ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Programación Matlab En Paralelo Sobre Clúster Computacional: Evaluación De Prestaciones

AUTOR: **Juan Francisco Rodríguez Pérez**
DIRECTOR(ES): **José Luis Sancho Gómez**
Rafael Verdú Monedero

Febrero/2010

Autor	JUAN FRANCISCO RODRÍGUEZ PÉREZ
E-mail del Autor	jfkkirchhoff@gmail.com
Director(es)	José Luis Sancho Gómez, Rafael Verdú Monedero
E-mail del Director	JoseL.Sancho@upct.es, rafael.verdu@upct.es
Codirector(es)	[Nombre] [Apellidos] , [Nombre] [Apellidos] , ...
Título del PFC	Programación Matlab En Paralelo Sobre Clúster Computacional: Evaluación De Prestaciones
Descriptores	Programación en paralelo, Matlab, Clúster, Procesamiento de imágenes, Operadores Morfológicos.

Resumen

Los objetivos de este proyecto fin de carrera son los de paralelizar una serie de programas para poder ser ejecutados en un Clúster con la ayuda de las funciones que nos proporciona Parallel Computing Toolbox de Matlab.

En este proyecto fin de carrera también se da una visión de cómo programar en paralelo con Matlab, para ello se han dispuesto una serie de ejemplos con los que es más fácil de comprender este tipo de programación.

Este tipo de programación paralela está orientada para que dichos programas sean ejecutados en un Clúster de ordenadores, por lo que también se profundiza en qué es y cómo funciona un Clúster.

Titulación	Ing. de Telecomunicación
Intensificación	En sistemas y redes de telecomunicación
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Febrero-2010

ÍNDICE DE CONTENIDOS

ÍNDICE_DE TABLAS Y FIGURAS

CAPÍTULO 1

INTRODUCCIÓN

1.1. Introducción

La programación secuencial es aquella en la que las instrucciones se ejecutan una detrás de otra a modo de secuencia, es decir, una instrucción no se ejecuta hasta que finaliza la anterior, sin embargo, la computación paralela es una técnica de programación en la que muchas instrucciones se ejecutan simultáneamente. Se basa en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de forma concurrente ("en paralelo").

Existen varios tipos de computación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas. Durante muchos años, la computación paralela se ha aplicado en la computación de altas prestaciones, pero el interés en ella ha aumentado en los últimos años debido a las restricciones físicas que se plantean.

La computación paralela se ha convertido en el paradigma dominante en la arquitectura de computadores. Sin embargo, recientemente, el consumo de energía de los ordenadores paralelos se ha convertido en una preocupación.

Los ordenadores paralelos se pueden clasificar según el nivel de paralelismo que admite su hardware: los ordenadores multinúcleo y multiproceso tienen varios

elementos de procesamiento en una sola máquina, mientras que los clúster, los MPP[1] y los grids[2] emplean varios ordenadores para trabajar en la misma tarea.

Los programas de ordenador paralelos son más difíciles de escribir que los secuenciales porque la concurrencia introduce nuevos tipos de errores de software. La comunicación y la sincronización entre las diferentes subtareas son típicamente las grandes barreras para conseguir un buen rendimiento de los programas paralelos.

El software se ha orientado tradicionalmente hacia la computación en serie. Para resolver un problema, se construye un algoritmo y se implementa en un flujo de instrucciones en serie. Estas instrucciones se ejecutan en la unidad central de procesamiento de un ordenador. En el momento en el que una instrucción se termina, se ejecuta la siguiente.

La computación paralela emplea elementos de procesamiento múltiple simultáneamente para resolver un problema. Esto se logra dividiendo el problema en partes independientes de tal manera que cada elemento de procesamiento pueda ejecutar su parte del algoritmo a la misma vez que los demás. Así se consigue más rapidez a la hora de realizar cualquier tipo de operaciones, lo cual desencadena una gran serie de ventajas, en el mundo de la tecnología.

1.2. Objetivos

Los objetivos de este proyecto fin de carrera son los de paralelizar una serie de programas ejecutándolos en un clúster, el clúster-GTTS, con la ayuda de las funciones que nos proporciona “Parallel Computing Toolbox” de Matlab.

En este proyecto también se pretende dar conocimiento de cómo paralelizar programas secuenciales gracias a esta “Toolbox” que nos ofrece Matlab, por lo que se muestran una serie de pequeños ejemplos ilustrativos, donde se podrá aprender las diferentes maneras de paralelizar programas y comprobar las ventajas de utilizar este tipo de programación a la hora de trabajar.

A efectos prácticos se paralelizarán cuatro funciones, cuya misión es el procesado de imágenes, donde se podrá observar con detalle las ventajas de la programación en paralelo con Matlab, utilizando como máquina de cómputo un clúster. Estas funciones son muy significativas a la hora de ver la comparación entre programación secuencial y programación paralela en Matlab, ya que el procesado de imágenes requiere un elevado número de operaciones de cálculo, cuyo tiempo de procesado se verá considerablemente reducido al transformar los programas secuenciales en programas paralelos.

---

[1] MPP: Proceso Masivamente Paralelo. Un sistema que utiliza más de 16 procesadores en una arquitectura de memoria distribuida, escalable y paralela, que puede escalar hasta más de 64 procesadores.

[2] El término *grid* se refiere a una infraestructura que permite la integración y el uso colectivo de ordenadores de alto rendimiento, redes y bases de datos que son propiedad y están administrados por diferentes instituciones.

En el presente trabajo se describirá de forma completa las partes de las que está compuesto el clúster, así como sus características. También se mostrará cómo trabajar con él, cuando se quiera procesar un programa en paralelo dado.

1.3. Introducción a los clústers

El origen del término clúster y del uso de este tipo de tecnología era desconocido hasta que comenzó a conocerse a finales de los años 50 y principios de los años 60.

En consecuencia, la historia de los primeros grupos de computadoras está más o menos directamente ligada a la historia de los principios de las redes, ya que una de las principales motivaciones para el desarrollo de una red es la de enlazar los recursos de computación de diferentes equipos, de ahí la creación de un clúster de computadoras. Las redes de conmutación de paquetes fueron conceptualmente inventados por la corporación RAND en 1962.

Utilizando el concepto de una red de conmutación de paquetes, el proyecto ARPANET logró crear en 1969 lo que fue posiblemente la primera red de computadoras basadas en el clúster de computadoras por cuatro tipos de centros informáticos (cada una de las cuales fue algo similar a un "clúster").

El proyecto ARPANET creció y se convirtió en lo que es ahora Internet, que se puede considerar como "la madre de todos los clúster" (como la unión de casi todos los recursos de cómputo, incluidos los clúster, que pasarían a ser conectados).

El primer producto comercial de tipo clúster fue ARCnet, desarrollada en 1977 por Datapoint pero no obtuvo un éxito comercial y los clústers no consiguieron tener éxito hasta que en 1984 VAXcluster produjeran el sistema operativo VAX/VMS. El ARCnet y VAXcluster no sólo son productos que apoyan la computación paralela, sino que también comparten los sistemas de archivos y dispositivos periféricos.

La idea era proporcionar las ventajas del procesamiento paralelo, al tiempo que se mantiene la fiabilidad de los datos y el carácter singular. VAXcluster y VMScluster están todavía disponibles en los sistemas de HP OpenVMS corriendo en sistemas Itanium y Alpha.

Otros dos principios comerciales de clústers notables fueron el Tandem Himalaya (alrededor de 1994 con productos de alta disponibilidad) y el IBM S/390 Parallel Sysplex (también alrededor de 1994, principalmente para el uso de la empresa).

La historia de los clústers de computadoras estaría incompleta sin señalar el papel fundamental desempeñado por el desarrollo del software de Parallel Virtual Machine (PVM). Este software de fuente abierta basado en comunicaciones TCP/IP permitió la creación de un superordenador virtual, un clúster HPC[3], realizada desde cualquiera de los sistemas conectados TCP/IP.

[3] HCP (High Performance Computing): Un clúster HPC es un clúster técnico de computación de alto rendimiento con cargas de trabajo distribuidas entre muchos "nodos computacionales". El clúster funciona como una entidad o sistema individual en tareas específicas, realizando operaciones de cálculo

De forma libre, los clústers heterogéneos han constituido la cima de este modelo logrando aumentar rápidamente en FLOPS[4] globalmente y superando con creces la disponibilidad incluso de los más caros superordenadores.

PVM y el empleo de PC y redes de bajo costo llevó, en 1993, a un proyecto de la NASA para construir supercomputadoras de clústers.

En 1995, la invención de "beowulf", un estilo de clúster, una granja de computación diseñada en base a un producto básico de la red con el objetivo específico de "ser un superordenador" capaz de realizar firmemente cálculos paralelos HPC.

Esto estimuló el desarrollo independiente de la computación Grid como una entidad, a pesar de que el estilo Grid giraba en torno al del sistema operativo Unix y el Arpanet.

1.3.1. Características del clúster de servidores

En este apartado se van a dar las características generales que puede poseer un clúster:

- *Instalación y configuración fácil.-* El servicio de clúster es una parte del conjunto del sistema operativo, el cual ya ha dejado de ser un componente opcional. Esto permite que un nodo de un clúster de servidores se configure sin medio de distribución y permite que un clúster de servidores se cree, o que se modifique la configuración, utilizando las herramientas de administración de un clúster desde una estación de administración remota. No se requieren reinicios para instalar una configuración de clúster de servidores.
  La acción de quitar un nodo de un clúster de servidores consiste en algo tan simple como su expulsión del clúster. Cualquier dato de configuración de clúster asociado con el nodo se suprime de forma automática, y no se requiere ningún reinicio.
  Cuando se configura un nodo del clúster de servidores, el proceso de configuración, valida la configuración de hardware y software para asegurar que cualquier incompatibilidad desconocida se detecta antes de finalizar la configuración del servicio del clúster. Muchas de las opciones de configuración son valores predeterminados proporcionados para facilitar y acelerar la configuración de un clúster de servidores que siga las prácticas recomendadas. Tras esta configuración, un clúster de servidor en funcionamiento puede personalizarse mediante las herramientas de administración del clúster de servidores.
  La infraestructura de configuración del clúster es una interfaz abierta disponible para otros fabricantes de software. Esto permite que las aplicaciones configuren sin problemas los recursos del clúster de servidores y modifiquen su

---

intensivo sobre grandes volúmenes de datos. El clúster es visto como un solo sistema por los sistemas ubicados en su exterior. Toda comunicación con las instancias ubicadas en el exterior del clúster se realiza únicamente a través de "nodos cabecera" dedicados. Todos los nodos de computación del clúster presentan la misma configuración.

[4] FLOPS: es el acrónimo de *Floating point Operations Per Second* (operaciones de punto flotante por segundo). Se usa como una medida del rendimiento de una computadora, especialmente en cálculos científicos que requieren un gran uso de operaciones de coma flotante.

configuración durante una instalación del clúster. La configuración de un clúster de servidores permite la creación de secuencias de comandos y está disponible a través de herramientas de línea de comandos y a través de la GUI del administrador de dicho clúster.

- *Sistema operativo*.- existen multitud de sistemas operativos posibles para el funcionamiento de un clúster, como son: GNU/Linux, Unix, Windows, Mac OS X, Solaris, etc. El sistema operativo utilizado por el clúster-GTTS, empleado en este proyecto fin de carrera, es GNU/Linux.

- *Compatibilidad con clústers de mayor tamaño*.- El aumento del número de nodos de un clúster de servidores permite al administrador tener muchas más opciones para implementar aplicaciones y proporcionar directivas de conmutación por error que cubran los riesgos y las expectativas de las empresas. Los clústers de gran tamaño proporcionan una mayor flexibilidad en la creación de clústers en múltiples sitios, dispersos geográficamente, proporcionados para la tolerancia a desastres y para evitar los tradicionales errores de nodos o aplicaciones.

- *Integrado con el servicio de Active Directory*.- La integración asegura que un objeto de equipo "virtual" se registra en Active Directory. Esto permite que las aplicaciones utilicen la delegación y la autenticación Kerberos[5] para obtener servicios de alta disponibilidad que se ejecuten en un clúster. El objeto de equipo proporciona también una ubicación predeterminada a los servicios compatibles con Active Directory para publicar puntos de control de servicio.

- *Mejoras en la red*.-Los clústers de servidores aprovechan una serie de mejoras importantes en la red. La lógica avanzada para la conmutación por error ahora es compatible cuando se ha producido una pérdida total de la comunicación interna; y el estado de la red para la comunicación pública de todos los nodos ahora se toma en cuenta antes de que se tome la decisión de la propiedad por quórum.
  La detección de medios proporciona una mejor protección a la conmutación por error. Como la detección de medios está deshabilitada de forma predeterminada, se preserva la función de red y todos los recursos dependientes de direcciones IP, permanecen con conexión. Si falla la comunicación de multidifusión por cualquier motivo, las comunicaciones internas vuelven a la difusión única. En cualquier suceso, todas las comunicaciones internas son firmadas y seguras.

- *Mejoras de capacidades de almacenamiento*.-Los clústers de servidores aprovechan las eficaces capacidades de almacenamiento. Los puntos de montaje ahora son compatibles con los discos compartidos y funcionan en la conmutación por error, proporcionando un espacio de nombres del sistema de archivos flexible. La caché de cliente (CSC), también conocida como "archivos sin conexión", ahora es compatible con los recursos compartidos de archivos en clúster y permite que un equipo cliente guarde en la caché datos almacenados en un recurso compartido de clúster.

[5] Kerberos es un protocolo de autenticación de redes de ordenador que permite a dos computadores en una red insegura demostrar su identida mutuamente de manera segura.

El Sistema de archivos distribuido (DFS) mejorado ahora incluye: múltiples raíces autónomas, conmutación por error independiente, compatibilidad con las configuraciones activas y permite compartir múltiples archivos en distintos equipos que deben agregarse a un espacio de nombres común.
Los servicios de clúster se han optimizado para las redes de área de almacenamiento (SAN), incluyendo restablecimientos de dispositivos seleccionados y requisitos de interconexión de almacenamiento.
Los discos compartidos ahora pueden ubicarse en la misma interconexión de almacenamiento, como los discos de archivo de volcado, de archivo de paginación y de arranque. Esto permite que un clúster de servidores tenga una única interconexión de almacenamiento. NOTA: esto únicamente está disponible si los fabricantes han configurado y habilitado estas configuraciones.

- *Recuperación de errores y solución de problemas más fácil*.- Se han realizado un importante número de mejoras en los archivos de registro del clúster de servidores para permitir una solución de problemas y una depuración más fácil. Entre estas mejoras cabe incluir: registros del clúster; registros de configuración; niveles de error; marca de tiempo de servidor local; GUID (identificador único global) en la asignación de nombres de recursos y el registro de sucesos.
  Existe una nueva herramienta de diagnóstico disponible en el Kit de recursos (ClusDiag) que permite que los registros de clústers y de sucesos de todos los nodos del clúster se correlacionen y comparen. En el caso de un error de disco, el Kit de recursos contiene una nueva herramienta (ClusterRecovery) que permite que los recursos del disco se reconstruyan y que el estado del clúster vuelva a crearse.

1.3.2. Características del equilibrio de carga en la red

Para una mayor rapidez y eficiencia se necesita un equilibrio de carga en la red, para optimizar al máximo la velocidad con la que se transmiten los datos.

- *Administrador de equilibrio de carga en la red*.- Esta utilidad proporciona un único punto de configuración y administración para los clústers de NLB. El Administrador de NLB[6] puede utilizarse para:
  - Crear nuevos clústers de NLB y propagar automáticamente los parámetros de clúster y las reglas de puerto a todos los hosts del clúster. También propaga los parámetros del host a hosts específicos de un clúster.
  - Agregar y quitar hosts, a o desde clústers de NLB.
  - Agregar automáticamente direcciones IP de clúster de servidores a TCP/IP.
  - Administrar los clústers existentes conectando con ellos o cargando su información de host a un archivo y guardando esta información para un uso posterior.
  - Configurar el NLB para equilibrar la carga de múltiples sitios Web o aplicaciones en el mismo clúster de NLB. Esto incluye agregar todas las direcciones IP de clúster a TCP/IP y controlar el tráfico enviado a aplicaciones específicas en hosts específicos del clúster.
  - Diagnóstico de clústers configurados indebidamente.

[6] NLB: Network Load Balancing ofrece una solución de alta disponibilidad para aplicaciones de servidor basadas en TCP/IP, capaz de ofrecer escalabilidad y alto rendimiento

- *Clústeres virtuales.*- Esta característica de puede utilizarse para:
  - Configurar reglas de puerto distintas para direcciones IP de clúster distintas, donde cada dirección IP de clúster corresponde a una aplicación o un sitio Web alojados en el clúster de NLB.
  - Filtrar el tráfico enviado a una aplicación o un sitio Web específico en un host específico del clúster.
  - Elegir el host del clúster que debería utilizarse para prestar servicio al tráfico enviado a una aplicación o un sitio Web específico alojado en el clúster.

- *Afinidad bidireccional.*- La utilización más habitual de la afinidad bidireccional es poner en el clúster, los servidores ISA (Internet Security and Acceleration) para el equilibrio de carga en servidores de seguridad y proxy. Habitualmente el NLB se utiliza conjuntamente con ISA para la publicación de Web y de servidor. Aunque la publicación de Web no requiere una afinidad bidireccional, la publicación de servidor sí. La afinidad bidireccional crea múltiples instancias de NLB en el mismo host, que trabajan en parejas para asegurar que las respuestas de los servidores publicados se enrutan a través de los servidores ISA adecuados en un clúster.

1.3.3. Clasificación de los Clústers

El término clúster tiene diferentes connotaciones para diferentes grupos de personas. Los tipos de clústers, establecidos en base al uso que se dé a los clústers y los servicios que ofrecen, determinan el significado del término para el grupo que lo utiliza.

Los clústers pueden clasificarse en base a sus características. Se pueden tener clústers de alto rendimiento (HPC – High Performance Clústers), clústers de alta disponibilidad (HA – High Availability) o clústers de alta eficiencia (HT – High Throughput).

- *Alto rendimiento:* Son clústers en los cuales se ejecutan tareas que requieren de gran capacidad computacional, grandes cantidades de memoria, o ambos a la vez. El llevar a cabo estas tareas puede comprometer los recursos del clúster por largos periodos de tiempo.
- *Alta disponibilidad:* Son clústers cuyo objetivo de diseño es el de proveer disponibilidad y confiabilidad. Estos clústers tratan de brindar la máxima disponibilidad de los servicios que ofrecen. La confiabilidad se provee mediante software que detecta fallos y permite recuperarse frente a los mismos, mientras que en hardware se evita tener un único punto de fallos.
- *Alta eficiencia:* Son clústers cuyo objetivo de diseño es el ejecutar la mayor cantidad de tareas en el menor tiempo posible. Existe independencia de datos entre las tareas individuales. El retardo entre los nodos del clúster no es considerado un gran problema.

Los clústers pueden también clasificarse como Clústers de IT Comerciales (Alta disponibilidad, Alta eficiencia) y Clústers Científicos (Alto rendimiento). A pesar de las discrepancias a nivel de requerimientos de las aplicaciones, muchas de las características de las arquitecturas de hardware y software, que están por debajo de las

aplicaciones en todos estos clústers, son las mismas. Más aún, un clúster de determinado tipo, puede también presentar características de los otros.

1.3.4. Componentes de un Clúster

En general, un clúster necesita de varios componentes de software y hardware para poder funcionar:

- *Nodos.*- Pueden ser simples ordenadores, sistemas multiprocesador o estaciones de trabajo (*workstations*). En informática, de forma muy general, un nodo es un punto de intersección o unión de varios elementos que confluyen en el mismo lugar. Ahora bien, dentro de la informática la palabra nodo puede referirse a conceptos diferentes según en el ámbito en el que nos movamos.
  En redes de computadoras cada una de las máquinas es un nodo, y si la red es Internet, cada servidor constituye también un nodo. En estructuras de datos dinámicas un nodo es un registro que contiene un dato de interés y al menos un puntero para referenciar (apuntar) a otro nodo. Si la estructura tiene sólo un puntero, la única estructura que se puede construir con él es una lista, si el nodo tiene más de un puntero ya se pueden construir estructuras más complejas como árboles o grafos.
  El clúster puede estar conformado por nodos dedicados o por nodos no dedicados. En un clúster con nodos dedicados, los nodos no disponen de teclado, mouse ni monitor y su uso está exclusivamente dedicado a realizar tareas relacionadas con el clúster. Mientras que, en un clúster con nodos no dedicados, los nodos disponen de teclado, mouse y monitor y su uso no está exclusivamente dedicado a realizar tareas relacionadas con el clúster, el clúster hace uso de los ciclos de reloj que el usuario del computador no está utilizando para realizar sus tareas.
  Cabe aclarar que a la hora de diseñar un clúster, los nodos deben tener características similares, es decir, deben guardar cierta similaridad de arquitectura y sistemas operativos, ya que si se conforma un clúster con nodos totalmente heterogéneos (existe una diferencia grande entre capacidad de procesadores, memoria, HD) será ineficiente debido a que el middleware delegará o asignará todos los procesos al nodo de mayor capacidad de cómputo y solo distribuirá cuando este se encuentre saturado de procesos; por eso es recomendable construir un grupo de ordenadores los más similares posible.

- *Almacenamiento.*- El almacenamiento puede consistir en una NAS, una SAN, o almacenamiento interno en el servidor. El protocolo más comúnmente utilizado es NFS (Network File System), sistema de ficheros compartido entre servidor y los nodos. Sin embargo existen sistemas de ficheros específicos para clúster como Lustre (CFS) y PVFS2. Tecnologías en el soporte del almacenamiento en discos duros:

  - IDE (PATA, Parallel ATA): Anchos de banda (Bw) de 33, 66, 100 y 133MBps.
  - SATA I (SATA-150): Bw 150 MBps.
  - SATA II (SATA-300): Bw 300 MBps.
  - SCSI: Bw 160, 320, 640MBps. Proporciona altos rendimientos.

- SAS (Serial Array SCSI): Aúna SATA II y SCSI. Bw 375MBps.
- Las unidades de cinta (DLTs) son utilizadas para backups por su bajo coste.

NAS (Network Attached Storage) es un dispositivo específico dedicado al almacenamiento a través de red (normalmente TCP/IP) que hace uso de un S.O. optimizado para dar acceso a través de protocolos CIFS, NFS, FTP o TFTP.

Por su parte, DAS (Direct Attached Storage) consiste en conectar unidades externas de almacenamiento SCSI o a una SAN (Storage Area Network) a través de fibra óptica. Estas conexiones son dedicadas.

Mientras NAS permite compartir el almacenamiento, utilizar la red, y tiene una gestión más sencilla, DAS proporciona mayor rendimiento y mayor fiabilidad al no compartir el recurso.

NAS vs. SAN:

- Cables: NAS usa Ethernet. SAN usa fibra óptica.
- Protocolo: NAS usa CIFS, NFS, ó HTTP sobre TCP/IP. SAN usa Encapsulated SCSI (iSCSI, cuando encapsula sobre TCP/IP, y FCP cuando encapsula directamente sobre fibra óptica).
- Manejo: en NAS, el NAS head, gestiona el sistema de ficheros. En SAN múltiples servidores manejan los datos.
- NAS nos permite acceder a un sistema de ficheros a través de TCP/IP usando CIFS (en el caso de Windows) ó NFS (en el caso de Unix/Linux.
- NAS es una solución más adecuada para:
  - Servidor de ficheros.
  - Almacenamiento de directorios de usuario.
  - Almacenamiento de archivos en general.

- Por su parte, una SAN es usada para acceder al almacenamiento en modo BLOQUE, a través de una red de fibra óptica con fibra óptica utilizando el protocolo SCSI.
- SAN es una solución más adecuada para:
  - Bases de datos.
  - Data warehouse.
  - Backup (al no interferir en la red del sistema).
  - Cualquier aplicación que requiera baja latencia y alto ancho de banda en el almacenamiento y recuperación de datos.

- *Sistema Operativo*.- Debe ser multiproceso, multiusuario. Otras características deseables son la facilidad de uso y acceso y permitir además múltiples procesos y usuarios. Un sistema operativo es un programa o conjunto de programas de computadora destinado a permitir una gestión eficaz de sus recursos. Comienza a trabajar cuando se enciende el computador, y gestiona el hardware de la máquina desde los niveles más básicos, permitiendo también la interacción con el usuario. Un sistema operativo se puede encontrar normalmente en la mayoría de los aparatos electrónicos que utilicen microprocesadores para funcionar, ya que gracias a estos podemos entender la máquina y que ésta cumpla con sus funciones (teléfonos móviles, reproductores de DVD, autoradios... y computadoras).

- *Conexiones de Red*.- Los nodos de un clúster pueden conectarse mediante una simple red Ethernet con placas comunes (adaptadores de red o NICs), o utilizarse tecnologías especiales de alta velocidad como Fast Ethernet, Gigabit Ethernet, Myrinet, Infiniband, SCI, etc.

- *Middleware*.- es un software que generalmente actúa entre el sistema operativo y las aplicaciones con la finalidad de proveer a un clúster lo siguiente:

    - Una interfaz única de acceso al sistema, denominada SSI (*Single System Image*), la cual genera la sensación al usuario de que utiliza un único ordenador muy potente.
    - Herramientas para la optimización y mantenimiento del sistema: migración de procesos, *checkpoint-restart* (congelar uno o varios procesos, mudarlos de servidor y continuar su funcionamiento en el nuevo host), balanceo de carga, tolerancia a fallos, etc.
    - Escalabilidad: debe poder detectar automáticamente nuevos servidores conectados al clúster para proceder a su utilización.

  Existen diversos tipos de middleware, como por ejemplo: MOSIX, OpenMOSIX, Cóndor, OpenSSI, etc.

  El middleware recibe los trabajos entrantes al clúster y los redistribuye de manera que el proceso se ejecute más rápido y el sistema no sufra sobrecargas en un servidor. Esto se realiza mediante políticas definidas en el sistema (automáticamente o por un administrador) que le indican dónde y cómo debe distribuir los procesos, por un sistema de monitorización, el cual controla la carga de cada CPU y la cantidad de procesos en él.

  El middleware también debe poder migrar procesos entre servidores con distintas finalidades:

    - balancear la carga: si un servidor está muy cargado de procesos y otro está ocioso, pueden transferirse procesos a este último para liberar de carga al primero y optimizar el funcionamiento.

- Mantenimiento de servidores: si hay procesos corriendo en un servidor que necesita mantenimiento o una actualización, es posible migrar los procesos a otro servidor y proceder a desconectar del clúster al primero.
- Priorización de trabajos: en caso de tener varios procesos corriendo en el clúster, pero uno de ellos de mayor importancia que los demás, puede migrarse este proceso a los servidores que posean más o mejores recursos para acelerar su procesamiento.

- *Ambientes de Programación Paralela.-* Los ambientes de programación paralela permiten implementar algoritmos que hagan uso de recursos compartidos: CPU (Central Processing Unit), memoria, datos y servicios.

1.3.5. Ventajas

Las aplicaciones paralelas escalables requieren: buen rendimiento, baja latencia, comunicaciones que dispongan de gran ancho de banda, redes escalables y acceso rápido a archivos. Un clúster puede satisfacer estos requerimientos usando los recursos que tiene asociados a él.

