lo. Si esto ocurre se envía el estado al resto de nodos, siempre a través de los procesos global correspondientes.

En el caso contrario de que el proceso reciba los estados del resto de nodos, la etiqueta correspondiente es “Remote_Change”. Cuando el estado ha sido recibido se comprueba si es receptor o deja de ser lo, para añadir o borrar respectivamente de la lista de receptores.

4.4.3. Tipos de dominios

Como se ha hablado en anteriores capítulos, se pueden crear una serie de grupos con un número de procesos, denominados *dominios*. Estos se crean debido a que si el número de nodos es muy elevado, se hace muy costoso realizar el intercambio de información entre todos ellos. En este proyecto se han implementado dos tipos: dominios solapados y dominios aleatorios.

A continuación se van a explicar los dos tipos, pero especificando que cambiaría respecto al reparto de la información entre todos los nodos.

4.4.3.1. Dominios solapados

En el caso de los dominios solapados, los nodos son seleccionados por orden del rango al que pertenecen, dentro del comunicador universal (MPI_COMM_WORLD). Pero se introducirá un aspecto importante que

denominaremos *solapamiento*, con el que habrá procesos que pertenezcan a más de un dominio. Este solapamiento sirve para que exista comunicación entre los distintos dominios. Así si un dominio está muy saturado y otro muy vacío se podrán comunicar y equilibrarse.

En este proyecto se ha seleccionado un solapamiento para que sea siempre la mitad del tamaño del dominio. Ya que si no fuese así, habría nodos que pertenecerían a un dominio y otros nodos a varios, y por lo tanto la tarea de implementarlo sería muy costosa.

En una primera opción se intentó implementar los dominios creando comunicadores para cada uno de ellos, pero surgió un problema. Este fue debido a que al crear los comunicadores se enviaba la información a través de ellos, pero luego al recibirla no se podía saber de qué comunicador venía. También se pensó enviar mediante cada comunicador de cada dominio y recibir del comunicador universal, pero MPI esto no lo permite ya que si se envía un mensaje con un comunicador, al recibir tiene que ser del mismo.

Todo esto se solucionó dejando de usar comunicadores y creando en cada nodo las listas correspondientes a los dominios a los que pertenecen, las cuales se van a llamar *listas de dominios*. Así en el momento de enviar la información sólo se hará a los nodos pertenecientes al dominio.

```
list_Domin1 = (int *) malloc ((sizeof (int))*(Tam_Domin-1));

list_Domin2 = (int *) malloc ((sizeof (int))*(Tam_Domin-1));
```

Figura 4.9. Generación de las listas de dominios

Antes de rellenar las listas con los datos correspondientes, se generan con un tamaño correspondiente al tamaño de cada valor añadido a ella (int) por el tamaño del dominio menos uno (figura 4.10). Esto último es debido a que en las listas el propio nodo no se va a enviar la información a sí mismo, por lo que no va a ser añadido.

Las listas de dominios se rellenan basándose en la posición del propio nodo. Ya que dichas listas pueden contener nodos con un identificador mayor o menor. Como en este proyecto cada nodo va a pertenecer siempre a dos dominios se crearán dos listas de dominios (en cada nodo). Esto se hace primero rellenando la primera lista y añadiendo los nodos que estén por encima de su rango (sumando valores al rango del nodo), mientras se comprueba que no se añadan más nodos de los que se debe. Esto se hace hasta que el módulo del identificador del nodo con el solapamiento sea igual al solapamiento menos uno.

Para los casos en los que el rango de los nodos esté por debajo del propio nodo se añadirán a la lista restando los sucesivos valores, al rango del propio nodo.

A continuación se puede ver un ejemplo con 8 nodos en un dominio con tamaño 4 y solapamiento 2 (pero sólo está representado el nodo 3, siendo N).

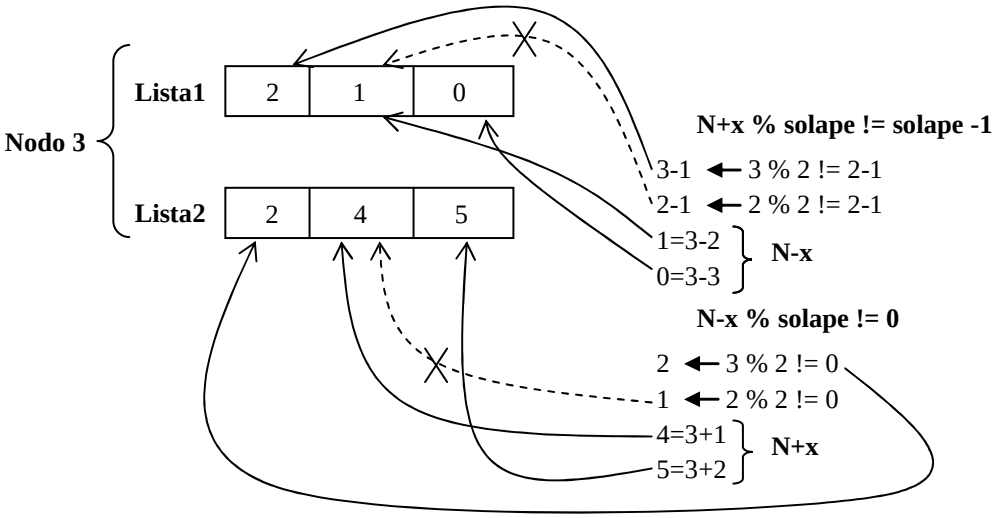


Figura 4.10: Ejemplo de creación de las listas de dominios.

A continuación habría que rellenar la segunda lista que se haría básicamente de la misma forma que la primera excepto que se empieza por añadir los valores que están por debajo del rango del nodo y seguidamente los que están por encima.

En el caso de los nodos que pertenezcan a una lista que incluya a los nodos extremos habría que comprobar que no llegan al final porque si esto ocurriese

se añadirían nodos negativos o nodos que ni existiesen. Esto se soluciona si cuando se llega al principio del identificador o se es el nodo primero (nodo 0), se iguala al número máximo de nodos menos 1 y se continúa desde ahí. Para el caso de que se llegue al máximo de nodos o se sea el último nodo se hará exactamente lo mismo pero igualando al número menor de nodos.

Estos dominios se implementan de la misma forma para los tres casos de reglas de intercambio de información, excepto que en el caso de la periódica las listas de dominios habría que declararlas como variables globales para poder utilizarlas en la función de tratamiento de señal.

Al realizar el envío del propio estado del nodo mediante estos dominios solapados, se van recorriendo las listas de dominios y enviando a cada nodo perteneciente a estas. Pero hay que tener en cuenta que puede haber nodos repetidos entre las dos listas si el solapamiento es mayor de uno. Por lo que se crea una lista provisional donde se añaden los nodos a los que se ha enviado la información, y se comprueba si el nodo al que se va a enviar aparece en esta lista. Si esto es así se pasa al siguiente nodo de la lista sin enviar nada.

