UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA
Departamento de Arquitectura de Computadoras y Automática

MODELO DE PROGRAMACIÓN PARA INFRAESTRUCTURAS GRID COMPUTACIONALES

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

José Herrera Sanz

Bajo la dirección de los doctores
Ignacio Martín Llorente
Rubén Santiago Montero

Madrid, 2008

- **ISBN: 978-84-692-1755-9**

Modelo de Programación para Infraestructuras Grid Computacionales

Tesis Doctoral

José Herrera Sanz

Madrid, Octubre de 2008

Dpto. de Arquitectura de Computadores y Automática

Universidad Complutense de Madrid

A mi padre y mi madre, por enseñarme
lo que es verdaderamente importante
en esta vida.

El hombre de negro huía
a través del desierto y
el pistolero iba
en pos de él.

La Torre Oscura,
Stephen King

Agradecimientos

Me gustaría agradecer a todas aquellas personas que de alguna u otra manera me han ayudado en estos últimos cinco años a la realización de esta tesis doctoral.

En primer lugar a mis directores de tesis, a Ignacio Martín Llorente por darme la oportunidad de realizar este trabajo dentro de su grupo de investigación, y a Rubén Santiago Montero por ayudarme tanto, tantísimo a finalizar esta tesis y por su infinita paciencia conmigo.

A mi padres, por aguantar mis cambios de humor y por apoyarme siempre en todo lo que he hecho aunque no estuvieran de acuerdo. Por darme la oportunidad de poder estudiar y hacer una carrera universitaria aún a costa de la economía familiar. A mis hermanas, sobrinos y cuñado por apoyarme durante todos estos años y estar siempre a mi lado dándome tanto cariño.

A Kis por ser mi compañera de aventuras y desventuras desde hace ocho años, por sus consejos y ánimos en los peores momentos y por hacerme reír con tontadas y tontunas siempre que ha tenido ocasión. A Bea y Fabian por estar dispuestos siempre a ayudarme en lo que sea, y por todos aquellos momentos que hemos vivido tanto dentro como fuera de España. A Benjamín, por ser un amigo de veras y estar siempre dispuesto cuando le he necesitado, y sobretodo por las risas que nos hemos echado siempre. A Carlos, que es como mi hermano pequeño y me ayudado a aumentar mi paciencia hasta limites insospechados. Y al resto de grupos de amigos que de alguna u otra manera me han enseñado algo, no siempre positivo, a lo largo de estos últimos años.

A Sara, por su mandala, y por todas las conversaciones que hemos tenido cuando compartíamos despacho, que me han servido para conocer otro lado de la vida. A Inma, mi primera "jefa", que me ilustró sobre tareas docentes y me ayudó mucho durante los primeros años de mi carrera universitaria. A Sonia, por todos los bueno momentos vividos y por el ánimo mutuo que hemos compartido y seguimos compartiendo. A Guadalupe, por hacerme reír con sus ocurrencias que me han alegrado algún que otro mal día. A Silvia, la otra friki del departamento, por sus risas, su continuo estado de buen humor, su objetividad y sabiduría, y por tener siempre buenas palabras hacia mi persona. A mi otra gran "jefa" en el departamento, Hortensia, por los divertidos momentos y las risas que nos pasamos en Rodas y cuando salimos por ahí a tomar algo. A Elena por todas las situaciones absurdas que hemos compartido. Y a Raquel, por compartir mi fanatismo y darme siempre tanto cariño.

A Marcos, yo si te pongo en mis agradecimientos, sobretodo por mostrarme el otro lado de la investigación que era desconocido para mi. A Juan Carlos, por todos sus consejos, por hacerme reír y en algunos casos enfadar, y por su apoyo durante estos años. A Daniel Mozos por preocuparse por mi y azuzarme para empezar a escribir la tesis. A mi "compañero" Fredy que me ha ilustrado sobre el cine oriental y de autor en versión original, algunas veces sin mucho éxito, y con el que he compartido muchos momentos, unos alegres y otros no tanto, pero que me ha ayudado mucho durante este

último año y ha sido un auténtico amigo de verdad. A mi "primo" Manu, por hacerme reír tanto, tanto en los cafés, por tener siempre una palabra agradable y de ánimo cuando me veía bajuno y por lo bien que nos lo hemos pasado montados en la "Stunt Fall". Y finalmente, al resto de compañeros de grupo de investigación, en especial a Eduardo Huedo por su ayuda durante la realización de esta tesis doctoral.

Contenido

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este capítulo realiza una descripción de las motivaciones y objetivos de la presente tesis doctoral que presenta un conjunto de aportaciones dividas en tres grandes bloques: el primero está constituido por un modelo de desarrollo que facilita la programación de aplicaciones Grid denominado DRMAA-Grid *W*ay; el segundo bloque lo constituye un algoritmo de planificación de tareas adaptado a entornos Grid heterogéneos y basado en el modelo de programación anterior; finalmente la última aportación presentada en esta tesis consiste en un simulador de entornos Grid heterogéneos basado en el modelo de programación DRMAA-Grid *W*ay y el simulador GridSim.

Este capítulo se estructura en tres secciones, la sección 1.1 que muestra las principales motivaciones que han dado lugar al desarrollo de esta tesis doctoral. La sección 1.2 explica los objetivos que se pretenden conseguir con la incorporación de las aportaciones presentadas en la tesis dentro del ámbito de la computación Grid. Y finalmente, la sección 1.3 presenta la estructura general de la tesis doctoral.

1.1. Motivacion

A lo largo de la última década los sistemas Grid han ido adquiriendo cada vez más relevancia y han dado lugar a una gran variedad de tesis doctorales con diferentes líneas de investigación. Una de las principales líneas de investigación en el ámbito de sistemas Grid heterogéneos, corresponde con lo que se denomina *Gridification* o Gridificación, que consiste en adaptar aplicaciones paralelas ya existentes a sistemas Grid. Sin embargo, esta tarea, al igual que la correspondiente con el desarrollo de aplicaciones nuevas a sistemas Grid, conlleva una gran dificultad asociada, ya que el

desarrollador de aplicaciones tiene que tener en cuenta muchos elementos a la hora de programar la aplicación. Aunque en la actualidad existen muchas herramientas que facilitan el desarrollo de este tipo de aplicaciones, no existe un modelo de desarrollo que permita implementar aplicaciones en entornos Grid independientemente del *middleware* utilizado e incluso del gestor de recursos utilizado. Por ello, esta tesis presenta un modelo de programación de aplicaciones Grid, denominado DRMAA-Grid *W*ay que, no sólo facilita el desarrollo de estas aplicaciones por parte del programador, sino que también dota a las aplicaciones implementadas bajo este modelo de características que permiten su ejecución de manera más eficiente.

Debido a la heterogeneidad intrínseca de los sistemas Grid, otra de las líneas de investigación importantes dentro de esta tecnología corresponde con la planificación de tareas. En este sentido se han realizado una gran cantidad de estudios y artículos de investigación que proponen distintas técnicas de planificación de tareas. Sin embargo, hasta la actualidad no existe ningún algoritmo de planificación adaptado y diseñado específicamente para entornos Grid heterogéneos que además incluya todas las aportaciones que ofrece el modelo de programación DRMAA-Grid *W*ay. De esta forma, se ha diseñado un nuevo algoritmo de planificación, basado en los algoritmos de planificación de iteraciones clásicos, que permite una distribución de tareas adaptada a entornos Grid heterogéneos y que además consigue disminuir el tiempo de ejecución total.

La investigación de nuevas técnicas y algoritmos de planificación en entornos Grid conlleva una dificultad añadida respecto a otros entornos de ejecución. Dicha dificultad radica en el tiempo físico necesario para ejecutar cada uno de los experimentos. Esto se debe a que los sistemas Grid están diseñados para ejecutar aplicaciones con una gran carga computacional, y por lo tanto todas las investigaciones sobre nuevas técnicas de planificación en Grid deben llevar asociadas ejecuciones de aplicaciones con cargas computacionales elevadas. De esta manera, el tiempo necesario para obtener los resultados de la investigación se incrementa considerablemente llegando a durar días, e incluso semanas. Para resolver estos problemas, han surgido un conjunto de herramientas que permiten el desarrollo de aplicaciones Grid, sin necesidad de disponer de un entorno Grid real, y que permiten simular la ejecución de aplicaciones con una gran carga computacional en un tiempo físico mucho menor. Sin embargo, ninguna de estas herramientas permite la ejecución de aplicaciones en un modelo semejante al modelo DRMAA-Grid *W*ay. Debido a ello se ha desarrollado un nuevo simulador basado en el conjunto de herramientas GridSim, que simula la ejecución de aplicaciones desarrolla-

das con el modelo DRMAA-Grid*Way*. Este nuevo simulador, ha sido una herramienta muy útil a la hora de comprobar la eficiencia y efectividad del nuevo algoritmo de planificación, presentado en esta tesis, antes de su ejecución en un entorno Grid real.

1.2. Objetivos

Los objetivos que presenta esta tesis se dividen en tres áreas de investigación Grid. La primera corresponde con el diseño de un modelo de programación para facilitar el desarrollo de aplicaciones en entornos Grid heterogéneos, independientemente del *middleware* utilizado. Además, el modelo debe incorporar a las aplicaciones ciertas características en su ejecución como flexibilidad, fiabilidad, transparencia y adaptación dinámica. La idea principal es que el desarrollador implemente el programa de manera muy similar a un programa secuencial, y que el modelo de desarrollo se encargue del resto de tareas como la gestión de recursos Grid, la planificación y la migración de trabajos, o comunicación con el *middleware* Grid correspondiente, entre otras. El modelo debe basarse en un API estándar que permita su ejecución en cualquier entorno distribuido y facilite de esta manera el despliegue de la aplicación.

El siguiente objetivo se encuentra dentro del área de los algoritmos de planificación de tareas y consiste en desarrollar un algoritmo de planificación de trabajos, o tareas, adaptado a entornos Grid heterogéneos que conviva con el modelo de desarrollo DRMAA-Grid*Way*. De esta manera el algoritmo, denominado *Grid Trapezoid Self-Scheduler* (GTSS), debe permitir la distribución uniforme de la carga computacional de los trabajos entre los nodos de un Grid heterogéneo y dinámico, a la vez que reduce el tiempo de ejecución de la aplicación. El objetivo final es que esta planificación se realice de forma totalmente trasparente tanto desde el punto de vista del usuario como del desarrollador de la aplicación.

El último objetivo corresponde con el área de investigación relacionada con los simuladores Grid. Esta tesis presenta el desarrollo de un nuevo simulador Grid adaptado al modelo DRMAA-Grid*Way* para la simulación de ejecuciones de aplicaciones Grid. Este simulador, además, debe permitir al desarrollador generar una gran variedad de bancos de pruebas para la ejecución de las aplicaciones. El objetivo principal del simulador consiste en servir como herramienta para analizar el comportamiento del planificador de tareas GTSS, antes de su ejecución en un entorno real Grid heterogéneo y distribuido.

1.3. Estructura

La presente tesis se divide en cinco capítulos principales:

- **Computación en GRID**. Este capítulo presenta el trabajo relacionado con la tecnología Grid, explicando en primer lugar las motivaciones que han llevado a la realización de esta tesis en esta tecnología. También se hará énfasis en analizar la estructura de la tecnología Grid así como su estándar de facto que corresponde con el *middleware* Globus Toolkit. La segunda parte del capítulo analiza el trabajo relacionado con los metaplanificadores de Grid más importantes que existen en la actualidad, y compara sus características con las del metaplanificador Grid *W*ay con el objetivo de justificar la elección de Grid *W*ay como parte del modelo de programación presentado en esta tesis. El capítulo termina con las conclusiones que han dado lugar a la elección de Grid *W*ay como metaplanificador del modelo DRMAA-Grid *W*ay.

- **Paradigmas de Desarrollo de Aplicaciones en Grid**. En este capítulo se analizan los características más importantes de los principales paradigmas de computación en Grid, indicando sus ventajas y desventajas, así como su adaptación al modelo de programación que se desea desarrollar. Posteriormente, se justifica la elección del estándar DRMAA en comparación con el resto de paradigmas analizados y se incluye una descripción de la especificación del estándar. Finalmente, se presenta el modelo de desarrollo DRMAA-Grid *W*ay y se remarca sus características fundamentales que le diferencian del resto de modelos existentes en la actualidad. El capítulo finaliza con un resumen de las principales características de este nuevo modelo de programación.