Los clústers ofrecen las siguientes características a un costo relativamente bajo:

- Alto Rendimiento.
- Alta Disponibilidad.
- Alta Eficiencia.
- Escalabilidad.

La tecnología clúster permite a las organizaciones incrementar su capacidad de procesamiento usando tecnología estándar, tanto en componentes de hardware como de software que pueden adquirirse a un costo relativamente bajo.

El clúster está diseñado para evitar un único punto de error. Las aplicaciones pueden distribuirse en más de un equipo, consiguiendo un grado de paralelismo y una recuperación de errores y proporcionando más disponibilidad.

Puede aumentar la capacidad de cálculo del clúster si agrega más procesadores o equipos.

El clúster aparece como la imagen de un único sistema para los usuarios finales, las aplicaciones y la red, proporcionando a la vez un único punto de control para los administradores, local o remotamente.

1.3.6. Ejemplos de Sistemas Clústers Implementados

En esta sección se dan a conocer brevemente algunos de los clústers implementados más importantes construidos:

- *Beowulf*.- Fue desarrollado por Donald Becker y Thomas Sterling en 1994. Construido con 16 computadores personales con procesadores Intel DX4 de 200 MHz, que estaban conectados a través de un switch Ethernet. El rendimiento teórico era de 3.2 GFlops.

Figura 1: Clúster Beowulf

- *Berkeley NOW*.- El sistema NOW de Berkeley estuvo conformado por 105 estaciones de trabajo Sun Ultra 170, conectadas a través de una red Myrinet. Cada estación de trabajo contenía un microprocesador Ultra1 de 167 MHz, caché de nivel 2 de 512 KB, 128 MB de memoria, dos discos de 2.3 GB, tarjetas de red Ethernet y Myrinet. En abril de 1997, NOW logró un rendimiento de 10 GFlops.

- *Google*.- Durante el año 2003, el clúster Google llegó a estar conformado por más de 15.000 computadores personales. En promedio, una consulta en Google lee cientos de megabytes y consume algunos billones de ciclos del CPU.

- *Clúster PS2*.- En el año 2004, en la Universidad de Illinois en Urbana-Champaign, Estados Unidos, se exploró el uso de consolas Play Station 2 (PS2) en cómputo científico y visualización de alta resolución. Se construyó un clúster conformado por 70 PS2; utilizando Sony Linux Kit (basado en Linux Kondora y Linux Red Hat) y MPI.

- *Clúster X*.- En la lista "TOP 500" de noviembre de 2004 fue considerado el séptimo sistema más rápido del mundo; sin embargo, para julio de 2005 ocupa la posición catorce. Clúster X fue construido en el Tecnológico de Virginia en el 2003; su instalación fue realizada por estudiantes del Tecnológico. Está constituido por 2200 procesadores Apple G5 de 2.3 GHz. Utiliza dos redes: Infiniband 4x para las comunicaciones entre procesos y Gigabit Ethernet para la administración. Clúster X posee 4 Terabytes de memoria RAM y 176 Terabytes de disco duro, su rendimiento es de 12.25 TFlops. Se lo conoce también como Terascale.

- *Red Española de Supercomputación*.- En el año 2007 se crea la Red Española de Supercomputación compuesta por 7 clúster distribuidos en distintas instituciones españolas, entre los que se encuentra el supercomputador Marenostrum (el clúster más veloz en el momento de su puesta en funcionamiento). Todos los clúster están formados por un número variable de nodos con procesadores

PowerPC 970 a 2.2GHz interconectados con una red Myrinet. El rendimiento de las máquinas oscilan entre los casi 65 TeraFLOPS proporcionados por las más de 10000 CPUs de Marenostrum, los casi 16 TeraFLOPS de Magerit con 2400 procesadores o los casi 3 TeraFLOPS de los 5 nodos restantes.

- *Thunder*.- Thunder fue construido por el Laboratorio Nacional Lawrence Livermore de la Universidad de California. Está conformado por 4096 procesadores Intel Itanium2 Tiger4 de 1.4GHz. Utiliza una red basada en tecnología Quadrics. Su rendimiento es de 19.94 TFlops. Se ubicó en la segunda posición del "TOP 500" durante junio de 2004, luego en la quinta posición en noviembre de 2004 y en la lista de julio de 2005 se ubicó en la séptima posición.

- *ASCI Q*.- ASCI Q fue construido en el año 2002 por el Laboratorio Nacional Los Álamos, Estados Unidos. Está constituido por 8192 procesadores AlphaServer SC45 de 1.25 GHz. Su rendimiento es de 13.88 TFlops. Se ubicó en la segunda posición del "TOP 500" durante junio y noviembre de 2003, luego en la tercera posición en junio de 2004, en la sexta posición en noviembre de 2004 y en la duodécima posición en julio de 2005.

1.4. Introducción a la computación en paralelo

Durante décadas, los ordenadores paralelos eran sinónimos de los supercomputadores, grandes y caros, las máquinas construidas por empresas como Cray e IBM, eran accesibles sólo para los laboratorios del gobierno y las grandes corporaciones. Solamente programadores expertos eran capaces de utilizar eficazmente estos sistemas. En la década de 1990, dos tipos de programación en paralelo llegaron a dominar la computación paralela: "Message Passing Interface" (MPI) y "Open Multiprocessing" (OpenMP). Sin embargo, MPI y OpenMP todavía requería una comprensión más profunda sobre la computación paralela.

Hoy en día, con el surgimiento de procesadores multinúcleo, la programación en paralelo está presente en todas partes. Las arquitecturas multinúcleo se han extendido rápidamente a todos los ámbitos de la informática, desde el incrustado en sistemas de ordenadores personales a supercomputadores de alto rendimiento.

El conocimiento y la experiencia en la programación en paralelo, lamentablemente, no han seguido el ritmo de la tendencia hacia el hardware paralelo. Las arquitecturas multinúcleo necesitan computación paralela y la gestión explícita de la jerarquía de memoria.

Mientras MPI y OpenMP todavía tienen un lugar en el mundo del multinúcleo, el proceso de aprendizaje de dichos lenguajes es demasiado fuerte para la mayoría de los programadores. Aunque hoy en dia, estos tipos de lenguajes nos llevan a las nuevas tecnologías, que son necesarias para que los procesadores multinúcleo sean accesibles a

una comunidad mayor. El procesamiento de imágenes (SIP) es uno de los campos que se beneficiará enormemente de estas tecnologías.

Estas nuevas tecnologías de software permiten ocultar la complejidad de las arquitecturas de múltiples núcleos, permitiendo a los programadores centrarse en los algoritmos en lugar de en las arquitecturas.

1.4.1. Capacidades técnológicas

Esta sección describe las diversas capacidades en software de muchas tecnologías que permiten a los programadores ofrecer programas para arquitecturas de múltiples núcleos.

Los diseños multinúcleo van desde, arquitecturas homogéneas (por ejemplo, de múltiples núcleos x86), a arquitecturas heterogéneas (por ejemplo, IBM Cell).

El movimiento eficiente de datos a través de la jerarquía de memoria es clave para lograr un alto rendimiento.

El paralelismo de datos es una forma de paralelismo donde se distribuyen los datos entre varios procesadores, cada procesador realiza el mismo cálculo, reduciendo así la cantidad total de trabajo en cada procesador. El paralelismo de datos es a menudo sinónimo de un solo modelo de programación programas-simples con datos-múltiples (SPMD), donde el mismo programa se ejecuta en varios procesadores, pero en los procesos existen datos diferentes.

El paralelismo de tareas es una forma de paralelismo. Las tareas se asignan a los diferentes procesadores. El paralelismo de tareas es a menudo sinónimo de un modelo de programación de programas-múltiples con datos-múltiples (MPMD), en el que diferentes procesadores ejecutan diferentes programas.

El mecanismo por defecto para la ejecución de programas en paralelo con procesadores de múltiples núcleos es el de ejecutar un proceso independiente en cada núcleo.

Una serie de tecnologías soportan múltiples arquitecturas de software, reduciendo al mínimo la dificultad de portar aplicaciones entre diferentes arquitecturas multinúcleo.

1.4.2. Lenguajes

La gran mayoría de lenguajes de programación están diseñados para la programación en serie. A continuación se presentan algunos lenguajes que soportan la programación en paralelo.

1.4.2.1. Unified Parallel C

La UPC es una extensión paralela del lenguaje C. Los programas de la UPC se modelan como un conjunto de hilos que comparten un único espacio de direcciones global que está lógicamente dividido entre los hilos, un modelo de programación

conocido como particiones “Global Address Space” (PGAS). Una parte de los hilos del espacio de direcciones están divididos en espacios privados y compartidos. Sólo los hilos locales pueden acceder a las variables en el espacio privado, cualquier hilo puede acceder a las variables en el espacio compartido. Las variables se asignan en el espacio compartido, añadiendo la palabra clave compartida para la declaración.

UPC apoya el paralelismo a través de las matrices de datos compartidos. Las matrices compartidas se dividen entre los hilos de tal manera que cada hilo tiene una parte de la matriz asignada en su espacio compartido. Cuando hace falta comunicación entre los hilos estos se comunican de manera transparente para el usuario. Este lenguaje, UPC, permite a los programadores escribir un programa en C en serie y después paralelizar el programa.

1.4.2.2. Sequoia

Sequoia es un proyecto de investigación en la Universidad de Stanford. Los objetivos de Sequoia, son los de lograr un alto rendimiento de la gestión de los datos en un procesador. Sequoia es una extensión sintáctica de C. A diferencia del modelo C, que consta de un único procesador, y el espacio de memoria única, Sequoia expone las jerarquías de memoria y coprocesadores.

Los programas de Sequoia se basan en funciones de cómputo denominadas tareas. A una tarea sólo se puede acceder a sus argumentos de la función y variables locales declaradas, como una función de C. Se permite que el mismo programa se pueda asignar a diferentes arquitecturas sin modificar el código del programa.

1.4.3. Otros lenguajes

Co-Array Fortran y Titanium son lenguajes PGAS **(**Partitioned Global Address Space). Al igual que UPC, se dan varias copias de un programa en diferentes núcleos, pero que comparten un espacio de direcciones global. Co-Array Fortran y Titanium añaden extensiones de los lenguajes Fortran y Java, respectivamente, para distribuir los datos. Cilk++ es una extensión paralela de C++ desarrollado por Cilk Arts y se basa en el lenguaje Cilk desarrollado en Massachusetts Institute of Technology (MIT). Cilk++ añade tres palabras clave, *cilk_spawn, cilk_sync*, y *cilk_for*, para marcar el paralelismo y la sincronización en un programa.

StreamIt fue desarrollado en el MIT, StreamIt tiene la ventaja del hecho de que las aplicaciones de *streaming*[7] a menudo realizan varias veces los mismos cálculos de forma regular, es decir, posee un modelo estático que se puede ejecutar como corrientes paralelas de instrucciones.

[7] Streaming es un término que se refiere a ver u oír un archivo directamente en una página web sin necesidad de descargarlo antes al ordenador.

Los sistemas de computación de alta productividad (HPCS) son desarrollados con lenguajes de computación de alto rendimiento para permitir aplicaciones paralelas y mejorar la productividad del futuro de los programadores.

1.4.4. Cálculo de propósito general en las GPU

Las GPU (unidades de procesamiento gráfico) están optimizadas para los algoritmos de gráficos, resultando en arquitecturas altamente paralelas. Los investigadores comenzaron a explorar el uso de GPU para acelerar los algoritmos no gráficos dando lugar a cálculos de propósito general en las GPU (GPGPU). Tempranamente, los algoritmos tuvieron que ser rediseñados para el uso de gráficos y operaciones de las API **(**Application Programming Interface) [por ejemplo, Open Graphics Library (OpenGL)].

En los últimos años, han aparecido más tecnologías de la programación general de las GPUs, eliminando la necesidad de un fondo en la programación de gráficos. Tres de las tecnologías que hacen esto son: “Compute Unified Device Architecture” (CUDA), “Brook+”, y “Open Computing Language” (OpenCL). Por su diseño, GPUs son coprocesadores y, por tanto, estas tecnologías tienen el mismo modelo de programación fundamental: una aplicación ejecutándose en un host, asigna un host y la memoria de la GPU, copia los datos de entrada desde la memoria del host a la memoria de la GPU, entonces se lanza en paralelo en los núcleos la GPU.

1.4.5. Lenguajes de alto nivel

Lenguajes de alto nivel (HLLs), tales como Mathematica y MATLAB, son atractivos para los ingenieros y científicos por sus altos niveles de abstracciones y entornos de programación interactiva. HLLs, sin embargo, sufren de una disminución del rendimiento, que a menudo resulta en tiempo de ejecución de la aplicación de horas o días. En el Capítulo 2 se ven varias tecnologías de apoyo a la computación paralela de MATLAB sin sacrificar la facilidad de uso del lenguaje de MATLAB, las cuales son: pMatlab, Star-P y Matlab Parallel Computing Toolbox.

1.5. Presentación del proyecto

En el siguiente capítulo, se describirán las partes hardware y software que hacen referencia al clúster-GTTS y los programas necesarios para trabajar con él, utilizando programación en paralelo. En la parte referida al clúster-GTTS, encontraremos sus características más importantes, así como imágenes del clúster en cuestión. En el apartado de la programación en paralelo se verá con profundidad el tipo de programación que nos permite la librería “Parallel Computing Toolbox” aportada por Matlab, que será con la que se trabaje en este proyecto fin de carrera. En el tercer capítulo, se mostrará de una forma detallada, cómo se implementan programas en paralelo con Matlab, que permitirán al usuario comprender mejor el funcionamiento de este tipo de programación basada en la librería “Parallel Computing Toolbox”. En el capítulo cuatro obtenemos unos resultados a partir de una serie de programas ejecutados en paralelo en el clúster-GTTS, que se compararán con ejecuciones en serie,

obteniendo así una visión global de cómo trabaja o funciona el clúster del laboratorio GTTS (Grupo de Teoría y Tratamiento de la Señal) que nos proporciona la Universidad Politécnica de Cartagena. Por último, las conclusiones y las líneas futuras se describen en el capítulo 5. Encontrando al final de este proyecto fin de carrera los anexos y la bibliografía.

CAPÍTULO 2

COMPUTACIÓN EN PARALELO

2.1. Introducción

Este capítulo se habla sobre la computación en paralelo, dividiendo éste en dos grandes partes, la parte hardware y la parte software. En la parte hardware se mostrarán las características del clúster, así como las partes de las que está compuesto y en la parte software se verán algunas de las tecnologías software multinúcleo profundizando en la tecnología que nos aporta Matlab, ya que es con la que se trabajará en este proyecto fin de carrera, específicamente con la librería "Parallel Computing Toolbox".

2.2. Hardware

Un clúster es un grupo de equipos que trabajan de manera conjunta para ejecutar un grupo de aplicaciones comunes y ofrecer la imagen de un único sistema al cliente y a la aplicación. Los equipos están conectados físicamente mediante cables y programados mediante software para clústers. Estas conexiones permiten que los equipos utilicen la conmutación por error y el equilibrio de carga, que no resultan posibles en equipos individuales. La conmutación por error, es el proceso automático que ocurre, cuando uno de los equipos presenta un error y deja de funcionar, y otro equipo se encargar de realizar el trabajo o las tareas del equipo que ha dejado de funcionar. El equilibrio de carga se dará cuando alguno de los equipos tenga mucho trabajo por realizar y parte de este trabajo se envia a otro equipo que contenga menos carga o no este realizando ningún trabajo, para así terminar dichos trabajos con mayor rapidez.

Hoy en día juegan un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno. La tecnología de clusters ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y software de misiones críticas, servidores Web y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

El cómputo con clústers, surge como resultado de la convergencia de varias tendencias actuales que incluyen la disponibilidad de microprocesadores económicos de alto rendimiento y redes de alta velocidad, el desarrollo de herramientas de software para cómputo distribuido de alto rendimiento, así como la creciente necesidad de potencia computacional para aplicaciones que la requieran.

De un clúster se espera que presente combinaciones de los siguientes servicios:

- Alto rendimiento
- Alta disponibilidad
- Equilibrio de carga
- Escalabilidad

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

Para que un clúster funcione como tal, no basta solo con conectar entre sí los ordenadores, sino que es necesario proveer un sistema de manejo del clúster, el cual se encargue de interactuar con el usuario y los procesos que corren en él para optimizar el funcionamiento.

2.2.1. Sistema Operativo del clúster-GTTS

GNU/Linux, es el sistema operativo del clúster-GTTS, utilizado en este proyecto, el cual proporcionará distintos tipos de servicios de clúster, entre ellos:

- *Alta disponibilidad y escalabilidad* para aplicaciones vitales como bases de datos, sistemas de mensajería y servicios de archivos e impresión. Los diversos servidores (nodos) de un clúster permanecen en comunicación constantemente. En caso de que uno de los nodos del clúster no esté disponible debido a un error o a que se están realizando tareas de mantenimiento, otro nodo comenzará inmediatamente a proporcionar servicios, un proceso denominado conmutación por error. Los usuarios que estén obteniendo acceso al servicio continuarán haciéndolo, sin saber que ahora el servicio lo proporciona un servidor distinto (nodo).
- *Linux Virtual Server*.- es una solución para gestionar balance de carga en sistemas Linux. Actualmente, la labor principal del proyecto LVS es desarrollar

un sistema IP avanzado de balanceo de carga por software (IPVS[8]), balanceo de carga por software a nivel de aplicación y componentes para la gestión de clústers.

2.2.2. Características del clúster-GTTS

El Clúster del laboratorio GTTS, con el que se ha realizado este proyecto fin de carrera, va a ser descrito a continuación. Se van a ir viendo cada una de las partes de las que está compuesto el clúster-GTTS. A continuación se empezarán a describir las partes externas del clúster-GTTS, como son el armario o rack que contiene los servidores, la unidad de distribución de alimentación, los paneles laterales del bastidor, el kit estabilizador del bastido y el kit de puesta a tierra del bastidor. En esta sección se describirán las características más importates, si se desea más información acerca de alguna parte especifíca consultar la referencia [4].

- **HP Rack 10642 G2 Shock Pallet - Rack - carbón, grafito metálico - 42U**

Figura 2: HP Rack 10642 G2 Shock Pallet

Tabla 1: Características del Rack 10642 G2 Shock Pallet

[8] IPVS: sistema IP avanzado de balanceo de carga por software implementado en el propio núcleo Linux y ya incluido en las versiones 2.4 y 2.6. La versión utilizada en el clúster utilizado en el proyecto fin de carrera es la 2.6.18-92 1.18.e15PAE.

Descripción del producto	HP Rack 10642 G2 Shock Pallet - rack - 42U
Tipo de producto	Rack
Color	Carbón, grafito metálico
Tamaño del rack	19"
Altura (unidades de bastidor)	42U
Material del producto	Metal
Dimensiones (Ancho x Profundidad x Altura)	59.7 cm x 101 cm x 200 cm
Peso	114.8 kg
Garantía del fabricante	3 años de garantía

- **HP PDU - Unidad de distribución de alimentación (montaje en bastidor) - CA 200/240 v -3680 VA - 16 conector/es de salida**

Figura 3: Unidad de distribución de alimentación

Tabla 2: Carácterísticas de la unidad de distribución de alimentación

Descripción del producto	HP PDU - Unidad de distribución de alimentación 3680 VA
Tipo de producto	Unidad de distribución de alimentación – montaje bastidor
Voltaje de entrada	CA 200/240 V
Potencia suministrada	3680 VA
Conector/es de salida	16 x alimentación IEC 320 EN 60320 C13
Dimensiones (Ancho x Profundidad x Altura)	44.5 cm x 14.2 cm x 4.1 cm
Peso	8.2 kg
Garantía del fabricante	3 años de garantía

- **HP – Panel lateral de bastidor – grafito metálico – 42U – 19’’**

Figura 4: Panel lateral de bastidor

Tabla 3: Características del panel lateral de bastidor

Descripción del producto	Panel lateral de bastidor – 42U
Cantidad	2 paneles laterales de bastidor
Color	Grafito metálico
Tamaño del rack	19’’
Altura (unidades de bastidor)	42 U
Material del producto	Metal

- **HP - kit de opción de estabilizador para bastidor - grafito**

Figura 5: kit de opción de estabilizador para bastidor

Tabla 4: Características del kip opción de estabilizador para bastidor

Descripción del producto	Kit de opción de estabilizador para bastidor
Color	Grafito
Material del producto	Metal

- **HP - kit de puesta a tierra de bastidor**

Figura 6: kit de puesta a tierra de bastidor

Una vez descritas las partes externas del clúster-GTTS, se van a detallar las partes internas y más fundamentales de éste, que son los servidores. Estos servidores, son simplemente ordenadores de gran capacidad de cómputo. El clúster-GTTS esta compuesto por 6 servidores, de los que hay 3 tipos diferentes, cada uno de ellos con unas características especificas, como se van a ver a continuación. En el Anexo II, se pueden ver ampliadas las características de algunas partes de las que se compone el clúster-GTTS (las partes ampliadas en el Anexo II vienen indicadas con una nota).

- **HP ProLiant ML350 G5 (458238-421) – Especificaciones**

Figura 7: HP ProLiant ML350 G5

Tabla 5: Procesador y memoria del HP ProLiant ML350 G5

Procesador y memoria	
Tipo de procesador	Procesador Quad Core Intel® Xeon® E5430
Velocidad del procesador	2,66 GHz
Número de procesadores	1 procesador
Actualización del procesador	Actualizable a doble procesamiento: 8 Cores
Núcleo de procesador disponible	Quad
Memoria caché interna	12 MB (2 x 6 MB) de caché de nivel 2
Chipset	Chipset Intel® 5000Z
Memoria	Módulos DIMM PC2-5300 con memoria intermedia completa (DDR2-667) que funcionan a 667 MHz
Memoria de serie	2 GB (2 x 1 GB) de memoria de serie
Bus frontal del procesador	Bus frontal de 1333 MHz
Memoria máxima	32 GB (8 x 4 GB) de memoria máxima
Ranuras de memoria	8 ranuras

NOTA: para más información acerca del HP ProLiant ML350 G5 ver Anexo II.

- **HP ProLiant ML570 G4 High Performance- Bastidor – 4 vías – 2 x Dual-Core Xeon 7140M / 3.4 GHz – RAM 4GB – SAS – hot-swap 2.5'' – sin disco duro – combinación de CD-RW / DVD-ROM – Gigabit Ethernet – Monitor : ninguno – 6U**

Figura 8: HP ProLiant ML570 G4

Tabla 6: Caracteristicas del HP ProLiant ML570 G4

Descripción del producto	HP ProLiant ML570 G4 High Performance - Dual-Core Xeon 7140M 3.4 GHz
Tipo	Servidor
Factor de forma	Se puede montar en bastidor - 6U
Dimensiones (Ancho x Profundidad x Altura)	48.3 cm x 71.1 cm x 28.2 cm
Peso	62.1 kg
Localización	Europa
Escalabilidad de servidor	4 vías
Procesador	2 x Intel Dual-Core Xeon 7140M / 3.4 GHz (Dual-Core)
Características principales del procesador	Hyper-Threading Technology, Intel Extended Memory 64 Technology, Intel Execute Disable Bit, Enhanced Intel SpeedStep Technology, Intel Virtualization Technology
Memoria caché	32 MB L3
Caché por procesador	16 MB
Memoria RAM	4 GB (instalados) / 64 GB (máx.) - DDR2 SDRAM - Código de corrección de errores (ECC) avanzado - 400 MHz - PC2-3200
Controlador de almacenamiento	Serial Attached SCSI (Serial Attached SCSI) - PCI Express (Smart Array P400) ; IDE
Bahías de almacenamiento de servidor	Hot-swap 2.5"
Disco duro	No.
Almacenamiento óptico	Combinación de CD-RW / DVD-ROM
Monitor	Ninguno
Memoria de vídeo	32 MB DDR SDRAM
Conexión de redes	Adaptador de red - PCI Express x4 - Ethernet, Fast Ethernet, Gigabit Ethernet - Puertos Ethernet : 2 x Gigabit Ethernet
Alimentación	CA 120/230 V (50/60 Hz)
Redundancia de alimentación	Sí
Garantía del fabricante	3 años de garantía - in situ

- **HP Proliant DL580 G5 2.93 130W 4P 8G**

Figura 9: HP Proliant DL580 G5

Tabla 7: Procesador y memoria del HP Proliant DL580 G5

Procesador y memoria	
Tipo de procesador	Procesador Intel® Xeon® X7350 Quad Core a 2,93 GHz
Velocidad del procesador	2,93 GHz
Número de procesadores	4 procesadores
Actualización del procesador	Hasta 4 procesadores
Núcleo de procesador disponible	Quad
Memoria caché interna	2 x 4 MB de caché de nivel 2
Chipset	Chipset Intel 7300 estándar
Memoria	DIMMs PC2-5300 con memoria intermedia completa (DDR2-667)
Memoria de serie	8 GB (4 x 2 GB) de memoria de serie
Bus frontal del procesador	Bus frontal a 1.066 MHz
Memoria máxima	256 GB
Ranuras de memoria	32 ranuras DIMM

NOTA: para más información acerca del HP Proliant DL580 G5 ver Anexo II.

- **HP Single Port - Disco duro - 146 GB - hot-swap - 2.5" SFF - SAS - 10000**

Figura 10: Disco duro

Tabla 8: Descripción general del disco duro

General

Tipo de dispositivo	Disco duro - hot-swap
Anchura	7 cm
Altura	1.5 cm

NOTA: para más información acerca del disco duro ver Anexo II.

El clúster del laboratorio GTTS se compone de cuatro HP Proliant ML350 G5, de un HP Proliant DL580 G5 y de un HP Proliant ML570 G4, donde cada uno de ellos posee 2 discos duros de 146 GB. El resultado se observa en la figura 11:

Figura 11: Clúster computacional de la UPCT

2.3. Software

Como ya se anunciaba en el Capítulo 1, existen 3 tipos de tecnologías que permiten desarrollar con Matlab programas en paralelo. En esta sección, se van a ver, en profundidad estas tecnologías, sobre todo la tecnología de "Parallel Computing Toolbox", que será la empleada a lo largo del proyecto.

2.3.1. pMatlab

Lincoln Laboratory del MIT (MIT-LL) desarrolla pMatlab, que apoya el paralelismo de datos mediante una programación llamada mapa. Para desarrollar una aplicación pMatlab, el usuario escribe un programa de Matlab. El programa es entonces paralelizado agregando mapas. Los mapas son objetos que describen cómo debe ser una matriz distribuida.

pMatlab distribuye y comunica los datos sin que el usuario tenga conocimiento de ello. pMatlab está escrito enteramente en el lenguaje Matlab funciona en cualquier arquitectura con el apoyo de Matlab. Esto se extiende desde los ordenadores personales hasta los clúster computacionales, los cuales contendrán procesadores en serie o multinúcleos homogéneos.

pMatlab introduce un nuevo tipo de datos: la matriz de distribución o dmat. Dmat es el tipo de dato de almacenamiento fundamental en pMatlab, equivalente al doble en Matlab. pMatlab puede trabajar con objetos dmat ya sean con dos, tres o cuatro dimensiones. Los objetos dmat pueden ser explícitamente construidos en pMatlab a través de una función constructora.