Para el caso del nodo que recibe información de otro nodo remoto, sería de la misma forma que en los casos en los que no existe un dominio.

4.4.3.2. Dominios aleatorios

En estos dominios, como su nombre indica, los nodos a los que se va a enviar la información se eligen aleatoriamente. Dependiendo del caso en el que se esté (periódica, bajo demanda o por eventos), se realizará la selección de nodos de forma aleatoria en el lugar donde corresponda.

En el momento de realizar el envío del estado del nodo, es cuando aleatoriamente se escogen los nodos entre cero y el valor máximo de nodos. Y hay que comprobar que el nodo al que se pretende enviar la información no es el propio nodo y que no se le ha enviado anteriormente. Por eso ocurre como en los dominios solapados, que se crea una lista provisional para comprobar que no se repite el envío a ningún nodo.

4.5. Proceso Balance

La función de este algoritmo es realizar la operación de equilibrio de carga. Para ello desarrolla la regla de inicialización, parte de la regla de localización y la operación en sí misma. A continuación se puede ver un pseudocódigo de este proceso.

```
Inicio
Mientras terminar sea igual a 0 ó haya tareas pendientes realiza:
        Espera hasta que recibe un mensaje
        Si recibe:
                Mensaje con etiqueta "Local_Change":
                        actual = recibo estado de load
                        Si actual distinto de emisor entonces:
                                max = recibo las tareas máximas que puede ejecutar
                                Mientras existan tareas realiza:
                                        Ejecuta los procesos
                                fin
                        si actual = emisor y existen tareas entonces:
                                Inicio operación de equilibrio
                        fin
                Mensaje con etiqueta "Remote_Petition":
                        Si  puede ejecutar las tareas remotas entonces:
                                Ejecutará los procesos remotos
                        fin
                Mensaje con etiqueta "Local_Petition":
                        Ejecuta las tareas locales
                Mensaje con etiqueta "Close":
                        terminar = 1 = Recibe acabar la ejecución
        fin
fin
```

Figura 4.11. Pseudocódigo del proceso balance.

Una de las funciones de este proceso es ejecutar las tareas a nivel local, es decir las que manda el usuario al propio nodo. Este tipo de comunicación tiene la etiqueta “Local_Petition” y recibirá un string de 32 caracteres con el nombre del proceso a ejecutar y su path relativo. Este string se guarda en la cola de procesos y dependiendo del estado en el que se encuentre el nodo será ejecutado de forma local o remota. Una vez que fuese ejecutado el proceso se borraría de la cola de procesos.

En el caso que se necesite ejecutar un proceso de forma remota, debido a que el nodo se ha saturado (estado emisor), se iniciará la operación de equilibrio de carga de trabajo. Para ello se necesita saber con qué nodos se puede realizar esta operación. El proceso global es el que nos comunica estos nodos, los cuales están en estado receptor. Una vez que se tiene la lista de nodos receptores, se envía la petición al correspondiente proceso balance de cada nodo (etiqueta “Remote_Petition”), recibiendo el máximo de tareas que puede ejecutar. A continuación se ordenan los nodos para que se ejecute el primer proceso en el que menor carga tenga, y se envían las tareas a los nodos hasta que se saturen dichos nodos o se acaben las tareas a enviar. Todo este proceso se realiza mediante las funciones de comunicación punto a punto de MPI.

Cuando el proceso balance de un nodo es el que recibe la petición de equilibrio mediante la etiqueta “Remote_Petition”, envía el máximo de tareas al proceso balance que le ha solicitado la operación. Si este nodo puede ejecutar los procesos remotos acepta la operación y ejecuta cuantos pueda hasta llegar al estado emisor. La manera de ejecutar los procesos remotos, o en otro caso locales, se produce haciendo un hijo mediante la función fork(). Este hijo ejecuta el proceso mediante la llamada a la instrucción exec y una vez que el proceso haya acabado de ejecutarse el hijo morirá.

Otro proceso con el que se comunica el proceso balance es con el load del mismo nodo mediante la etiqueta “Local_Change”. Esto ocurre ya que el balance necesita saber en qué estado se encuentra y qué cantidad de tareas puede ejecutar, para poder tomar la decisión de realizar la operación de equilibrio o no.

Capítulo 5

Experimentación

5.1. Descripción de los experimentos

Una vez implementadas todas las variantes de la regla de información explicadas en el capitulo anterior, es necesario realizar una serie de experimentos para validar y verificar su correcto funcionamiento. Así mismo estos experimentos nos servirán para tener una primera comparativa entre las distintas reglas y sacar una serie de conclusiones respecto a su rendimiento en los algoritmos de carga de trabajo.

Por lo tanto se explicarán como se han realizado estos experimentos así como también el sistema en el cual se han realizado.

5.1.1. Descripción del sistema

Para la realización de los experimentos se ha escogido como entorno de trabajo un multicomputador o clúster de memoria distribuida. Este clúster, llamado **calderón**, pertenece al grupo ATC de la Universidad de Cantabria, y está compuesto por unos 75 nodos de diferentes características. Esto quiere decir que existen nodos con diferente cantidad de núcleos y diferentes potencias de cómputo.

Además utiliza un sistema operativo Linux Debian y una red de interconexión basada en una topología en 'estrella' con enlaces de 10 Gigabit Ethernet, de

fibra óptica. Esta topología está compuesta a su vez, por switches Juniper EX4200.

Debido a que este proyecto se basa en la utilización de un clúster homogéneo, se han escogido una serie de nodos con las mismas características, es decir con un procesador AMD single core. El número de nodos disponible para la realización de estas pruebas ha sido 8. Esto se debe a que era necesario disponer de un entorno de trabajo aislado del resto de usuarios, de forma que los experimentos fueran controlados y repetibles.

Como ya se sabe, la comunicación entre procesos en el clúster se realiza mediante paso de mensajes, es decir a través del interfaz MPI. En este proyecto se utiliza la versión 1.2.7.

5.1.2. Descripción de los escenarios

En este proyecto se ha comprobado la regla de información en dos escenarios diferentes. El primero consiste en que a un nodo se le envían todas las tareas a repartir. En ese caso éste nodo empieza a realizar las ejecuciones en local y cuando su estado pase a emisor comenzará a lanzar tareas al resto de nodos disponibles, siguiendo el algoritmo descrito en los capítulos 3 y 4. A partir de ahora este caso se le llamará '**escenario 1**'.

El segundo escenario (**escenario 2**) consiste básicamente en lo mismo pero en vez de enviar todas las tareas a un nodo, se mandan a más de uno. Por lo que en este caso las tareas las reparten más de un nodo.