- **Estudio de Perfiles de Aplicaciones Grid con DRMAA-Grid *W*ay**. Con el objetivo de justificar la efectividad y eficiencia del modelo DRMAA-Grid *W*ay, este capítulo analiza su comportamiento en la ejecución de aplicaciones con perfil típico dentro de la tecnología Grid. Para ello, en primer lugar se presentará cada uno de los perfiles de ejecución indicando las principales motivaciones de su uso, para después estudiar el comportamiento del desarrollo de dichas aplicaciones en el modelo DRMAA-Grid *W*ay. La última sección de este capítulo presenta un nuevo tipo de algoritmo genético distribuido adaptado a entornos Grid heterogéneos, que utiliza el modelo DRMAA-Grid *W*ay para su desarrollo. El capítulo termina

con un resumen de las principales características del modelo DRMAA-Grid*W*ay que se deducen con el análisis de los resultados presentados en este capítulo.

- **Planificación de Iteraciones en Grid**. Este capítulo presenta un nuevo algoritmo de planificación adaptado a entornos Grid heterogéneos. En primer lugar se analiza el trabajo relacionado de los principales algoritmos de planificación de iteraciones que existen en la actualidad, desde los algoritmos estáticos, hasta los dinámicos distribuidos. Se estudia también sus carencias en relación a la adaptación de estos algoritmos clásicos a un entorno Grid heterogéneo, con el objetivo de motivar el desarrollo de un nuevo algoritmo de planificación. La segunda parte del capítulo presenta el nuevo algoritmo de planificación diseñado para su ejecución en entornos Grid, y analiza sus principales características así como los principales componentes que constituyen dicho algoritmo. El capítulo finaliza con un resumen de las principales aportaciones de este nuevo algoritmo.

- **Análisis del Algoritmo GTSS**. Con la finalidad de analizar el comportamiento del algoritmo GTSS, este capítulo se divide en dos partes fundamentales. La primera de ellas corresponde a la presentación de un nuevo simulador de entornos Grid adaptado al modelo DRMAA-Grid*W*ay, necesario para estudiar el comportamiento del algoritmo GTSS antes de su ejecución en un entorno real. Para ello, se analizan los principales simuladores Grid existentes en la actualidad, y se justifica la elección del conjunto de herramientas GridSim, como base fundamental para el desarrollo del nuevo simulador. La segunda parte del capítulo analiza los resultados obtenidos en la ejecución del algoritmo GTSS, tanto en un entorno simulado con Grid*W*aySim como en una plataforma Grid real. También se pone de manifiesto la adaptabilidad del modelo presentado, ya que la plataforma Grid real donde se realiza la ejecución utiliza un *middleware* distinto al que se ha utilizado a lo largo de los capítulos previos de la tesis. Finalmente pone de manifiesto las diferencias de ejecución existentes en un entorno real, y en el simulador Grid*W*aySim. El capítulo concluye con un resumen de las principales aportaciones presentadas.

Este trabajo concluye con un capítulo dedicado al resumen de las aportaciones que presenta esta tesis doctoral, incluyendo sus principales publicaciones. También se analizará el trabajo futuro que se origina a partir esta tesis doctoral.

Capítulo 2

Computación en GRID

El objetivo de este capitulo consiste en resumir las principales características del modelo de computación distribuida Grid, ya que esta tecnología ha sido utilizada para desarrollar la presente tesis doctoral. También mostrará los elementos que dificultan el desarrollo de aplicaciones Grid y las principales motivaciones que llevan a la creación de un modelo de programación que facilite la implementación de estas aplicaciones. La primera sección presenta una introducción del capítulo a través de la descripción de los paradigmas de computación distribuida más utilizados en la actualidad. Seguidamente en la sección 2.2 se detallarán las características más importantes de la tecnología Grid junto con sus principales aplicaciones y aportaciones así como su arquitectura y modelo de aplicación. En esta misma sección también se analizará el estandar *de facto* utilizado en esta tecnología, el paquete de herramientas Globus Toolkit. La sección 2.3 presentará los metaplanificadores más comúnmente utilizados en el modelo de computación distribuida Grid, y realizará una descripción más detallada del metaplanificador utilizado para la realización de esta tesis, el metaplanificador Grid *Way*. También se detallarán aspectos como su arquitectura y las ventajas que presenta este metaplanificador en comparación con el resto de metaplanificadores. El capítulo finalizará con una sección dedicada a las conclusiones que resumirá las principales motivaciones que han llevado al desarrollo del modelo de programación presentado en esta tesis.

2.1. Introducción

En diversas ocasiones autores de renombre como Ian Foster en [FK99] han comparado le tecnología Grid con la Red Eléctrica (*Electrical Power Grid*). La comparación

a la que se refieren estos autores es la que se realiza en el ámbito de acceso a los recursos. Es decir, la situación ideal consistiría en que cualquier entidad con necesidad de procesar una gran cantidad de datos, pudiera acceder a un conjunto de recursos computacionales geográficamente dispersos (como las centrales eléctricas) con la misma facilidad con la que un electrodoméstico se conecta a la red eléctrica [FK99]. De esta manera, una determinada organización no se vería obligada a adquirir un nuevo recurso computacional cada vez que lo necesitara, con el coste en tiempo y dinero que este proceso lleva implícito. Además, tampoco se encargaría de la gestión y el mantenimiento de estos recursos, incrementando así la velocidad en el proceso de computación de los datos. Para dar respuesta a estas necesidades, los autores Ian Foster y Carl Kesselman propusieron a mediados de los 90 un nuevo paradigma de programación denominado *Grid computing*, nombre derivado de su analogía a la red eléctrica, que tiene como objetivo principal facilitar el acceso a recursos computacionales distribuidos.

En la actualidad, existen otros modelos de computación en red que aportan mecanismos para aprovechar al máximo los recursos distribuidos que generalmente se encuentran infrautilizados, estos son:

- ***Cluster Computing***. Formado por un *cluster* dedicado de equipos como alternativa a la adquisición de un equipo multiprocesador. La principal ventaja de este modelo de computación en red consiste en el aumento de la relación coste/rendimiento, sin embargo su mayor desventaja radica en la dificultad de programación y mantenimiento de estos sistemas. Los *clusters* suelen estar gestionados por software de planificación como MOSIX [MOX99] de la Universidad de Israel o PBS [PBS08] desarrollado inicialmente en NASA.

- ***Intranet Computing***. Consiste en la unión de potencia computacional desaprovechada en los recursos hardware distribuidos en una red de área local. Su principal ventaja consiste en que proporciona un rendimientos semejante al ofrecido por sistemas de alto rendimiento con un coste económico casi nulo. Las herramientas SGE [SGE94] de Sun Microsystems, LSF [LSF01] de Platform Computing o Condor [CON08] de la Universidad de Wisconsin permiten la gestión oportunista de los recursos.

- ***Internet Computing***: Este modelo de computación aprovecha la potencia de los recursos distribuidos por Internet siguiendo el modelo cliente/servidor y obteniendo de esta manera un gran rendimiento en la ejecución de aplicaciones.

Sin embargo, estas ejecuciones están limitadas por el bajo ancho de banda y por la escasa seguridad en Internet. Dentro de este campo se pueden encontrar aplicaciones pertenecientes a los *Volunteer Computer Grids* como la dedicadas a investigaciones astronómicas o de enfermedades como el cáncer. Un ejemplo de este tipo de aplicaciones son el proyecto seti@home [SET03] o el proyecto BOINC [BOI08].

Aunque estos modelos posibilitan el aprovechamiento eficiente de los recursos dentro de una misma organización, ninguno permite unir dominios de administración diferentes, manteniendo la política de seguridad de cada centro y las herramientas de planificación. Así, la tecnología Grid surge como una alternativa que además de presentar todas las características expuestas anteriormente está basada en interfaces de acceso y protocolos abiertos, estándar y de propósito general. La computación en Grid permite también el acceso a recursos heterogéneos de diferentes organizaciones, ofreciendo un conjunto de protocolos, tecnologías y metodologías que facilitan dicho acceso.

2.2. Tecnología de Computación en Grid

El principal objetivo de la tecnología Grid consiste en unir de forma segura recursos de todo tipo, no sólo capacidad de cálculo y almacenamiento, conservando las políticas de seguridad, las herramientas de gestión interna y todos los componentes autónomos de las entidades que interviene en dicha unión. En la actualidad, existen diversas definiciones de la tecnología Grid, la primera definición fue propuesta por I. Foster y C. Kesselman en [FK99]:

> *"A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities."*

Esta definición se centra en el aspecto computacional de la tecnología Grid. Desde este punto de vista, la tecnología Grid debe proporcionar el acceso a recursos computacionales avanzados de manera robusta. También debe permitir que dicho acceso se realice mediante un interfaz uniforme, desde cualquier lugar y con un coste mínimo. Los mismos autores propusieron posteriormente en [FKT01] una nueva definición:

> *"Flexible, secure and coordinated resource sharing among dynamic collections of resources, individuals and institutions known as virtual organizations."*

Esta definición se centra en lo relativo a la compartición de recursos, de forma flexible segura y coordinada entre individuos, instituciones (conocidas como organizaciones virtuales) y colecciones de recursos. Las Organizaciones Virtuales (VO, *Virtual Organization*) están compuestas por agrupaciones de recursos de varias organizaciones distintas que colaboran para alcanzar una meta común. El uso de las VOs facilita la gestión de los recursos y de la seguridad. Además, la pertenencia a una VO no es permanente y por lo tanto puede cambiar según las necesidades de las organizaciones.

La definición mas extendida por la comunidad Grid es la propuesta por Ian Foster en [Fos02]:

> *A Grid is a system that...*
>
> 1. *...coodinates resources that are not subject to a centralized control...*
> 2. *...using standard, open, general-purpose protocols and interfaces...*
> 3. *...to deliver nontrivial qualities of services.*

Esta definición impone tres requisitos fundamentales que todo sistema debe cumplir para que pueda ser considerado como sistema Grid, a saber:

1. Debe carecer de un control centralizado, de esta manera se imposibilita la formación de cuellos de botella y además facilita la incorporación de nuevos nodos permitiendo así un mejor aprovechamiento de recursos heterogéneos.

2. Debe estar basado en protocolos e interfaces estándar, abiertos y de propósito general. Así, se simplifica la ejecución de aplicaciones a través del Grid, permitiendo que puedan coexistir aplicaciones con distintas políticas de autenticación, autorización o acceso a los recursos.

3. Debe proporcionar calidades de servicio no triviales. Es decir, la tecnología Grid debe ofrecer calidades de servicio superiores a las proporcionadas por la unión de los nodos individuales.

Desde este punto de vista, por ejemplo, un sistema de gestión de clusters, como SGE [SGE94], LSF [LSF01] o PBS [PBS08], instalado en un computador paralelo o

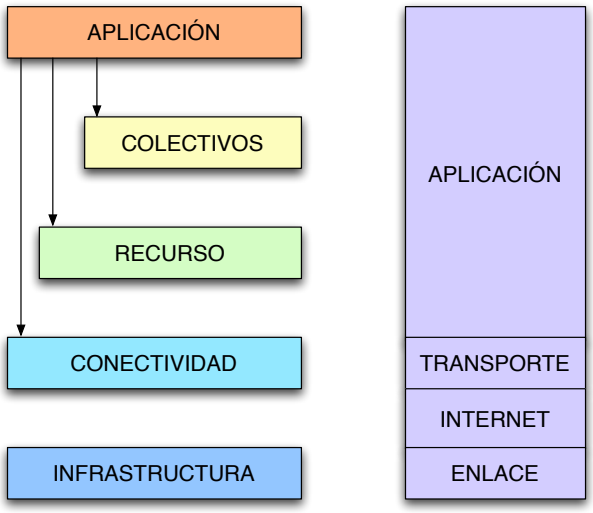

Figura 2.1: Arquitectura GRID vs Arquitectura Internet

sobre una red local, aunque proporciona garantías de calidad de servicio no es un Grid por sí mismo. Esto se debe principalmente a que estos sistemas disponen de un control centralizado de los componentes que gestionan y además utilizan interfaces y protocolos propietarios.

2.2.1. Arquitectura GRID

En el artículo *"Anatomy of the Grid: Enabling Scalable Virtual Organizations"* (2000) [FKT01], Ian Foster, Carl Kesselman, y Steven Tuecke proponen una arquitectura general que cataloga todos los componentes que forman un sistema Grid de acuerdo a su función y propósito [SC05]. La figura 2.1 muestra un esquema de esta arquitectura:

- **Infraestructura (*Fabric*)**. La infraestructura del Grid está compuesta por los recursos computacionales que se desean compartir. Estos pueden ser, ordenadores individuales, conjunto de ordenadores, *clusters*, supercomputadores y sistemas de almacenamiento. En esta capa también se incluye la infraestructura de red y sus mecanismos de gestión y control.

- **Conectividad (*Conectivity*)**. La capa de conectividad incluye los protocolos de comunicación y de seguridad que permiten la comunicación entre los recursos computacionales. Entre estos protocolos se pueden encontrar la pila de protocolos

TCP/IP, protocolos en redes de alta velocidad, SSL o Certificados X.509. Esta capa es especialmente importante ya que en la computación Grid intervienen múltiples recursos de distintas organizaciones con distintas políticas de seguridad.