En la actualidad, pMatlab posee cuatro funciones del constructor de Matlab: zeros, ones, rand, y spalloc. Cada una de estas funciones acepta el mismo conjunto de parámetros como sus correspondientes funciones de Matlab, añadiéndole un parámetro del mapa. El parámetro que se le añade, el mapa, describe cómo distribuir el objeto dmat entre varios procesadores.

```
D = ones(M, P);  % Crea una matriz distribuida(dmat)MxM, de unos
% con el mapa P

D = zeros(M, N, R, P); % Crea una matriz distribuida MxNxR de ceros usando un
% mapa P

D = rand(M, N, R, S, P); % Crea una matriz distribuida MxNxRxS con números
% aleatorios y el mapa P.

D = spalloc(M, N, X, P); % crear una matriz distribuida dispersa de tamaño
%MxN con X valores distintos de cero con el mapa P
```

Figura 12: funciones pMatlab

Cada mapa contiene 4 componentes:

- Grid.- es un vector que indica cómo se divide la matriz distribuida (dmat). Si el grid fuese [2,3], la matriz se dividirá, para la primera dimensión, en 2 procesos y para la segunda dimensión, en 3 procesos. La matriz quedará distribuida de la siguiente manera:

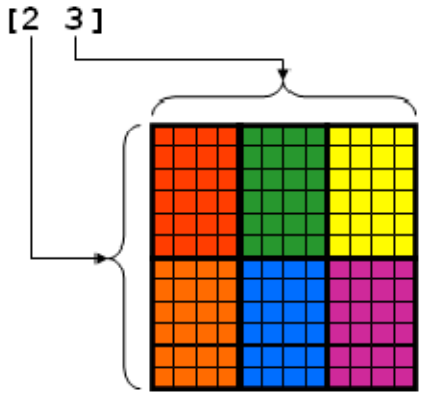

Figura 13: Matriz distribuida [2, 3]

- Distribution.- especifica el orden de cómo se distribuyen los datos de cada dimensión a los procesadores. Existen 3 tipos de distribución:
  - Block.- cada procesador contiene un solo bloque de datos.
  - Cyclic.- los datos se intercalan entre los procesadores.
  - Block-cyclic.-bloques de datos se intercalan entre los procesadores.

- Processor list.-especifica cómo se podrían asignar los procesadores a las diferentes dimensiones.
- Overlap.-es un vector que especifica la cantidad de datos que se pueden superponer en cada procesador.

Dos de las operaciones más fundamentales de Matlab son subíndice de asignación (subsasgn) y subíndice de referencia (subsref). En el siguiente ejemplo se muestra como realizar los subíndices de asignación en pMatlab.

```
N = 1000000;
% A y B tienen diferentes mapas
mapA = map([1 Np], {}, 0:Np-1);
mapB = map([Np 1], {}, 0:Np-1);
A = zeros(N, mapA);
B = rand(N, mapB);
% Invoca a “subsasn” de pMatlab
% Los datos en B son remapeados al mapa y asignados a A
A(:,:) = B;
% Realiza una copia de la memoria
% A es remplazado por una copia de B
```

Figura 14: Ejemplo de subíndices de asignación con pMatlab

Aquí se muestra un ejemplo de cómo realizar subíndices de referencia en pMatlab:

```
A(:,:) % Referencia a toda la matriz
A(i,j) % Referencia a un elemento de la matriz
A(i:k, j:l) % Referencia a una submatriz de la matriz
```

Figura 15: Ejemplo de subíndices de referencia con pMatlab

Estos operadores aritméticos +, *, .*, ==, se utilizan de igual forma en pMatlab que en Matlab:

```
N = 1000000;

M1 = map([1 Np], {}, 0:Np-1);
M2 = map([Np 1], {}, 0:Np-1);

% Se anadem 2 matrices distribuidas (dmat)
A = rand(N, M1); % NxN dmat mapped to M1
B = rand(N, M1); % NxN dmat mapped to M1

C1 = A + B; % Result is mapped to M1
C2 = zeros(N, M2); % NxN dmat mapped to M2
C2(:,:) = A + B; % Result is remapped to M2

D = rand(N, M1); % NxN dmat mapped to M1
E = rand(N); % NxN double
F1 = D * E; % Result is mapped to M1
F2 = zeros(N, M2); % NxN dmat mapped to M2
F2(:,:) = D * E; % Result is remapped to M2

G = rand(N, M1); % NxN dmat mapped to M1
H = rand(N, M1); % NxN dmat
I1 = (G == H); % Result is mapped to M1
I2 = zeros(N, M2); % NxN dmat mapped to M2
I2(:,:) = (G == H); % Result is remapped to M2
```

Figura 16: **Ejemplo de operadores aritméticos con pMatlab**

Varias funciones matemáticas, incluyendo las funciones simples, como el valor absoluto (abs) y funciones avanzadas como FFT se han añadido a pMatlab. La llamada a estas funciones se realiza de la misma forma que en Matlab. A continuación se muestra un ejemplo:

```
N = 1000000;

M1 = map([1 Np], {}, 0:Np-1);
M2 = map([Np 1], {}, 0:Np-1);

A = rand(N, M1); % NxN dmat mapped to M1
B1 = abs(A); % Result is mapped to M1
```

Figura 17: **Ejemplo de funciones avanzadas con pMatlab**

2.3.2. Star-P

Star-P es un producto interactivo de supercomputación, inicialmente dirigido a Matlab, pero que se ha expandido a otros HLLs, por ejemplo, Python. Ambos pMatlab y Star-P comparten un enfoque similar al apoyo datos paralelos. En primer lugar, el usuario escribe un programa en serie funcionalmente correcto. Star-P indica las dimensiones de la matriz de distribución mediante el etiquetado de los argumentos con *p. Un ejemplo de la asignación sería: a = rand(100*p).

Star-P tiene un cliente-servidor basado en una arquitectura donde el cliente es una sesión de Matlab que se ejecuta en el ordenador del usuario y el servidor es un clúster que ejecuta el software de Star-P.

Para una mayor compresión de cómo funciona el método Start-P de Matlab, se va a ilustrar un ejemplo sencillo. En este ejemplo se crea una matriz aleatoria de tres dimensiones, en el que se calcula la inversa de esta matriz. En la imagen de la figura 18 podemos ver la implentación de este ejemplo en serie; en la imagen de la figura 19 se observa este mismo ejemplo implementado con Start-P, para ello se ha añadido el parametró *p en la última dimensión de la matriz a la hora de crearla. Con Start-P se utiliza la función “ppeval” para enviar a ejecutar de forma paralela el trabajo deseado, esta función se utiliza de forma análoga a la función “feval” de “Parallel Computing Toolbox”; y en la imagen de la figura 20 se ha implementado un ejemplo con una matriz de dos dimensiones utilizando la función “feval”, ya que “feval” devuelve un error al introducirle, como parámetro de entrada, una matriz de tres dimensiones.

```
%implementación en serie
a=rand(100,100,500);
for i=1:500
    b(:,:,i)=inv(a(:,:,i));
end
```

Figura 18: Ejemplo implementado en serie

```
%implementación en paralelo con Star-P
a=rand(100,100,500*p);
b=ppeval('inv',a)
```

Figura 19: Figura 18 implementado con Star-p

```
a=rand(100,200);
b=feval('inv',a)
```

Figura 20: Ejemplo con la función feval

2.3.3. Parallel Computing Toolbox

En esta sección del presente capítulo se va a describir las características de "Parallel Computing Toolbox" de Matlab, para una mayor comprensión de en qué se basa la programación paralela a través de Matlab.

La librería "Parallel Computing" de Matlab, nos permite desde un cliente de Matlab trabajar con varias sesiones de Matlab a la vez. "Parallel Computing Toolbox", permite trabajar hasta con 4 "workers"[9] en una máquina local, además de la sesión del cliente, aunque si es en un clúster donde se quieren tener los "workers", se trabajará a través del software "Matlab Distributed Computing Sever", que nos permitirá trabajar con todos aquellos "workers" que el clúster posea.

En este proyecto fin de carrera el clúster que se utilizará tendrá 6 "workers", por lo que como máximo se podrá estar trabajando con 6 "workers" a la vez y no con más. Si se le indicara un número superior de "workers", Matlab nos daría un error en consola.

2.3.3.1. Parallel Computing Toolbox y Matlab Distributed Computing Server

"Parallel Computing Toolbox" y el software "Matlab Distributed Computing Server" permiten coordinar y ejecutar operaciones independientes de Matlab de forma simultánea en un clúster, acelerando así la ejecución de grandes trabajos de Matlab.

Un "Job" o trabajo es una operación de gran envergadura. Este trabajo se divide en diferentes segmentos llamados tareas. Se podría dividir el trabajo en tareas idénticas, pero las tareas no tienen porque ser idénticas.

La sesión de Matlab en que el trabajo y sus tareas están definidos, se llama periodo de sesión de cliente. A menudo, "Parallel Computing Toolbox" utiliza la sesión de cliente para llevar a cabo la definición de puestos de trabajo y tareas. "Matlab Distributed Computing Server" realiza la ejecución de su trabajo mediante la evaluación de cada una de sus tareas, devolviendo el resultado a su sesión de cliente.

El administrador de trabajo o "job manager" es la parte del software del servidor que coordina la ejecución de los trabajos y de la evaluación de sus tareas. El "job manager" distribuye las tareas de evaluación para cada servidor de sesiones de Matlab llamados "workers". El uso del "job manager" de MathWorks es opcional, el reparto de tareas a los trabajadores también puede ser realizado por un programador tal como "Windows Compute Cluster Server" (CCS) o "Plataforma LSF planificador", aunque el clúster de

[9] Workers: son los diferentes módulos de los que se compone el clústers, los que se encargan de ejecutar los trabajos.

este proyecto fin de carrera utilizará un “job manager”, para el reparto de las tareas a los trabajadores.

A continuación se muestra el esquema donde queda detallada cada una de las partes de la configuración básica para una computación paralela.

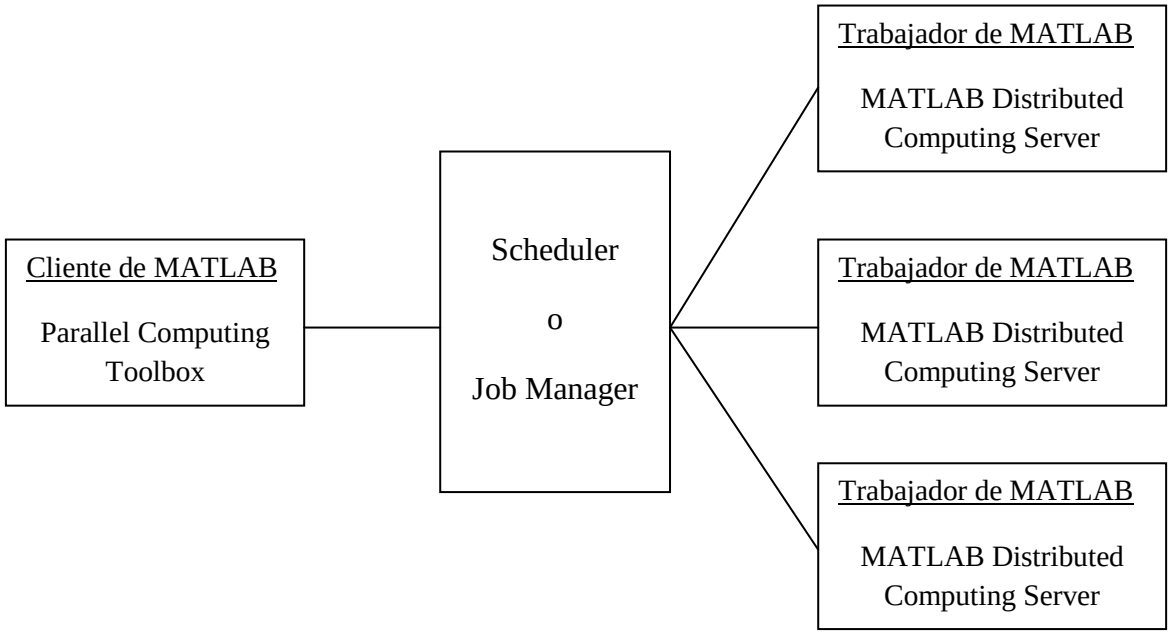

Figura 21: Esquema de una configuración básica para una computación paralela

2.3.3.2. Casos típicos de uso de la programación en paralelo

Existen tres casos típicos de uso de la programación en paralelo. Uno de estos casos son los bucles *for*, en los que se ejecutan repetitivamente líneas de código similares. Otro caso típico es la descarga de trabajo, es decir, se deja libre el cliente de Matlab mientras un “worker” realiza dicho trabajo. Y el último de los casos típicos es el de los grandes conjuntos de datos, que se encarga de distribuir el trabajo en los diferentes workers para manejar los datos de forma más sencilla.

A continuación se estudia cada uno de estos casos con más profundidad.

a) for-loops

Muchas aplicaciones involucran múltiples líneas de código, algunas de las cuales son repetitivas. A menudo se pueden utilizar bucles (*for-loops*) para resolver estos casos. La capacidad de ejecutar código en paralelo, en un equipo o en un clúster, puede mejorar significativamente el rendimiento de muchas aplicaciones. Estas aplicaciones se pueden clasificar en dos tipos:

- Aplicaciones de barrido de parámetros:
    - Muchas iteraciones.- Un barrido podría tardar mucho tiempo porque incluye muchas iteraciones. Cada iteración por sí sola no tardaría mucho tiempo en ejecutarse, pero para completar miles o millones de iteraciones en serie podría tardar mucho tiempo.
    - Iteraciones largas.- Un barrido podría no tener muchas iteraciones, pero cada iteración podría tardar mucho tiempo para ejecutarse.

  Normalmente, la única diferencia entre las iteraciones se define por los diferentes datos de entrada. En estos casos, la capacidad de ejecutar las iteraciones, de barrido por separado y al mismo tiempo, puede mejorar el rendimiento. La evaluación de iteraciones en paralelo es una forma ideal para barrer grandes o múltiples conjuntos de datos. La única restricción a los bucles en paralelo es que no se permite a las iteraciones depender de ninguna otra iteración.

- Test de suites con partes independientes.- Para las aplicaciones que se ejecutan con una serie de tareas no relacionadas, se pueden ejecutar de forma simultánea en diferentes sitios. Puede que no se haya utilizado un bucle *for*, para el caso que comprende estas tareas distintas, pero puede que un bucle *for* en paralelo ofrezca la solución adecuada.

"Parallel Computing Toolbox" mejora el rendimiento de la ejecución del bucle ya que permite que varios "workers" de Matlab ejecuten las iteraciones del bucle simultáneamente. Por ejemplo, un bucle de 100 iteraciones podría ejecutarse en un clúster de 20 "workers", por lo que cada uno de los "workers" solo tendría que ejecutar 5 iteraciones del bucle. La mejora en la velocidad ejecución no será exactamente 20 veces más rápida debido a la velocidad de comunicación entre las distintas aéreas y el tráfico de red, aunque la diferencia será muy notable.

b) Offloading work o descarga de trabajo

Cuando se trabaja de forma interactiva en una sesión de Matlab, se puede descargar dicho trabajo a un "worker" de Matlab. El comando para realizar esta acción es *asynchoronous*, lo que significa que el actual periodo de nuestra sesión de Matlab no estará bloqueado, y se podrá seguir con el periodo de sesiones interactivas propias, mientras que el "worker" está ocupado evaluando el código enviado. El "worker" de Matlab puede funcionar tanto en la misma máquina que el cliente, o en una máquina remota como puede ser un clúster.

c) Grandes conjuntos de datos

Si se tiene una matriz que es demasiado grande para la memoria de un ordenador dado, no se puede manejar fácilmente en una sola sesión de Matlab. "Parallel Computing Toolbox" permite la distribución de la matriz entre varios trabajadores de Matlab, de modo que cada "worker" sólo contiene una parte de la matriz. Sin embargo,

se puede operar en todo el conjunto como una sola entidad. Cada "worker" sólo opera en su parte de la matriz, y los trabajadores de forma automática se transfieren datos entre sí cuando sea necesario, como por ejemplo, en la multiplicación de matrices. Un gran número de operaciones de la matriz y de las funciones han sido mejoradas para trabajar directamente con las matrices distribuidas.

2.3.3.3. Parallel for loops

El concepto básico de un bucle paralelo en Matlab es el mismo que el bucle estándar de Matlab, es decir, ejecuta una serie de declaraciones (en el cuerpo del ciclo) en un rango de valores. Parte del cuerpo del *parfor* se ejecuta en el cliente de Matlab (con el que se envía el *parfor*) y parte se ejecuta en paralelo en los trabajadores o workers de Matlab. Los datos necesarios con los que opera *parfor*, se envían desde el cliente a los trabajadores, donde se realizan la mayoría de los cálculos y los resultados se envían de vuelta al cliente.

Debido a que varios workers de Matlab pueden ser computacionalmente concurrentes al mismo tiempo en el mismo ciclo, un ciclo *parfor* puede proporcionar un rendimiento significativamente mejor que un ciclo normal *for*.

Cada ejecución del cuerpo de un bucle *parfor* es una iteración. Los "workers" de Matlab evalúan las iteraciones sin ningún orden en particular y de forma independiente unas de otras. Debido a que cada iteración es independiente, no hay garantía de que las iteraciones se sincronicen en modo alguno, ni hay necesidad de esto. Si el número de trabajadores es igual al número de iteraciones del bucle, cada trabajador realiza una iteración del bucle. Si hay mas iteraciones que "workers", algunos "workers" realizaran más de una iteración, en este caso un trabajador puede recibir múltiples iteraciones a la vez para reducir el tiempo de comunicación.

a) Cuándo utilizar parfor

Un bucle *parfor* es útil en situaciones donde se necesitan muchas iteraciones de un bucle, donde estas iteraciones son de fácil cálculo. *Parfor* divide las iteraciones del bucle en grupos para que cada trabajador realice una parte del número total de iteraciones. Los bucles *parfor* también son útiles cuando se tienen iteraciones donde cada una de ellas tarda un gran tiempo en ejecutarse, porque los trabajadores pueden realizar estas iteraciones de forma simultánea.

No se puede utilizar un bucle *parfor* cuando una iteración del bucle depende de los resultados de otras iteraciones. Cada iteración debe ser independiente de todas las demás. Dado que existe un coste en la comunicación en el bucle *parfor*, no hay ninguna ventaja si el número de operaciones que se tienen que realizar son muy pequeñas, ya que el tiempo de comunicación será mayor que el de realizar dichas operaciones.

b) Matlabpool

Esta función, *matlabpool*, se usa para reservar un número de "workers" de Matlab para la ejecución de un bucle *parfor*. Dependiendo de su "scheduler" podrían estar ejecutándose de forma remota en un clúster, o puede que se ejecuten localmente en una máquina cliente de Matlab. A través de la configuración paralela se identificará el tipo de "scheduler" y clúster que se utilizará.

Si se ejecuta el comando *matlabpool open*, por defecto comienzan cuatro sesiones de trabajo de Matlab en la máquina local del cliente. Para cerrar estas sesiones se utiliza el comando *matlabpool close*.

Si *matlabpool* no se está ejecutando, el bucle *parfor* se ejecuta en serie en el cliente, ya que *matlabpool* es el que inicia la sesión en paralelo y si no se arranca no se ejecutará como tal.

Si se desea arrancar una sesión en paralelo en una máquina remota como podría ser un clúster, primero se debería crear una nueva configuración paralela en Matlab donde está maquina remota conste como tal, si le damos el nombre a esta configuración de *maquina_remota*, por ejemplo, para arrancar dicha sesión deberemos teclear la siguiente sentencia en la línea de comandos de Matlab: *matlabpool open maquina_remota*.

Si se quiere incluir el número de workers con los que se desea trabajar solo hay que incluir tal número a continuación del nombre dado a la configuración paralela en la sentencia anterior.

c) Creación de un bucle parfor

Para asegurarnos de que será un acierto utilizar un bucle *parfor*, hay que estudiar que todas las iteraciones son totalmente independientes unas de otras. Si unas iteraciones dependen de otras este bucle no podrá ser evaluado en paralelo, de modo que un bucle *for* normal no será fácilmente convertible a un bucle *parfor*.

Los ejemplos siguientes producen resultados equivalentes. El ejemplo de la izquierda con un bucle *for* y a la derecha un bucle *parfor*:

```
clear A

for i = 1:8
   A(i) = i;
end
```

```
clear A

parfor i = 1:8
   A(i) = i;
end
```

Figura 22: Comparación entre for y parfor

d) Diferencias entre bucles for y bucles parfor

Debido a que los bucles *parfor* no son exactamente iguales a los bucles *for*, hay comportamientos especiales a tener en cuenta. Como se ve en el ejemplo de la figura 23,

cuando se asigna a una variable una matriz (*A*) dentro del bucle *parfor*, los elementos de la matriz están disponibles después del bucle, de la misma manera que con el bucle *for*.

Sin embargo, imaginemos que se utiliza una variable donde los valores de la variable *A* no dependen de la variable *i*:

```
clear A
d = 0; i = 0;
for i = 1:4
   d = i*2;
   A(i) = d;
end
A
d
i
```

```
clear A
d = 0; i = 0;
parfor i = 1:4
   d = i*2;
   A(i) = d;
end
A
d
i
```

Figura 23: Comparación de variables con for y parfor

Aunque el valor de los elementos de *A* sean iguales, el valor de *d* no es el mismo. En el código de arriba a la izquierda, las iteraciones se ejecutan en secuencia, de modo que después *d* tiene el valor que tenía en la última iteración del bucle. En el código de la derecha donde está el bucle *parfor*, las iteraciones se ejecutan en paralelo, no en secuencia, de modo que sería imposible asignar un valor a *d* después de finalizar este bucle. Esto también se aplica a la variable *i*. Por lo tanto, el comportamiento del bucle *parfor* se define de modo que no afecte a los valores de *d* e *i* fuera del bucle y sus valores seguirán siendo los mismos antes y después del bucle. Así, un código donde resida un *parfor* requiere que cada iteración de este bucle sea independiente de las otras iteraciones y que todo el código que sigue al bucle *parfor* no dependa de la secuencia de iteración del bucle.

e) Clasificación de las variables en el entorno del bucle parfor

Cuando un nombre en un bucle *parfor* es reconocido como una referencia a una variable, se clasifica en una de las categorías que se muestran a continuación. Un bucle *parfor* genera un error si contiene cualquier variable que no puede ser categorizada únicamente en una clase de variable o si las variables violan sus restricciones de categoría al ser clasificadas.

- Variables “Loop”.- sirve como índice del bucle para los arrays.
- Variables “Sliced”.- es un array cuyos segmentos son operados por diferentes iteraciones del bucle.
- Variables “Broadcast”.- es una variable definida antes del bucle, cuyo valor se utiliza dentro del bucle, pero no es asignado dentro del bucle.
- Variables “Reduction”.- es una variable que acumula un valor a través de las iteraciones del bucle, independientemente del orden de la iteración.
- Variables “Temporary”.- es una variable creada dentro del bucle, pero a diferencia de la variable del tipo “sliced”, no estará disponible fuera del bucle.

En el siguiente trozo de código se puede ver un ejemplo de cada tipo de las variables descritas anteriormente:

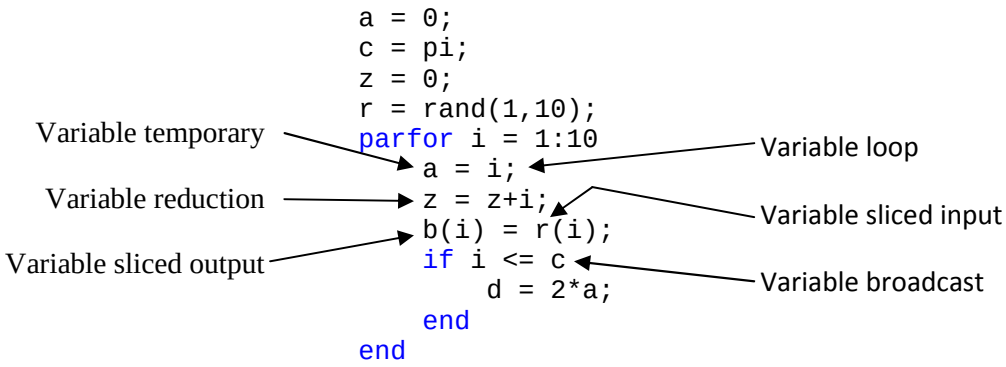

Figura 24: Tipos de variables en un bucle parfor

2.3.3.4. Interactive Parallel mode (pmode)

El modo paralelo interactivo, se ejecuta con el comando *pmode*. El software de Matlab permite trabajar de forma interactiva, con un trabajo desarrollado en paralelo, simultáneamente en varios laboratorios. Los laboratorios son los distintos módulos de trabajo de este modo interactivo, como se puede ver en la figura 25. Los comandos que se escriban en la ventana del *pmode* se ejecutarán en paralelo en todos los laboratorios, al mismo tiempo. Cada laboratorio, ejecuta los comandos en su propio espacio de trabajo con sus propias variables.

Los laboratorios trabajan de forma sincronizada entre ellos. En consecuencia, cuando se ejecuta una orden o instrucción, se espera a que todos los laboratorios que están resolviendo dicha instrucción, hayan finalizado sus operaciones correspondientes. Sólo cuando todos los laboratorios estén parados, se comenzará a ejecutar el siguiente comando tecleado en la ventana de comandos de *pmode*. La visualización de este modo de trabajo interactivo para 4 laboratorios es:

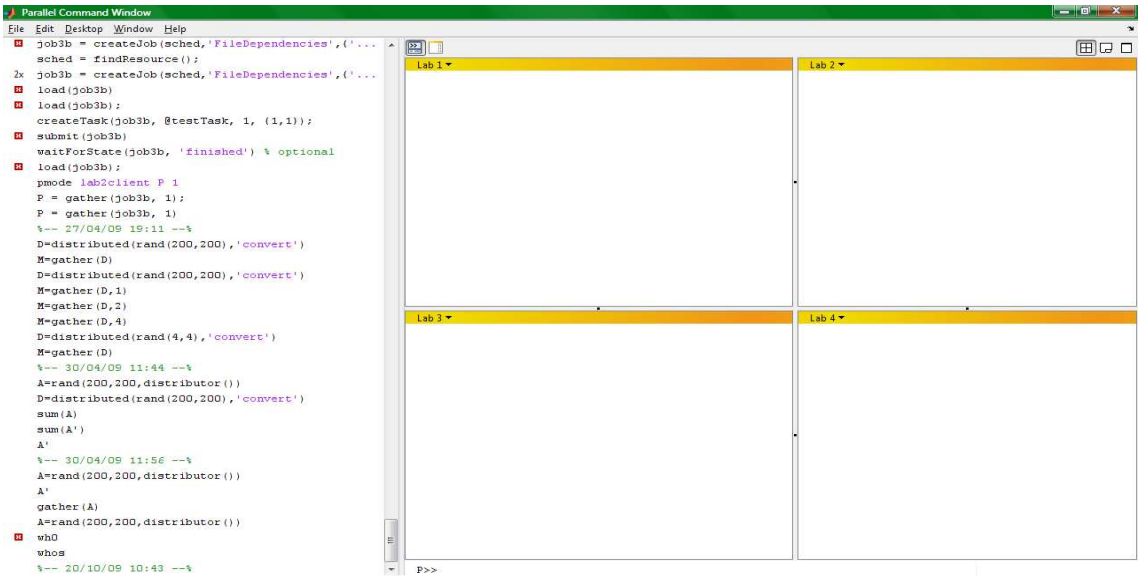

Figura 25: Modo interactivo para la programación en paralelo con matlab (pmode)

Un ejemplo sencillo que ilustraría cómo funciona este modo interactivo sería:

- Introducimos en la línea de comandos: A=rand(200,200, distributor())
- Ejecutamos, y observaremos que en cada uno de los laboratorios o "labs" se crea la cuarta parte de la matriz total.
- Despues podemos operar con dicha matriz, por ejemplo haciendo la matriz transpuesta de A, para ello introducimos en la línea de comandos A=A' y lo ejecutamos.
- Y para que nos muestre la matriz completa, es decir, para juntar todas las partes utilizamos la función gather(A).