El número de tareas a enviar para realizar las pruebas no ha sido ni un número muy pequeño, ya que no se podría comprobar correctamente el funcionamiento de la regla de información, ni muy grande porque el tiempo de ejecución sería muy largo. Por lo que se ha escogido empíricamente un valor entre medio de 160 tareas. Cada una de estas tareas es enviada cada segundo, esto se realiza así para que exista un cierto margen.

Como se ha explicado antes, en el caso del **escenario 1** las 160 tareas se envían a un nodo, pero en el otro caso se mandan a dos nodos, correspondiendo 80 tareas a un nodo para repartir y 80 al otro.

Para comprobar la regla de información se realizan una serie de medidas.

- **Tiempo óptimo (top)** que se define como el tiempo que tarda en ejecutar las tareas un nodo, sin la existencia de un algoritmo de equilibrio de carga. Como en este proyecto se envían 160 tareas y se tienen 8 nodos, el reparto óptimo serían 20 tareas por cada nodo. De acuerdo con esto, se mide el tiempo que se tarda en ejecutar estas 20 tareas en cada nodo. Como los nodos son homogéneos el tiempo que tarda un nodo en ejecutar las tareas es el mismo que cualquiera de los otros, siendo el tiempo óptimo el mismo para todos los nodos.

- **Tiempo de reparto (trep)** que como su nombre indica, es el tiempo que se tarda en repartir todas las tareas. Es decir el tiempo desde que comienza la ejecución del experimento hasta que finaliza el reparto de tareas, si bien muchas de ellas estarán todavía en ejecución.

- **Tiempo de ejecución (tejec)** es el tiempo que se tarda en realizar todas las tareas. Es decir, es el tiempo medido desde que se comienza la ejecución del experimento hasta que todas las tareas lanzadas han finalizado su ejecución por completo.

- **Envíos totales (env tot)** es una variable que engloba todas las comunicaciones que realiza cada nodo (proceso global) con el resto de ellos para enviar el estado en el que se encuentra cada uno. En cada tipo de proceso global ya sea bajo demanda, periódica o por eventos, cada comunicación se realizará como se explicó en el apartado 2.3.2.2.

- **Envíos aceptados (env acep)** es el número de peticiones de equilibrio de carga que realmente se ejecutan. Esto quiere decir, que de las peticiones de equilibrio que se solicitan a otros nodos, cuántas de ellas se producen.

- **Envíos no aceptados (env no acep)** es igual que en el caso anterior pero en vez de contar los casos en los que se produce una operación de equilibrio, ocurre al contrario, serán medidas las solicitudes de operación de equilibrio de carga que no se llegan a producir, es decir cuántas peticiones se solicitan con datos obsoletos.

Otro valor del que se va a hacer uso para la comprobación del funcionamiento es la eficiencia (ε) [9]. Este valor no se mide directamente, sino que se va a calcular en función de algunos de los parámetros anteriormente medidos. En concreto la fórmula correspondiente es: $\varepsilon = \frac{env\ acep}{env\ tot * n^o\ tareas} * \frac{top}{tejec}$

En cuanto a los dominios es necesario elegir unos datos con los que trabajar en la experimentación. En el caso de los dominios estáticos se han escogido dos tipos: 4 nodos por dominio con un solapamiento de 2 nodos y 2 nodos por dominios con un solapamiento de 1 nodo. En el otro caso en el que los dominios son aleatorios, se han realizado las pruebas con todos los casos posibles desde 2 nodos por dominio hasta 7 nodos por dominio. Pero solo se ha escogido comentar 3 de los 5 tipos de dominios aleatorios, debido a que algunos eran muy parecidos y se sacaban casi las mismas conclusiones.

Cuando el proceso global es de tipo periódico es necesario elegir un tiempo de periodicidad. En este proyecto se van a realizar los experimentos cambiando este tiempo para así poder comprobar que sucede en el equilibrio de carga si se utiliza un tiempo muy grande o uno muy pequeño. Los tiempos utilizados son 1 segundo, 3 segundos, 5 segundos y 10 segundos.

5.2. Resultados del escenario 1

En este apartado se van a poder observar las tablas y gráficas de los experimentos que se han realizado en un tipo de escenario. En este escenario, como se ha explicado en el apartado anterior, las tareas se envían a un nodo (nodo 0) y este mismo es el encargado de realizar el reparto cuando el nodo no sea capaz de seguir ejecutando tareas (se sature).

A continuación se pueden ver las tablas de los distintos experimentos que se han realizado, haciendo uso de los parámetros explicados en el apartado 5.1.2.

El primero es el caso (tabla 5.1) en el que no existe ningún dominio, es decir el nodo 0 puede repartir las tareas entre todos los nodos.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1694	139	61	393	430	0,921	4,72E-4
Periódica 1	396	23065	139	127	416	453	0,874	3,29E-05
Periódica 3	396	7392	139	245	400	438	0,904	1,06E-4
Periódica 5	396	4704	138	354	424	462	0,857	1,57E-4
Periódica 10	396	2688	136	845	484	521	0,76	2,4E-4
Bajo deman	396	3612	140	73	392	428	0,925	2,24E-4

Tabla 5.1: Experimentos sin dominios.

Las siguientes tablas han sido realizadas mediante dominios, es decir, que el nodo 0 solo repartirá entre nodos que pertenezcan a dicho dominio.

En las dos primeras tablas (tabla 5.2 y 5.3) se realizan los experimentos con dominios estáticos.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	592	105	16	1093	1131	0,35	3,88E-04
Periódica 1	396	17312	106	20	1086	1123	0,353	1,35E-05
Periódica 3	396	5840	106	104	1099	1137	0,348	3,95E-05
Periódica 5	396	3616	103	227	1134	1171	0,338	6,02E-05
Periódica 10	396	1936	99	472	1214	1252	0,316	1,01E-4
Bajo deman	396	2394	106	14	1091	1128	0,351	9,72E-05

Tabla 5.2: Experimentos con dominio estático 2.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1390	133	44	548	585	0,677	4,05E-4
Periódica 1	396	21735	132	61	547	585	0,677	2,57E-05
Periódica 3	396	7360	131	201	557	594	0,667	7,42E-05
Periódica 5	396	4680	130	331	592	629	0,63	1,09E-4
Periódica 10	396	2480	129	677	624	661	0,599	1,95E-4
Bajo deman	396	3395	132	38	558	594	0,667	1,62E-4

Tabla 5.3: Experimentos con dominio estático 4.