- **Gestión de recursos individuales (*Resource*)**. Esta capa incluye servicios y protocolos para el control y gestión de recursos individuales. En particular existen dos tipos de protocolos principales:
  - *Protocolos de información*. Permiten obtener información sobre un determinado recurso (características técnicas, carga actual, precio, número de procesadores o memoria disponible).
  - *Protocolos de gestión*. Permiten el control de un determinado recurso, esto es: acceso, arranque, parada, monitorización, contabilidad o auditoria del recurso.
- **Colectivos (*Collective*), gestión de conjuntos de recursos**. Esta capa engloba los servicios que gestionan conjuntos de recursos. Los servicios más comunes que se pueden encontrar en esta capa son:
  - *Servicios de directorio*. Permiten descubrir los recursos de organizaciones virtuales, así como consultar sus propiedades.
  - *Servicios de planificación y asignación*. Permiten la correcta asignación de cada tarea a un recurso.
  - *Servicios de monitorización y diagnostico*. Informan sobre el estado de los recursos del Grid.
  - *Servicios de contabilidad*. Realizan operaciones de cálculo del coste de la utilización de varios recursos heterogéneos.
  - *Servicios de gestión de datos*. Gestionan las bases de datos necesarias en cada recurso.
- **Aplicaciones (*Applications*)**. Las aplicaciones Grid acceden a la infraestructura del Grid a través de las distintas capas. Según las exigencias de la aplicación, puede ser necesario pasar por todas las capas o conectarse directamente a la infraestructura. Las principales aplicaciones que pueden beneficiares del uso de la tecnología Grid son las siguientes [Sot03]:

- *Supercomputación distribuida.* Se basa en aplicaciones cuyas necesidades son imposibles de satisfacer por una única organización. Estas aplicaciones se distinguen porque requieren demandas puntuales e intensivas de computación. Ejemplos de estas aplicaciones son simulaciones de complejos fenómenos físicos o cálculos numéricos [NUG00].
- *Sistemas distribuidos en tiempo real.* Son sistemas que generan un flujo de datos a alta velocidad que debe ser analizado en tiempo real. Algunos ejemplos son, e-Medicine, experimentos de física de alta energía, control remoto de un recurso no-trivial (microscopios o equípo médico) [Res03].
- *Servicios puntuales.* Se producen mediante accesos puntuales a uno o varios recursos. Son semejantes a las dos aplicaciones anteriores, pero se diferencian en que estos servicios no se refieren a 'potencia computacional' y no tienen porque ser en tiempo real. Un ejemplo de este tipo de aplicación es el acceso a hardware específico para ciertos tipos de análisis (químico o biológico) [AEH+04].
- *Procesos intensivos de datos.* Son aplicaciones que trabajan con grandes volúmenes de datos y que son imposibles de almacenar en un único nodo. En su lugar, los datos se distribuyen a lo largo del Grid [GRI00].
- *Entornos virtuales de colaboración (Teleinmersión).* Este tipo de aplicaciones utilizan la potencia computacional y la naturaleza distribuida del Grid para crear entornos virtuales en 3D distribuidos [REA07].

Aunque existen diversas implementaciones de la tecnología Grid como GRIA [STRZ05], UNICORE [Rom02] o ARC [EnK+07], para el desarrollo de esta tesis se ha escogido la implementación denominada Globus Toolkit. La elección de dicha implementación se debe a que esta es la más utilizada por la comunidad Grid y además es la considerada como el estándar *de facto* de esta tecnología. La implementación Globus Toolkit está compuesta por un conjunto de servicios y protocolos proporcionados por el proyecto Globus [GT08]. La siguiente sección describe las principales características de este proyecto, en su última versión Globus Toolkit 4 (GT4).

2.2.2. Globus ToolKit

El estándar *de facto* para la implementación de aplicaciones en sistemas Grid está constituido por un conjunto de herramientas denominado *Globus Toolkit* [GT08].

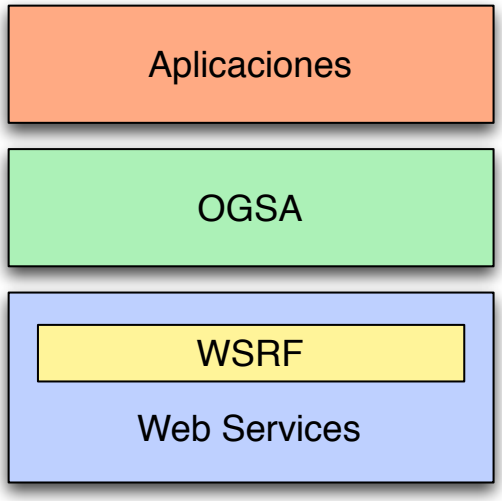

Figura 2.2: Estructura en capas de GT4

Aunque para el desarrollo de la presente tesis se han utilizado tanto la versión *Globus ToolKit 2* [FK97] (sin servicios Web) como la versión *Globus ToolKit 4* [Fos05] (con servicios Web), esta sección se centrará en el análisis de la versión 4 ya que incorpora toda la funcionalidad contenida en la versión 2 más las funcionalidades añadidas por los servicios web. *Globus Toolkit* es un software libre, de código abierto y organizado como una colección de componentes debilmente acoplados, cuyo desarrollo está liderado por diversos grupos de investigación ubicados en la *University of Southern California* y en la *University of Chicago*.

La figura 2.2 muestra el diagrama que relaciona GT4 con tres elementos clave de esta versión, los servicios Web, la arquitectura OGSA y la especificación WSRF. Antes de especificar la arquitectura general de Globus toolkit en su versión 4 es importante realizar un análisis previo de estos elementos:

- **Servicios Web**. Permiten implementar aplicaciones cliente/servidor con las siguientes características:
  - Son independientes de plataforma, es decir, un servicio implementado en Java con Windows, puede ser utilizado por un cliente que este implementado en lenguaje C bajo OSX.
  - La trasferencia de mensajes se realiza utilizando el protocolo HTTP, facilitando de esta manera la comunicación entre las aplicaciones.

  - Debido a su naturaleza, las aplicaciones desarrolladas a través de los servicios web se ajustan a entornos débilmente acoplados y heterogéneos.

  Por todas estas características, los servicios web se presentan como una buena opción para el desarrollo de aplicaciones en sistemas Grid.

- **OGSA**. Este acrónimo hace referencia a las siglas *Open Grid Services Architecture*, y consiste en la especificación de una arquitectura de servicios abiertos en Grid. Esta arquitectura ha sido desarrollada por el Global Grid Forum [GGF08], y define una arquitectura abierta, común y estandar, para todas aquellas aplicaciones basadas en sistemas Grid [SC05]. El principal objetivo de la arquitectura OGSA consiste en la estandarización de los servicios que comúnmente se pueden encontrar en un sistema Grid. Servicios como la gestión de trabajos y recursos o seguridad. Dicha estandarización se realiza mediante la especificación de interfaces estándar y de requisitos para cada uno de los servicios especificados. Entre estos servicios se encuentran: (i) Servicio de Gestión de VO, controla los usuarios y nodos que pertenecen a una determinada VO; (ii) Servicio de Gestión y Descubrimiento de Recursos, permite que las aplicaciones puedan ejecutarse en el recurso más apropiado; y (iii) Servicio de Gestión de Trabajos, para facilitar la ejecución de trabajos por parte de los usuarios. Todos estos servicios que especifica la arquitectura OGSA, deben ser servicios con estado.

- **WSRF**. Los servicios web proponen una alternativa para la implementación de aplicaciones en Grid ya que permiten el desarrollo de los servicios especificados en la arquitectura OGSA. Sin embargo, la arquitectura OGSA define servicios con estado obligando de esta manera a que los servicios web definidos sea servicios con estado. Para facilitar el desarrollo de estos servicios, la especificación WSRF (*Web Services Resource Framework*) propone una infraestructura de desarrollo que permite la implementación de los servicios propuestos por la arquitectura OGSA a través de servicios web con estado. Para almacenar toda la información relativa al estado de los servicios web, la infraestructura WSRF define una nueva entidad denominada *resource* (recurso), es importante destacar que esta nueva entidad es una estructura lógica, y por lo tanto, no se debe confundir con el recurso físico asociado al Grid. Los recursos de GT4 se identifican a través de una clave única y su estructura depende del servicio web al que esté asociado. Dicha estructura la define el desarrollador del servicio web. La tupla formada

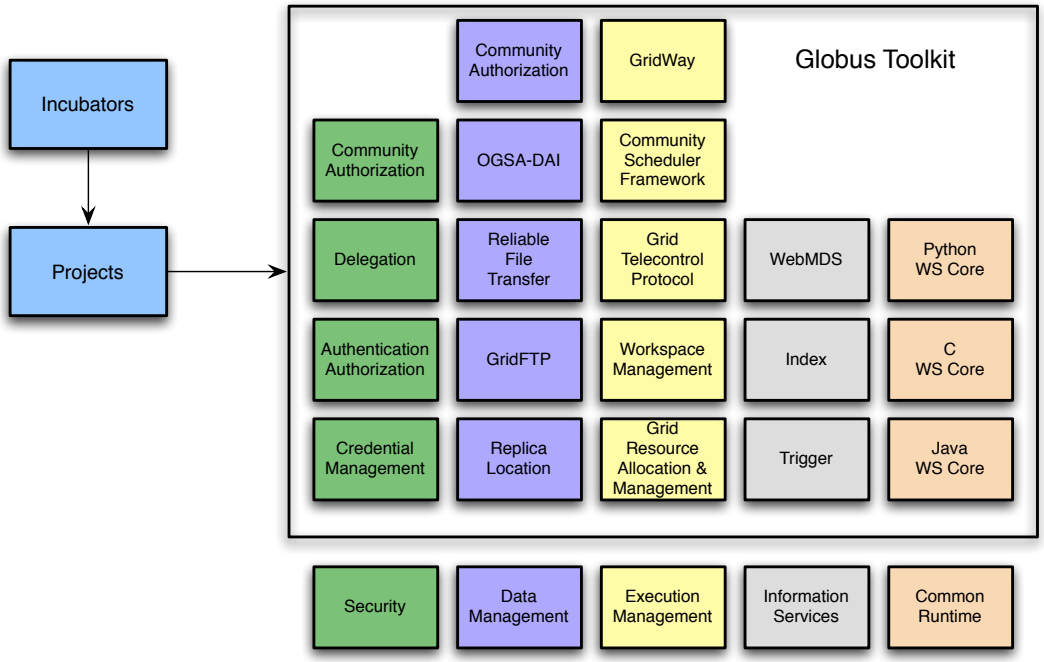

Figura 2.3: Componentes de Globus versión 4

por el servicio web y su recurso asociado se denomina *WS-Resource*. WSRF está constituido por un conjunto de especificaciones que definen la gestión de los *WS-Resources*, estas son: (i) *WS-ResourceProperties* define un conjunto de interfaces para el acceso, modificación y consulta de las propiedades de cada recurso Globus; (ii) *WS-ResourceLifeTime* define diferentes mecanismos de gestión del ciclo de vida de los recursos Globus; (iii) *WS-ServiceGroup* define operaciones para la gestión de grupos de *WS-Resources*; (iv) *WS-BaseFaults* ofrece un mecanismo estándar de representación de errores ocurridos durante la invocación de un *WS-Service*.

La figura 2.3 muestra la arquitectura general del conjunto de herramientas Globus en su versión 4, *GT4*. Esta arquitectura se divide en cinco grandes areas: Seguridad (*Security*), Gestión de Datos (*Data Management*), Gestión de Ejecución (*Execution Management*), Servicios de Información (*Information Services*) y Componentes Comunes (*Common Runtime*). En las siguientes secciones se detallan las características de los principales componentes de esta arquitectura.

Infraestructura de Seguridad

Los componentes de seguridad de la versión 4 de Globus forman la Infraestructura de Seguridad Grid (GSI, *Grid Security Infrastructure)*. Dicha infraestructura facilita la seguridad de las comunicaciones así como la aplicación de políticas uniformes a través de los distintos sistemas. La infraestructura de seguridad de Gobus, se basa en las siguientes necesidades o motivaciones [FKTT98]:

- *Comunicación Cifrada*. GSI ofrece una comunicación segura y autenticada entre los distintos elementos que componen un Grid.
- *Registro Único*. Para los usuarios del Grid, incluyendo delegación de credenciales para computaciones con múltiples recursos.