2.3.3.5. Evaluación de funciones síncronas y asíncronas en un clúster

En muchos casos, las tareas de un puesto de trabajo son todas iguales, o hay un número limitado de diferentes tipos de tareas en un trabajo. "Parallel Computing Toolbox" ofrece una solución para estos casos que alivia de tener que definir las tareas y trabajos individuales evaluando una función en el clúster. Existen dos formas de evaluar una función en un clúster:

- *Evaluando funciones síncronamente*.-se evalúa una función en el clúster mientras que el cliente de Matlab está bloqueado.

  Para evaluar una función en un clúster de computadoras con *dfeval*, es necesario tener información básica como: la función a ser evaluada, el número de tareas a dividir el trabajo y la variable donde se devuelven los resultados. Una evaluación síncrona (*sync*) significa que la sesión de Matlab quedará bloqueada hasta que la evaluación de dicha función este completada y los resultados son asignados a la variable indicada. Por lo que *dfeval*, proporcionándole la información necesaria, maneja gracias a "Parallel Computing Toolbox", todos los aspectos relacionados con el trabajo de la evaluación de la función, es decir, al ejecutar *dfeval* esta "Toolbox" realiza los siguientes pasos:

  1. Encuentra el "scheduler" o "job manager"
  2. Crea un "job"
  3. Crea las tareas
  4. Envia el trabajo a la cola del "job manager" o del "scheduler"
  5. Recupera los resultados del trabajo
  6. Destruye el trabajo

  Al permitir que el sistema realice todos los pasos para crear y ejecutar un trabajo con una simple llamada a la función, no proporciona la misma flexibilidad que la ofrecida por "Parallel Computing Toolbox". Sin embargo, esta estrecha funcionalidad satisface las necesidades de muchas aplicaciones. La función *dfeval* tiene las siguientes limitaciones:

  - Se pueden pasar propiedades a los objetos de trabajo, pero no se puede establecer cualquier propiedad para una tarea específica.
  - Todas las tareas de un trabajo deben tener el mismo número de argumentos de entrada.

- Todas las tareas de un trabajo deben tener el mismo número de argumentos de salida.
- Si se utiliza un "scheduler" en vez de un "job manager", se debe configurar en la función *dfeval*.
- No se tiene acceso directo al "job manager", ni al trabajo, ni a las tareas, es decir, no hay objetos para manimular en el "workspace" de Matlab. La función *dfevalasync* devuelve un objeto de trabajo o "job".
- Sin acceso a los objetos y sus propiedades, no se tiene control sobre el manejo de errores.

Veamos cómo sería la función *dfeval* para un supuesto caso en el que la función a realizar es *myfun* con tres argumentos de entrada y dos de salida, para ejecutar un trabajo con cuatro tareas:

```
[X, Y] = dfeval(@myfun, {a1 a2 a3 a4}, {b1 b2 b3 b4}, {c1 c2 c3 c4});
```

¿Por qué se devuelven los resultados en dos variables? Porque los resultados de este trabajo serán asignados a dos arrays, X e Y, a causa de la función. Estos arrays tendrán cuatro elementos cada uno, para las cuatro tareas. El primer elemento de X tendrá el primer argumento de la primera tarea y el primer elemento de Y tendrá el segundo argumento de la primera tarea.

La siguiente tabla muestra cómo *dfeval* divide esto en tareas y dónde devuelve los resultados:

Tabla 9: Funcionamiento de la función dfeval

Llamada a la función task	Resultados
myfun (a1, b1, c1)	X{1}, Y{1}
myfun (a2, b2, c2)	X{2}, Y{2}
myfun (a3, b3, c3)	X{3}, Y{3}
myfun (a4, b4, c4)	X{4}, Y{4}

Si no usásemos *dfeval* lo equivalente sería, ejecutar de una en una cada operación, como se puede observar a continuación:

```
[X{1}, Y{1}] = myfun(a1, b1, c1);
[X{2}, Y{2}] = myfun(a2, b2, c2);
[X{3}, Y{3}] = myfun(a3, b3, c3);
[X{4}, Y{4}] = myfun(a4, b4, c4);
```

- *Evaluando funciones asíncronamente*.-se evalúa una función en el clúster mientras que el cliente de Matlab continua, es decir, no queda bloqueado, que es lo contrario que pasaba en el caso anterior.

Si se desea enviar un trabajo al "job manager" y obtener acceso a la línea de comandos mientras el trabajo se ejecuta se tiene que usar la función asíncrona *dfevalasync*. La función *dfevalasync* opera de la misma forma que *dfeval*, excepto que no bloquea la línea de comandos de Matlab y no devuelve directamente los resultados. Para el mismo caso que en el punto anterior (evaluando funciones síncronas) la expresión para ejecutar dicho ejemplo pero en el caso asíncrono sería:

```
job1 = dfevalasync(@averages, 2, c1, c2, c3, 'jobmanager',…
'MyJobManager', 'FileDependencies', {'averages.m'});
```

Donde cabe destacar que el número de argumentos devueltos es 2, como se indica en la función *dfevalasync*.

2.3.3.6. Programando trabajos distribuidos y trabajos paralelos

Un trabajo distribuido es aquel donde las tareas no se comunican entre sí. No es necesario que las tareas se ejecuten simultáneamente y un trabajador puede ejecutar varias tareas de un mismo trabajo.

Los trabajos paralelos son aquellos en los que los trabajadores o workers pueden comunicarse entre sí durante la evaluación de sus tareas.

2.3.3.7. Configuración del clúster-GTTS para trabajar desde Matlab

En esta sección, se va a comprender como se debe configurar el clúster-GTTS, utilizado en el proyecto fin de carrera, para que cuando se ejecute un script desde una máquina local, éste sea enviado y ejecutado en el clúster-GTTS, siempre y cuando en el script también se le indique que se desea que ejecutar en el clúster-GTTS.

Para acceder a Manage Configurations se despliega la lista de opciones del menú Parallel del escritorio de Matlab:

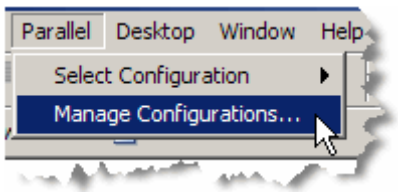

Figura 26: Localización de Manage Configurations en Matlab

Cuando se abre por primera vez Manage Configurations, solo aparece una configuración en la lista de configuraciones creadas llamada *local*, esta es la configuración por defecto:

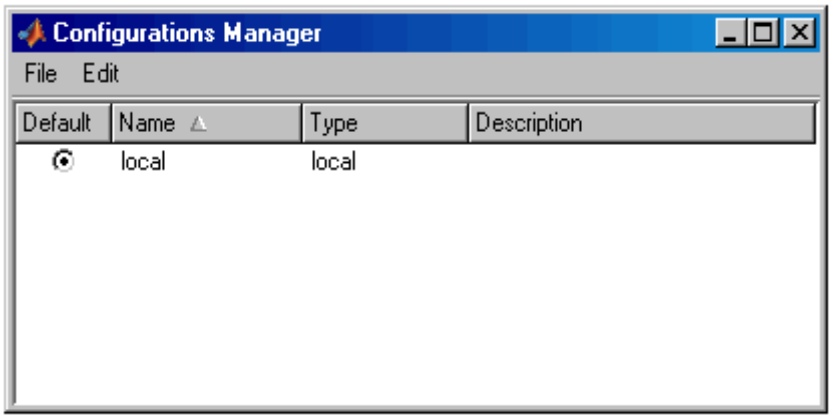

Figura 27: Manage Configurations

Para crear una nueva configuración hay que seleccionar la opción File→New, una vez aquí nos aparecerá una lista de los tipos de "scheduler" posibles a la hora de crear la nueva configuración. El clúster-GTTS posee un "scheduler" de tipo "jobmanager". Una vez elegido el tipo se nos abre una nueva ventana que posee 4 pestañas donde podremos configurar las opciones deseadas, en este caso son:

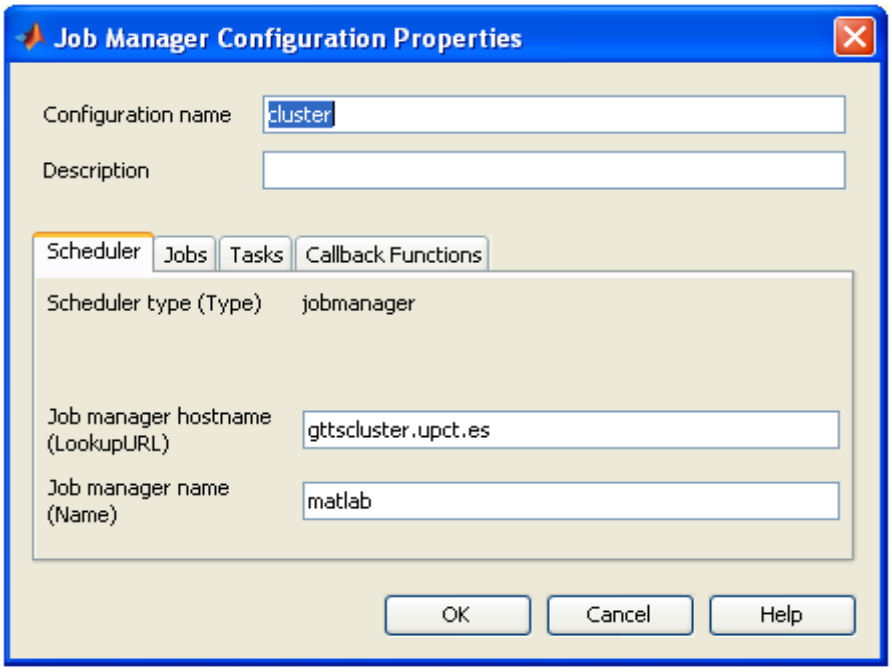

Figura 28: Nueva configuración con un scheduler del tipo jobmanager, pestaña Scheduler

En la parte que representa la figura 28, introducimos el nombre del ordenador que posee el Job manager, en este caso gttscluster.upct.es. En la siguiente figura, en la 29, se tiene que indicar las rutas donde se tienen guardados los scripts en la máquina local, que posteriormente quedrán ser ejecutados en el clúster-GTTS.

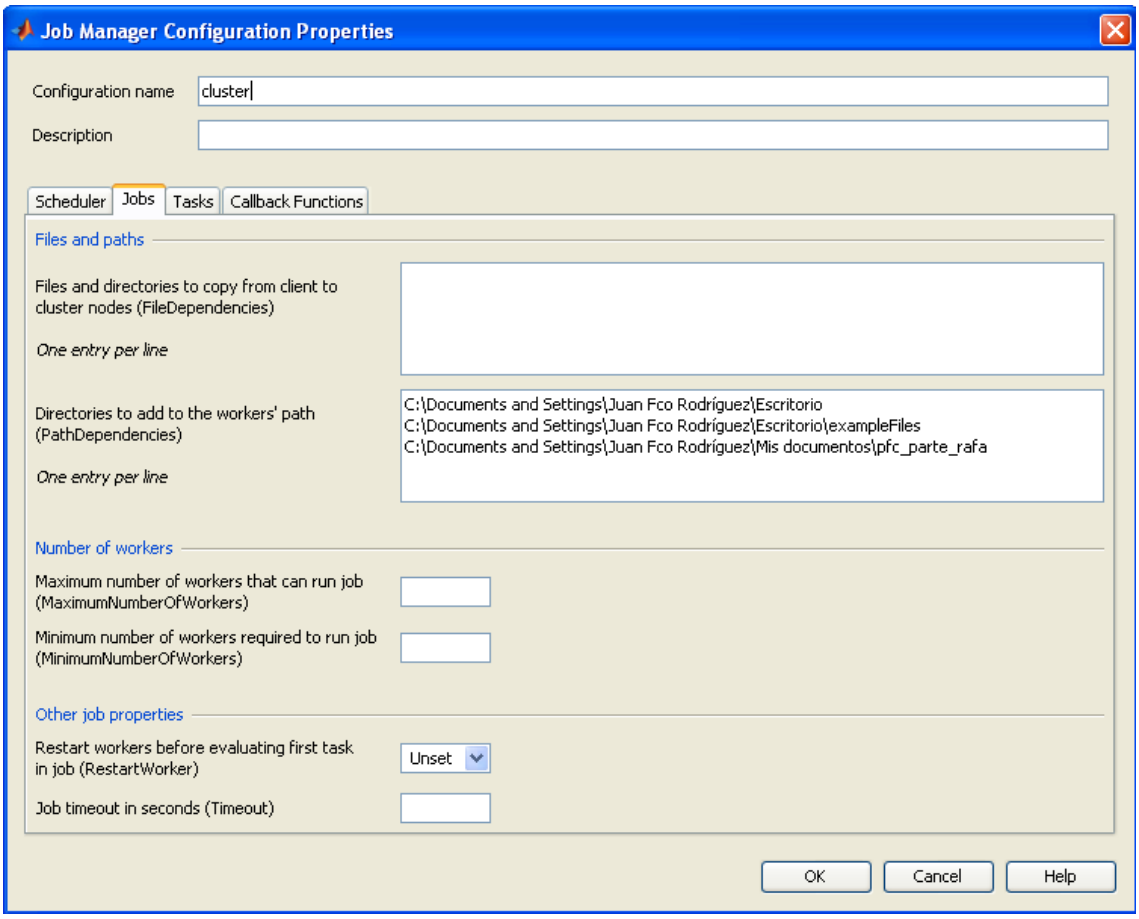

Figura 29: Nueva configuración con un scheduler del tipo jobmanager, pestaña Jobs

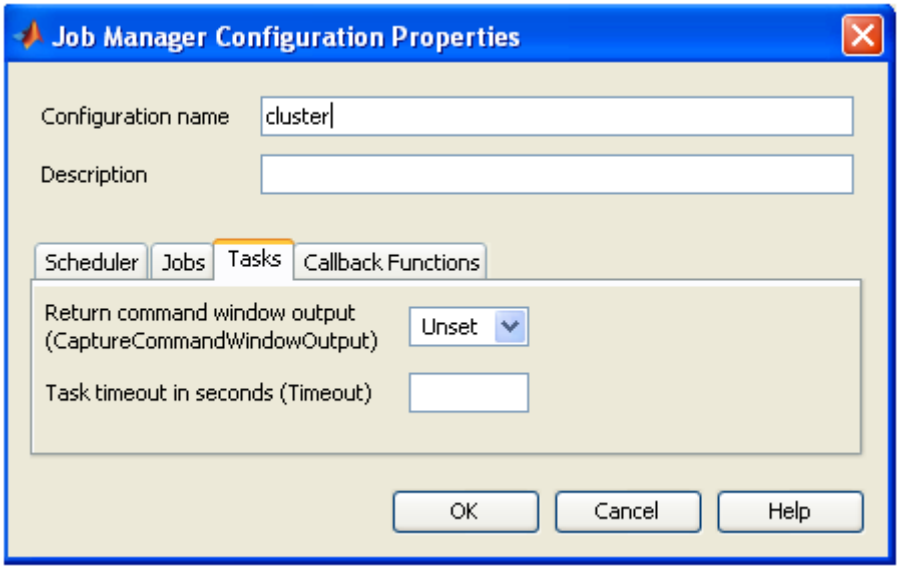

Figura 30: Nueva configuración con un scheduler del tipo jobmanager, pestaña Tasks

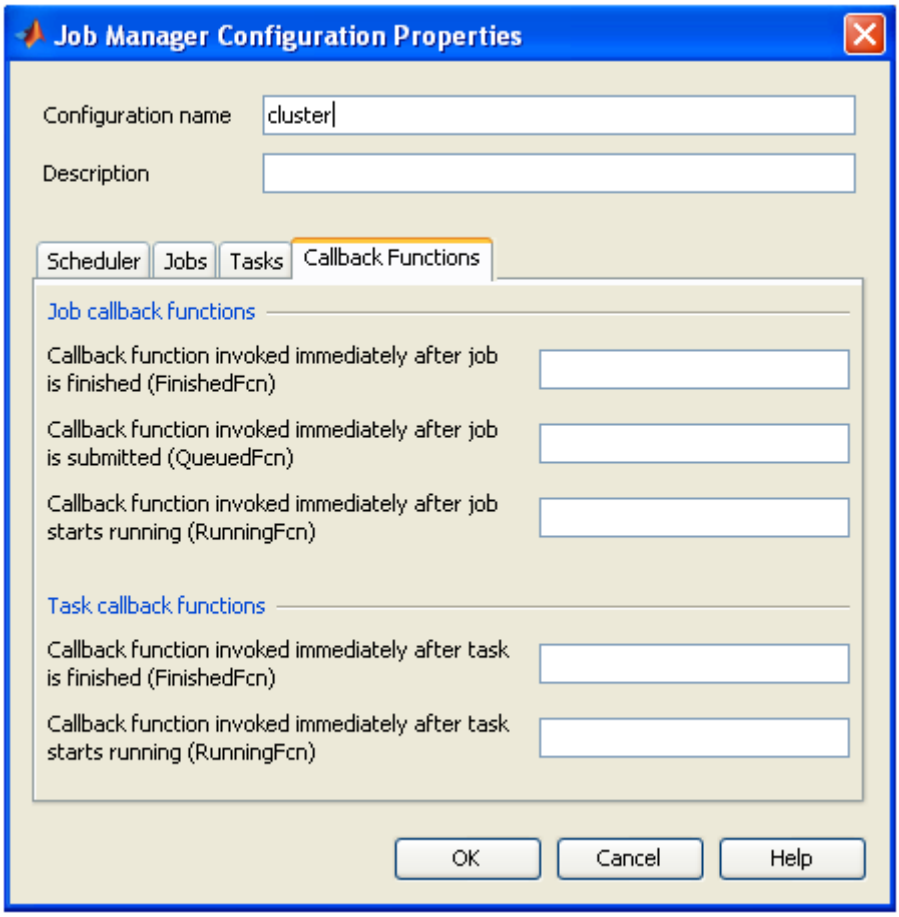

Figura 31: Nueva configuración con un scheduler del tipo jobmanager, pestaña Callback Functions

Una vez hecho todo esto y se pulsado el botón OK nos aparecerá un nuevo nombre en la lista de Configurations Manager llamado cluster. Esta configuración será la que se utilizará en este proyecto fin de carrera.

Para que el clúster-GTTS ejecute los script que se le envían desde una máquina remota, es necesario que dichos script también se encuentren presentes en el clúster-GTTS. Para ello se utiliza el programa WinSCP que nos permite enviar los ficheros deseados desde nuestra máquina cliente al clúster-GTTS. Una vez enviados al clúster-GTTS este replica estos ficheros a cada uno de los nodos a través del sistema de archivos NFS. En la figura 32 podemos ver el aspecto del programa WinSCP. En la parte izquierda del programa se encuentra la máquina cliente y en la derecha el clúster-GTTS. Para pasar archivos de la máquina cliente al clúster basta con arrastrarlos de un lado al otro.

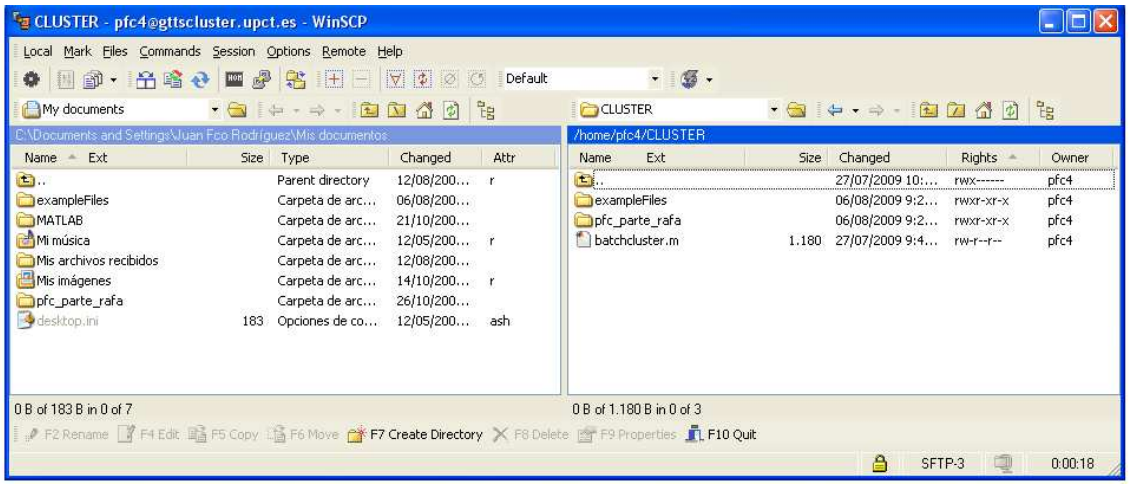

Figura 32: Programa WinSCP

2.4. Resumen

En este capítulo 2, encontramos dos bloques bien diferenciados. Por un lado, se describe el clúster-GTTS, que es el clúster utilizado a lo largo de este proyecto fin de carrera, como se comprobará a lo largo de los capítulo 3 y 4. Por otro lado, se dan a conocer los tres tipos de programación en paralelo existentes que proporciona Matlab: pMatlab, Star-P y Parallel Computing Toolbox. En este proyecto, se utilizará la programación en Matlab paralela asociada a la tecnología de "Parallel Computing Toolbox" como se verá en los dos siguientes capítulos. Por ello la tecnología "Parallel Computing Toolbox" se ha descrito con mayor detalle. Aun así, para más información consultar la referencia [3].

CAPÍTULO 3

IMPLEMENTACIÓN PARALELA CON MATLAB

3.1. Programación en paralelo con Parallel Computing Toolbox de Matlab

En este capítulo se va a explicar cada uno de los ejemplos utilizados para comprender la programación en paralelo con Matlab a través de la librería "Parallel Computing Toolbox". Se trata de ir aprendiendo, en cada uno de los ejemplos, un caso concreto y específico de la programación en paralelo de Matlab, con ejemplos básicos y muy sencillos de entender, para cualquier usuario.

3.1.1. Paralelizando un bucle for

Hay ocasiones en las que es necesario repetir el mismo conjunto de instrucciones muchas veces, cambiando algunos detalles. Para ejecutar este conjunto de instrucciones en Matlab tenemos el comando *for*. Para determinar cuano termina él comando for se utiliza *end.* La forma de utilizar esto se muestra a continuación:

```
for contador=inicio:paso:fin
   sentencias
end
```

Figura 33: Bucle for

Este bucle pone en marcha una variable llamada *contador* que comienza en el valor asignado por la variable *inicio* hasta el valor de la variable *fin* aumentando este valor en cada iteración según el valor indicado en la variable *paso*. Cada vez que las *sentencias*

se ejecutan, *contador* aumenta en un valor igual a la variable *paso* (que si se omite, se le asigna automáticamente el valor uno). Cuando *contador* llega al valor *fin*, el bucle se acaba y el programa continúa con las sentencias que existan más allá de *end*. Así es como funciona el bucle *for* en la programación secuencial. Un ejemplo sería:

```
N = 100;
M = 200;
a = zeros(N,1);

tic;
for i = 1:N
    a(i) = a(i) + max(eig(rand(M)));
end
toc;
```

Figura 34: Ejemplo de un bucle for en serie que será paralelizado

En este ejemplo se crean *N* variables llamadas *a,* cada una diferenciada de las demás por el índice *i,* cada una de estas variables *a* se crean con el bucle *for* y su valor vendrá determinado por el valor del parámetro *i*, que irá aumentando de uno en uno en cada iteración del bucle *for*, desde el valor 1 al 100; la función *rand(M)* devuelve una matriz de *MxM*, donde *M* tiene un valor de 200; la función *eig* lo que hace es devolver un vector de valores propios de dicha matriz y la función *max* elige de entre esos valores el más grande; inicialmente todos los valores de *a(i)* son ceros ya que la sentencia *a = zeros(N,1);* inicializa la variable *a* asignándole una columna de *N* ceros. Las sentencias *tic;* y *toc;* nos devolverán el tiempo que ha transcurrido en ejecutarse el bucle completo.

Para paralelizar este bucle solo hay que hacer los siguientes cambios:

```
N = 100;
M = 200;
a = zeros(N,1);
matlabpool open 2
tic;
parfor i = 1:N
    a(i) = a(i) + max(eig(rand(M)));
end
toc;
matlabpool close
```

Figura 35: Ejemplo de la figura 34 paralelizado

Con estos cambios realizados ya se ha paralelizado el programa, es decir, ahora el bucle *for* se ejecutará de forma diferente al caso anterior.

Cuando se ejecuta la sentencia *matlabpool open* lo que ocurre es que el cliente de Matlab conecta con el número de "workers" que se indiquen, arrancando la configuración en paralelo descrita, en el caso en el que no se especifique la configuración que es, tomará la que tenga asignada por defecto. El número dos indica cuantos "workers" en paralelo se van a ejecutar. Los "workers" o "labs" son los procesos paralelos que se encargan de ejecutar los trabajos que se les envían desde el cliente. Las opciones posibles al ejecutar *matlabpool open* son:

matlabpool open conf poolsize

- *conf* indica la configuración descrita en “Configurations Manager” que se encuentra en Matlab→Parallel→Select Configuration
- *Poolsize* es el tamaño del número de “workers” que se quieren utilizar.

Si sólo se ejecuta *matlabpool open* sin ambas opciones, *conf* y *poolsize*, se ejecutará la configuración por defecto.

La palabra *par* que va delante de *for*, indica que dicho *for* se ejecutará en paralelo, por lo que si hay 100 iteraciones y dos “workers” se ejecutarán 50 iteraciones en cada uno de los “workers” correspondientes, reduciendo así el tiempo de ejecución.

Una vez terminada la ejecución se cierra la configuración en paralelo con la sentencia *matlabpool close*.

Para ver el contraste entre la programación secuencial y la programación en paralelo, solo hay que ejecutar dichos programas. Se verá que el programa secuencial tarda bastante más en terminar de ejecutarse en comparación con el programa en paralelo.