Las tres últimas tablas (tabla 5.4, 5.5 y 5.6) se refieren a los experimentos realizados con los dominios aleatorios.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	864	77	1202	1659	1696	0,234	1,3E-4
Periódica 1	396	9774	139	245	412	449	0,882	7,84E-05
Periódica 3	396	3495	138	467	441	495	0,8	1,97E-4
Periódica 5	396	2301	135	881	484	521	0,76	2,79E-4
Periódica 10	396	1416	130	1674	595	631	0,628	3,6E-4
Bajo deman	396	1602	138	36	414	450	0,88	4,74E-4

Tabla 5.4: Experimentos con dominio aleatorio 3.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1495	121	755	774	819	0,484	2,45E-4
Periódica 1	396	15960	139	150	404	442	0,896	4,88E-05
Periódica 3	396	5400	138	308	410	448	0,884	1,41E-4
Periódica 5	396	3600	136	497	455	492	0,805	1,9E-4
Periódica 10	396	2080	134	1125	526	564	0,702	2,83E-4
Bajo deman	396	2600	140	54	399	437	0,906	3,05E-4

Tabla 5.5: Experimentos con dominio aleatorio 5.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1953	139	65	396	433	0,915	4,07E-4
Periódica 1	396	22232	139	106	402	439	0,902	3,53E-05
Periódica 3	396	7560	139	226	408	446	0,888	1,02E-4
Periódica 5	396	5075	136	373	459	497	0,797	1,34E-4
Periódica 10	396	2856	134	853	515	553	0,716	2,1E-4
Bajo deman	396	3675	139	72	402	440	0,9	2,13E-4

Tabla 5.6: Experimentos con dominio aleatorio 7.

Con los datos correspondientes a las tablas anteriores, se han realizado unas gráficas para poder comparar de forma más cómoda los datos.

Por un lado se ha escogido realizar una gráfica del cociente top/tejec, que representa el cociente entre el tiempo que debería tardar el clúster en realizar las 160 tareas, pero de forma equilibrada (cada nodo 20), y el tiempo que tarda

con el algoritmo de equilibrio de carga. Por lo que el valor que más se acerque a uno será el más próximo al valor óptimo.

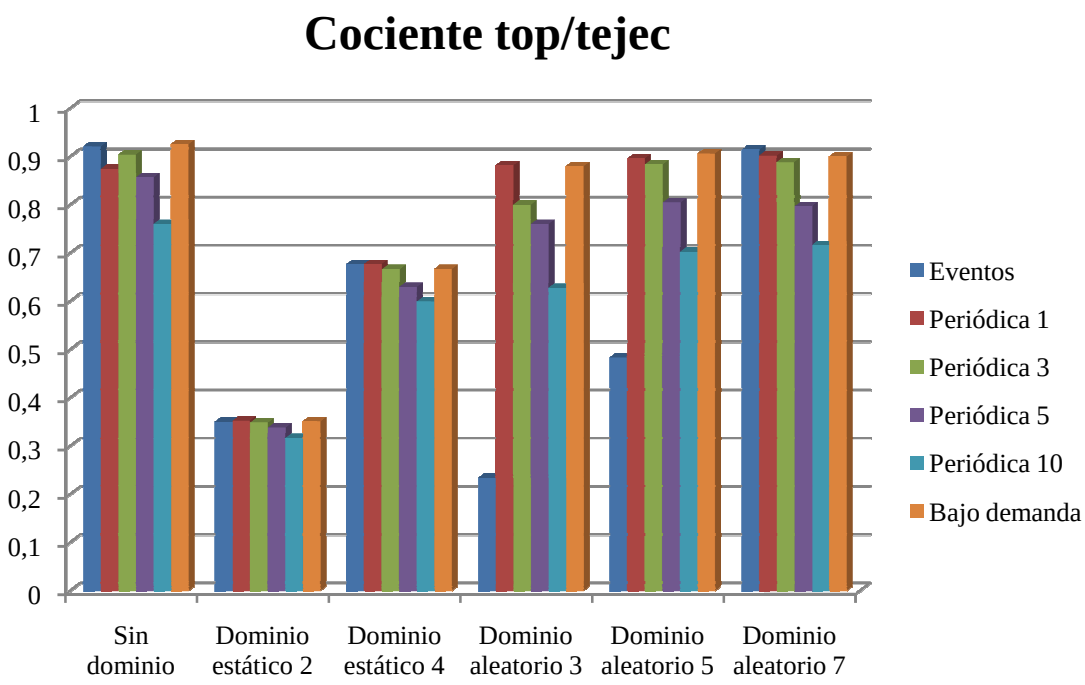


Figura 5.1: Representación del Cociente top/tejec.

En esta gráfica (figura 5.1) se han representado los procesos global respecto de las opciones de diseño en el eje de abscisas y los valores del cociente de top/tejec en el eje de ordenadas. Estos valores solo varían de 0 a 1 ya que no puede ser mayor que uno porque nunca va a ser mejor el tiempo de ejecución que el tiempo óptimo.

Analizando los valores de la gráfica se puede ver, que en general, el dominio estático 2 es el que menos se acerca al tiempo óptimo, con diferencia. Otra cosa que se puede ver es que los dominios estáticos son los peores respecto a los demás. Esto es debido a que el dominio estático siempre trabaja con los mismos nodos. No ocurre lo mismo con los dominios aleatorios ya que no se utilizan los mismos nodos debido a que se escogen aleatoriamente. Aunque sí utilizan el mismo número de nodos, igual que los estáticos. También se podría decir que cuando no se utilizan dominios se usan los mismos nodos, pero en este caso el número de nodos es mayor ya que se utilizan todos ellos. Todo esto

es lógico ya que se tiene un número muy bajo de nodos y las reglas con dominios tienen sentido cuando el número de nodos crece mucho y considerarlos a todos es muy costoso desde el punto de vista de la comunicación.

Si se mira cual de todos los procesos se acerca más al tiempo óptimo ese sería el proceso bajo demanda en el caso que no hubiese dominios. Pero si se comparan cada proceso en todas las opciones de diseño se puede ver que el proceso periódica 1 es la que más estable se mantiene entre estos diseños. Esto puede ser debido a que, al ser la que más actualizada tiene sus nodos, en cuanto existe un nodo disponible se utiliza. Debido a esto mismo, el proceso periódico 10, respecto a los procesos periódicos, es el que más se aleja del tiempo óptimo, como se puede ver en la gráfica.

Fijándose solo en los procesos periódicos, los que mejores resultados dan son con 1 segundo y con 3 segundos, por lo que con 5 segundos y 10 segundos se ve que están muy lejos de proporcionar valores adecuados. Esto también está relacionado con el período de envío de las tareas al clúster. En un clúster más estable (mayor intervalo de tiempo entre tareas) se puede ampliar este valor.