Por otro lado, tal y como muestra la figura 2.3, la infraestructura de seguridad (GSI) está formada por los siguientes componentes:

- *Autorización Comunitaria (Community Autorization)*. Permite la gestión de políticas de autorización de los recursos de las organizaciones virtuales (V0) por parte de estas.
- *Delegación (Delegation)*. Es una extensión del protocolo TLS estándar que reduce el número de veces que el usuario debe introducir su contraseña. Permite, de este modo, la creación de un certificado delegado (*proxy certificate*) para el acceso a los recursos del Grid. Un proxy consiste en un nuevo certificado (con una nueva clave pública) y una nueva clave privada. El nuevo certificado contiene la identidad del propietario, modificada para indicar que es delegado. El nuevo certificado es firmado por el propietario, en lugar de la CA. Los certificado delegados tienen un tiempo de vida limitado (12 horas por defecto).
- *Autenticación y Autorización (Autentication and Autorization)*. GSI incluye herramientas para el control de acceso a los recursos y servicios, permitiendo el uso de distintos métodos de autorización para dicho acceso. Para ello, GSI utiliza el lenguaje estándar SAML de OASIS [SAM08] que define las estructuras necesarias para llevar a cabo los diferentes mecanismos de acceso a los servicios y recursos que constituyen el Grid.
- *Gestión de Credenciales (Credential Management)*. Este componente incluye una autoridad de certificación básica denominada *SimpleCA*, necesaria para usuarios

que no disponen del acceso a una autoridad de certificación real. También incluye un contenedor de credenciales en línea.

Gestión de Datos

Globus incluye componentes para el descubrimiento, transferencia, gestión y acceso de datos en entornos Grid de alto rendimiento. Los componentes más importantes son:

- *GridFTP*. Proporciona transferencias de datos seguras y confiables entre los nodos del Grid [ABB+02, ABF+02]. El protocolo GridFTP, se basa en el protocolo estándar FTP (*File Transfer Protocol*), extendiendo dicho protocolo con otras funcionalidades como transferencias multicanal, auto-ajuste, segmentación de comandos, transferencia entre terceros y seguridad.

- *Servicio de Transferencia de Ficheros Fiable (Reliable File Transfer, RFT)*. Es un componente que utiliza GridFTP internamente, y permite transferencias de grandes cantidades de datos entre los nodos. Además, añade nuevas funcionalidades sobre GridFTP como la posibilidad de reanudar transferencias interrumpidas.

- *Localización de Replicas (Replica Location Service, RLS)*. Permite a los usuarios localizar donde se encuentran las replicas de los datos dentro de una organización virtual.

- *Replica de Datos (Data Replication Service, DRS)*. El servicio de réplica de datos utiliza los servicios RLS y RFT para garantizar la disponibilidad de las copias locales de las réplicas en aquellos nodos que las necesiten.

- *OGSA-DAI*. El servicio de integración y acceso a datos OGSA permite que el acceso y la integración de datos en un Grid esté disponible en diferentes formatos, (ficheros de texto, bases de datos o ficheros XML).

Gestión de Ejecución

El conjunto de componentes que intervienen en la gestión de ejecución, se encargan del despliegue, monitorización y ejecución de programas. La gestión de ejecución está constituida por los siguientes servicios:

- *GRAM*. El gestor de recursos de Globus, GRAM (*Grid Resource Allocation and Management*) [CFK+98], proporciona tanto la ejecución remota como su control y

monitorización. GRAM proporciona un único interfaz estándar para la ejecución de trabajos, simplificando de esta forma el uso de sistemas remotos.

- *Interfaz de Planificadores (Community Scheduler Framework, CSF)*. Este servicio ofrece un interfaz común a planificadores de recursos como PBS, Condor, LSF o SGE.

- *Gestor de Espacios de Trabajo (Management WorkSpace)*. Este componente permite crear y gestionar de forma dinámica espacios de trabajo en nodos remotos.

- *Grid*W*ay*. Desde la versión 4.0.5 Gobus Toolkit incluye Grid *W*ay, un metaplanificador que permite el uso eficiente y fiable de recursos computacionales (*clusters*, servidores o supercomputadores), gestionados por distintos sistemas LRM (*Local Resource Management*), como PBS, SGE, LSF o Condor dentro de una misma organización o dispersados en distintos dominios de administración. Este componente se explicará con más detalle en la sección 2.3.4.

Servicios de Información

Los servicios de información constituyen un importante pilar dentro de la infraestructura Grid, ya que proporciona servicios vitales para el descubrimiento y la monitorización, permitiendo de esta manera, planificar y adaptar el comportamiento de las aplicaciones. Este servicio se denomina como MDS (*Monitoring and Discovery Service*) y proporciona un acceso uniforme, flexible, escalable y eficiente a la información estática y dinámica de recursos. Los servicios de información más importantes son:

- *Servicio de Índice (Index Service)*. Este componente es el encargado de gestionar la incorporación de recursos dentro de las V.O.

- *Servicio Acciones Disparadas por Eventos (Trigger Service)*. Recolecta información de los recursos, permitiendo de esta manera que se puedan realizar acciones sobre el recurso basadas en los datos obtenidos por este servicio.

- *WebMDS*. Permite a los usuarios obtener información de monitorización utilizando un interfaz web.

2.3. Metaplanificadores en Grid

Aunque Globus ofrece un conjunto de servicios para facilitar la ejecución de trabajos en Grid, el desarrollo de aplicaciones bajo esta tecnología, requiere de un esfuerzo adicional por parte del usuario, ya que tiene que tener en cuenta aspectos tan importantes como la complejidad de la computación en Grid, la heterogeneidad, el dinamismo y una elevada tasa de fallos de los recursos. En los últimos años han aparecido una serie de herramientas que facilitan la gestión de los recursos así como el desarrollo de aplicaciones Grid, dichas herramientas se denominan metaplanificadores. Los metaplanificadores realizan tareas de gestión de recursos, gestión de trabajos, seguridad o gestión de información. Entre los metaplanificadores más importantes se encuentran CSF (*Community Scheduler Framework*) [CSF08], GSB (*Grid Service Broker*) [GSB08], GRMS (*GridLab Resource Management System*) [GRM08] y Grid*W*ay [GW08].

2.3.1. CSF

El metaplanificador CSF (*Community Scheduler Framework*) desarrollado por *Platform Computing* junto con la Universidad Jili en China, es una implementación en código abierto de servicios básicos Grid. CSF utiliza los servicios ofrecidos por la herramienta Globus, y presenta un entorno de ejecución de trabajos cuya gestión puede llevarse acabo utilizando diferentes gestores de recursos como LSF, PBS o SGE. Además, CSF permite la reserva de recursos así como mecanismos simples de planificación.

La figura 2.4 muestra la arquitectura del metaplanificador CSF. Como se puede observar el metaplanificador se compone de [Smi03]: (i) Servicio de Trabajos, crea, monitoriza y controla los trabajos computados; (ii) Servicio de Reservas, garantiza que el recurso está disponible para un trabajo en ejecución; (iii) Servicio de Planificación, permite definir políticas de planificación a nivel de las VO y a diferentes niveles de gestión de los recursos. Estos elementos se comunican con el gestor de recursos GRAM, o con el adaptador de gestor de recursos RM, que ofrece un interfaz de servicio entre el protocolo de servicios Grid y los gestores de recursos LSF o Altair PBS [NSJ04]. Los gestores de recursos GRAM y RM se comunican a su vez con los gestores de recursos locales (SGE, PBS, LSF). Finalmente, los recursos se monitorizan a través del Servicio Global de Información.

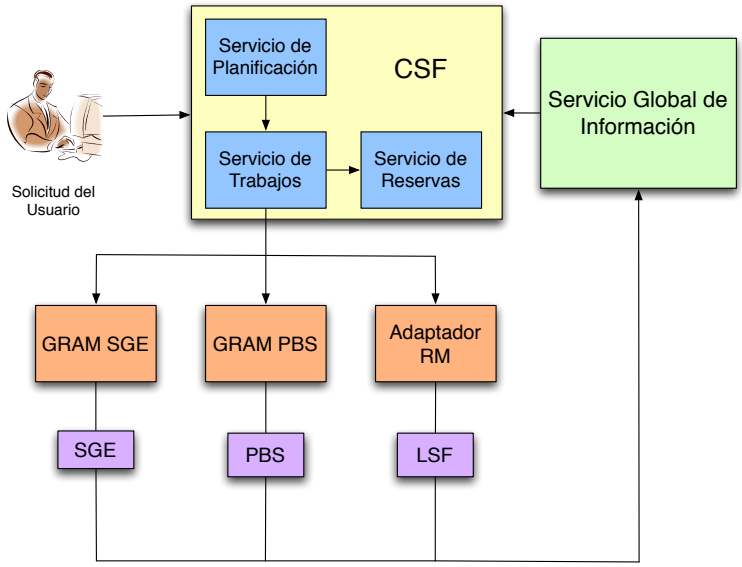

Figura 2.4: Arquitectura CSF

2.3.2. GSB

El metaplanificador GSB (*Grid Service Broker*) se encarga de realizar diversas funciones requeridas por aplicaciones Grid, como descubrir los recursos que más se ajustan a una determinada aplicación, planificar los trabajos de acuerdo a un *deadline* preestablecido o gestionar los errores que se producen durante la ejecución [VBW04]. Además, este metaplanificador gestiona la comunicación entre recursos con distintos *middleware* como Globus, Alchemi o Unicore, facilitando el acceso al Grid por parte de usuarios no familiarizados con los sistemas distribuidos.

La figura 2.5 muestra un esquema de la arquitectura del planificador GSB. La principal diferencia de este metaplanificador con respecto al metaplanificador CSF, reside en el Gestor de *Middleware*, ya que permite al usuario ejecutar trabajos independientemente del *middleware* Grid que se encuentre en el recurso remoto. Facilitando de esta manera la ejecución de los trabajos por parte del usuario. Otro modulo a tener en cuenta es el gestor de parámetros, que se encarga de obtener los parámetros de la aplicación y adaptarlos al *middleware* apropiado.

2.3.3. GRMS

GRMS (*GridLab Resource Management System*) es un metaplanificador incluido dentro del proyecto GridLab [GL06], basado en Globus 2.4, que contiene mecanismos

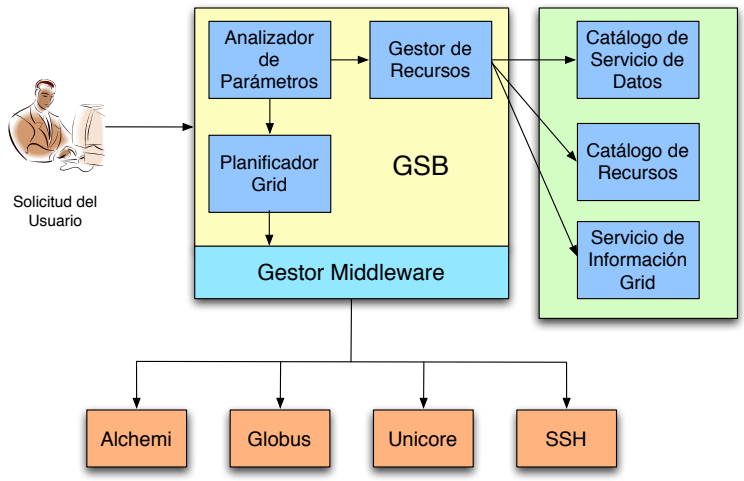

Figura 2.5: Arquitectura GSB

de planificación simple, permite la migración de trabajos, incorpora un propio lenguaje de descripción de trabajos denominado GLJ (*GridLab Job Definition*) y permite la ejecución de trabajos con dependencias. Además también hace uso de los servicios web para permitir su ejecución en versiones posteriores de Globus.

La figura 2.6 representa un esquema de los módulos más importantes de la arquitectura del metaplanificador GRMS. Como se puede observar, este metaplanificador se compone de los siguientes módulos: (i) Módulo de Gestión de *Workflows*, que permite la ejecución de trabajos con dependencias; (ii) Cola de Trabajos, se encarga de encolar los trabajos que van llegado al metaplanificador según el algoritmo FIFO (*First In First Out*); (iii) *Broker*, dirige todos los procesos involucrados en la ejecución de los trabajos, como por ejemplo, la elección del recurso más apropiado para la ejecución de un trabajo, para ello se comunica con el Registro de Trabajos, el Gestor de Trabajos y el modulo de Descubrimiento de Recursos.
La diferencia más destacable con respecto a los anteriores metaplanificadores reside en la incorporación del módulo gestor de *workflows*, aún así no incorpora tantas funcionalidades como los metaplanificadores presentados en secciones previas.