3.1.2. Enviar un script a un clúster-GTTS

La función *batch()* nos permite enviar un script escrito en Matlab para ser evaluado por el clúster-GTTS, es una función abreviada, que nos evita tener que crear los trabajos (“job”) y las tareas (“task”), ya que lo hace de forma automática. Si queremos enviar por ejemplo el siguiente script llamado *testBach.m*:

```
A = rand(100,100);
B = max(A);
```

Figura 36: script testBach.m

La forma de enviar este script al clúster-GTTS y que lo ejecute es:

```
tic
job2a = batch('testBatch', 'configuration','cluster');
wait(job2a);  % only can load when job is finished
toc
sprintf('Finished Running Job')
load(job2a)  % loads all variables back
sprintf('Loaded Variables into Workspace')
% load(job2a, 'A');  % only loads variable A
destroy(job2a)  % permanently removes job data
sprintf('Test Completed')
```

Figura 37: script submitJob2a.m

Los parámetros de la función *batch()* son:

- j = batch ('aScript') → el script aScript.m se ejecutará en el worker definido según la configuración paralela por defecto. La función *batch* devuelve el objeto de trabajo j como identificador de lo que se ha ejecutado del script.

- j = batch (..., 'p1', v1, 'p2', v2, ...) → permite parejas de parámetros-valores que modifican el comportamiento del trabajo. Los parámetros que se aceptan son:

  - 'Workspace'- para definir el workspace del worker justo antes de que se llame al script. El nombre de los campos de la estructura define los nombres de las variables, y los valores de los campos son asignados a las variables del workspace. Por defecto estos parámetros tienen un campo para cada variable en el workspace donde *batch* es ejecutado.
  - 'Configuration'- una única cadena que es el nombre de la configuración paralela usada para encontrar el clúster correcto. Por defecto esta es la cadena devuelta por "defaultParallelConfig".
  - 'PathDependencies' - una cadena o array de cadenas definen una ruta para ser añadido a los workers de Matlab, antes de ejecutar el script.
  - 'FileDependencies' – una cadena o array de cadenas. Cada cadena en la lista identifica cualquiera a un archivo o a un directorio, el cual es transferido al worker.
  - 'CurrentDirectory' –una cadena para indicar en qué directorio se ejecuta el script. Esto no garantiza que este directorio exista en el worker. El valor por defecto para esta propiedad es el *cwd* de Matlab donde el comando *batch* es ejecutado, a menos que "FileDependencies" sean definidas.
  - 'CaptureDiary' – un indicador booleano para indicar que la toolbox debe recoger el diario de la función. Consultar sobre la función diario (diary) para obtener información sobre los datos recogidos. Por defecto es falso.
  - 'Matlabpool' – un entero escalar positivo que define el numero de labs a ejecutar en matlab. El valor por defecto es 0, por lo que el script sólo se ejecutara en un solo worker sin matlabpool.

El resto de funciones que aparecen en el ejemplo tienen el siguiente significado: la función "wait()" bloquea la línea de comandos de Matlab hasta que se devuelven los datos tras realizar todas las operaciones; la función "load()" carga los datos devueltos en el "workspace" de Matlab; la función "destroy()" elimina el "job" o trabajo creado, una vez que ya no hace falta.

3.1.3. Ejecutar un bucle parfor enviándolo al clúster-GTTS a través de un script

Otra forma de ejecutar un bucle *for* en paralelo es utilizando la función "batch()", es decir, si tenemos un script llamado *testParforBach.m*, tal que:

```
N=50;
M=200;

A=zeros(N,1);

parfor i=1:N
    A(i)=A(i)+max(eig(rand(M)));
end
```

Figura 38: script testParforBach.m

Para ejecutar este código en el clúster-GTTS queriendo que solo trabajen dos "workers", utilizando la función "batch()", habrá que escribir un código tal que:

```
tic
job2b =
batch('testParforBatch','configuration','cluster','matlabpool',2);
wait(job2b);  % only can load when job is finished
toc
sprintf('Finished Running Job')
load(job2b);  % loads all variables back
sprintf('Loaded Variables into Workspace')
%load(job2b, 'A');  % only loads variable A
destroy(job2b)  % permanently removes job data
sprintf('Test Completed')
```

Figura 39: script submitJob2b.m

A diferencia del apartado 1.2, en este caso se ejecutará un script que contiene un bucle *parfor*. Como se puede ver se ha incluido un nuevo par de parámetro-valor que son *matlabpool* como parámetro y "2" como valor de dicho parámetro. El resto de código realiza las mismas funciones, espera (a través de la función "wait()") a que el trabajo haya finalizado antes de cargar las variables devueltas con la función "load()". Eliminando finalmente los datos correspondientes al trabajo creado, una vez obtenidas las variables deseadas. Si por ejemplo sólo deseamos cargar una variable en concreto, como podría ser en este caso, la variable *A*, en vez de ejecutar "load(job2b)" que carga todas las variables devueltas del trabajo job2b, ejecutamos "load(job2b, 'A')", cargando así en el "workspace" sólo el valor de la variable *A* devuelta después de ejecutar dicho trabajo.

3.1.4. Ejecutar diferentes tareas en el clúster-GTTS

Para que se ejecuten diferentes tareas en el clúster-GTTS, donde estas tareas ejecutan una función definida de Matlab, se hará lo siguiente:

```
%% This script submits a job with 3 tasks
sched = findResource('scheduler','configuration','cluster');
job3a = createJob(sched);
createTask(job3a, @sum, 1, {[1 1 3]});
createTask(job3a, @sum, 1, {[2 2]});
createTask(job3a, @sum, 1, {[3 3]});
tic
submit(job3a)
waitForState(job3a, 'finished') %optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job3a)
toc
sprintf('Got Output Arguments')
destroy(job3a) % permanently removes job data
sprintf('Test Completed')
```

Figura 40: script submitJob3a.m

En este caso, creamos un trabajo a partir de la función "createJob()", este trabajo contendrá 3 tareas a ejecutar por el clúster-GTTS como se ha indicado en la configuración a través de la función "findResource()". Cada una de estas tareas ejecuta una operación, en este caso una operación suma, con una función predeterminada de Matlab, la función "sum". Los argumentos de entrada para la función "sum", son los que se indican en la cuarta posición, de los parámeros que posee la función task. En el

tercer parámetro de la función task, se indica el número de argumentos de salida, que tendrá dicha operación, en este caso 1. Una vez definido por completo el trabajo a ejecutar se envía dicho trabajo al administrador de trabajo, gracias a la función "submit()", una vez enviado el trabajo esperamos a que termine de ejecutarlo por completo, para ello se utiliza la función "waitFoState()". Para poder recoger los resultados devueltos se utiliza la función "getAllOutputArguments()". Por último, al igual que en los casos anteriores, se elimina el trabajo.

3.1.5. Ejecutar diferentes tareas en un clúster-GTTS cuando la función es definida por el usuario

Para que se ejecuten diferentes tareas en el clúster-GTTS, donde estas tareas ejecutan una función definida por un usuario a través de un script, se hará lo siguiente:

```
% This script submits a job with 3 tasks
sched = findResource('scheduler','configuration','cluster');
job3b = createJob(sched,'FileDependencies',{'testTask.m'});
createTask(job3b, @testTask, 1, {1,1});
createTask(job3b, @testTask, 1, {2,2});
createTask(job3b, @testTask, 1, {3,3});
tic
submit(job3b)
waitForState(job3b, 'finished') % optional
toc
sprintf('Finished Running Job')
results = getAllOutputArguments(job3b);
sprintf('Got Output Arguments')
destroy(job3b) % permanently removes job data
sprintf('Test Completed')
```

Figura 41: script submitJob3b.m

Este caso introduce la peculiaridad, de que las tareas, utilizan una función descrita por un usuario, en vez de una predefinida de Matlab como en el caso del apartado anterior. Para poder utilizar esta función no predefinida de Matlab, se tiene que declarar de alguna forma. A la hora de crear el trabajo, es cuando se le indica al programa cual es la función que se utilizan en dichas tareas. Por lo demás el resto del código realiza las mismas operaciones que él en caso del apartado anterior 1.4.

La función *testTask.m* devolverá la suma de los dos valores de entrada que se le suministren. Esta función, no predefinida en Matlab, se muestra a continuación:

```
function total = testTask(x1,x2)
total = x1+x2;
end
```

Figura 42: script testTask.m

3.1.6. Cuándo usar la función *createJob* y cuándo usar *createMatlabPoolJob*

A continuación se muestra cómo utilizar la función "createMatlabPoolJob()" a diferencia de la función estudiada anteriormente "createJob()". La función "createJob()" como se ha visto se utilizaba para crear un trabajo a partir de una función simple, sin

embargo “createMatlabPoolJob()” crea un trabajo a partir de una función en la cual reside un *parfor*, como lo es la función *testParforJob.m*. En este primer trozo de código se ve el script necesario para crear el trabajo y esperar los resultados del clúster-GTTS:

```
% This script submits a function that contains parfor
sched = findResource('scheduler','configuration','cluster');
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'testParforJob.m'});
createTask(job4, @testParforJob, 1, {});
set(job4, 'MaximumNumberOfWorkers', 4);
set(job4, 'MinimumNumberOfWorkers', 4);
tic
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job4)
toc
sprintf('Got Output Arguments')
destroy(job4) % permanently removes job data
sprintf('Test Completed')
```

Figura 43: script submitJob4.m

Como se puede ver, la forma de enviar el trabajo, esperar a que termine de ejecutarse, de obtener los datos y de eliminar el trabajo creado, se realiza de la misma forma que anteriormente.

Podemos observar que la forma de decir cuántos “workers” son los que van a trabajar para ejecutar este trabajo, se asignan a través de la función “set()”, la cual se utiliza dos veces, una para modificar el parámetro que indica del número máximo de trabajadores y otra para modificar el parámetro que asigna el número mínimo de trabajadores. Si ambos parámetros, son iguales, estamos forzando a la máquina que trabaje con ese número determinado de “workers”. Si por ejemplo ponemos cómo número máximo de “workers” 4 y como mínimo 1 y el clúster-GTTS no tiene ningún otro trabajo ejecutándose, este trabajará siempre con el número máximo, pero si el clúster-GTTS tiene todos los “workers” ocupados, el administrador de trabajo, pondrá en cola el trabajo que enviamos asta que se libere uno de ellos para poder ejecutar este trabajo. Es decir podemos hacer que sea más flexible a la hora de repartir el trabajo en los “workers”.

Aquí tenemos la función que contiene el *parfor*, la función *testParforJob.m*:

```
function A = testParforJob
%% Simple parfor function to illustrate MatlabPoolJob submission

N=50;
M=200;
A=zeros(N,1);
tic

parfor i=1:N
    A(i)=A(i)+max(eig(rand(M)));
end
```

```
toc
```

Figura 44: script testParforJob.m

3.1.7. Cuándo usar la función *createParallelJob*

A diferencia de las dos funciones anteriores, "createJob()" y "createMatlabPoolJob()", esta función "createParallelJob()" sólo admite una sola tarea:

```
%% Script submits a data parallel job, with one task
sched = findResource('scheduler','configuration','cluster');
job5 = createParallelJob(sched);
createTask(job5, @labindex, 1, {});
set(job5, 'MaximumNumberOfWorkers', 4);
set(job5, 'MinimumNumberOfWorkers', 4);
tic
submit(job5)
waitForState(job5, 'finished') % optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job5);
toc
sprintf('Got Output Arguments')
destroy(job5); % permanently removes job data
sprintf('Test Completed')
```

Figura 45: script submitJob5.m

3.2. Resumen

En este tercer capítulo, se ha mostrado el cómo se debe trabajar con la libería "Parallel Computing Toolbox" de Matlab a través de ejemplos muy básicos donde cada uno de ellos representa un caso en concreto. Cada uno de estos casos, representan las situaciones más comunes que se pueden presentar a la hora de programar un código en paralelo con Matlab. El trabajo que realiza "Parallel Computing Toolbox", de una forma u otra queda resumido en los siguientes pasos:

1. Encuentra el "scheduler" o "job manager"
2. Crea un "job"
3. Crea las tareas
4. Envia el trabajo a la cola del "job manager" o del "scheduler"
5. Recupera los resultados del trabajo
6. Destruye el trabajo

En el siguiente capítulo, se analizaran las prestaciones del clúster-GTTS y los resultados que proporciona, este clúster, al ejecutar unas funciones morfológicas y los ejemplos descritos en este capítulo. Todos los códigos de estas funciones y ejemplos están programados en paralelo con Matlab a través de la tecnología "Parallel Computing Toolbox".

CAPÍTULO 4

ANÁLISIS DE PRESTACIONES Y RESULTADOS

4.1. Funciones morfológicas

En este capítulo se va a estudiar el comportamiento de las funciones morfológicas a través de la programación en paralelo y con ayuda del clúster-GTTS, comparándo estos resultados con los obtenidos ejecutando dichas funciones en serie. También se analizarán los ejemplos estudiados en el capítulo anterior. Con todo esto podremos entender de una forma más lógica la manera de trabajar que tiene el clúster-GTTS.

4.1.1. Introducción

De forma general, morfología se refiere al estudio de la forma y de la estructura. En el campo del procesado de la imagen, proporciona una herramienta para la extracción de componentes, útil en la representación y descripción de la forma de una región. La morfología matemática emplea la teoría de conjuntos para representar las formas de los objetos en una imagen. De este modo, las operaciones morfológicas se pueden describir simplemente añadiendo o eliminando píxeles de la imagen binaria original. En las imágenes binarias, los conjuntos pertenecen al espacio 2-D de los enteros (Z^2), donde cada elemento del conjunto es un vector 2-D de coordenadas (x,y).

Una imagen es una representación de 2 dimensiones del mundo visual. Según González [6], una imagen es una función de dos dimensiones f(x,y) de una intensidad de luz, donde x e y se refieren a las coordenadas espaciales y el valor de f en cualquier punto (x,y) es proporcional al brillo (o nivel de grises) de una imagen en cualquier punto.

El procesamiento de imágenes se utiliza para la descripción de operaciones realizadas en imágenes para lograr un propósito [7]. Ya sea para convertir una imagen en una forma que sea más fácil de transmitir mediante una telecomunicación, restaurarla en la memoria de una computadora o extraer solamente cierta información que sea de mayor prioridad en el estudio de algo en particular para alguien.

Algunas de las operaciones principales en el procesamiento de imágenes son el escalamiento, codificación, extracción de características, reconocimiento de patrones, entre otras [8].

Toda la teoría de la morfología matemática se basa en un par de operaciones elementales a partir de las cuales se puede generar cualquier otra operación.

4.1.2. Erosión y Dilatación

Son las dos operaciones morfológicas básicas, a partir de las cuales se definen todas las demás. Siendo *A* y *B* conjuntos de Z^2 , la *erosión* de *A* con respecto a *B* se define:

$$A\Theta\, B=\{X|(B)_x \subseteq A\}$$

es decir, está formada por el conjunto de puntos 'x' que hacen que *B*, trasladado según el vector 'x', esté completamente contenido dentro del conjunto *A*. Otra posible definición es la intersección de todas las traslaciones de *A* con respecto a un vector perteneciente al conjunto $\hat{B}$:

$$A\Theta\, B=\cap\{(A)_b\,|b \in \hat{B}\} \quad (\textit{Definición de Sternberg})$$

Si el conjunto *B* es simétrico con respecto al origen, $\hat{B} = B$ y, entonces, la definición de Sternberg coincide con la *resta de Minkowski*.

La *dilatación* de *A* con respecto a *B* se define:

$$A\oplus B =\{X|(\hat{B})_x \cap A\neq\varnothing\}$$

lo que significa, que está constituida por todos los puntos 'x' tales que al reflejar *B* y luego desplazar con respecto a 'x', el conjunto resultante se solape con *A*, al menos en un punto.

Al igual que en la erosión, existe otra definición alternativa para la dilatación:

$$A\oplus B =\cup\{(A)_b|b\in B\} \quad (\textit{Suma de Minkowski})$$

que representa la unión de todas las traslaciones de *A* con respecto a los puntos que forman *B*.

El conjunto *B* se denomina *Elemento Estructural* y cada punto de los conjuntos es un *píxel* de la imagen. En imágenes binarias, los píxeles tienen sólo dos valores posibles: ON (valor binario 1)/OFF (valor binario 0).

La erosión pone a OFF píxeles que estaban a ON en la imagen original. Un ejemplo de aplicación de la erosión es la eliminación de píxeles que, por razones de umbral (en la segmentación de la imagen) o por ruido, están a ON cuando deberían estar a OFF. La elección de los píxeles a borrar depende de la forma del elemento estructural. El resultado de la operación es la reducción del tamaño de la imagen original. Es lo que se puede comprobar en el siguiente ejemplo, donde se erosiona un triángulo utilizando un disco como elemento estructural.

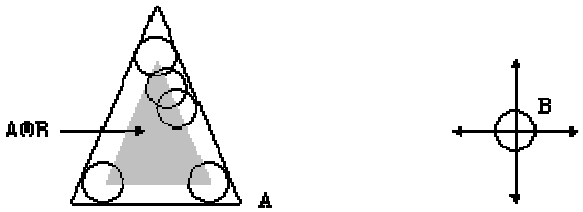

Figura 46: Erosión de la figura A con el elemento estrucutral B

En el caso de la dilatación, la imagen se expande con respecto a la original, lo que implica la puesta a ON de píxeles que originalmente estaban en OFF. Como ejemplo, se muestra la dilatación de un triángulo mediante un disco.

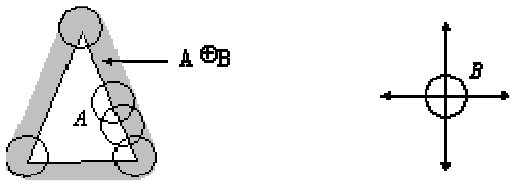

Figura 47: Dilatación de la figura A con el elemento estrucutral B

La erosión más simple (*erosión clásica*) es aquella en la que se eliminan los píxeles que tocan a alguno que es parte del fondo de la imagen. Esto supone eliminar una línea de píxeles alrededor del borde de los objetos de la imagen. De forma análoga, la mecánica de la *dilatación clásica* es añadir a un objeto (poner en ON) aquellos píxeles que tocan a alguno del objeto en cuestión. Es decir, se expandirá una línea de píxeles alrededor de la periferia del objeto de la imagen original.

Dos de las propiedades de la erosión y de la dilatación son:

- Ambas son operaciones monótonamente crecientes con respecto a un elemento estructural dado.
  Como la dilatación es conmutativa, también será monótonamente creciente con respecto a una imagen de entrada. Para la erosión no se cumple.
  De aquí se concluye que es el tamaño y forma del elemento estructural lo que determina el mayor o menor crecimiento/reducción de la imagen.
- Erosión y dilatación son operaciones duales.
  Según esto, erosionar una imagen equivale a la dilatación de su complemento, es decir, la erosión del objeto que está en primer plano de la imagen equivale a la

dilatación del fondo de la misma. Un razonamiento similar se puede aplicar a la dilatación: expande la imagen en primer plano y encoge su fondo.

A continuación se muestran dos ejemplos de cómo se dilata y erosiona una imagen a partir de un elemento estructural que se muestra, junto con la operación que se está realizando:

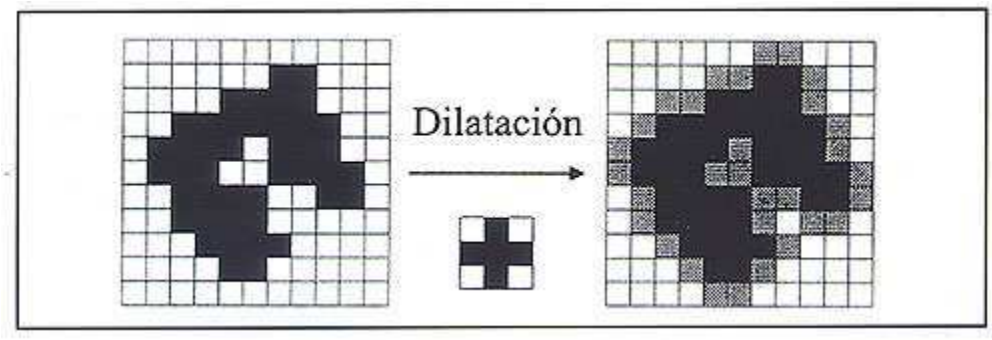

Figura 48: Ejemplo de dilatación

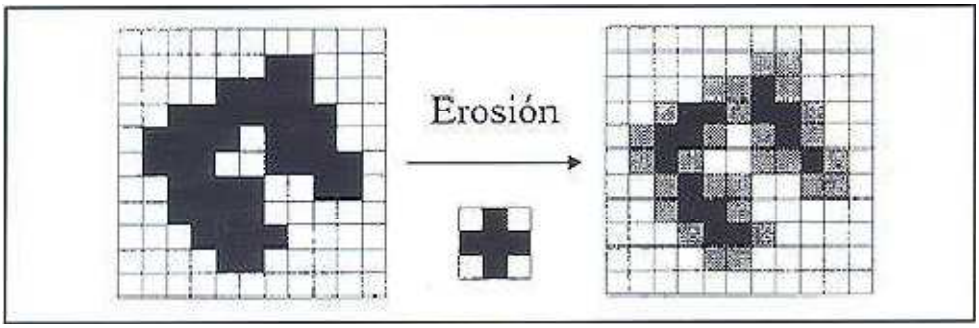

Figura 49: Ejemplo de erosión

4.1.3. Erosión y Dilatación en escala de grises

Dentro del algoritmo las imágenes son tratadas en escala de grises, y no en color. En computación, una escala de grises es una escala empleada en las imágenes digitales en las que el valor de cada píxel corresponde a una graduación de gris. Las imágenes representadas de este tipo están compuestas de sombras de grises, que van desde el negro más profundo variando gradualmente en intensidad de grises hasta llegar al blanco.

A continuación se van a definir la erosión y la dilatación para el procesado de imágenes en escala de grises en términos de máximos y minimos a través de la vecindad de los píxeles.

Se define una imagen en escala de grises como una función de dos dimensiones f(x,y) donde los valores de ‘x’ e ‘y’ determina un pixel en concreto de la imagen.

La dilatación en escala de grises de f por el elemento estructural b, denotada como f⊕b, se define como:

$$(f \oplus b)(x,y) = \max\{f(x-x',y-y')+b(x',y') | (x',y') \in D_b\}$$

donde D_b es el dominio de b. Esta formulación se define como el máximo de las translaciones de f por los elementos pertenecientes a b, es decir, el valor de dilatación de un píxel (x,y) es el máximo de la imagen en la ventana de vecindad definida por el elemento estructurante cuando su origen se sitúa en (x,y).

La erosión en escala de grises de f por el elemento estructural b, denotada como fΘb, se define como:

$$(f\Theta b)(x,y)= \min\{f(x+x',y+y')-b(x',y')|(x',y')\in D_b\}$$

donde D_b es el dominio de b. Esta formulación se define como el mínimo de las translaciones de f por los elementos pertenecientes a b, es decir, el valor de la erosión de un píxel (x,y) es el mínimo de la imagen en la ventana de vecindad definida por el elemento estructurante cuando su origen se sitúa en (x,y).

4.1.4. Apertura y Cierre

Son dos operaciones secundarias de gran importancia, que usan como base las operaciones erosión y dilatación. Sus definiciones para imágenes binarias e imágenes en escala de grises son las mismas. La *apertura* de una imagen *f* con un elemento estructural *b* se define como la combinación de una erosión seguida por una dilatación:

$$f \circ b=(f\Theta b)\oplus b$$

es decir, la apertura de un objeto de la imagen de entrada está formada por la unión de todas las traslaciones del elemento estructural que estén completamente contenidas dentro del objeto en cuestión.

La operación de apertura puede verse como si *b* se deslizara por el interior del objeto, sin rotar, haciendo el borde de *f* como tope para el avance de *b*. Si el elemento estructural tiene un borde suave (p. ej. un disco), entonces el efecto de la apertura es la eliminación de los salientes y picos en los que no cabe el elemento estructural. De este modo, si se aplica la apertura a un triángulo, siendo *b* un disco (cuyo centro coincide con el origen), el resultado de la operación es el mismo triángulo pero con las esquinas redondeadas. Cualquier otro elemento estructural produciría un efecto distinto, pero con la característica común de que la superficie resultante sería menor a la del objeto original.

En general, el cierre de un objeto, con un disco como elemento estructural, redondea las esquinas y entrantes que se extienden hacia el interior del objeto en cuestión. Es como si el disco se deslizara por la parte externa del borde del objeto. Se define como una dilatación seguida de una erosión:

$$f \bullet b=(f\oplus b)\Theta b$$

Algunas propiedades básicas de la apertura y el cierre son:

- La apertura disminuye el área del objeto, mientras que el cierre la amplia.

- Apertura y cierre son operaciones duales.
- Tanto la apertura como el cierre son operaciones "*idempotentes*".
  Esta propiedad es particularmente interesante ya que implica que sucesivas aperturas o cierres de una imagen con el mismo elemento estructural no altera el resultado. Este tipo de transformaciones se corresponden con los filtros paso banda ideales (no realizables) del filtrado lineal convencional, en el cual una vez que una imagen ha sido idealmente filtrada, posteriores filtrados paso banda no modifican el resultado.

4.1.5. Funciones morfológicas

En este proyecto fin de carrera las operaciones de erosión, dilatación, apertura y cierre, están implementadas en las funciones SVMMerode, SVMMdilate, SVMMopen y SVMMclose, respectivamente, como se pueden observar en el anexo.

Existe un script desde el que se llaman a estas funciones. En este script se realizan previamente unas operaciones novedosas que se encargan de buscar las direcciones vectoriales asociadas a una imagen, donde el elemento estructural será diferente en cada situación durante el procesado de dicha imagen, obteniendo unos resultados donde se resalta el relieve de la imagen dada [1,2].

Acontinuación se va a analizar todas las funciones referentes a operaciones morfológicas a través del script *scriptstarrynight.m*. Las distintas operaciones a realizar son las de erosión que se realizará con la función *SVMMerode.m*, la dilatación a través de la función *SVMMdilate.m*, la apertura con *SVMMopen.m* y el cierre con la función *SVMMclose.m*. Cada una de estas funciones tiene el objetivo de modificar una imagen, para poder destacar distintos matices de dicha imagen, dependiendo de las necesidades del usuario. Estas funciones en un principio han sido proporcionadas por Rafael Verdú, escritas con programación en serie a través de Matlab. En este proyecto fin de carrera, se han modificado, estas funciones, reescribiéndolas con programación en paralelo en Matlab, para comprobar las ventajas en este tipo de programación. Para el procesado de imágenes, se requiere una gran capacidad de cálculo, ya que una imagen puede estar compuesta por cientos o miles de píxeles, que hay que recorrer, para poder procesarla.

Este tipo de programación paralela nos ayudará de forma muy eficaz, como se va a ir comprobando a continuación con ayuda del clúster del laboratorio GTTS de la universidad. La imagen de entrada que se va a proporcionar al script *scriptstarrynight.m*, va a ser la siguiente. Es una imagen en escala de grises. Conforme se vaya utilizando una función u otra para procesar esta imagen se irán viendo los cambios surgidos en ella.

También se irán comprobando los tiempos que el clúster-GTTS tarda en procesar esta imagen dependiendo del número de "workers" utilizado en cada una de las distintas operaciones que se le pueden hacer.