Analizando más profundamente la gráfica, en el caso en el que no se utiliza ningún dominio se puede ver en la gráfica que la periódica 10 es la que peor tiempo tiene. Esto es debido a que el tiempo de actualización es muy grande y utiliza datos obsoletos en muchas ocasiones. También se puede observar que el proceso que más se acerca a uno, es decir al tiempo óptimo, es el proceso bajo demanda pero por muy poca diferencia, ya que el proceso por eventos está a menos de una centésima. Si se comparan solo los procesos periódicos, la de 3 segundos es la que mejor funciona.

Si se miran los dominios estáticos se puede ver que cualquier tipo de proceso es mejor cuando el tamaño del dominio es 4. Esto tiene sentido ya que cuanto mayor sea el dominio, el nodo 0 tendrá más nodos con los que repartir las tareas. Observando solo los dominios estáticos con tamaño 2, el mejor de ellos es el proceso periódica 1 aunque existe una centésima de diferencia con el proceso bajo demanda y dos centésimas con el proceso por eventos. En el caso del dominio estático 4, el proceso por eventos y el periódica 1 son los que más

se acercan al tiempo óptimo ya que los dos tienen exactamente los mismos valores. Ocurre lo mismo con el proceso bajo demanda y el periódica 3 pero serían los segundos que mejor tiempo tienen.

En el caso de los dominios aleatorios el que mejores tiempos tiene es el dominio con tamaño 7, aunque el tiempo del proceso bajo demanda es 6 milésimas menos que en el dominio con tamaño 5 y el proceso periódica 5 una centésima menos. Pero en general es el que mejores tiempos proporciona. Esto ocurre debido a que los nodos totales son 8 y al utilizar un tamaño de dominio de 7 nodos el reparto de carga es muy parecido. Como se puede ver en la gráfica los valores entre este tipo y sin dominios son bastante parecidos.

El proceso por eventos en el dominio aleatorio con tamaño 3 y tamaño 5 es el que peor tiempo da. Esto es debido a que al ser los nodos escogidos aleatoriamente, puede ocurrir que un nodo cambie a estado receptor y el nodo 0 que no se entere de esta situación, porque no se ha seleccionado para comunicarse con él. Por lo que no ejecutará con él ninguna operación de equilibrio de carga aunque en realidad si esté disponible. Esto ocurre solo en el proceso por eventos porque solo se comunica la información cuando se produce un cambio de estado.

La siguiente gráfica (figura 5.2) representa la eficiencia, es decir cuál de los procesos es más eficiente con el algoritmo de equilibrio de carga.

Ya no se solo se compararán tiempos como en la gráfica anterior, sino que también estarán en función de los envíos de información que realicen los procesos y del número de equilibrios de carga solicitados que siguen adelante.

Al igual que en la gráfica anterior, se caracteriza por los procesos global en función de las opciones de diseño (abscisas) y los valores de la eficiencia (coordenadas). Estos valores están representados de 0 a 0.0005 ya que en este escenario ninguno de ellos supera este último valor.

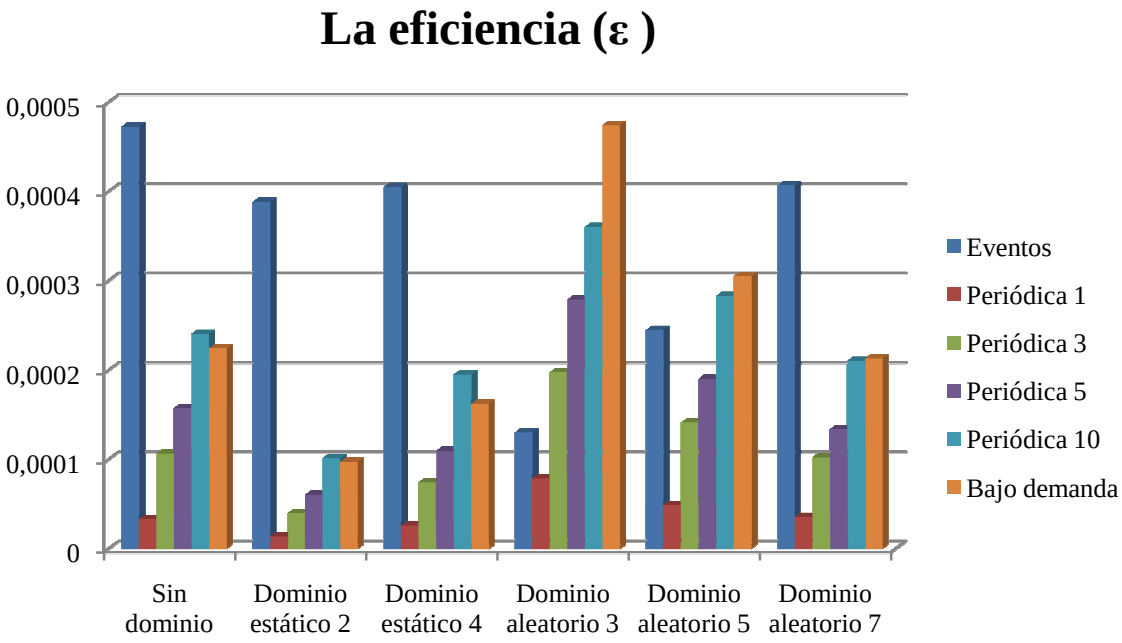


Figura 5.2: Representación de la eficiencia.

El proceso bajo demanda con dominio aleatorio 3 tiene la mejor eficiencia de la gráfica. Aunque el proceso por eventos cuando no existe dominio está por debajo a 1 millonésima.

En general se puede ver que en todas las opciones de diseño es más eficiente el proceso por eventos. En los dos casos en que no lo es, es debido a que al ser aleatorio, el nodo que reparte las tareas no siempre tiene correctamente actualizado el estado de otro nodo, como se ha explicado en la gráfica anterior ya que ocurría lo mismo.

Comparando las eficiencias se puede observar que el proceso periódica 1 no tiene un valor muy bueno. Esto es debido a que no sale rentable realizar tantas comunicaciones. Por ejemplo, si se observa la tabla 5.3, se puede ver que los envíos totales del periódica 1 son 21735, siendo el que le sigue en número de envíos el periódica 1 con 7360 comunicaciones.

Mirando solo los procesos periódicos existe una misma secuencia en todas las opciones de diseño. Esta secuencia coloca como la regla con peor eficiencia a la periódica 1 (como se ha comentado anteriormente), después la de 3

segundos, la de 5 segundos y la que mejor eficiencia tiene es la de 10 segundos. Ocurre al revés que en la otra gráfica ya que la de 10 segundos era la peor y aquí es la mejor (siempre respecto a las reglas periódicas). Esto es debido a que aunque realiza muchas menos comunicaciones y utiliza a veces valores obsoletos, los equilibrios de carga que se realizan son parecidos a los otros tipos de periódica. Con lo que sale más rentable realizar menos comunicaciones si al final se van a obtener resultados parecidos para el equilibrio de la carga.