2.3.4. Grid *W*ay

El metaplanificador Grid *W*ay desarrollado por el grupo de Arquitectura de Sistemas Distribuidos de la Universidad Complutense de Madrid, se encuentra ubicado dentro del proyecto Globus, y permite la compartición fiable y eficiente de recursos Grid,

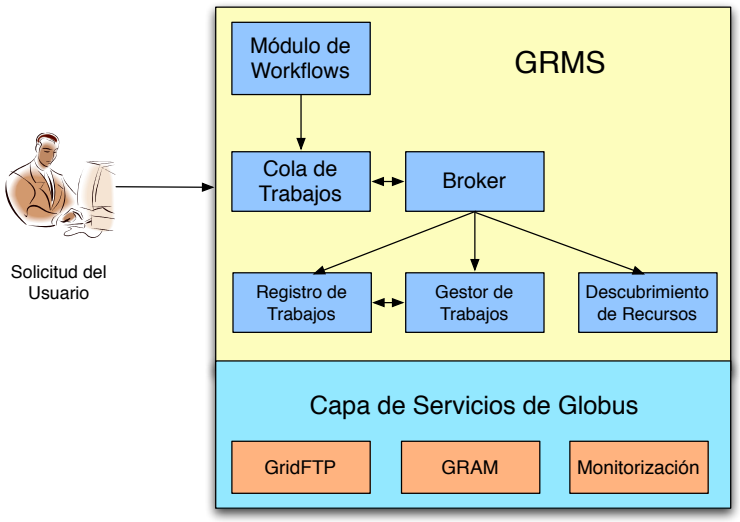

Figura 2.6: Arquitectura GRMS

como clusters, servidores y supercomputadores. Estos recursos pueden estar gestionados por diferentes gestores de recursos locales (LRM, *Local Resource Management*), como pueden ser PBS, SGE, LSF o Condor. Las características más significativas del metaplanificador Grid*Way* son:

- *Arquitectura fiable y extensible.* Grid*Way* al estar diseñado de manera modular, permite la adaptación a diferentes infraestructuras como EGEE [EGE08], TeraGrid [TG08], OSG [OSG08] o NorduGrid [NG08].

- *Soporte para estándares OGF.* El metaplanificador Grid*Way* implementa estándares OGF(*Open Grid Forum*) [OGF08] como el interfaz de programación DRMAA [DRM08a], para distintos lenguajes de programación como C, Java o Perl. De esta manera, asegura la compatibilidad entre aplicaciones que se ejecuten en recursos cuyos LRM implementen dicho estándar, como ocurre con SGE, Condor y Torque. Además, permite la ejecución, monitorización, sincronización y control de trabajos usando el estandar JSDL [JSD08] de OGF.

- *WorkFlows.* Permite a los usuarios ejecutar trabajos con dependencias.

- *Ejecución Adaptativa.* El metaplanificador Grid*Way* realiza migración de trabajos, siempre que sea requerida por el usuario, permitiendo de esta manera que la ejecución de los trabajos se adapte al estado del Grid.

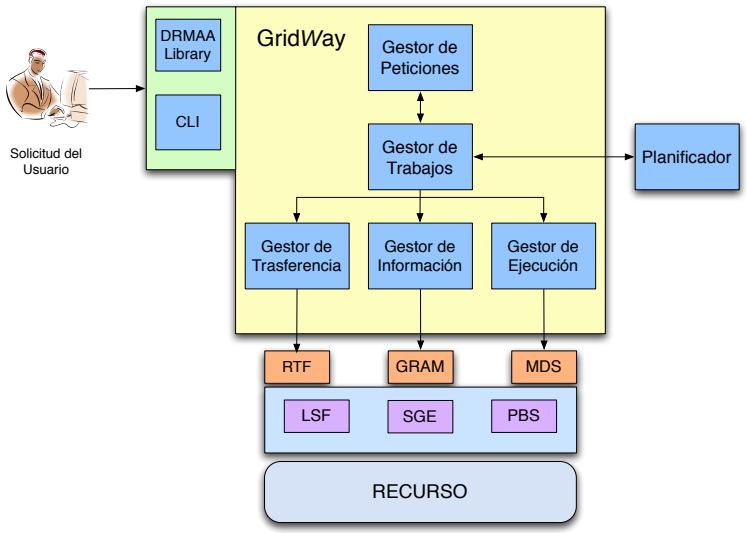

Figura 2.7: Arquitectura Grid *Way*

Arquitectura Grid *Way*

La figura 2.7 representa las principales características de la arquitectura Grid *Way*. En primer lugar, el usuario lanza los trabajos utilizando el estándar DRMAA o a través del intefaz en línea de comandos (CLI, *Command Line Interfaz*). Estas peticiones son tratadas por el Gestor de Peticiones que se comunicará con al Gestor de Trabajos, para que la ejecución del trabajo se realice de forma satisfactoria. El Gestor de Trabajos se comunicará con el planificador, para así realizar una planificación eficiente del trabajo a ejecutar. Seguidamente, el Gestor de Trabajos, utilizará el Gestor de Transferencia, el Gestor de Información y el Gestor de Ejecución para lanzar los trabajos a un recurso concreto. Dicho recurso utilizará el LRM adecuado, para ejecutar el trabajo localmente.

Además de la arquitectura mostrada en la figura 2.7, Grid *Way* ofrece otros módulos que aumentan la funcionalidad de este metaplanificador: (i) Cola de Transferencia SGE, permite la ejecución en Globus con Grid *Way* de trabajos que originalmente se enviaron desde un cluster gestionado por Sun Grid Engine; (ii) GridGateWay, ofrece un interfaz de servicios web para Grid *Way* que permite la creación de estructuras Grid jerárquicas. Finalmente, existen otros componentes desarrollados por otros grupos como TIGRE-GW-Client o GWGUI [GW08].

Grid *Way* incorpora diversas funcionalidades en comparación con los metaplanificadores expuestos anteriormente, sobretodo a lo referente a la ejecución de trabajos

de una manera sencilla y fiable siguiendo además los estándares propuestos por OGF. Además, Grid*Way* aglutina las principales ventajas ofrecidas por los metaplanificadores CSF, GSB o GRMS.

2.4. Conclusiones

En este capítulo se ha presentado la tecnología Grid y su estándar *de facto* correspondiente al conjunto de herramientas constituido por Globus Toolkit, que facilita tareas como la gestión de recursos heterogéneos o la ejecución de trabajos en Grid. Sin embargo, la implementación de aplicaciones Grid no es una tarea trivial ya que el programador debe considerar determinadas características intrínsecas de la computación en Grid como el dinamismo del sistema, la heterogeneidad de los nodos o una alta tasa de fallos existente. Para resolver estas carencias, han surgido unas herramientas denominadas metaplanificadores que, en general, proponen diferentes soluciones que facilitan la gestión de recursos y trabajos en Grid heterogéneos. En este capítulo se han presentado los metaplanificadores más utilizados en la actualidad, CSF (*Community Scheduler Framework*), GSB (*Grid Service Broker*), GRMS (*GridLab Resource Management System*) y Grid*Way*. Con el análisis de cada uno de estos metaplanificadores, se ha llegado a la conclusión que el metaplanificador que reúne un conjunto de características más completo es Grid*Way*, ya que permite la ejecución de trabajos de una manera sencilla y fiable siguiendo además los estándares propuestos por OGF. Además, Grid*Way* dispone de una completa CLI (*Command Line Interface*) que permite al usuario tener control en todo momento de los trabajos en ejecución. Por otra parte, Grid*Way* se encarga también de la gestión de los recursos Grid, desvinculando de esta manera al usuario de esta tarea y facilitando así una ejecución más eficiente de los trabajos.

Aunque Grid*Way* ofrece muchas ventajas para la ejecución eficiente de trabajos en Grid heterogéneos, el desarrollo de aplicaciones en estos entornos, sigue necesitando de un estándar de programación que permita el desarrollo de aplicaciones Grid de manera fiable y sencilla. Por lo tanto, es evidente la necesidad de otro componente que se sitúe entre el desarrollador de la aplicación y el metaplanificador, en este caso Grid*Way*, que facilite la implementación y el despliegue de aplicaciones en Grid heterogéneos. Este es el caso del API DRMAA, que ha sido diseñado como estándar para el desarrollo de aplicaciones distribuidas independientemente del gestor de recursos utilizado. De

esta manera, el API DRMAA, junto con el metaplanficador Grid*W*ay constituyen el modelo de programación presentado en esta tesis doctoral.

Capítulo 3

Paradigmas de Desarrollo de Aplicaciones en Grid

El desarrollo de aplicaciones en entornos Grid requiere por parte del programador de un conocimiento completo de esta tecnología así como de una gran experiencia en la implementación de aplicaciones para este entorno. Este capítulo presenta un nuevo modelo de desarrollo de aplicaciones en Grid cuyo objetivo es facilitar la implementación de estas aplicaciones ofreciendo además un interfaz de programación estándar. De esta manera, el modelo abstrae al desarrollador de tareas relacionadas con la gestión, mantenimiento y disponibilidad de los recursos conservando el control sobre los trabajos que constituyen la aplicación Grid, además, permite desacomplar las aplicaciones de la infraestructura Grid donde se ejecuten. Este nuevo paradigma, denominado modelo DRMAA-Grid *Way*, se basa en tres elementos principales, Globus Toolkit, el metaplanificador Grid *Way* y el API para el desarrollo de aplicaciones distribuidas DRMAA.

Como introducción a los distintos paradigmas de desarrollo en Grid que se utilizan en la actualidad, la sección 3.1 analiza las principales características de estos paradigmas mostrando las ventajas y desventajas de cada una de las soluciones presentadas. Seguidamente, la sección 3.5 muestra las principales aportaciones que incorpora el API estándar DRMAA, enfatizando los motivos por los cuales se ha escogido este paradigma de programación y no cualquiera de los presentados en las secciones previas. Finalmente, en la subsección 3.5.3, se presenta el modelo del programación de desarrollo de aplicaciones Grid propuesto en esta tesis denominado DRMAA-Grid *Way*.

3.1. Introducción

La complejidad intrínseca de las aplicaciones Grid ha facilitado la aparición de herramientas e interfaces que ayudan a los usuarios en su desarrollo y ejecución, de entre todas ellas se pueden encontrar:

- **Portales Web**. Permiten la ejecución de aplicaciones en entornos Grid a través de servidores Web, un ejemplo de este tipo de interfaces de usuario es el proyecto In-VIGO [ACC+05] desarrollado en la Universidad de Florida. In-VIGO permite la ejecución vía Web de aplicaciones en un entorno distribuido formado tanto por recursos físicos como virtuales, de esta manera desvincula a los clientes de las aplicaciones de la complejidad inherente de la computación Grid.

- **GRID superscalar**. Desarrollado en el Centro Nacional de Supercomputación (BSC-CNS) GRID superscalar [BLS+03] se compone de un entorno de programación Grid que permite la ejecución eficiente de programas en Grid computacionales. Para ello, *paraleliza* en tiempo real y a nivel de tarea aplicaciones secuenciales, generalmente de grano grueso, que posteriormente se ejecutarán en un Grid.

- **Introduce**. Está constituido por un conjunto de herramientas que facilitan el desarrollo de servicios web en Grid. Introduce [HOL+07] se distribuye conjuntamente con Globus Toolkit desde la versión 4.0.2 y permite la implementación de servicios web, métodos y recursos Globus utilizando un interfaz gráfico. Así, el desarrollador solo es responsable de la implementación de la lógica actual de los métodos que describe, ya que, Introduce se encargará de incorporar todo el código relacionado con la arquitectura Grid simplificando de esta manera la labor del desarrollador.

- **Interfaces Específicas**. Algunos proyectos Grid incorporan dentro de su infraestructura las herramientas necesarias para facilitar el desarrollo de aplicaciones Grid en su marco de ejecución. Este es el caso del proyecto $caBIG^{TM}$(*cancer Biomedical Informatics* $Grid^{TM}$) [caB08] que consiste en una red de información distribuida para la investigación sobre la enfermedad del cáncer. El proyecto $caBIG^{TM}$ ofrece al usuario un conjunto de aplicaciones, bases de datos, aplicaciones web y herramientas software que facilitan la recolección, análisis e integración de los datos necesarios para llevar a cabo dicha investigación.

Aunque estas herramientas facilitan la ejecución y el desarrollo de aplicaciones Grid bajo determinadas condiciones, lo deseable seria disponer de un modelo general para el desarrollo de dichas aplicaciones. De este modo, aunque Globus Toolkit junto con la herramienta Grid*W*ay facilitan la ejecución de trabajos en Grid, el desarrollo de aplicaciones en estos entornos sigue necesitando de esfuerzo adicional por parte del usuario, ya que este se ve en la necesidad de comprender detalladamente el comportamiento del metaplanificador y del *middleware* a utilizar para que la aplicación llegue a buen término. Para facilitar estas labores al desarrollador el paradigma o modelo de desarrollo debe ofrecer las siguientes características:

- **Transparencia**. El modelo de desarrollo debe abstraer al desarrollador de todas las tareas relacionadas con la gestión de los recursos del Grid, así como de las tareas vinculadas con la planificación de los trabajos que se ejecuten. Sin embargo, el desarrollador debe tener en todo momento control sobre los trabajos en ejecución para así poder comprobar y modificar el estado de los mismos.

- **Fiabilidad**. Si durante la ejecución de la aplicación un recurso falla, el modelo debe permitir que la ejecución continúe enviando, por ejemplo, los trabajos asignados a ese recurso a otro que se encuentre disponible. Sea cual sea la tarea que se lleve acabo, esta debe ser transparente tanto desde el punto de vista del desarrollador como del usuario de la aplicación.