Figura 50: Imagen original, sin procesar

4.1.5.1. Análisis de la erosión en paralelo

En este apartado del capítulo vamos a estudiar cómo se comporta la función *SVMMerode.m* a través del script *scriptstarrynight.m* que se ejecutará en paralelo en el clúster del laboratorio GTTS. Para ver el comportamiento del clúster-GTTS al trabajar en modo paralelo iremos cambiando el número de "workers" con el que se desea ejecutar el script para ir viendo que los resultados devueltos en cada caso son idénticos pero con diferentes tiempos en las ejecuciones debido a los diferentes factores que intervienen y los cambios que se realizan.

En la siguiente tabla se muestran los diferentes tiempos de ejecución para unos "workers" determinados, junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del clúster-GTTS.

Tabla 10: Tiempos de ejecución con la erosión

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	140,7	196,1	150,2	114,2	108,6	103,5

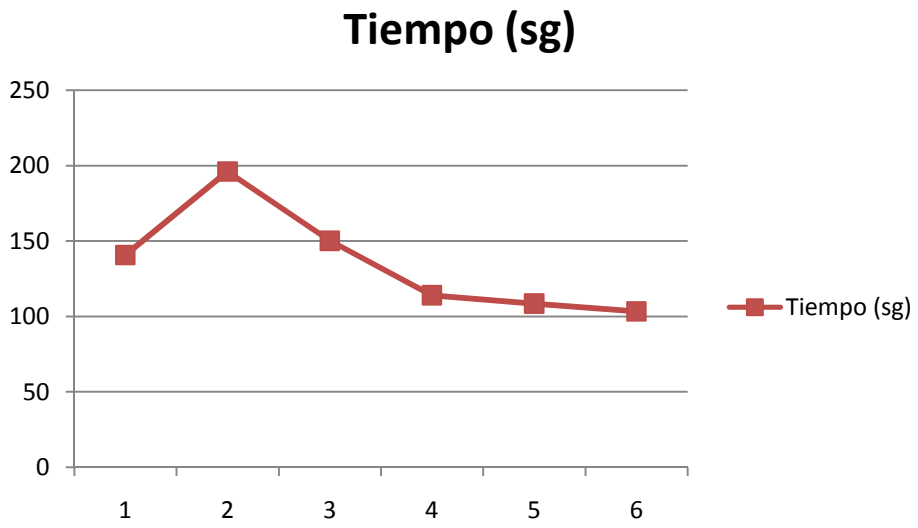

Figura 51: Gráfico de los tiempos de ejecución frente al número de workers usados, en la erosión

Se puede ver en el gráfico de tiempos frente al número de "workers" que están trabajando, que los tiempos de ejecución van decreciendo conforme se va aumentando el número de "workers", sin embargo cuando se utilizan 2 "workers" tarda más tiempo en ejecutar la función *SVMMerode.m* que cuando trabaja con un solo "workers", esto es debido a que el tiempo que tardan los datos en recorrer la red, es decir, el tiempo de red, no es favorable ya que tarda más tiempo en distribuir el trabajo entre los "workers" y devolver los resultados que lo que tardan dichos "workers" en realizar la ejecución, sin embargo este tiempo (el tiempo de red) ya no es tan significativo cuando se utilizan 4 "workers", ya que la rapidez de ejecución es tan grande que por mucho que tarden los datos en distribuirse y devolver los datos sigue siendo menor que si trabajásemos con menos "workers". Imaginemos un ejemplo, para comprender esto mejor, en el que se trabaja con 2 y 4 "workers": imaginemos que el tiempo de red, es decir, el que se tarda en distribuir el trabajo cuando se trabaja con 2 "workers" es de 20 segundos y el de ejecución en los "workers" de una función dada es de 40 segundos, tendríamos un tiempo total de 60 segundos para la ejecución de dicha tarea. Después al ejecutar esta misma operación con 4 "workers"el tiempo de red seria un poco mayor ya que el trabajo hay que dividirlo entre más "workers" supongamos que unos 30 segundos, pero el tiempo de ejecución se vería reducido prácticamente a la mitad por lo que los "workers" tardarían en ejecutar la operación unos 20 segundos, por lo que en total tardaría unos 50 segundos, reduciendo así el tiempo total de procesado. Con esto también podemos observar que trabajar con 4 "workers" y trabajar con 2 "workers" no implica exactamente reducir en la mitad el tiempo de procesado.

Esto también depende de cuánto trabajo tiene que repartir el "job manager" o administrador de trabajo, si no hay muchas operaciones que repartir, este tardará menos tiempo, por lo que el tiempo de red se verá disminuido.

En programas con un tiempo de ejecución bastante grande el tiempo de red es menos significativo y con un número de “workers” elevado el tiempo de ejecución es menor.

Como podemos ver el tiempo en ejecutar una función depende de la combinación de varios factores, que son el número de operaciones a ejecutar y el número de “workers” con el que se trabaja, de estos dependen el tiempo de red y el tiempo de procesado.

En los casos en los que se está procesando un bucle *parfor*, se sabe que las iteraciones se ejecutan unas independientes de otras, por lo que no hace falta que exista comunicación entre los “workers” o estaciones de trabajo, por lo que el tiempo de distribución en estos casos es nulo. El problema es que no se ha podido paralelizar el doble bucle *for* sino que solo se ha paralelizado el bucle *for* mas interno, ya que en este caso las iteraciones si dependen unas de otras, por esta causa no se puede paralelizar el bucle más exterior. Con esto podemos concluir que en este caso al no poder paralelizar los dos bucles, solo se paraleliza el bucle interno, influirá a la hora de la obtención del tiempo de procesado total de la función de erosión, obteniendo así los valores que nos devuelve el clúster-GTTS, que son observados en la tabla dada.

En serie con un ordenador de las siguientes características:

Tabla 11: Caracteristicas de un ordenador

Procesador	Intel ® Core™2 Duo CPU P8400 @ 2.25 GHz 2.27 GHz
Memoria (RAM)	4,00 GB
Tipo de sistema	Sistema operativo de 32 bits

tarda 467,8 segundos en ejecutar estas operaciones a la imagen de entrada.

El resultado obtenido una vez ejecutada dicha función es el que se muestra en la figura siguiente. Este resultado es idéntico al que se obtendría si se ejecutara este programa en serie, la única diferencia es que se ha obtenido en un tiempo mucho más reducido.

Figura 52: Imagen erosionada

4.1.5.2. Análisis de la dilatación en paralelo

En este caso vamos a analizar cómo se comporta la función *SVMMerode.m* programada en paralelo a través del script *scriptstarrynight.m* que se lanzará en el clúster del laboratorio GTTS de la universidad para su ejecución. Para ver el comportamiento del clúster-GTTS con esta función se irán modificando el número de "workers" utilizados para la ejecución de esta operación morfológica. Todos los resultados que sean obtenidos con los diferentes "workers" son iguales pero vemos que los tiempos en cada ejecución, dependiendo del número de "workers" usado, son diferentes.

En la siguiente tabla se muestran los diferentes tiempos de ejecución para unos "workers" determinados, junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del clúster-GTTS.

Tabla 12: Tiempos de ejecución con la dilatación

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	140,8	201,5	144,9	119,8	101,8	100,0

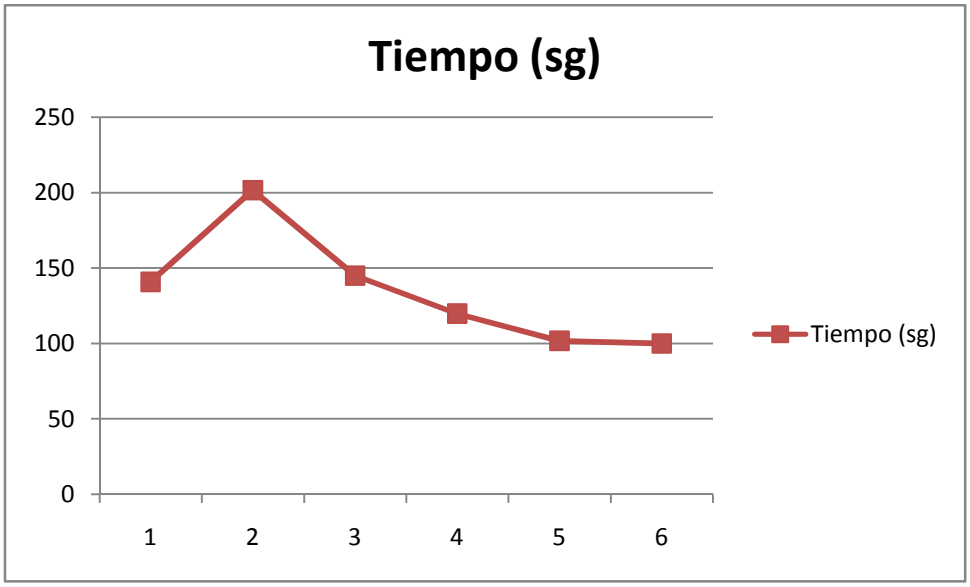

Figura 53: Gráfico de los tiempos de ejecución frente al número de workers usados, en la dilatación

Como en el caso anterior se ha tenido que paralelizar un bucle *for* que no es el más exterior de los bucles anidados que hay en el código de la función *SVMMdilate.m*. En serie, con un ordenador de las características de las de la tabla 11, la ejecución e esta función tarda 484,3 segundos. Como vemos se reduce considerablemente el tiempo en ejecución en paralelo en cualquier caso, en comparación de cuando se ejecuta en serie.

En este caso de la dilatación, se observa que los tiempos son muy similares a los tiempos obtenidos en el apartado anterior de la erosión. Esto es debido a que los códigos de dichas funciones son muy similares como se puede apreciar en el Anexo I, del capítulo 6.

El resultado obtenido tras ejecutar dicha función es el que se muestra en la siguiente figura. Este resultado es idéntico del que se obtendría si se ejecutara este programa en serie, la única diferencia es que se ha obtenido en un tiempo mucho más reducido.

Figura 54: Imagen dilatada

4.1.5.3. Análisis de la apertura en paralelo

En este apartado se va a estudiar el comportamiento del clúster-GTTS con la función *SVMMopen.m* programada en paralelo ejecutando el script *scriptstarrynight.m*. Para poder analizar el comportamiento del clúster-GTTS en más en profundidad se ejecutará dicho script, varias veces, modificando el número de "workers" utilizados. Cabe destacar que aunque se han ido modificando el número de "workers" usados, el resultado obtenido ha sido exactamente el mismo, lo único que ha cambiado en cada ejecución ha sido el tiempo de procesado.

En la siguiente tabla se muestran los diferentes tiempos de ejecución para unos "workers" determinados, junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del clúster-GTTS.

Tabla 13: Tiempos de ejecución de la apertura

Nº de workers	1	2	3	4	5	6
Tiempo(sg)	35641,6	44278,9	26306,8	18275,1	13192,7	11712,9
Tiempo (h)	9,9	12,3	7,31	5,08	3,67	3,25

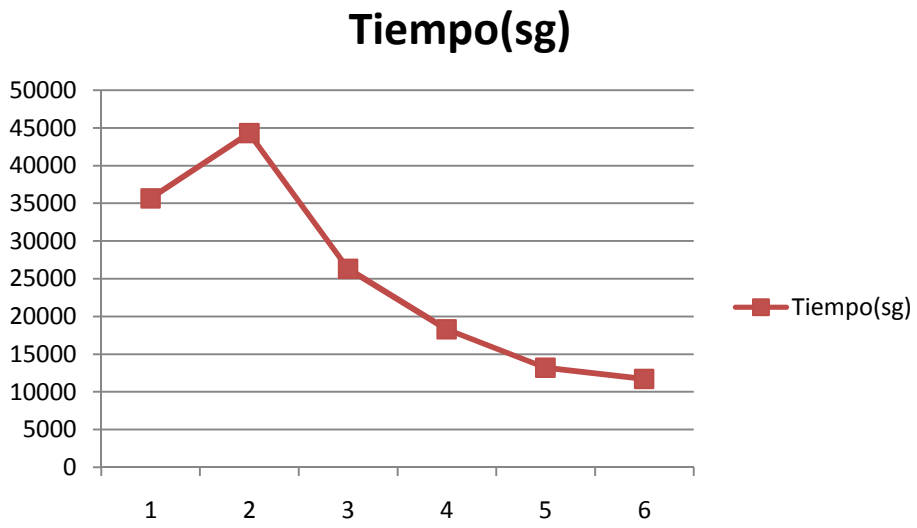

Figura 55: Gráfico de los tiempos de ejecución frente al número de workers usados, en la apertura

Como se puede ver a diferencia de los casos anteriores, aquí los tiempos de simulación son bastante más elevados, ya que el procesado de apertura es más elaborado y contiene más operaciones a realizar. En este caso no vamos a tener tiempo asociado a la comunicación entre "workers" ya que el *parfor* se ha podido insertar en el bucle más externo posible.

En estas ejecuciones se ve que conforme aumentamos el número de "workers" el descenso en el tiempo de ejecución va disminuyendo prácticamente de forma lineal. ¿Por qué aumenta el tiempo de ejecución al pasar de trabajar con un worker al trabajar con dos y no se reduce a la mitad que es lo que se razonaría antes de hacer ninguna ejecución? Al igual que ocurría en el caso de la erosión, esto es debido a que el tiempo que tardan los datos en recorrer la red para ser ejecutados no es favorable ya que tarda más tiempo en distribuir el trabajo entre los "workers" y devolver los resultados, que lo que tardan dichos "workers" en realizar la ejecución, sin embargo este tiempo (el tiempo de red) ya no es tan significativo cuando se utiliza un cierto número de workers, en este caso a partir de 3 "workers", ya que la rapidez de ejecución es tan grande que por mucho que tarden los datos en distribuirse y devolver los datos, sigue siendo menor que si se trabajase con menos "workers".

Estas ejecuciones son realizadas forzando al clúster-GTTS a trabajar con el número de "workers" indicado, es decir, en el código de las funciones se ha dicho que como mínimo y como máximo trabaje con un mismo número de "workers", forzando al clúster-GTTS que no trabaje ni con mas ni con menos de los "workers" deseados, pero qué ocurre si por ejemplo ponemos como mínimo 1 "worker" y como máximo que pueda utilizar 6 "workers". Para ello se ha realizado una simulación con este caso y se ha visto que los resultados son prácticamente iguales que si forzamos a trabajar al clúster-GTTS con 6 "workers", cabe decir que el clúster-GTTS no tenía ningún otro trabajo ejecutándose, por lo que este se ha dedicado en exclusiva, al estar libre, a la ejecución de esta función. El tiempo de simulación teniendo como mínimo un

"workers" y como máximo 6 es de 11716,824826 segundos frente a los 11712,914625 segundos que tarda en ejecutar forzando al clúster-GTTS a trabajar con 6. Como se puede ver la diferencia en ambos casos es prácticamente inexistente ya que simplemente se varía un resultado del otro en un 0,03% del tiempo, siendo un poco más lenta la simulación en la que no se fuerza que el clúster-GTTS trabaje con 6 "workers". De esto podemos concluir que el si el clúster-GTTS no tiene ningún trabajo pendiente y está libre, este utiliza el mayor de su potencial para que el trabajo enviado, sea ejecutado con la mayor eficiencia posible de forma automática, dentro del rango establecido.

El resultado obtenido una vez ejecutada esta función es el que se muestra en la figura siguiente. Este resultado es idéntico al que se obtendría si se ejecutara este programa en serie, la única diferencia es que se ha obtenido en un tiempo mucho más reducido.

Figura 56: Imagen procesada con la apertura

4.1.5.4. Análisis del cierre en paralelo

En esta sección del capítulo se va a analizar el comportamiento del clúster-GTTS con la función *SVMMclose.m* programada en paralelo ejecutando el script *scriptstarrynight.m*. Para poder estudiar el comportamiento del clúster-GTTS en más en profundidad se ejecutará este script, varias veces al igual que en los casos vistos hasta ahora, modificando el número de "workers" utilizados. Aunque se han ido modificando el número de "workers" usados, el resultado obtenido ha sido exactamente el mismo, lo único que ha cambiado en cada ejecución ha sido el tiempo de procesado, como observamos en la siguiente tabla que recoge el tiempo de procesado de cada una de las ejecuciones realizadas para los diferentes "workers".

En la siguiente tabla se muestran los diferentes tiempos de ejecución para unos "workers" determinados, junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del clúster-GTTS.

Tabla 14: Tiempos de ejecución de la cerradura

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	36100,8	44874,3	30910,6	18646,9	13101,9	11780
Tiempo (h)	10,03	12,47	8,59	5,18	3,64	3,27

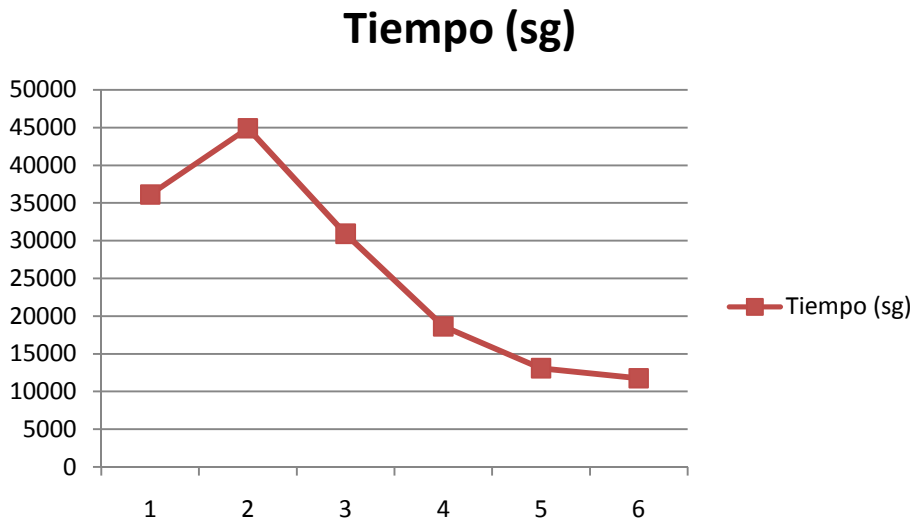

Figura 57: Gráfico de los tiempos de ejecución frente al número de workers usados, en la cerradura

Si observamos los códigos de las funciones *SVMMopen.m* y *SVMMclose.m* se ve que la función *SVMMclose.m* se sirve de la función *SVMMopen.m* para realizar sus operaciones por lo que los tiempos de ejecución son muy similares, siendo los de la función *SVMMclose.m* un poco más elevados ya que realiza algunas operaciones más.

Como se observa, trabajar con el clúster-GTTS, es un modo muy eficiente a la hora de reducir tiempo, en este caso, en el procesado de imágenes, llegando a procesar una imagen en poco más de tres horas al utilizar todos los "workers" del clúster-GTTS.

Para poder hacernos una idea de la eficiencia del clúster-GTTS a la hora de trabajar en paralelo frente a trabajar en serie con un ordenador normal de las características que se muestran en la tabla 11, se ha ejecutado el script *scriptstarrynight.m* haciendo uso de la función *SVMMopen.m*, observamos que para el caso en el que **se ejecuta con el clúster-GTTS en paralelo con 6 "workers",** se obtienen los resultados de procesar una

imagen **en 3,25 horas**, y para el caso del **ordenador Core2Duo** que ejecuta **en serie** este mismo programa con la misma imagen, **tarda 28,4 horas** en ejecutarlo. Como se puede ver la diferencia es abismal.

Figura 58: Imagen tras el proceso de la cerradura

4.2. Ejemplos

En esta segunda parte del capítulo, se da a conocer el comportamiento de los ejemplos en el clúster-GTTS, aunque su misión principal en este proyecto final de carrera es más bien el dar al usuario el conocimiento suficiente para poder programar en paralelo con Matlab.

4.2.1. Resultado de enviar un script a través de la función *batch* al clúster-GTTS

En esta sección se va a analizar cómo se comporta la función "batch()" en el clúster-GTTS a través del ejemplo creado en el script *SubmitJob2a.m*. Al igual que en el resto de casos anteriores, se irán cambiando el número de "workers" usados desde uno hasta seis que es el número máximo de "workers" que posee el clúster-GTTS, para ir viendo cómo van cambiado el tiempo de procesado de este ejemplo.

En la tabla siguiente se muestran los diferentes tiempos de ejecución obtenidos para los distintos "workers", junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 15: Tiempos de ejecución del ejemplo submitJob2a.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	7,339631	7,101522	3,831176	3,798970	3,954024	0,783414

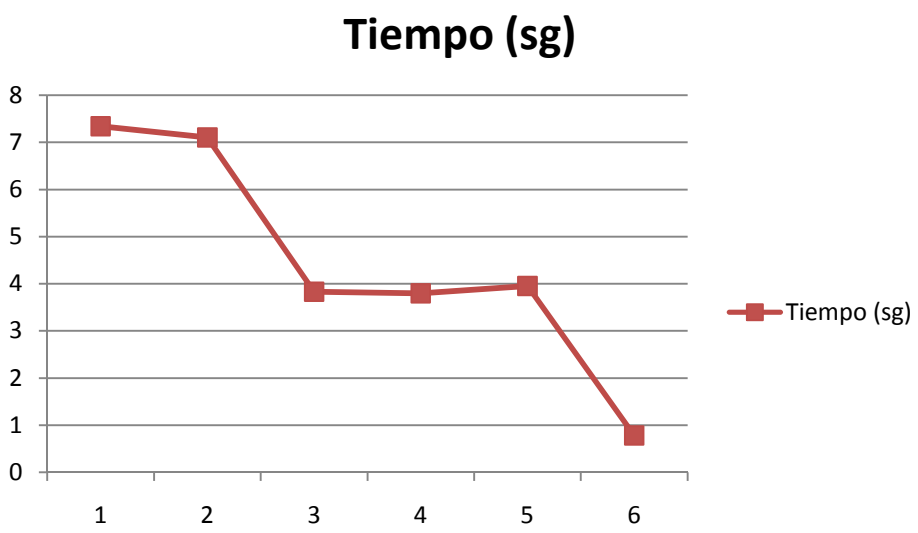

Figura 59: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob2a.m

Una vez realizadas estas ejecuciones podemos decir que el script que contiene la función "batch()" se va viendo ejecutado cada vez en un periodo más reducido de tiempo conforme se va aumentando el número de "workers" con el que trabaja el clúster-GTTS.

4.2.2. Resultado de enviar un script que contiene un bucle parfor a través de la función *batch* al clúster-GTTS

En este apartado se va a ejecutar el script que contiene el ejemplo *submitJob2b.m*, con los diferentes "workers" que posee el clúster-GTTS, para poder así comparar los tiempos de ejecución. Este ejemplo trabaja, al igual que el ejemplo anterior con la función "batch()", pero en este caso el script que se envía a través de esta función contiene un bucle *parfor*. Como sabemos la finalidad que tiene la función "batch()" es la de enviar un script para ser ejecutado en el clúster-GTTS.

En la tabla siguiente se muestran los diferentes tiempos de ejecución obtenidos para los distintos "workers", junto con una pequeña gráfica de esta tabla donde podremos sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 16: Tiempos de ejecución del ejemplo submitJob2b.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	6,512657	5,626535	5,117437	4,846196	4,776871	0,761752

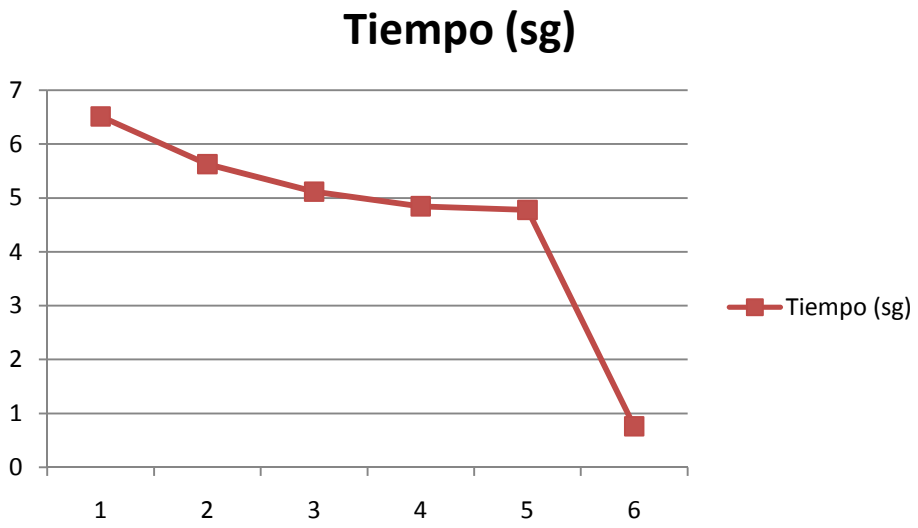

Figura 60: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob2b.m

Como se puede observar tanto mirando en la tabla como a través de la gráfica, los tiempos de ejecución del script que contiene el bucle *parfor* enviado al clúster-GTTS gracias a la función "batch()", se van viendo reducidos conforme aumentamos el número de "workers" tal y como era de esperar. Obteniendo una gran reducción del tiempo de procesado de esta función cuando se utilizan todos los "workers" del clúster-GTTS.

4.2.3. Resultado de enviar unas tareas para ser ejecutadas en el clúster-GTTS

En este apartado del capítulo se va a ejecutar el ejemplo *submitJob3a.m* a través del clúster-GTTS, para poder ver su comportamiento en él. Este ejemplo tiene la finalidad de dar a entender cómo se definen tareas para que el clúster-GTTS las resuelva. Estas tareas, en este caso en concreto, utilizan funciones predefinidas de Matlab. En este ejemplo se utiliza la función "createJob()" para la creación del trabajo que será enviado al clúster-GTTS.

En la siguiente tabla se muestran los diferentes tiempos de ejecución obtenidos con el clúster para los distintos "workers", junto con una gráfica de esta tabla donde podremos sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 17: Tiempos de ejecución del ejemplo submitJob3a.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	0,863743	0,497606	0,460808	0,041420	0,039531	0,043400

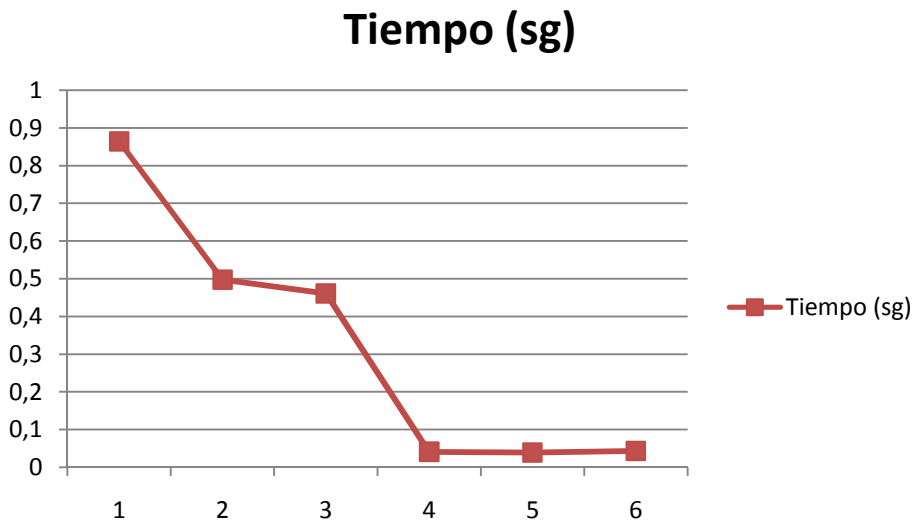

Figura 61: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob3a.m

Tras realizar las ejecuciones aumentando el número de "workers" vemos que el tiempo de ejecución de este ejemplo va disminuyendo. Observamos una gran reducción del tiempo de procesado cuando se pasa de estar trabajando con 3 "workers" a trabajar con 4. Esto se debe a que en este caso se está trabajando con 3 tareas y cuando estas se reparten entre más "workers" que tareas existen, el tiempo de comunicación se ve drásticamente reducido.