Comparando los dominios estáticos sigue siendo mejor la situación con un tamaño de dominio de 4 nodos, pero en este caso existe una gran diferencia entre el proceso por eventos y el resto.

Si se miran solo los dominios aleatorios, el más eficiente sería el tamaño del dominio igual a 3, excepto en el proceso por eventos que sería el peor.

En este tipo de escenario se puede sacar la conclusión de que en general es más eficiente el proceso por eventos.

5.3. Resultados del escenario 2

En este caso el escenario se basa en enviar las 160 tareas repartidas entre dos nodos (nodo 0 y nodo 4), es decir 80 a cada uno de ellos, para que estos las repartan.

A continuación se muestran las tablas correspondientes a los experimentos que se han realizado, dependiendo de la opción de diseño que se haya escogido.

La primera tabla (tabla 5.7) corresponde al caso en el que no existen dominios, es decir el nodo 0 reparte entre todos los demás.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1960	121	161	395	432	0,917	3,54E-4
Periodica 1	396	22981	123	208	414	452	0,876	2,93E-05
Periodica 3	396	4648	124	476	419	457	0,867	1,45E-4
Periodica 5	396	4704	124	532	424	462	0,857	1,41E-4
Periodica 10	396	2576	118	1077	463	501	0,79	2,26E-4
Bajo deman	396	5544	123	500	393	430	0,921	1,28E-4

Tabla 5.7: Experimentos sin dominio.

Las dos siguientes tablas (tabla 5.8 y 5.9) son experimentos realizados con dominios estáticos cambiando el tamaño del dominio (2 y 4 nodos). Por lo que el reparto solo se realizara con los nodos que pertenezcan a dicho dominio.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	546	105	14	549	587	0,675	8,11E-4
Periodica 1	396	9340	107	69	588	625	0,634	4,54E-05
Periodica 3	396	2956	104	131	559	596	0,664	1,46E-4
Periodica 5	396	1904	104	279	600	637	0,622	2,12E-4
Periodica 10	396	1230	93	516	774	811	0,488	2,31E-4
Bajo deman	396	2340	106	20	552	590	0,671	1,9E-4

Tabla 5.8: Experimentos con dominio estático 2.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1400	121	97	428	465	0,852	4,6E-4
Periodica 1	396	17080	121	99	432	469	0,844	3,74E-05
Periodica 3	396	5800	119	356	440	477	0,83	1,07E-4
Periodica 5	396	3640	120	436	459	497	0,797	1,64E-4
Periodica 10	396	1880	116	941	475	513	0,772	2,98E-4
Bajo deman	396	4225	121	103	428	465	0,852	1,52E-4

Tabla 5.9: Experimentos con dominio estático 4.

En el caso de los dominios aleatorios se muestran tres tablas (tabla 5.10, 5.11 y 5.12) correspondientes a dominios con tamaño 3, 5 y 7 nodos.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	831	111	2362	527	564	0,702	5,86E-4
Periodica 1	396	10338	118	329	435	472	0,839	5,99E-05
Periodica 3	396	3432	119	596	434	472	0,839	1,82E-4
Periodica 5	396	2040	118	949	429	467	0,848	3,07E-4
Periodica 10	396	1149	112	1362	484	521	0,76	4,63E-4
Bajo deman	396	2418	123	109	406	444	0,892	2,84E-4

Tabla 5.10: Experimentos con dominios aleatorios 3.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1325	117	1451	476	513	0,772	4,26E-4
Periodica 1	396	16880	122	211	426	464	0,853	3,86E-05
Periodica 3	396	5720	122	440	435	471	0,841	1,12E-4
Periodica 5	396	3440	121	640	435	472	0,839	1,84E-4
Periodica 10	396	1840	116	1153	465	525	0,754	2,97E-4
Bajo deman	396	4050	123	108	415	452	0,876	1,66E-4

Tabla 5.11: Experimentos con dominios aleatorios 5.

	top	env tot	env acep	env no acep	trep	tejec	Cociente top/tejec	ε
Eventos	396	1946	123	137	396	433	0,915	3,61E-4
Periodica 1	396	23198	127	177	418	456	0,868	2,97E-05
Periodica 3	396	7392	124	375	402	439	0,902	9,46E-05
Periodica 5	396	5152	123	595	465	503	0,787	1,18E-4
Periodica 10	396	2632	116	1054	475	512	0,773	2,13E-4
Bajo deman	396	5523	124	155	396	433	0,915	1,28E-4

Tabla 5.12. Experimentos con dominios aleatorios 7.

Utilizando los datos anteriores de las tablas se ha realizado la siguiente gráfica, que corresponde al cociente entre el tiempo óptimo de cada proceso entre el tiempo de ejecución que tarda el programa en realizar las 160 tareas con el algoritmo de equilibrio de carga.

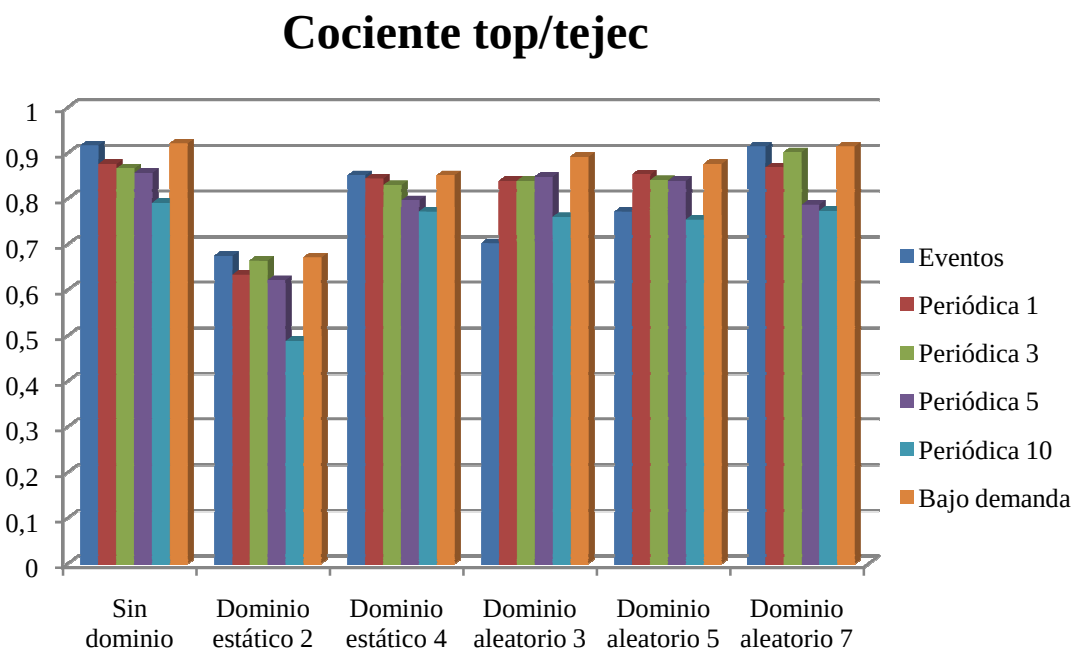


Figura 5.3: Representación del Cociente top/tejec.