- **Adaptación Dinámica**. Los trabajos puden migrar a los recursos que mejor se adapten a sus necesidades durante la ejecución de la aplicación. Además, si se incorpora un nuevo recurso al Grid, este podrá ser utilizado durante el resto de la ejecución de la aplicación.

Todas estas características permiten desacoplar las aplicaciones Grid de la infraestructura Grid a la que se encuentre asociada. En este capítulo se presenta un nuevo modelo de desarrollo que cumple todas estas características facilitando de esta manera el desarrollo de aplicaciones Grid. Este modelo de desarrollo está formado por el *middleware* Globus, el metaplanificador Grid*W*ay y el paradigma de programación de aplicaciones distribuidas DRMAA. En el capítulo anterior se expusieron las principales características del *middleware* Globus y del metaplanificador Grid*W*ay que motivan a la elección de estas herramientas para constituir el modelo de desarrollo de aplicaciones Grid que se presenta en este capítulo. A continuación se analizarán los paradigmas de programación de aplicaciones distribuidas más utilizados concretamente, MPICH-G2, SAGA,

Grid-RPC y DRMAA, y se justificará el por qué de la elección de paradigma DRMAA para la creación de este nuevo modelo de desarrollo.

3.2. MPICH-G2

MPICH-G2 es una versión adaptada a entornos Grid de una implementación del estándar MPI denominada MPICH. El estandar MPI (*Message Passing Interface*) es una especificación basada en el paso de mensajes realizada por consenso por el MPI Forum [MPI08] para los lenguajes de programación C, C++ y Fortran. Concretamente, MPI especifica un modelo de programación para desarrollar aplicaciones que pueden ser utilizadas en sistemas de SPMD (*Single Program Multiple Data*). Siguiendo este modelo, un mismo programa se ejecuta en todos los nodos del sistema, mientras que los datos son distribuidos equitativamente entre cada uno de los nodos, de tal forma que cada nodo procesa solo aquellos datos que le corresponden. De esta manera, la comunicación entre los distintos procesos, recuérdese que cada proceso ejecuta el mismo programa, se realiza a través de llamadas a funciones de envío y recepción de mensajes. El estándar MPI ha ido evolucionando desde la versión MPI 1.0 publicada en el año 1994, hasta su versión más reciente MPI 2.0 [MPI98], que incorpora extensiones para la ejecución de trabajos según el modelo MPMD (*Multiple Program Multiple Data*), permitiendo a cada proceso ejecutar un programa distinto.

Aunque está orientado preferiblemente a sistemas homogéneos, MPI permite exportar sus aplicaciones a sistemas multiprocesadores, multicomputadores o sistemas heterogéneos. Además, al ser un estándar basado en funciones facilita la portabilidad y el desarrollo de aplicaciones, ya que el usuario desarrolla la aplicación paralela como si de un programa secuencial se tratase incorporando las funciones MPI que sean necesarias. Existen diversas implementaciones del estándar MPI, como LAM/MPI [LAM08] o MPICH [MPI05]. Es importante destacar, que ninguna de estas implementaciones está orientada a entornos Grid, sino que son implementaciones que siguen estrictamente el modelo de paso de mensajes especificado en el estándar MPI.

MPICH-G2 [KTF02] es la implementación del estandar MPI, concretamente de la versión MPI 1.1, para entornos Grid. Dicha implementación utiliza servicios de la herramienta Globus para realizar el paso de mensajes entre los diferentes nodos. El paradigma de programación MPICH-G2 es una versión renovada de MPICH-G, y permite la ejecución de programas desarrollados bajo la plataforma MPI aprovechando todos

los servicios y facilidades que ofrece la herramienta Globus, como pueden ser gestión de trabajos, gestión de seguridad y servicios de conversión de datos. Básicamente MPICH-G2 ofrece un interfaz que realiza la paralelización de la aplicación, conviertiendo sus datos en mensajes que permitirán la comunicación entre sistemas con distintas arquitecturas.

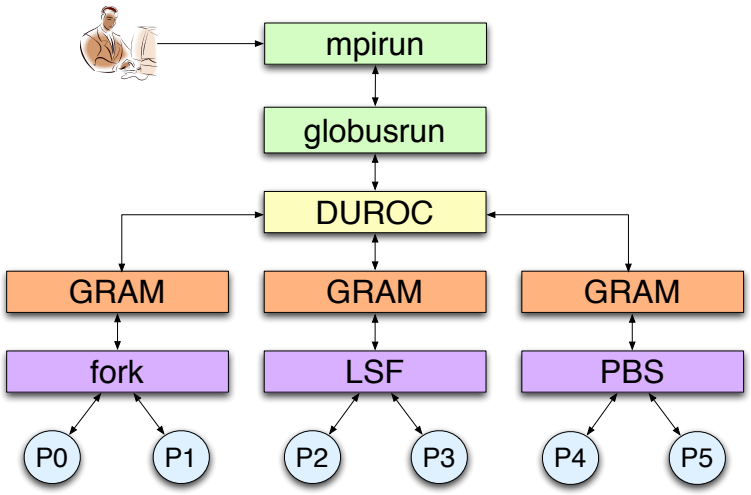

Figura 3.1: Arquitectura General de MPICH-G2

La figura 3.1 muestra un esquema de la arquitectura general del estándar MPICH-G2 [KTF02]. Inicialmente, el usuario realiza un programa, por ejemplo en lenguaje C, al que incorpora las funciones definidas en las librerías de MPICH-G2. Seguidamente, ejecuta el comando `mpirun` que realiza las tareas necesarias para permitir la ejecución del programa utilizando el paradigma MPICH-G2 como por ejemplo describir el trabajo a ejecutar utilizando el lenguaje RSL (*Resource Specification Language*) [CFK+98]. Con toda la información contenida en el fichero RSL, MPICH-G2 utiliza la librería DUROC (*Dinamically Updated Request Online Coallocator*) [CFK99] para planificar y ejecutar los trabajos en los recursos Grid especificados en dicho fichero. Si dichos nodos se encuentran disponibles DUROC utilizará GRAM para enviar los trabajos a los LRMS de cada nodo, en caso contrario se debe modificar el fichero RSL para consultar la disponibilidad de otros nodos del Grid.

La figura 3.2 presenta un programa en MPICH-G2 que implementa una aplicación sencilla maestro-esclavo. Dicha aplicación realiza una comunicación en anillo entre un nodo maestro y un conjunto de nodos exclavos especificados en la llamada a dicha aplicación a través del comando `mpirun`. El objetivo de la aplicación consiste en con-

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int numprocs, my_id, passed_num, trip;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

    if (numprocs > 1)
    {
        if (my_id == 0)  /* Proceso Maestro */
        {
            passed_num = 1;
            MPI_Send(&passed_num, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            MPI_Recv(&passed_num, 1, MPI_INT, numprocs-1, 0, MPI_COMM_WORLD, &status);
            printf("Maestro: fin del viaje: después de recibir %d pasajes", passed_num);
        }
        else            /* Proceso Esclavo */
        {
           MPI_Recv(&passed_num, 1, MPI_INT, my_id-1, 0, MPI_COMM_WORLD, &status);
           passed_num ++;
           MPI_Send(&passed_num, 1, MPI_INT, (my_id+1)%numprocs, 0, MPI_COMM_WORLD);
        } /* endif */
    }
    MPI_Finalize();
    exit(0);
} /* end main() */
```

Figura 3.2: Ejemplo de un programa básico en MPICH-G2

tabilizar el número total de pasajes de un determinado viaje. Para ello, cada uno de los nodos incrementa en una unidad la variable `passed_num` encargada de almacenar el número total de pasajes. Como se puede observar, la comunicación entre los distintos procesos se realiza a través de envío y recepción de mensajes utilizando las llamadas `MPI_Send` y `MPI_Receive`. Además, las llamadas `MPI_Comm_size` y `MPI_Comm_rank` permiten al desarrollador conocer tanto el número de procesos que se van a utilizar como el identificador único de cada uno de estos procesos.

El paradigma MPICH-G2 se ha utilizado para el desarrollo de aplicaciones distribuidas en diversos proyectos como por ejemplo GRADS [GRA01], sin embargo no incluye todas las características necesarias para su consideración como modelo de programación en entornos Grid. Ya que un modelo de programación Grid debería incorporar características como transparencia, fiabilidad y adaptación dinámica. Aunque MPICH-G2 facilita la labor al desarrollador gracias al uso del comando `mpirun` y la librería DUROC, la trasparencia que ofrece este paradigma no es suficiente ya que el usuario debe controlar los recursos donde se realizará la ejecución de la aplicación. Además, debido al uso de la barrera DUROC, es necesario que todos los recursos especificados se encuentren disponibles durante la ejecución de la aplicación para que esta se realice con

éxito.
MPICH-G2 no permite la adaptación dinámica de las aplicaciones y por lo tanto desaprovecha las capacidades computacionales de los nuevos recursos que se incorporen al Grid una vez iniciada la ejecución de la aplicación. Tampoco se ajusta a los patrones típicos de aplicaciones Grid, como en el caso de aplicaciones de alta productividad o de flujo de trabajo, incrementando además el tiempo de ejecución de las aplicaciones ya que el estándar MPI está pensado para el desarrollo de aplicaciones en sistemas fuertemente acoplados, y por lo tanto no tiene en cuenta las latencias que se producen en las redes publicas. Finalmente, es importante destacar que MPICH-G2 realiza la comunicación directa con Globus, por lo que desaprovecha todas las facilidades y funcionaliades que ofrecen los metaplanificadores.

3.3. SAGA

El paradigma de programación SAGA [KMHA06] (*Simple API for Grid Applications*) es una especificación para aplicaciones Grid definida por OGF [OGF08] (*Open Grid Forum*), que surge con el objetivo de simplificar el desarrollo de aplicaciones en Grid. Para ello, proporciona un API simple que permite estandarizar la sintaxis y semántica de dichas aplicaciones. Además, el interfaz propuesto por SAGA permite el desarrollo de aplicaciones independientes del *middleware* utilizado. Las principales características del API SAGA son:

- **Simplicidad**. SAGA es fácil de usar, instalar, administrar y mantener.
- **Uniformidad**. SAGA ofrece soporte a diversos lenguajes de programación como C++ o Java, así como una semántica y estilo conforme a las diferentes funcionalidades del Grid.
- **Escalabilidad**. Contiene mecanismos que permiten la ejecución de una misma aplicación en una gran variedad de sistemas, añadiendo soporte para distintos *middleware*.
- **Modularidad**. El API SAGA ofrece un entorno de trabajo fácilmente extensible por parte del desarrollador de aplicaciones.

La figura 3.3 muestra un esquema general de la arquitectura SAGA. Como se puede comprobar, la arquitectura está compuesta por varios módulos:

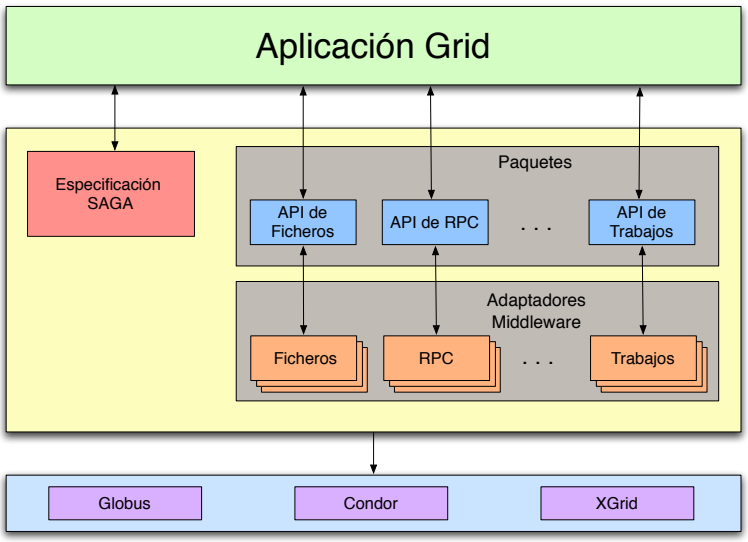

Figura 3.3: Arquitectura General de SAGA

- **Especificación SAGA**. Contiene todo el motor necesario para que el desarrollador implemente una aplicación Grid utilizando las sentencias proporcionadas por el API SAGA.

- **Paquetes SAGA**. Proporcionan sentencias especificas para la gestión de ficheros, RPC, trabajos, espacio de nombres y flujo de datos.

- **Adaptadores Middleware**. Se encargan de realizar la traducción de las sentencias proporcionadas por el módulo denominado "Paquetes SAGA" a sentencias especificas del *middleware*.

De entre todos los módulos que componen la arquitectura SAGA el más importante es el correspondiente al módulo denominado "Especificación SAGA", ya que constituye el eje central de este paradigma de programación. A continuación se detallan los elementos más importantes que componen dicho módulo:

- **Gestor de Espacio de Nombres**. Se encarga de gestionar el espacio de nombres de cada uno de los ficheros físicos y directorios que componen la aplicación que se está desarrollando siguiendo el estándar POSIX. Define, por tanto, una estructura jerárquica compuesta por directorios y entradas, donde cada directorio puede contener a su vez otro directorios o entradas, y cada entrada puede contener un objeto SAGA. El gestor de espacio de nombres ofrece sentencias para manejar

enlaces simbólicos, lista de control de acceso, búsquedas de directorios y entradas, y operaciones basadas en patrones.

- **Gestor de Ficheros**. Permiten la manipulación y control de los ficheros a través de sentencias SAGA basándose en el estándar POSIX. Estas sentencias pueden leer, escribir y realizar búsquedas sobre los ficheros físicos, independiente del *middleware* utilizado.

- **Gestor de Trabajos**. Realiza la gestión de trabajos a través de sentencias basadas en el API DRMAA, pero sin completar todos los servicios que ofrece este API. Por tanto el desarrollador puede crear trabajos, describir cada uno de los elementos del trabajo (definidos en el estándar JSDL) y gestionar el estado de cada trabajo.

- **Gestor de Replicas**. Ofrece servicios de replicas de ficheros y directorios similar a Globus RSL, añadiendo a las llamadas ofrecidas por el gestor de espacio de nombres llamadas para la gestión de replicas.

- **Gestor de Flujo de Datos**. Realiza la gestión de conexión de *sockets* similar a los *sockets* BSD.

Además, SAGA también ofrece llamadas para el control de sesiones dentro de la aplicaciones Grid. Permitiendo de esta manera agrupar conjuntos independientes de objetos SAGA dentro de una misma sesión. Finalmente la especificación SAGA se realiza a través del lenguaje SIDL (*Scientific Interface Definition Language*) [SID07], existiendo en la actualidad versiones de SAGA para los lenguajes C++ y Java que utilizan dicha especificación.

La figura 3.4 presenta un ejemplo sencillo de un programa en C++ con la especificación SAGA. Dicho programa realiza dos funciones básicas, en primer lugar prepara y envía al Grid un programa simple para su ejecución y en segundo lugar espera por la finalización de su ejecución. Como se puede observar, la espera que realiza es de las denominadas esperas activas, es decir, el programa pregunta constantemente por el estado del trabajo en ejecución hasta que este termina de ejecutarse, esto es debido a que el API de SAGA no incluye un comando que espere por la finalización de los trabajos.

Aunque el API SAGA se utiliza en diversos proyectos como GridSAT [CW03] ofreciendo muchas ventajas en el desarrollo de aplicaciones en sistemas Grid, el compor-