4.2.4. Resultado de enviar unas tareas, donde las funciones son definidas por el usuario, para ser ejecutadas en el clúster-GTTS

En esta sección se va a analizar el script que contiene el ejemplo submitJob3b.m, con los diferentes "workers" que tiene el clúster-GTTS, comparando posteriormente los tiempos que tarda este ejemplo en ser ejecutado con los distintos "workers" asignados. Este ejemplo tiene una finalidad parecida al ejemplo anterior, que es la de definir unas tareas para ser ejecutadas en el clúster-GTTS, la diferencia es que ahora las tareas se crean a partir de funciones definidas por el usuario en vez de funciones predefinidas por Matlab, que era lo que ocurría en el caso del ejemplo submitJob3a.m. En este ejemplo, al igual que en el anterior, se utiliza la función "createJob()" para la creación del trabajo que será enviado al clúster-GTTS.

En la siguiente tabla se muestran los diferentes tiempos de ejecución obtenidos con el clúster-GTTS para los distintos "workers", junto con una gráfica de esta tabla donde después se podrán sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 18: Tiempos de ejecución del ejemplo submitJob3b.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	0,812750	0,529908	0,545988	0,024861	0,024856	0,024027

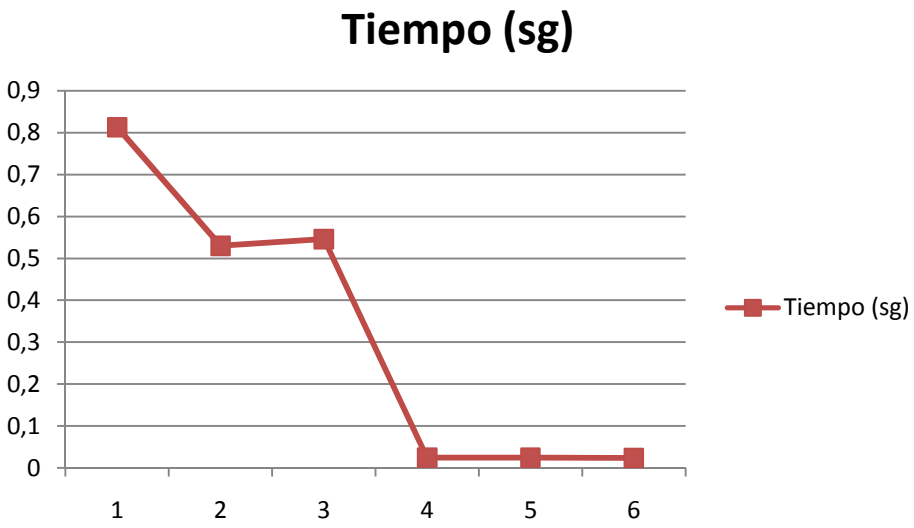

Figura 62: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob3a.m

Observando los resultados obtenidos tras ejecutar este ejemplo aumentando el número de "workers", podemos decir que conforme se van aumentando el número de "workers" utilizados el tiempo de ejecución va disminuyendo, con un gran descenso cuando se pasa de 3 a 4 "workers", esto es debido a que en este caso se trabajan con tres tareas, al igual que en el caso anterior.

4.2.5. Resultado de un ejemplo, con el que se crea un trabajo a partir de la función *createMatlabPoolJob*, para ser ejecutado por el clúster-GTTS

En este apartado se va a estudiar el ejemplo SubmitJob4, cuya peculiaridad reside en la utilización de la función "createMatlabPoolJob()" a la cual se le pasa un script que contiene un *parfor*. En este ejemplo, como en los demás será ejecutado aumentando el número de "workers" para ver el comportamiento que tiene en el clúster-GTTS.

En la siguiente tabla se muestran los diferentes tiempos de ejecución obtenidos con el clúster-GTTS para los distintos "workers", junto con una gráfica de esta tabla donde se podrán sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 19: Tiempos de ejecución del ejemplo submitJob4.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	3,416021	5,788659	4,970493	4,327126	4,077706	4,101698

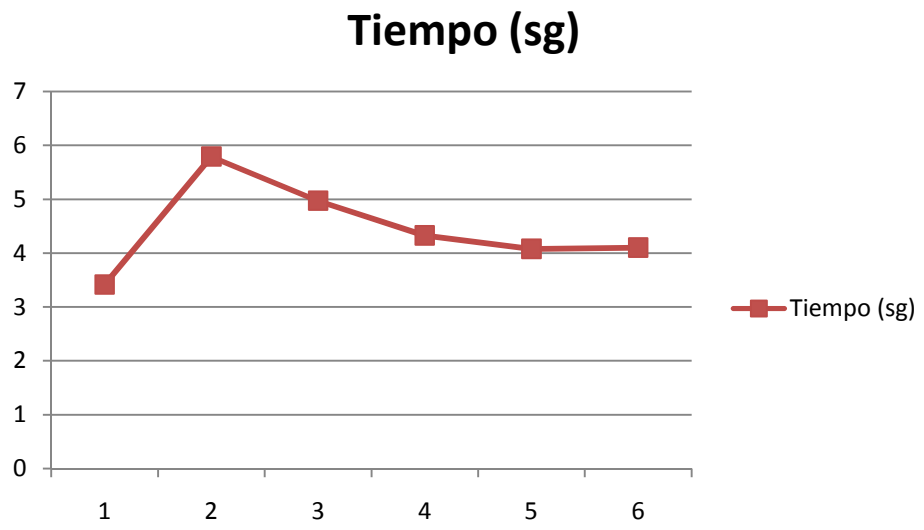

Figura 63: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob4.m

Si nos fijamos en el código de este ejemplo podremos observar que el trabajo creado sólo posee una tarea para ejecutar en el clúster-GTTS. Como bien sabemos el clúster-GTTS está compuesto por 6 “workers”, donde el administrador de trabajo es el que se encarga de repartir el trabajo que le llega a estos “workers”. Cuando el trabajo que llega al administrador de trabajo está compuesto por tareas, este reparte las tareas, asignando una tarea a cada “worker”, si existen más tareas que “workers”, algunos “workers” poseerán más de una tarea. Pero, ¿qué ocurre cuando existe una sola tarea y se le exige que trabaje con más de un “worker”? Que es lo que ocurre en este caso. Pues lo que ocurre es que el clúster-GTTS trabaja de forma irregular tardando más tiempo del necesario para resolver dicho trabajo, como vemos en la gráfica el tiempo más óptimo es cuando se trabaja con un solo “worker”. También se observa que esto no ocurría en el caso en el que se trabajaba con la función “createJob()” (esta función crea el trabajo que será enviado al clúster-GTTS), sin embargo tanto con la función que se utiliza en este ejemplo, la función “createMatlabPoolJob()”, como con la función “createParallelJob()” que se utiliza en el siguiente ejemplo, el número de tareas influye según el número de “workers”.

4.2.6. Resultado de un ejemplo, con el que se crea un trabajo a partir de la función *createParallelJob*, para ser ejecutado por el clúster-GTTS

En esta sección se va a analizar el ejemplo SubmitJob5. Este ejemplo se basa en la utilización de la función "createParallelJob()". Esta función crea un trabajo y a ese trabajo solo se le puede asignar una tarea. Como hasta ahora se va a ejecutar este ejemplo aumentando el número de "workers" para ir viendo el tiempo que tarda en procesarlo en cada caso.

En la siguiente tabla se muestran los diferentes tiempos de ejecución obtenidos con el clúster-GTTS para los distintos "workers", junto con una gráfica de esta tabla donde se podrán sacar conclusiones acerca del comportamiento del clúster-GTTS.

Tabla 20: Tiempos de ejecución del ejemplo submitJob5.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	0,412234	0,501546	0,710270	0,733552	0,715180	0,851683

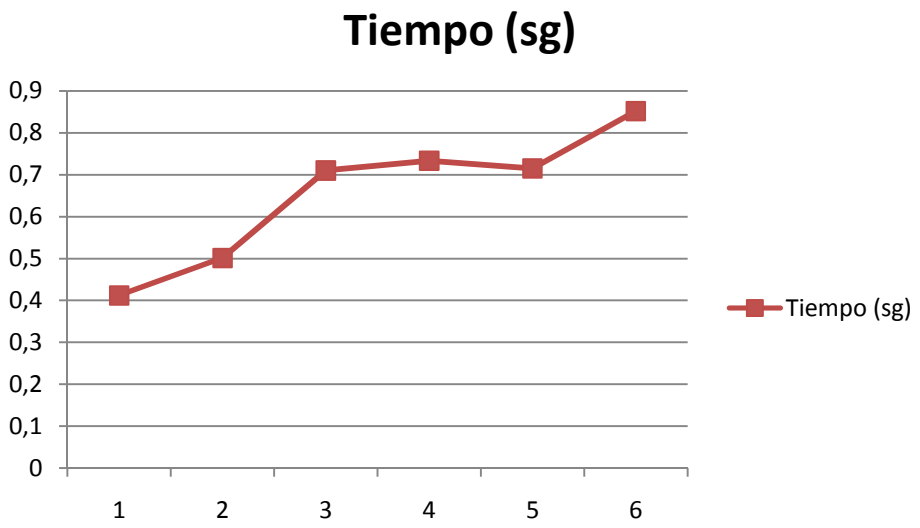

Figura 64: Gráfico de los tiempos de ejecución frente al número de workers usados, en submitJob5.m

En este ejemplo observamos que los tiempos son muy irregulares, ya que lo lógico sería que el tiempo de procesado de este ejemplo fuese disminuyendo conforme se van aumentando el número de "workers" con los que trabaja el clúster-GTTS, pero si nos fijamos en este ejemplo, sólo se puede crear una tarea para un trabajo, por lo que si no hay más de una tarea no se pueden repartir entre los diferentes "workers", ya que una tarea será asignada a un "worker", y si hay mas tareas que "workers", algunos "workers" se le asignaran más de una tarea. Al no poder dividirse esta tarea en operaciones más pequeñas cuando se utilizan más de un "worker" para resolver esta

operación el clúster-GTTS no opera de la forma que debería, como podemos observar en la línea de tiempo que se muestra en la gráfica. Por ello el mejor valor de entre los 6 tomados es cuando se trabaja con un solo "worker" ya que sólo existe una sola tarea a resolver.

4.2.7. Resultados de la ejecución de un bucle parfor

En este caso se estudia el comportamiento de un bucle *parfor* más sencillo que los bucles *parfor* mostrados con las operaciones morfológicas. En este caso no hay comunicación entre los "workers" ya que las operaciones son independientes entre sí. Para la ejecución de este ejemplo también se ha ido aumentando el número de "workers" con el que se realizaban las simulaciones y se han obtenido los datos que se muestran en la siguiente tabla.

Tabla 21: Tiempos de ejecución del ejemplo parforExample1.m

Nº de workers	1	2	3	4	5	6
Tiempo (sg)	6,664699	2,872960	2,242581	1,762906	1,397387	1,223873

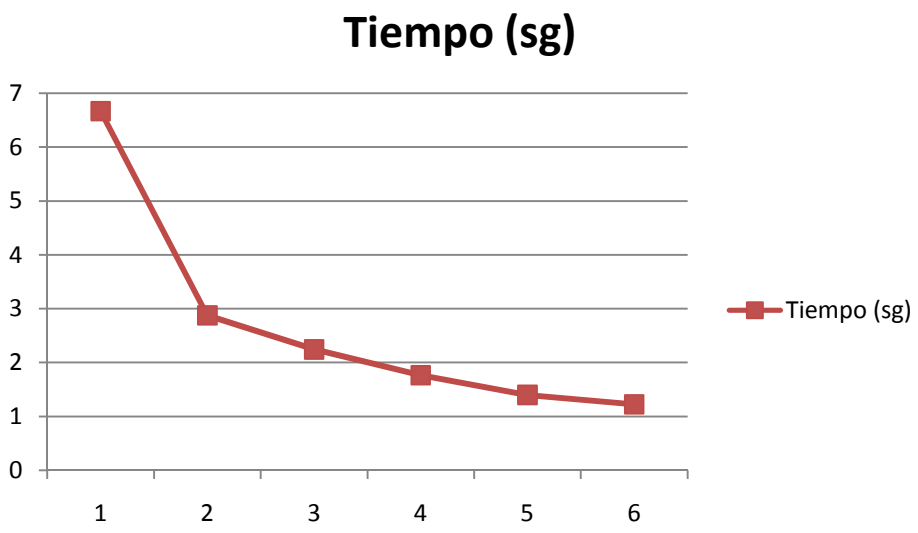

Figura 65: Gráfico de los tiempos de ejecución frente al número de workers usados, en parforExample1.m

Como podemos ver en este ejemplo, los tiempos van decreciendo conforme vamos aumentado el número de "workers", es un ejemplo prácticamente ideal.

Este último ejemplo también se ha procesado en serie y su tiempo de ejecución ha sido de 9,2327 segundos frente a los 1,2238 que se consiguen al procesar este ejemplo, parforExample1.m, en paralelo trabajando con los 6 "workers" del clúster-GTTS. Como

podemos ver nos ahorramos unos 8 segundos en una ejecución en la que el tiempo no es muy elevado.

4.3. Resumen

En este capítulo cuatro, se analiza el comportamiento del clúster-GTTS mediante la ejecución de diferentes funciones y ejemplos. Todo el trabajo que realiza el clúster-GTTS es totalmente opaco al usuario, es decir, el usuario solo sabe que ha enviado un trabajo al clúster-GTTS y le devuelve los resultados. Por ello en este capítulo se intenta dar a conocer, a través, de multiples ejecuciones con diferentes programas el funcionamiento de dicho clúster. Como se ha podido observar, trabajar con un clúster a la hora de procesar grandes cantidades de datos, nos ahorra mucho tiempo ya que su capacidad de procesamiento es muy elevada a la hora de compararla con un simple ordenador. Y si a ello le sumamos la programación en paralelo, el tiempo en procesar un programa en éste clúster, se ve drásticamente reducido. Así, podemos concluir que trabajar conjuntamente con el clúster-GTTS y la programación en paralelo que nos aporta Matlab, con su tecnología “Parallel Computing Toolbox”, es un gran avance para mejorar los tiempos de ejecución, tan importantes hoy en día.

CAPÍTULO 5

CONCLUSIONES Y LÍNEAS FUTURAS

5.1. Conclusiones

En esta sección del capítulo, se van a exponer las conclusiones que se han sacado a lo largo de todo el proyecto fin de carrera. Estas conclusiones, se han estructurado en función de los diferentes capítulos, ya que en cada uno de ellos, se resuelven problemas específicos:

En el primer capítulo, simplemente encontramos el planteamiento del proyecto, dando a conocer un poco al usuario qué es la programación en paralelo, y una pequeña introducción para comprender por qué se ha usado un clúster en este proyecto fin de carrera en vez de un ordenador corriente.

En el capítulo 2, encontramos dos bloques bien diferenciados, donde se describen las partes hardware y software que hacen referencia al clúster-GTTS y lo programas necesarios para trabajar con él, utilizando programación en paralelo. Por un lado, se han descrito las características más importantes del clúster-GTTS (es el clúster utilizado a lo largo de este proyecto fin de carrera), así como el aspecto de cada de las partes de la que está compuesto este clúster. Por otro lado, en el capítulo 2, también se han dado a conocer los tres tipos de programación en paralelo existentes que proporciona Matlab, describiendo su funcionamiento y aportando ejemplos aclaratorios. Estos tres tipos de programación en paralelo que nos ofrece Matlab son: pMatlab, Star-P y Parallel Computing Toolbox. En este proyecto, se ha utilizado la programación en Matlab

paralela asociada a la tecnología de “Parallel Computing Toolbox” como se puede comprobar en los capítulos 3 y 4. Por ello, la tecnología “Parallel Computing Toolbox” se ha descrito con mayor detalle que las otras dos tecnologías, en el segundo capítulo. Se ha elegido esta tecnología por su flexibilidad a la hora de poder reescribir un código en serie ya existente a un código en paralelo.

En el tercer capítulo, se ha mostrado de forma más detallada, cómo se deben implementar programas en paralelo con Matlab. Con ello, se ha conseguido, permitir a un usuario, que desee conocer esta tecnología, comprender mejor el funcionamiento de este tipo de programación basada en la librería “Parallel Computing Toolbox”. Para esto, se ha mostrado, con detalle, cómo se debe trabajar con la libería “Parallel Computing Toolbox” de Matlab, explicando su funcionamiento con ayuda de ejemplos muy básicos donde cada uno de ellos representa un caso en concreto. Cada uno de estos casos, representan las situaciones más comunes que se pueden dar a la hora de estar programando un código en paralelo con Matlab.

En el capítulo cuatro, se ha analizado el comportamiento del clúster-GTTS (clúster perteneciente al departamento Grupo de Teoría y Tratamiento de la Señal, que nos proporciona la Universidad Politécnica de Cartagena) mediante la ejecución de diferentes funciones y ejemplos. Las funciones han sido proporcionadas por Rafael Verdú, habiendo sido modificadas debidamente, en este proyecto fin de carrera, para que estas funciones pudiesen ser ejecutadas en paralelo, ya que los códigos fuentes de las funciones que han sido proporcionados, estaban escritos en serie. Todo el trabajo que realiza el clúster-GTTS es totalmente opaco al usuario, es decir, el usuario solo sabe que ha enviado un trabajo al clúster-GTTS y éste le devuelve los resultados. Por ello, en el capítulo cuatro se ha dado a conocer, a través, de múltiples ejecuciones con diferentes programas, el funcionamiento de dicho clúster. Como se ha podido observar, trabajar con un clúster a la hora de procesar grandes cantidades de datos nos ahorra mucho tiempo, ya que su capacidad de procesamiento es muy elevada a la hora de compararla con un simple ordenador. Y si a ello le sumamos la programación en paralelo que nos ofrece Matlab, el tiempo en procesar un programa en éste clúster, se ve drásticamente reducido, como se ha podido ir comprobando a lo largo del capítulo cuatro. También se ha comprobado el tiempo que se tardaría en ejecutar algunos de estos programas en serie, para tener una mayor visión, del tiempo que se puede ahorrar utilizando este tipo de programación. Así, podemos concluir que trabajar conjuntamente con el clúster-GTTS y la programación en paralelo que nos aporta Matlab, con su tecnología “Parallel Computing Toolbox”, es un gran avance para mejorar los tiempos de ejecución, tan importantes hoy en día.

5.2. Líneas futuras

En esta sección del capítulo cinco, se va a desarrollar una idea que podría llevarse a cabo en un futuro, para la ampliación de este proyecto fin de carrera. La idea es la siguiente:

Implementación de un simulador, que simule el funcionamiento de un clúster, a través de Simulink. Implementando cada una de sus partes en bloques diferentes y en el que se puedan añadir y quitar "workers" y clientes, dependiendo del clúster deseado. Los bloques a desarrollar serian: Cliente de matlab, Job Manager y worker. Donde la visualización de los bloques finalinalizados serían:

Una vez implementados estos bloques, se podrá formar el simulador deseado. Imaginemos que queremos tener dos ordenadores trabajando en el clúster, el cual posee tres workers, su estructura sería la de la figura 66. Estos bloques deben estar programados de forma que en el cliente se le pueda introducir un script el cual será procesado por el clúster y los resultados puedan ser vistos también desde el cliente. El Job Manager deberá de encargarse de distribuir el trabajo entre los worker, así como coordinar las operaciones entre los distintos workers. Y los workers serán los encargados de ejecutar cada una de las operaciones encomendadas por el Job Manager, devolviéndole los resultados.

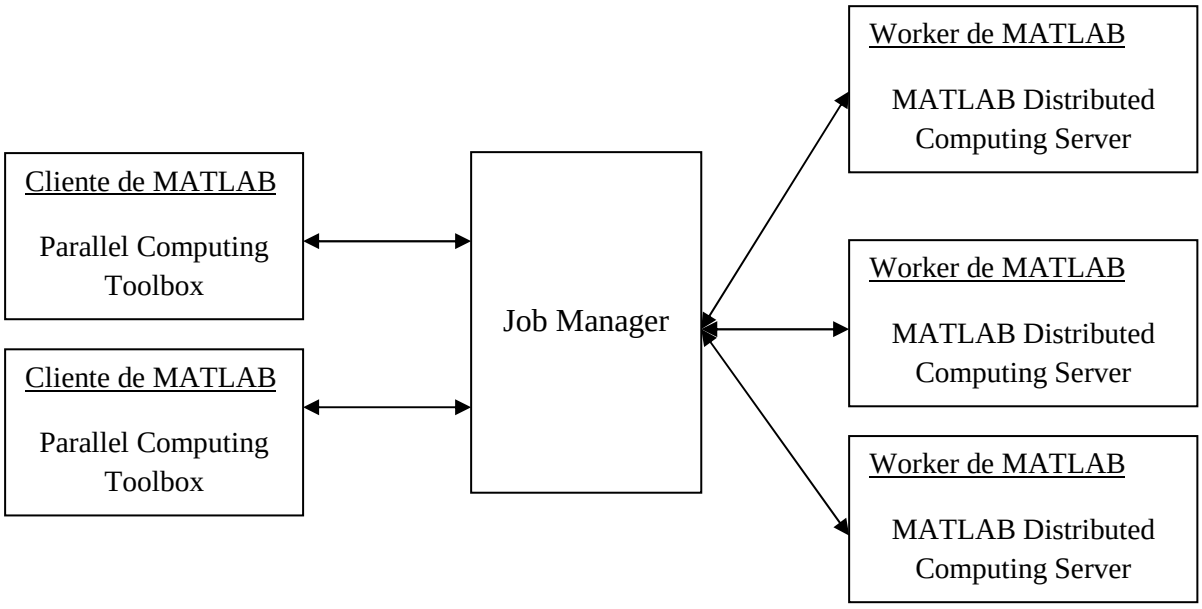

Figura 66: Simulador de un clúster, a partir de bloques simulink.

ANEXOS

En esta sección se encuentran todos los códigos morfológicos de los programas mencionados a lo largo de los capítulos, así como las características de algunos módulos del clúster-GTTS. Los códigos de los programas serán incluidos en la primera parte del anexo (Anexo I) y las características del clúster-GTTS en la segunda parte (Anexo II).

1. Anexo I

Se van a exponer los códigos tanto en serie como en paralelo de las operaciones morfológicas utilizadas a lo largo del proyecto final de carrera.

NOTA: en los scriptstarrynight.m ya sean en serie o en paralelo, se utilicen unas funcione u otras las operaciones que se realizan en los apartados de Imagen, ASG (Averaged Square Gradient), ASGVF y las operaciones morfológicas, por lo que sólo se mostrarán estas operaciones en el script del punto 1.1, en los demás se omitirá.

1.1. Código de la función erosión, junto con el script necesario, para ser ejecutada en serie.

Scriptstarrynight.m en serie

```
%% Imagen
im1 = double((imread('imagen.bmp')));
disp(max(im1(:)))
[m, n]   = size(im1);
paso     = 4;
[xx, yy] = meshgrid(1:paso:n, 1:paso:m);

%% ASG
lpfasgl  = 7; %2*round(m/30)+1;        % LowPassFilter
AveragedSquaredGradient Length
[dx, dy] = asg(im1, lpfasgl);       % Averaged Square Gradient
angulod  = -angle(dx +j*dy)*180/pi;
absd     = abs(dx + j*dy);
dxn      = dx./(absd+eps);
dyn      = dy./(absd+eps);

%% ASGVF
niter     = 50;
etaini    = 1;
etafin    = 1;
factoreta = 10;%sqrt(10);
```

```
[vx, vy]  = asgvfme(dx, dy, niter, etaini, etafin, factoreta);
angulov   = -angle(vx +j*vy)*180/pi;
absv      = abs(vx + j*vy);
vxn       = vx./(absv+eps);
vyn       = vy./(absv+eps);

%% Morphological operations
mask1 = double(edge(im1,'Canny'));
mask1 = double(bwmorph(mask1,'clean'));
mask1 = double(imdilate(mask1,ones(3)));
mask3 = double(imdilate(mask1,ones(3)))-mask1;
mask5 = ones([m,n])-mask3-mask1;

lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

im1recte = SVMMerode( im1, angulov, lengthMatrix, widthMatrix);
figure, imshow(im1recte/255);
```

SVMMerode.m en serie

```
function imout = SVMMerode(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMerode(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology erosion
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);

maximin = max(imin(:));

maxWidth   = max(widthMatrix(:));
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);
imin2      = maximin*ones(mim + 2*(m2-1), nim + 2*(m2-1)); %
boundaries -> maximo
imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imin;

i_ini_img  = m2;
i_end_img  = m2 + mim - 1;
j_ini_img  = m2;
j_end_img  = m2 + nim - 1;

imout2     = zeros(size(imin2));

tot = 100;
```

```
for i = i_ini_img:i_end_img
   for j = j_ini_img:j_end_img;
       se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
       se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
       se  = (imfill(conv2(double(se1),double(se2)),'holes')>0);
       [mse, nse] = size(se);
       i_ini = ceil(mse/2);
       j_ini = ceil(nse/2);
       auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-j_ini+1):(j+j_ini-1));
       auxpr = auxin.*se;
       auxse = auxpr(se);
       imout2(i,j) = min(auxse(:));
   end
   if mod(i,5)==0
       fprintf(1, '%3d', round((i-m2)/mim*tot));
       if (rem(i,100) == 0)      fprintf(1, '\n');   end
   end
end

imout = imout2(m2:(m2+mim-1),m2:(m2+nim-1));

fprintf(1, '\n');
```

1.2. Código de la función erosión, junto con el script necesario, para ser ejecutada en paralelo.

Este código está preparado para ser ejecutado por un solo “worker” como se indica en las líneas

```
set(job4, 'MaximumNumberOfWorkers', 1);
set(job4, 'MinimumNumberOfWorkers', 1);
```

de *scriptstarrynight.m*. Para cambiar el número de “workers” deseado simplemente hay que modificar el tercer parámetro de esta función, sabiendo que en “MinimunNumberWorkers” el valor más bajo es 1 y el valor más alto es 6 siempre y cuando sea menor o igual que “MaximumNumberWorkers” que tendrá un rango de 1 a 6, ya que 6 es el número de “workers” que posee el clúster-GTTS y no podremos usar mas “workers” de los que existen.