En el eje de abscisas se representan las distintas opciones de diseño con las que se han realizado las pruebas, siendo el eje de ordenadas los valores correspondientes al cociente dicho anteriormente. Estos valores van de 0 a 1 ya que nunca va a ser mayor el tiempo de ejecución con el algoritmo de equilibrio que el tiempo óptimo. Por lo que está representado cada proceso global en función de la opción de diseño en la que se encuentre y el valor que tiene del cociente.

Analizando la gráfica se puede observar que el dominio estático con tamaño 2 es el que peor tiempo tiene, aunque si se compara con la misma gráfica del apartado 5.2 se ve que ha habido una mejora ya que el cociente ha aumentado.

Si se sigue comparando con la del apartado 5.2 se observa que el dominio estático con tamaño 4 ha mejora bastante ya que no existe tanta diferencia con el resto. Y que el proceso por eventos también mejora en los dominios aleatorios 3 y 5, ya que al ser dos nodos los que reparten las tareas, cada uno las reparte con un dominio distinto, por lo que en total se dispone del doble de

nodos con los que repartir dichas tareas. Por lo que casi no ocurre lo explicado en el apartado anterior con los procesos por eventos en estos dominios.

Mirando la gráfica a nivel general se ve que el proceso bajo demanda en la opción sin dominio es la que mejor resultado tiene, y que la periódica 10 es la peor de todas las periódicas en todas las opciones de diseño.

Cuando se comparan solo los dominios aleatorios se ve que el de tamaño 7 es el mejor en general aunque el proceso periódica 5 es mejor en el de tamaño 3.

También se ha realizado una gráfica con la eficiencia para ver qué proceso es más eficiente en cada diseño y en general. Esta gráfica está representada en su eje de abscisas por las opciones de diseño y en su eje de ordenadas por los valores de dicha eficiencia. Cada tipo de proceso es representado por un color en cada opción de diseño.

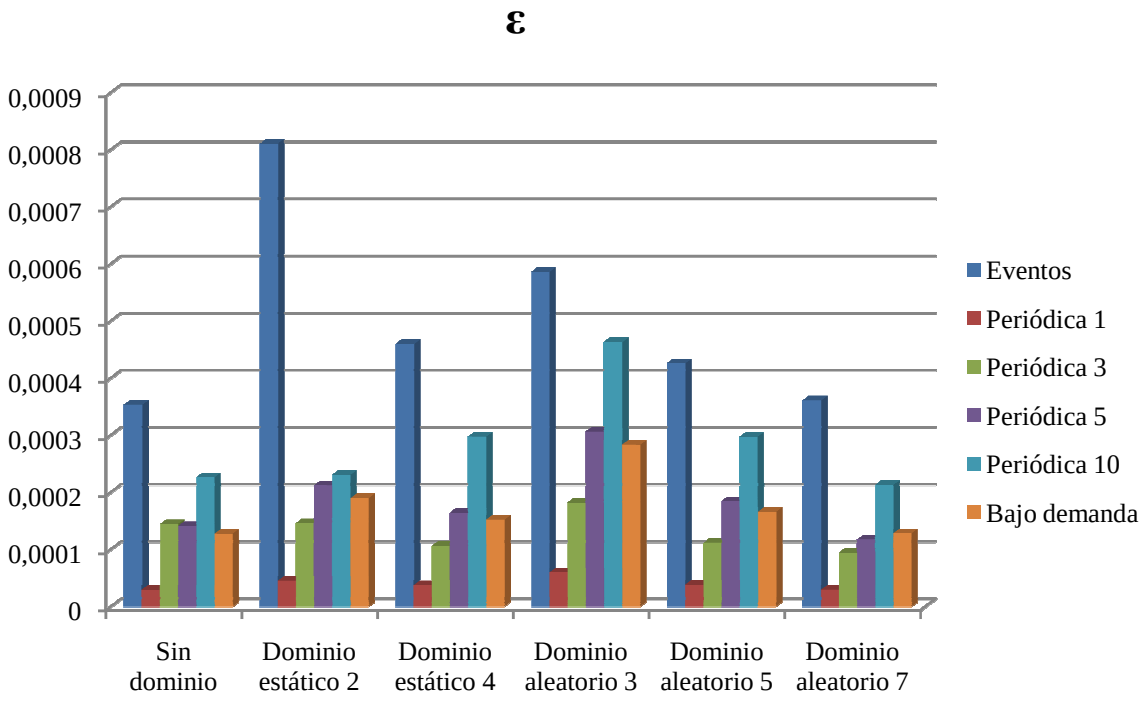


Figura 5.4: Representación de la eficiencia.

Analizando la gráfica (figura 5.4) se observa que el proceso por eventos es la más eficiente con diferencia en cualquiera de las opciones de diseño. Esto es debido a que es un término medio de los otros dos procesos ya que el bajo demanda no realiza muchas comunicaciones, pero al hacer la operación de equilibrio tarda más tiempo que el resto. Por otro lado, la regla periódica no tiene este último problema, pero si se pone un tiempo muy pequeño de periodicidad se realizan muchas comunicaciones innecesarias y si se utiliza un tiempo más grande no habría tantas comunicaciones, pero en muchas ocasiones se trabajaría con datos obsoletos.

Lo segundo que se observa es que el proceso periódica con 10 segundos es el segundo más eficiente. Sucede esto ya que no se realizan muchas comunicaciones y no existe mucha diferencia del número de equilibrios que se realizan.

Por último el proceso que peor eficiencia tiene en todos los diseños es el proceso periódica 1 porque al hacer tantas comunicaciones al final realiza equilibrios de carga parecidos a los demás.

La conclusión más contundente que se saca en este escenario al igual que en el escenario 1, es que el proceso por eventos es el más eficiente de todos. Si se observan las gráficas con las del escenario 1 se puede ver que los tiempos y la eficiencia mejoran por lo que otra conclusión sería que al repartir las tareas entre más de un proceso el equilibrio de carga es más eficiente.

Capítulo 6

Conclusiones y Líneas futuras

6.1. Conclusiones

En este capítulo se comentarán las distintas conclusiones que se han podido extraer de la realización del proyecto, tanto a nivel personal como técnico.