```
#include <string>
#include <saga/saga.hpp>
int main(int argc, char **argv)
{
    saga::job_description jobdef;
    jobdef.set_attribute("Executable", "job.sh");
    saga::job_service js;
    saga::job job = js.create_job("remote.host.net", jobdef);
    job.run();
    while( job.get_state() == saga::job::Running )
    {
        std::cout << "Job running with ID: "
        << job.get_attribute("JobID") << std::endl;
        sleep(1);
    }
}
```

Figura 3.4: Ejemplo de un programa básico en SAGA

tamiento que presenta es muy similar al de un pseudometaplanificador. De echo, tal y como se muestra en la figura 3.3 SAGA se encuentra en una capa intermedia entre el *middleware*, en el caso de la presente tesis sería Globus, y el usuario. Por tanto a igual que ocurre con MPICH, obvia todas las facilidades y servicios que ofrecen los metaplanficadores presentados en el capítulo anterior. De echo los autores comparan en varias ocasiones SAGA con MPI definiendolo como: "*SAGA es el homologo Grid de MPI*"[SAG07]. Además, la gestión de trabajos que realiza, aunque ofrece al desarrollador muchas operaciones, no es tan completa como la que propone el estándar DRMAA, ver sección 3.5.

3.4. Grid-RPC

Grid-RPC [SNM+02] constituye un paradigma de programación compuesto por un mecanismo de llamadas a procedimientos remotos (RPC, *Remote Procedure Calls*) [BN84] para entornos Grid propuesto por los autores C. Lee et al en [LMT+01]. Grid-RPC permite el desarrollo de aplicaciones en Grid a través de la comunicación entre el usuario y el *middleware* utilizando llamadas a procedimientos remotos. De esta manera, abstrae al programador de las tareas de comunicación entre los elementos dinámicos del Grid así como de la gestión de seguridad que todo entorno Grid requiere.

La figura 3.5 muestra un esquema del comportamiento básico del paradigma de

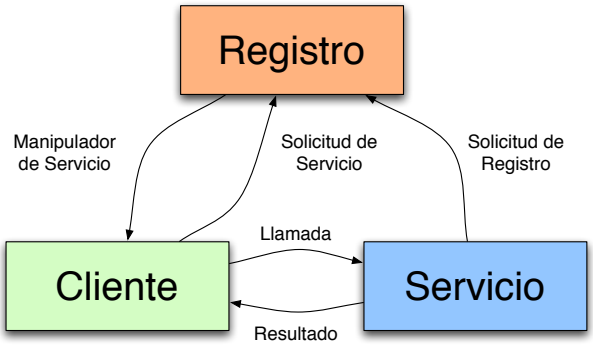

Figura 3.5: Modelo General de GridRPC

programación Grid-RPC. En primer lugar, un servicio determinado solicita su registro, enviando su información al registro de servicios. Posteriormente, una vez que el servicio está registrado, el cliente puede solicitar dicho servicio al registro de servicios. De esta manera, el cliente recibirá el manipulador del servicio especifico, que le permite realizar la llamada al procedimiento remoto. Una vez realizada la llamada al servicio, el cliente recibe el resultado de dicha llamada. Como se puede comprobar este comportamiento es muy similar a los que existen en otras arquitecturas como ocurre en el caso de CORBA. Es importante añadir, que el servicio puede ser un servicio web si el cliente está tratando con el *middleware* Globus. Los aspectos más importantes del paradigma Grid-RPC son:

- **Manipulador de Funciones e Identificadores de Sesión**. El manipulador de funciones se encarga de obtener la instancia concreta de una función de un determinado servicio a partir del nombre de dicha función. Por otro lado, el identificador de la sesión representa a una determinada llamada RPC no bloqueante.

- **Funciones de Inicio y Finalización**. Las llamadas realizadas antes de la inicialización o después de la finalización no serán válidas.

- **Funciones de Gestión de Funciones Remotas**. Permite la creación y destrucción de manipuladores de función remotos.

- **Funciones de Llamadas GridRPC**. Permiten realizar llamadas a procedimientos remotos, tanto de forma bloqueante como no bloqueante, así como con distintos argumentos de entrada.

- **Funciones de Espera y Control**. Permiten comprobar el estado de las solicitudes así como esperar a la finalización de las llamadas no bloqueantes previamente enviadas.
- **Funciones de Informes de Error**. Ofrecen una descripción de los errores que se hayan podido producir durante la ejecución.
- **Funciones de Pila de Argumentos**. Permiten construir los argumentos de una llamada en tiempo de ejecución.

Estas funciones facilitan la gestión de llamadas a procedimientos remotos a través del Grid, permitiendo de esta manera que la aplicación cliente tenga un aspecto similar a la de una aplicación RPC clásica.

```
#include <string>
#include <grpc.h>
int main(int argc, char **argv)
{
    grpc_function_handle_t handle;
    grpc_sessionid_t id;
    int i, time, ret, a, b, result;
    time = 1;
    a=4;
    b=5;

    if (ret = grpc_initialize(argv[1])) ==  GRPC_NO_ERROR)
    {
        grpc_function_handle_init(&handle,"magi1.cpe.ku.ac.th",
        "/example/plus");

        if ((grpc_call_async(&handle, &id, a, b, &result) &&
             grpc_wait_all() == GRPC_NO_ERROR)
          grpc_function_handle_destruct(&handles[i]);