Scriptstarrynight.m en paralelo

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

sched = findResource('scheduler','configuration','cluster');
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'SVMMerode.m'});
createTask(job4, @SVMMerode, 1, {im1, angulod, lengthMatrix,
widthMatrix});
set(job4, 'MaximumNumberOfWorkers', 1);
set(job4, 'MinimumNumberOfWorkers', 1);
```

```
tic
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
im1recte = getAllOutputArguments(job4);
toc
sprintf('Got Output Arguments')
destroy(job4) % permanently removes job data
sprintf('Test Completed')

figure, imshow(im1recte{1,1}/255);
```

SVMMerode.m en paralelo

```
function imout = SVMMerode(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMerode(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology erosion
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);

maximin = max(imin(:));

maxWidth   = max(widthMatrix(:));
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);
imin2      = maximin*ones(mim + 2*(m2-1), nim + 2*(m2-1)); %
boundaries -> maximo
imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imin;

i_ini_img  = m2;
i_end_img  = m2 + mim - 1;
j_ini_img  = m2;
j_end_img  = m2 + nim - 1;

imout2     = zeros(size(imin2));

tot = 100;

for i = i_ini_img:i_end_img
   parfor j = j_ini_img:j_end_img;
       se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
       se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
```

```
        se  = (imfill(conv2(double(se1),double(se2)),'holes')>0);
        [mse, nse] = size(se);
        i_ini = ceil(mse/2);
        j_ini = ceil(nse/2);
        auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-j_ini+1):(j+j_ini-1));
        auxpr = auxin.*se;
        auxse = auxpr(se);
        imout2(i,j) = min(auxse(:));
    end
    if mod(i,5)==0
        fprintf(1, '%3d', round((i-m2)/mim*tot));
        if (rem(i,100) == 0)      fprintf(1, '\n');   end
    end
end

imout = imout2(m2:(m2+mim-1),m2:(m2+nim-1));

fprintf(1, '\n');
```

1.3. Código de la función de dilatación y su correspondiente script, en serie

Scriptstarrynight.m en serie

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

im1recte = SVMMdilate( im1, angulov, lengthMatrix, widthMatrix);
figure, imshow(im1recte/255);
```

SVMMdilate.m en serie

```
function imout = SVMMdilate(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMdilate(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology dilation
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);

maxWidth   = max(widthMatrix(:));
```

```
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);
imin2      = zeros(mim + 2*(m2-1), nim + 2*(m2-1)); % boundaries -> 0
imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imin;

i_ini_img  = m2;
i_end_img  = m2 + mim - 1;
j_ini_img  = m2;
j_end_img  = m2 + nim - 1;

imout2     = zeros(size(imin2));

tot=100;

for i = i_ini_img:i_end_img
   for j = j_ini_img:j_end_img;
       se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
      m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
       se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
      m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
       se  = (imfill(conv2(double(se1),double(se2)),'holes')>0);
       [mse, nse] = size(se);
       i_ini = ceil(mse/2);
       j_ini = ceil(nse/2);
       auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-j_ini+1):(j+j_ini-1));
       auxpr = auxin.*se;
       auxse = auxpr(se);
       imout2(i,j) = max(auxse(:));
   end
   if mod(i,5)==0
       fprintf(1, '%3d', round((i-m2)/mim*tot));
       if (rem(i,100) == 0)      fprintf(1, '\n');   end
   end
end

imout = imout2(m2:(m2+mim-1),m2:(m2+nim-1));
fprintf(1, '\n');
```

1.4. Código de la función de dilatación y su correspondiente script, en paralelo

Scriptstarrynight.m en paralelo

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

sched = findResource('scheduler','configuration','cluster');
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'SVMMdilate.m'});
createTask(job4, @SVMMdilate, 1, {im1, angulod, lengthMatrix,
widthMatrix});
set(job4, 'MaximumNumberOfWorkers', 1);
set(job4, 'MinimumNumberOfWorkers', 1);
tic
```

```
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
im1recte = getAllOutputArguments(job4);
toc
sprintf('Got Output Arguments')
destroy(job4) % permanently removes job data
sprintf('Test Completed')

figure, imshow(im1recte{1,1}/255);
```

SVMMdilate.m en paralelo

```
function imout = SVMMdilate(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMdilate(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology dilation
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);

maxWidth   = max(widthMatrix(:));
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);
imin2      = zeros(mim + 2*(m2-1), nim + 2*(m2-1)); % boundaries -> 0
imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imin;

i_ini_img  = m2;
i_end_img  = m2 + mim - 1;
j_ini_img  = m2;
j_end_img  = m2 + nim - 1;

imout2     = zeros(size(imin2));

tot=100;

for i = i_ini_img:i_end_img
   parfor j = j_ini_img:j_end_img;
       se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
      m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
       se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
      m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
       se  = (imfill(conv2(double(se1),double(se2)),'holes')>0);
       [mse, nse] = size(se);
       i_ini = ceil(mse/2);
       j_ini = ceil(nse/2);
       auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-j_ini+1):(j+j_ini-1));
```

```
        auxpr = auxin.*se;
        auxse = auxpr(se);
        imout2(i,j) = max(auxse(:));
    end
    if mod(i,5)==0
        fprintf(1, '%3d', round((i-m2)/mim*tot));
        if (rem(i,100) == 0)      fprintf(1, '\n');   end
    end
end

imout = imout2(m2:(m2+mim-1),m2:(m2+nim-1));

fprintf(1, '\n');
```

1.5. Código de la función de apertura y su correspondiente script, en serie

Scriptstarrynight.m en serie

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

im1recte = SVMMopen( im1, angulov, lengthMatrix, widthMatrix);
figure, imshow(im1recte/255);
```

SVMMopen.m serie

```
function imout = SVMMopen(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMopen(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology opening
%
%       imin:           input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);

maximin = max(imin(:));    % maximo valor de intensidad

maxWidth   = max(widthMatrix(:));
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);

imout  = zeros(mim, nim);
imin3  = zeros(mim, nim, maximin);
```

```
imout3 = zeros(mim, nim, maximin);

for z = 0:maximin
    imin3(:,:,z+1) = (imin>=z);
end

fprintf(1, '\n');
%en este for le ponemos el parfor
for z = 0:maximin
    imaux = imin3(:,:,z+1);
    munos = ones(mim,nim);
    if max(max(abs(imaux-munos)))>0
        imin2      = ones(mim + 2*(m2-1), nim + 2*(m2-1)); % creo
bordes a uno
        imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imaux;%imin;

        i_ini_img  = m2;
        i_end_img  = m2 + mim - 1;
        j_ini_img  = m2;
        j_end_img  = m2 + nim - 1;

        imout2     = zeros(size(imin2));

        for i = i_ini_img:i_end_img
            for j = j_ini_img:j_end_img;
                se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
                se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
                se  =
(imfill(conv2(double(se1),double(se2)),'holes')>0);
                [mse, nse] = size(se);
                i_ini = ceil(mse/2);
                j_ini = ceil(nse/2);
                auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1));
                auxpr = auxin.*se;
                auxse = auxpr(se);
                if (sum(se(:))==sum(auxse(:))) % el ee esta contenido
                    imout2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1))=imout2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1)) + se;
                end
            end
            if maximin==1
                fprintf(1, '%3d', round((i-m2)/mim*100));
                if (rem(i,35) == 0)      fprintf(1, '\n');   end
            end
        end
        imout3(:,:,z+1) = double(imout2(m2:(m2+mim-1),m2:(m2+nim-
1))>0)*z;
    else
        imout3(:,:,z+1) = double(munos)*z;
    end
    fprintf(1, '%3d', round(z/maximin*100));
    if (rem(z,35) == 0)      fprintf(1, '\n');   end
end

for i = 1:mim
    for j = 1:nim
```

```
        aux = imout3(i,j,:);
        imout(i,j) = max(aux(:));
    end
end

fprintf(1, '\n')
```

1.6. Código de la función de apertura y su correspondiente script, en paralelo

Scriptstarrynight.m en paralelo

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

sched = findResource('scheduler','configuration','cluster');
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'SVMMopen.m'});
createTask(job4, @SVMMopen, 1, {im1, angulod, lengthMatrix,
widthMatrix});
set(job4, 'MaximumNumberOfWorkers', 1);
set(job4, 'MinimumNumberOfWorkers', 1);
tic
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
im1recte = getAllOutputArguments(job4);
toc
sprintf('Got Output Arguments')
destroy(job4) % permanently removes job data
sprintf('Test Completed')

figure, imshow(im1recte{1,1}/255);
```

SVMMopen.m en paralelo

```
function imout = SVMMopen(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = SVMMopen(imin, degreeMatrix, lengthMatrix, widthMatrix);
%
%       Spatially-Variant Mathematical Morphology opening
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)

[mim, nim] = size(imin);
```

```
maximin = max(imin(:));    % maximo valor de intensidad

maxWidth   = max(widthMatrix(:));
maxLength  = max(lengthMatrix(:));
m          = max([maxLength; maxWidth]);
m2         = m;%ceil(m/2);

imout  = zeros(mim, nim);
imin3  = zeros(mim, nim, maximin);
imout3 = zeros(mim, nim, maximin);

for z = 0:maximin
    imin3(:,:,z+1) = (imin>=z);
end

fprintf(1, '\n');

parfor z = 0:maximin
    imaux = imin3(:,:,z+1);
    munos = ones(mim,nim);
    if max(max(abs(imaux-munos)))>0
        imin2      = ones(mim + 2*(m2-1), nim + 2*(m2-1)); % creo
bordes a uno
        imin2(m2:(m2+mim-1),m2:(m2+nim-1)) = imaux;%imin;

        i_ini_img  = m2;
        i_end_img  = m2 + mim - 1;
        j_ini_img  = m2;
        j_end_img  = m2 + nim - 1;

        imout2     = zeros(size(imin2));

        for i = i_ini_img:i_end_img
            for j = j_ini_img:j_end_img;
                se1 = getnhood(strel('line',lengthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+00));
                se2 = getnhood(strel('line', widthMatrix(i-m2+1,j-
m2+1),degreeMatrix(i-m2+1,j-m2+1)+90));
                se  =
(imfill(conv2(double(se1),double(se2)),'holes')>0);
                [mse, nse] = size(se);
                i_ini = ceil(mse/2);
                j_ini = ceil(nse/2);
                auxin = imin2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1));
                auxpr = auxin.*se;
                auxse = auxpr(se);
                if (sum(se(:))==sum(auxse(:))) % el ee esta contenido
                    imout2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1))=imout2((i-i_ini+1):(i+i_ini-1),(j-
j_ini+1):(j+j_ini-1)) + se;
                end
            end
            if maximin==1
                fprintf(1, '%3d', round((i-m2)/mim*100));
                if (rem(i,35) == 0)      fprintf(1, '\n');   end
            end
        end
        imout3(:,:,z+1) = double(imout2(m2:(m2+mim-1),m2:(m2+nim-
1))>0)*z;
```

```
    else
        imout3(:,:,z+1) = double(munos)*z;
    end
    fprintf(1, '%3d', round(z/maximin*100));
    if (rem(z,35) == 0)      fprintf(1, '\n');   end
end

for i = 1:mim
    for j = 1:nim
        aux = imout3(i,j,:);
        imout(i,j) = max(aux(:));
    end
end

fprintf(1, '\n')
```

1.7. Código de la función cierre y su correspondiente script, en serie

Scriptstarrynight.m en serie

```
lengthMatrix = 7*ones(m,n);
widthMatrix  = 1*ones(m,n);

im1recte = SVMMclose( im1, angulov, lengthMatrix, widthMatrix);
figure, imshow(im1recte/255);
```

SVMMclose.m en serie

```
function imout = SVMMclose(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = rectgrayMMSVclose(imin, degreeMatrix, lengthMatrix,
widthMatrix);
%
%       Spatially-Variant Mathematical Morphology closing
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)


maxg  = max(imin(:));
imin2 = maxg - imin;
imout = SVMMopen(imin2, degreeMatrix, lengthMatrix, widthMatrix);
imout = maxg - imout;
```

1.8. Código de la función cierre y su correspondiente script, en paralelo

Scriptstarrynight.m en paralelo

Este script es idéntico a cuando se ejecutan en serie, lo que cambia es la función *SVMMclose.m* que se ve a continuación.

SVMMclose.m en paralelo

```
function imout = SVMMclose(imin, degreeMatrix, lengthMatrix,
widthMatrix)
%
% imout = rectgrayMMSVclose(imin, degreeMatrix, lengthMatrix,
widthMatrix);
%
%       Spatially-Variant Mathematical Morphology closing
%
%       imin:          input image, binary (0,1) or gray-level (0-255)
%       degreeMatrix:  matrix with the orientation (in degrees) of the
data
%       lengthMatrix:  matrix with the length of the structuring
element (must be an odd number)
%       widthMatrix:   matrix with the width of the structuring
element (must be an odd number)


maxg  = max(imin(:));
imin2 = maxg - imin;
%imout = SVMMopen(imin2, degreeMatrix, lengthMatrix, widthMatrix);

sched = findResource('scheduler','configuration','cluster');
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'SVMMopen.m'});  %%cuando se trata de una funcion en vez de
%script es mejor utilizar createMatlabPoolJob que el
%job=batch(.nombresript..) ver submitJob2b.m
createTask(job4, @SVMMopen, 1, {imin2, degreeMatrix, lengthMatrix,
widthMatrix});
set(job4, 'MaximumNumberOfWorkers', 4);
set(job4, 'MinimumNumberOfWorkers', 4);
tic
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
im1recte = getAllOutputArguments(job4);
toc
sprintf('Got Output Arguments')
destroy(job4) % permanently removes job data
sprintf('Test Completed')

imout = maxg - imout;
```

2. Anexo II

En este apartado del anexo se amplía la información de algunos módulos del clúster-GTTS de la UPCT.

2.1. HP Proliant ML350 G5 (458238-421) – Especificaciones

Tabla 22: Unidades internas del HP ProLiant ML350 G5

Unidades internas	
Unidades internas	La configuración estándar no incluye discos duros
Velocidad de la unidad de disco duro	No aplicable
Controlador de almacenamiento	Controlador Smart Array E200i/128, caché de escritura respaldada por batería (BBWC), (RAID 0/1/1+0/5)
Ranuras de expansión	6 ranuras de expansión (tres PCI Express, tres PCI-X)
Unidad de disco flexible	Ninguna incluida de serie
Unidades ópticas	DVD-ROM 16x de serie

Tabla 23: Caracteristicas del sistema del HP ProLiant ML350 G5

Características del sistema	
Formato	Torre de 5U
Chipset	Chipset Intel® 5000Z
Interfaz de red	Adaptador de red Gigabit NC373i multifunción integrado con TCP/IP Offload Engine, incluyendo compatibilidad con iSCSI acelerado por medio de un kit opcional de licencias
Puertos de E/S externos	Paralelo - Puerto paralelo opcional disponible; Serie - 1 (2º puerto serie opcional disponible); Dispositivo de puntero (ratón) - 1; Gráficos - 1; Teclado - 1; Red RJ-45 - 2 (1 dedicado para iLO 2); Puertos USB 2.0 - 6 (2 frontales, 2 posteriores, 2 internos)
Ranuras de expansión	6 ranuras de expansión (tres PCI Express, tres PCI-X)
Tipo de fuente de alimentación	Fuente de alimentación de 800 W de conexión en caliente, cumple con la marca CE (línea de 1000 W); Una segunda fuente de alimentación opcional proporciona redundancia.
Requisitos de alimentación	Tensión de entrada de 100 a 120 VCA, 200 a 240 VCA; 50/60 Hz
Sistemas operativos compatibles	Microsoft® Windows® 2000 Server y Advanced Server; Microsoft® Windows® Server 2003/R2 (Standard, Web y Enterprise Editions); Microsoft® Windows® Small Business

	Server 2003; Microsoft® Windows® Small Business Server 2003, R2; Microsoft® Windows® Server 2008; Novell NetWare 6.5/Open Enterprise Server; Red Hat Enterprise Linux; SUSE Linux Enterprise Server; SCO OpenServer 5.0.7/6.0; SCO UnixWare 7.1.3/7.1.4
Medidas del producto (P x A x L)	21,8 x 59,6 x 46,7 cm
Peso del producto	30,4 kg
Cumplimiento de estándares del mercado	Compatible con ACPI V2.0; Compatible con PCI 2.2; Compatible con PXE; Compatible con WOL; Compatible con PCI-X 1.0a; Certificado por Novell; Certificaciones del logotipo Microsoft®; USB 2.0
Gestión de seguridad	Contraseña de encendido; Contraseña de configuración; Control de arranque desde disquetera; Control de interfaz serie y paralelo; Bloqueo de configuración de discos; Seguridad mediante el botón de encendido
Facilidad de mantenimiento	Puerto paralelo y segundo puerto serie opcionales disponibles sin usar una ranura PCI; Apertura de chasis y acceso a componentes sin necesidad de utilizar herramientas; Retirada de la placa base sin necesidad de herramientas; Solución de raíles universales que admite entornos de bastidor con orificios cuadrados y redondos
Garantía	3 años en piezas, 3 años en mano de obra, 3 años de soporte en casa del cliente

2.2. HP ProliantDL580 G5 2.93 130W 4P 8G

2.2.1. Introducción

El HP ProLiant DL580 G5 es la mejor plataforma para virtualización de su tipo, combina la nueva tecnología de procesador Intel Xeon®, una escalabilidad máxima y una alta disponibilidad. Este servidor de 4 ranuras ofrece flexibilidad y capacidad de mantenimiento inigualables con un tamaño versátil de 4U optimizado para bastidor. Según las últimas tecnologías de normas de industria, el DL580 G5 proporciona los niveles más altos de rendimiento que demandan las aplicaciones informáticas de gran intensidad hoy en día. Un gran número de funciones de alta disponibilidad aseguran un tiempo máximo de actividad. La gestión remota junto con la tecnología Integrated Lights-Out 2 (iLO2) permite llevar a cabo la administración desde un explorador Web estándar sin necesidad de visitar el servidor. Posee una arquitectura con grandes posibilidades de ampliación que ofrecen la máxima flexibilidad de implementación de aplicaciones que permiten añadir E/S, procesamiento, memoria o almacenamiento según sea necesario.

2.2.2. Características

Los últimos procesadores de 4 y 6 núcleos Intel® Xeon® 7400 ofrecen una posibilidad máxima de ampliación según sus necesidades; Gran ancho de banda y flexibilidad para ampliación gracias a las 8 ranuras PCI Express y la opción de añadir 3 ranuras PCI Express o PCI-X adicionales; Las 32 ranuras DIMM admiten hasta 256GB de DIMMs de memoria intermedia completa PC2-5300 (DDR2-667); Los clientes que busquen almacenamiento adicional pueden llenar hasta 16 unidades SAS/SATA de conexión en caliente con 4 TB de almacenamiento interno; Funciones estándar que mejoran el rendimiento son las tarjetas de red Gigabit integradas y el controlador Smart Array P400i con hasta 512 MB de caché de escritura respaldada por la batería.

Mantenimiento sin herramientas y acceso de servicio a los componentes de conexión en caliente externos e internos sin sacrificar por ello la capacidad de ampliación interna; Hasta 32 módulos de memoria, 2 compartimentos para soporte, 11 ranuras PCI y 16 unidades de disco duro interno proporcionan la flexibilidad necesaria para realizar implementaciones repetidas veces en una gran variedad de entornos de aplicación; Unidad DVD compacta extraíble y unidades de cinta de media altura extraíbles accesibles con facilidad desde la parte frontal del servidor. Puertos USB (dos frontales, dos posteriores y dos internos) incrementan las opciones de soportes, incluida la capacidad de arrancar desde un lápiz USB; Los kits de guías encajables y un diseño sin cables creado especialmente para el tamaño 4U permiten implementaciones sin complicaciones en bastidores con orificios ajustables o cuadrados; El System Insight Display frontal ofrece indicaciones visuales instantáneas sobre condiciones de fallo dentro de todos los subsistemas principales del servidor, permite una respuesta rápida para los sucesos de servicio y reduce el tiempo de inactividad y los costes de servicio y mantenimiento informáticos.

El estándar Integrated Lights-Out 2 (iLO 2) combina una serie de diagnósticos y funciones básicas de gestión seguras con la presencia y control virtuales esenciales para la gestión de servidores ProLiant en todo el centro de datos o a nivel mundial; ProLiant Essentials Foundation Pack, que se incluye con cada servidor ProLiant, le permite instalar y configurar rápidamente los sistemas, gestionar los cambios de forma proactiva y asegurar el funcionamiento continuado de los servidores; SmartStart configura el hardware, carga controladores optimizados y ayuda a instalar el software para alcanzar un nivel óptimo de fiabilidad, rendimiento y disponibilidad del sistema; SmartStart Scripting Toolkit simplifica al máximo las implantaciones de grandes volúmenes de servidores, ya que incluye utilidades de duplicación que crean y copian archivos de configuración y de comandos; Para ampliar sus capacidades con herramientas como ProLiant Essentials Rapid Deployment Value Pack que automatiza el proceso de implantar y suministrar las configuraciones particulares del software del servidor.

Tabla 24: Unidades internas HP Proliant DL580 G5

Unidades internas	
Unidades internas	La configuración estándar no incluye unidades de disco duro
Velocidad de la unidad de disco duro	No aplicable
Controlador de almacenamiento	Contr. HP Smart Array P400i integ./512 MB caché BBWC
Ranuras de expansión	Más ranuras PCI-Express con hasta once disponibles con tarjeta opcional; Estándar de ocho ranuras: (4) ranuras PCI-E x8 de altura completa, (1) ranura PCI-E x4 de longitud completa y (3) ranuras PCI-E x4; La tarjeta opcional añade (3) ranuras PCI-X o (3) ranuras PCI-E x8
Unidad de disco flexible	Ninguna entregada de serie
Unidades ópticas	Unidad DVD extraplana estándar

Tabla 25: Características del sistema del HP Proliant DL580 G5

Características del sistema	
Formato	Bastidor de 4U
Chipset	Chipset Intel 7300 estándar
Interfaz de red	Dos adaptadores de red Gigabit NC373i multifunción integrados con TCP/IP Offload Engine, incluyendo compatibilidad con iSCSI acelerado por medio de un kit opcional de licencias de ProLiant Essentials.
Puertos de E/S externos	Serie - 1; Dispositivo señalador (ratón) - 1; Vídeo - 1 frontal; 1 posteriores; Teclado - 1; puerto USB (2.0) - 5 en total: 2 frontales; 2 posteriores; 1 interno; Gestión remota iLO2 - 1; Red RJ-45 - 2
Ranuras de expansión	Más ranuras PCI-Express con hasta once disponibles con tarjeta opcional; Estándar de ocho ranuras: (4) ranuras PCI-E x8 de altura completa, (1) ranura PCI-E x4 de longitud completa y (3) ranuras PCI-E x4; La tarjeta opcional añade (3) ranuras PCI-X o (3) ranuras PCI-E x8
Tipo de fuente de alimentación	Fuente alimentación CA 800/1200W; Suprime la necesidad de fuente de alimentación
Requisitos de alimentación	De 90 a 132 VCA, de 180 a 264 VCA, de 47 a 63 Hz
Sistemas operativos compatibles	Microsoft® Windows® Server; Microsoft® Windows® Server Hyper V; Red Hat Enterprise Linux (RHEL); SUSE Linux Enterprise Server (SLES); Oracle Enterprise Linux (OEL); Solaris 10 para sistemas basados en arquitectura x86/x64; NetWare; VMware; Citrix Essentials for XenServer
Medidas del producto (P x A x L)	48,3 x 67,3 x 17.6 cm

Peso del producto	45,4 kg
Cumplimiento de estándares del mercado	ACPI 2.0. ACPI 1.0a; Compatible con PCIE 1.0a; admite PXE; admite WOL; Compatible con la ampliación de direcciones físicas (PAE); Certificaciones del logotipo Microsoft®; Compatible con USB 2.0
Gestión de seguridad	Contraseña de encendido; Contraseña de teclado; Activación/desactivación de puerto USB externo; Modo de servidor de red; Control de interfaz serie; Contraseña de administrador; Unidad de CD-ROM o DVD extraíble
Facilidad de mantenimiento	Puertos vídeo delantero y trasero; Los módulos alimentación de procesador, el procesador con acceso frontal y la memoria permiten al servidor permanecer firmemente instalado en el bastidor durante el mantenimiento; El sistema de guías de implantación rápida incluye guías deslizantes universales, un brazo de administrador de cables ambidiestro y palancas de liberación rápida para agilizar y simplificar tareas de mantenimiento y permite acceder a los principales componentes del servidor en el propio bastidor sin el uso de herramientas. Acceso a los componentes del sistema sin herramientas para facilitar el mantenimiento en bastidor; CSR (reparación por parte del cliente): Los productos HP se han diseñado con piezas que puede reparar y sustituir el usuario para reducir el tiempo de reparación
Fuente de alimentación redundante	Cuatro fuentes de alimentación (tercera y cuarta fuente para redundancia).
Garantía	3 años en piezas, 3 años mano de obra, 3 años de soporte a domicilio

2.3. HP Single Port – Disco duro – 146 GB – hot-swap – 2.5’’ SFF – SAS -10000

Tabla 26: Descripción del disco duro

Disco duro

Factor de forma	2.5" SFF
Capacidad	146 GB
Tipo de interfaz	Serial Attached SCSI

Tabla 27: Presaciones del disco duro

Prestación

Índice de transferencia de la unidad	300 MBps (externo)
Tiempo de búsqueda	4.1 ms (media) / 8 ms (máx.)
Tiempo de búsqueda de pista a pista	0.6 ms
Velocidad del eje	10000 rpm

Tabla 28: Expansión o conectividad del disco duro

Expansión / Conectividad

Interfaces	1 x Serial Attached SCSI - SAS interno de 29 patillas (SFF-8482)
Compartimentos compatibles	1 x hot-swap - 2.5" SFF

Tabla 29: Garantia del fabricante del disco duro

Garantía del fabricante

Servicio y mantenimiento	3 años de garantía
Detalles de Servicio y Mantenimiento	Garantía limitada - 3 años

Tabla 30: Parámetros del entorno del disco duro

Parámetros de entorno

Temperatura mínima de funcionamiento	10 °C
Temperatura máxima de funcionamiento	35 °C

BIBLIOGRAFÍA

1. R. Verdú, Jesús Angulo, Jean Serra, "Spatially-Variant Anisotropic Morphological Filters Driven by Gradient Fields", Lecture Notes in Computer Science (LNCS): 9th International Symposium on Mathematical Morphology, Springer, vol. 5720, pp. 115-125, Groningen (The Netherlands), Aug. 2009.

2. R. Verdú, Jesús Angulo, "Spatially-Variant Directional Mathematical Morphology Operators Based on a Diffused Average Squared Gradient Field", Lecture Notes in Computer Science (LNCS): Advanced Conceps for Intelligent Vision Systems, vol. 5259, pp. 542-553, Juan-les-Pins (France), Oct. 2008.

3. Ayuda de Matlab

4. www.hp.com

5. Códigos fuente (Erosión, Dilatación, Apertura y Cierre) de Rafael Verdú Monedero

6. Digital Image Processing Using Matlab - Gonzalez Woods & Eddins

7. D. Pearson, ed. Procesamiento de Imágenes (McGraw-Hill, Maidenhead, 1991)

8. R. Chellappa and S. Chatterjee, "Classification of textures using gaussia markov random fields," IEEE Transactions on Acoustics Speech and Signal Processing, vol. 33, 1985.