A nivel personal se ha aprendido el manejo de un nuevo modelo de programación paralela que no se conocía ni habían hablado de él en la titulación. También se ha tenido la posibilidad de trabajar con un sistema de computación de altas prestaciones, como es el clúster calderón, del grupo ATC, de la Universidad de Cantabria. Al utilizar este clúster se ha trabajado en remoto en una máquina que ha sido completamente ajena. Lo cual ha servido para aprender más sobre el manejo de una máquina a través únicamente de comandos. Debido a esto último se tuvo unos problemas iniciales al no haber trabajado nunca en remoto y especialmente debidos a la compleja configuración de la seguridad del sistema de ficheros de red.

Todo ello ha aportado más soltura en cuestiones relativas a la programación y al manejo de grandes sistemas.

Por otro lado, desde un punto de vista técnico es importante destacar que con este proyecto se ha cumplido con todos los objetivos propuestos en el primer capítulo de esta memoria. Es decir, se ha hecho el estudio del modelo de programación de paso de mensajes, concretado en el estándar MPI. Se ha puesto especial énfasis en las comunicaciones, si bien otros aspectos como la

sincronización de procesos y los comunicadores han tenido que ser estudiados y empleados a lo largo de la realización del proyecto. Con la finalización de este proyecto se cree haber alcanzado un dominio suficiente de la programación de entornos, lo que puede permitir en un futuro inmediato abordar otros proyectos similares.

También se ha estudiado como funciona un algoritmo de equilibrio de carga de trabajo dinámico en clusters. Ha sido necesario entender cada una de las reglas o políticas que lo componen, si bien para este estudio se partía de un proyecto de fin de carrera anterior. Al principio se tuvo problemas al poner en marcha el algoritmo de dicho proyecto. Uno de ellos estuvo relacionado con los nodos, debido a que los que se utilizaron ya no estaban disponibles. Los otros problemas fueron en relación al código. Con ello se ha aprendido el funcionamiento de un clúster en relación con la administración de la carga de trabajo.

La complejidad de este tipo de algoritmos está fundamentalmente en la arquitectura de comunicaciones que tienen que gestionar. Existen distintos tipos de procesos y cada uno de ellos tiene unas necesidades de información, que debe estar actualizada para poder tomar las decisiones adecuadas. Estas comunicaciones tienen un impacto muy fuerte en el rendimiento y sobre todo en la escalabilidad del algoritmo, por lo que en este proyecto se ha planteado un estudio exhaustivo para analizar cuál de las posibles soluciones son las más adecuadas.

También se han estudiado las distintas opciones de diseño para esta última regla con lo que se ha aprendido como pueden cambiar los resultados de este algoritmo de carga al tener diferentes diseños. En cuanto a estos diseños hubo dificultades al implementar los algoritmos sobre todo en relación con los problemas de la comunicación. Ya que había que encontrar una forma de comunicarse sólo entre nodos de un mismo dominio. Todos los problemas fueron satisfactoriamente resueltos y finalmente todas las técnicas propuestas se implementaron y validaron.

Finalmente se presentan una serie de conclusiones generales extraídas de los resultados obtenidos en la experimentación. Estas conclusiones se obtienen de

los experimentos basándose en como varía el funcionamiento del algoritmo según las distintas políticas de la regla de intercambio de información. Si se analiza cada escenario por separado existe una conclusión general, que indica que la regla basada en **eventos** es la más eficiente respecto a los otros 5 tipos de políticas en los dos escenarios. Por otro lado, si se comparan los dos escenarios para ver en cuál de ellos este proceso por eventos es más eficiente, sería el escenario 2, es decir, el que reparte las tareas mediante dos nodos. Esto es lógico ya que al ser un nodo más, las tareas se reparten entre más nodos.

Al analizar sólo la relación entre el tiempo óptimo y de ejecución, no existe un proceso que destaque en todas las opciones de diseño. Pero sí se puede decir que el proceso **bajo demanda** es el que mejor tiempo de ejecución tiene en todos o casi todos los diseños, sin distinguir entre escenarios. Esto es debido a que este proceso siempre que necesita realizar un equilibrio lo hace con datos actualizados, ya que cuando necesita realizarlo es cuando pide los estados.

6.2. Líneas futuras

Una vez sacadas las conclusiones de este proyecto surgen otras líneas de estudio. Es decir, se puede seguir completando el estudio de la regla de intercambio de información y del algoritmo de equilibrio de carga en general.

Una de ellas es la realización de un estudio más profundo con las opciones de diseño, es decir implementar otros tipos de dominios o en los dominios estáticos estudiar cómo funcionaría si el solapamiento no es siempre la mitad del tamaño del dominio. También se podría aumentar el número de nodos para ver el impacto de la regla en la escalabilidad del algoritmo.

Otro caso que se podría estudiar más profundamente sería la periodicidad del proceso periódica, es decir que ocurriría si cuando este tiempo es constante se aumenta dicho tiempo de comunicación y si al ser muy variable se disminuye.

Si se habla de líneas futuras en general, se podría realizar el estudio más profundamente de cada regla del algoritmo de equilibrio de carga, como se ha hecho en este proyecto con la regla de intercambio de información.

Capítulo 7

Bibliografía

[1] Francisco Almeida… [et al]. *Introducción a la programación paralela.* Paraninfo, Madrid. 2008.

[2] Gregory F.Pfister. *In search of clusters.* Prentice Hall PTR, 1998.

[3] Flavio Mauricio Gallardo Padilla. *Diseño de una solución para servidores de alta disponibilidad y balanceo de carga con open source.* Nelio Brito (dir.). Proyecto de Grado. Universidad Alfredo Pérez Guerrero. 2011.

[4] José Miguel Alonso. *Programación de aplicaciones paralelas con MPI (Message Passing Interface).* Facultad de Informática UPV/EHU. 1997.

[5] J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill. 2004.

[6] MPI Forum. *MPI: A Message-Passing Interface Standard*. http://www.mpi-forum.org/. 2009.

[7] Francisco Hidrobo, Hebert Hoeger. *Introducción a MPI (Message Passing Interface).* http://webdelprofesor.ula.ve/ingenieria/hhoeger/Introduccion_MPI.pdf.

[8] C. Xu, F. Lau. *Load balancing in parallel computers: Theory and Practice*. Kluwer Academic Publishers, Boston. 1997.

[9] Marta Beltrán Pardo. *Equilibrio de Carga en Clusters Heterogéneos.* Tesis Doctoral. Universidad rey Juan Carlos. 2005.

[10] Pablo Dosal Viñas. *Algoritmo de Equilibrio de Carga de Trabajo para Clústers Heterogéneos.* Proyecto de Fin de Carrera. Universidad de Cantabria. 2010.