        grpc_finalize();
    }
    exit(0)
}
```

Figura 3.6: Ejemplo de un programa básico en GridRPC

Un ejemplo de código en Grid-RPC es el que aparece representado en la figura 3.6, dicho programa ejecuta en un nodo remoto una instancia de un procedimiento previamente implementado denominado “plus” que realiza la suma de dos números, para a continuación esperar por la finalización de dicho procedimiento. Para ello el nodo destino debe tener el ejecutable de dicho programa, ya que desde el cliente se realizará una llamada asíncrona a dicho procedimiento de forma remota. Como se puede

observar, Grid-RPC utiliza llamadas para realizar ejecución remota de procedimientos, dificultando de esta manera el desarrollo de aplicaciones, ya que el proceso de paralelización de dichas aplicaciones se realiza a través de la distribución de procedimientos y funciones en lugar de distribuir trabajos completos. Además, debido a que dichos procedimientos deben estar implementados en el nodo destino, el desarrollador debe conocer de antemano los nodos que contienen dichos métodos. Por lo tanto el paradigma Grid-RPC carece de características fundamentales (dinamismo, transparencia, fiabilidad y adaptación dinámica) que faciliten y abstraigan al desarollador de las tareas de comunicación, planificación o gestión de recursos.

Con el objetivo de incrementar el dinamismo de las aplicaciones Grid-RPC han surgido diversos proyectos que desarrollan implementaciones de dicho paradigma como NetSolve [AAB+02] y Ninf-G [TNS+03]. Estas implementaciones ofrecen al programador infraestructuras para el desarrollo y despliegue de funciones y procedimientos remotos, facilitando al desarrollador la gestión de dichas funciones y procedimientos en los nodos remotos a través del uso de *stubs* y de interfaces IDL. Aunque estas implementaciones ha sido utilizadas en variedad de proyectos de investigación como, por ejemplo, el estudio de simulaciones climatológicas en AIST-TeraGrid [AIS08], estas mejoras no logran suplir las deficiencias inherentes al propio paradigma, ya que, como se ha comentado previamente, no lo abstraen al desarollador de las tareas relacionadas con la gestión de recursos. Además al estar ubicado en una capa intermedia entre el *middleware* y el programador, desaprovecha las ventajas que ofrece el uso de los metaplanificadores presentados en el capítulo 2.

3.5. DRMAA

DRMAA (*Distributed Resource Management Application API*) es una especificación desarrollada por el grupo de trabajo DRMAA-WG [DRM07] perteneciente al *Global Grid Forum* (GGF). Esta especificación es utilizada para la ejecución y el control de los trabajos en uno o varios DRMS (*Distributed Resource Management Systems*). El ámbito de esta especificación consiste en todas aquellas funcionalidades de alto nivel que son necesarias para que una determinada aplicación pueda enviar trabajos a sistemas DRM incluyendo operaciones sobre dicho trabajos como por ejemplo, operaciones de finalización, suspensión y reinicio. El objetivo de este especificación es facilitar una interfaz de aplicación directa entre los sistemas DRM actuales y los programadores de

aplicaciones, portales o los distribuidores de software independiente.

3.5.1. Aportación del Estándar DRMAA

La especificación DRMAA constituye un interfaz homogéneo y portable a diferentes DRMS para gestionar el envío, monitorización y control de trabajos, y para la recuperación del estado de los trabajos finalizados. Las implementaciones y las aplicaciones distribuidas de DRMAA, no tienen que particularizarse para un entorno DRMS determinado, o para una políticas de desarrollo determinadas. Para ello se utiliza las categorías de trabajos y la especificación nativa del DRMAA. De esta manera, las políticas especificas de la aplicación se pueden traducir en simples cadenas de caracteres que interpretarán las implementaciones que utilicen el API DRMAA.

DRMAA ofrece interfaces de "categorías de trabajos" que encapsulan los detalles concretos del recurso donde se ejecutarán los trabajos, ocultando dichos detalles a las aplicaciones que utilicen la interfaz DRMAA. De esta forma, el administrador del recurso, puede crear una categoría de trabajo, sujeta a una determinada aplicación que se ejecutará usando DRMS. El nombre de dicha categoría se especifica como un atributo de ejecución del trabajo. De esta forma, la implementación DRMAA, puede usar ese nombre de la categoría para manejar los recursos específicos y los requisitos funcionales de los trabajos pertenecientes a esa categoría.

Por lo tanto, el concepto de categorías ofrece ocultación de las políticas determinadas de cada recurso a los trabajos que se estén ejecutando usando el API DRMAA. Por otro lado para facilitar la implementación de dichas categorías, se debe mantener una categoría por cada política del recurso utilizada. De esta forma, la especificación nativa de las categorías, permite interpretar cada política a través de una simple cadena de caracteres, que será interpretado por cada una de las librerías del API DRMAA.

De esta manera, el API DRMAA permite el desarrollo de aplicaciones en Grid ocultado al usuario cualquier detalle relativo al gestor de recursos distribuidos o al *middleware* utilizado. Por lo tanto, el desarrollador realizará un programa de forma secuencial, con sentencias muy parecidas al POSIX de Unix, y DRMAA, el metaplanificador o gestor de recursos distribuidos, y el *middleware* se encargarán del resto. En contrapartida, los resultados parciales de las ejecuciones deberán ser almacenados en ficheros ubicados en memoria secundaria aumentado en determinadas ocasiones, el tiempo total de ejecución.

3.5.2. Especificación del Estándar

Existen diversas especificaciones del API para diferentes lenguajes de programación, en esta tesis se detallarán los aspectos más importantes de la especificación del estándar para los lenguajes C y Java. Desde el punto de vista de estos lenguajes, la especificación DRMAA se compone de funciones, en el caso del lenguaje C, o de métodos y clases en el caso del lenguaje Java. El objetivo de esta sección consiste en presentar una visión global de la especificación DRMAA, por ello únicamente se comentarán los aspectos más generales de esta especificación, indicando las funciones o métodos más importantes de dicha API.

Antes de especificar las funciones más importantes que ofrece el estándar DRMAA, es importante explicar el concepto de sesión. Toda implementación de una aplicación que se realice bajo el API DRMAA debe estar dentro de una sesión DRMAA. Es decir, cualquier llamada a un método o función del API que no se encuentre dentro de una sesión, previamente iniciada, no será tratada y se enviará un mensaje de error. Por lo tanto, todas las llamadas DRMAA deberán estar ubicadas entre la llamadas de inicio y de fin de sesión. El motivo de la existencia de sesiones es permitir la independencia de los elementos que se encuentren dentro de una misma sesión del resto de elementos semejantes que se encuentren en ejecución. Como se puede observar el concepto de sesión DRMAA es muy similar al concepto de sesión en SAGA.

La especificación del API DRMAA se divide en cuatro grandes grupos de métodos (clases) o funciones, para unificar conceptos, se denominarán tanto a los métodos como a la funciones, llamadas, ya que el objetivo es presentar una estructura general de las funcionalidades del API, y no una descripción de cada implementación realizada. Por lo tanto el estandar API se divide en cuatro grupos de llamadas:

- **Llamadas de Sesión**. Corresponden a las llamadas encargadas de inicializar y finalizar la sesión DRMAA. La llamada encargada de iniciar la sesión, inicializa las estructuras de datos necesarias en una sesión de DRMAA, mientras que la llamada encargada de finalizar la sesión libera las estructuras de datos, dejando la aplicación como se encontraba antes del inicio de la sesión.

- **Llamadas de Definición de Trabajos**. Permiten la manipulación de entidades de definición, es decir, plantillas de trabajos. De esta forma, permite establecer parámetros tales como el fichero ejecutable, sus argumentos, flujos de salida estándar, variable de entorno y directorio de trabajo local. DRMAA permite extender

estas llamadas a llamadas propias y especificas del gestor de recursos a utilizar. Dependiendo del lenguaje de programación estas llamadas pueden ser un conjunto de definiciones, funciones y constantes declaradas en un módulo, o pueden constituir una clase base a partir de la cual se pueden extender todas las clases hijas que se deseen.

- **Llamadas de Ejecución de trabajos**. Estas llamadas son el eje fundamental dentro de la API DRMAA, se encargan de enviar los trabajos para su ejecución. Permiten el envío de un trabajo independiente, así como el envío de un conjunto de trabajos. Tanto en un caso como en el otro, el API DRMAA devuelve un identificador único que representa al trabajo enviado al Grid para su ejecución dentro de la sesión DRMAA.

- **Llamadas de Control y Monitorización de Trabajos**. En este grupo se engloban todas aquellas llamadas que se utilizan para controlar y sincronizar trabajos así como para monitorizar su estado. Las llamadas de control permiten parar la ejecución, reenviar o matar tanto un trabajo específico como la totalidad de trabajos que estén en ejecución en una determinada sesión. Por otro lado, las llamadas de sincronización permiten esperar a la finalización de ejecución de un trabajo determinado, de un conjunto de trabajos o de todos los trabajos de la sesión. Finalmente, las llamadas de monitorización devuelven el estado actual de un determinado trabajo.

Esta especificación permite el desarrollo de programas a través de un estándar facilitando así el despliegue de aplicaciones en Grid. Además, el uso del API DRMAA permite desarrollar aplicaciones Grid independientemente del gestor de recursos utilizado. También proporciona al desarrollador llamadas para gestión de los trabajos en ejecución, posibilitando de esta manera la implementación de aplicaciones compatibles con los perfiles de ejecución tipicos de Grid, como aplicaciones de alta productividad o aplicaciones maestro-esclavo. Y lo más importante, facilita el desarrollo de aplicaciones que se desacomplen de la infraestructura Grid donde se vayan a ejecutar. Debido a todas estas características, que diferencian este paradigma de los presentados en secciones anteriores, se ha decido su inclusión en el modelo de programación presentado en esta tesis.

3.5.3. Modelo de Programación DRMAA-Grid *W*ay

En la actualidad existen diversas implementaciones del estándar DRMAA en sistemas como Condor, PBS/Torque o LSF [DRM08b], sin embargo esta tesis presenta la primera implementación de este estándar para los lenguajes C y Java. Concretamente presenta la implementación del estándar DRMAA para el metaplanificador Grid *W*ay, constituyendo de esta manera el modelo de programación en Grid denominado modelo DRMAA-Grid *W*ay.

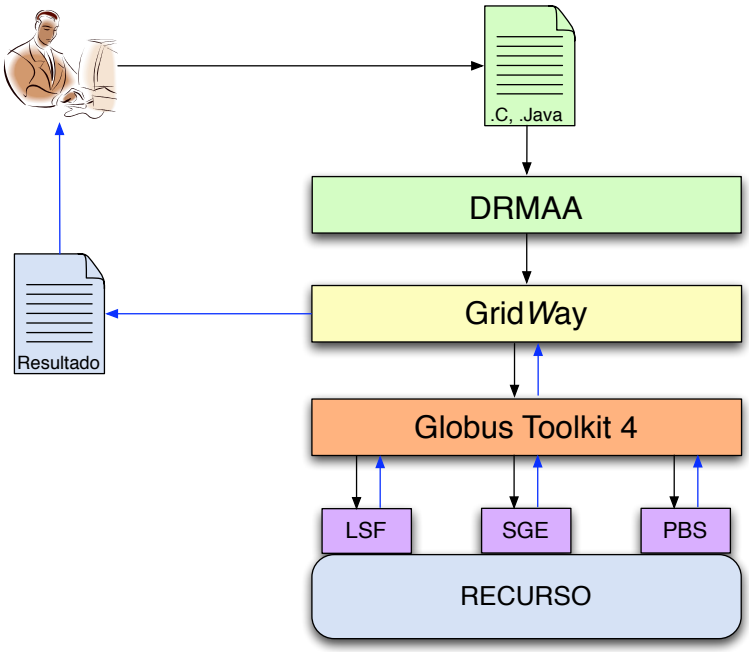

Figura 3.7: Cliclo de Desarrollo con DRMAA y Grid *W*ay.

La figura 3.7 representa el ciclo de desarrollo de una aplicación en Grid utilizando el modelo de programación DRMAA-Grid *W*ay. Como se puede observar, el usuario desarrolla un programa en lenguaje C o Java añadiendo las llamadas necesarias del API DRMAA. Posteriormente, generará un ejecutable utilizando la librerías de metaplanificador Grid *W*ay y del API DRMAA. Dicho programa se ejecutará en un nodo del Grid y distribuirá sus trabajo utilizando el *middleware* Globus, de tal manera que desde el punto de vista del usuario de la aplicación el comportamiento del programa será similar al de un programa secuencial. Finalmente, gracias al uso de Globus y Grid *W*ay, el usuario obtendrá los resultados de la ejecución de dicho programa.

La figura 3.8 muestra un programa sencillo utilizando el API DRMAA que ejecuta

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    char                    *error, *attr_value;
    char                    *job_id, *job_id_out;
    int                     stat;
    drmaa_job_template_t    *jt;
    drmaa_attr_values_t     *rusage;

    drmaa_init(NULL, error);
    drmaa_allocate_job_template(&jt, error);
    drmaa_set_attribute(jt, DRMAA_JOB_NAME, "ht2", error);
    drmaa_set_attribute(jt, DRMAA_REMOTE_COMMAND, "/bin/ls", error);

    drmaa_run_job(job_id, DRMAA_JOBNAME_BUFFER-1, jt, error);
    drmaa_wait(job_id, job_id_out, &stat, DRMAA_TIMEOUT_WAIT_FOREVER, &rusage,
               error);

    drmaa_wexitstatus(&stat, stat, error);

    while (drmaa_get_next_attr_value(rusage, attr_value)
           != DRMAA_ERRNO_NO_MORE_ELEMENTS)
        fprintf(stderr,"\t%s\n",attr_value);

    drmaa_release_attr_values(rusage);
    drmaa_delete_job_template(jt, error);
    drmaa_exit(error);

  return 0;
 }
```

Figura 3.8: Ejemplo de un programa básico en DRMAA

de forma remota el comando `ls` de UNIX y espera por el fin de ejecución de dicho programa, además, muestra por pantalla las estadísticas de ejecución del trabajo enviado al Grid, indicando por ejemplo, el tiempo dedicado a la trasferencia de ficheros y el dedicado a la ejecución del programa en el nodo remoto. Como se puede observar, en ninguna sentencia del código se hace referencia al nodo o nodos donde se ejecutará el programa, ya que dicha gestión la realizarán conjuntamente Grid *W*ay y Globus de manera trasparente al usuario. El resultado de dicha ejecución se almacenará en un fichero que se alojará en la ubicación especificada por el desarrollador o el usuario de la aplicación

El programa presentado en la figura 3.8 es solo una pequeña muestra de la funcionalidad que ofrece el modelo DRMAA-Grid *W*ay. Ya que el modelo permite también ejecutar conjuntos de trabajos, esperar por la finalización tanto de un trabajo individual, como de un subconjunto de trabajos, o por todos los trabajos de una misma sesión. Además el desarrollador tiene control de los trabajos en ejecución pudiendo eliminarlos, pararlos o reiniciarlos. Para la ejecución de los programas es necesario que

el desarrollador genere un *template* o plantilla donde defina las características de los trabajos a ejecutar. Dicha plantilla se genera a través de las llamadas de definición de trabajos analizadas en la sección 3.5.2.

La utilización del estándar DRMAA en conjunción con el metaplanificador Grid *W*ay y el *middleware* Globus permite al desarrollador implementar programas para entornos Grid heterogéneos desacoplando el desarrollo de dichos programas de la infraestructura Grid donde se vayan a ejecutar. Esta característica es fundamental en el modelo DRMAA-Grid *W*ay ya que permite el desarrollo de aplicaciones Grid independientes del entorno Grid donde se ejecuten. Además el modelo permite al desarrollador y al propio usuario de la aplicación desvincularse de las tareas asociadas a la gestión de los recursos Grid donde se realice la ejecución, permitiendo la incorporación en dicha ejecución de nuevos nodos o aumentando la fiabilidad de la aplicación a través migración de trabajos. Este modelo, además, permite el desarrollo de aplicaciones compatibles con los perfiles de ejecución más comunes que se pueden encontrar en entornos Grid heterogéneos como aplicaciones maestro-esclavo o aplicaciones de alta productividad.

3.6. Conclusiones

La primera fase de investigación de la presente tesis se inició en septiembre del año 2003 con la publicación de la especificación 0.9 del estándar DRMAA para el lenguaje C. El estándar DRMAA proponía un conjunto de llamadas independientes del gestor de recursos utilizado, que permitían la ejecución, gestión y control de trabajos de forma remota. Debido a estas características se ideó un modelo de programación para aplicaciones Grid basado en el estandar DRMAA, el metaplanificador Grid *W*ay y el *middleware* de Grid Globus. Para desarrollar dicho modelo se implementó el estándar DRMAA para el metaplanificador Grid *W*ay siendo esta la primera implementación de dicho estándar.

Este capítulo ha analizado cada uno de los paradigmas de desarrollo de aplicaciones más utilizados en la actualidad. Concretamente presenta el comportamiento y la arquitectura de MPICH-G2, SAGA y Grid-RPC, con el objetivo de comparar las características de cada uno, para justificar la elección del API DRMAA como paradigma de programación del modelo DRMAA-Grid *W*ay. La característica fundamental del API DRMAA, que ha motivado a su elección, ha sido la posibilidad de desarrollar aplicaciones Grid desacopladas de la infraestructura donde se ejecutan. Esta característica es

única del API DRMAA, ya que propone un conjunto de llamadas independientes del gestor de recursos utilizado, que permiten la ejecución, gestión y control de trabajos de forma remota.

El capítulo concluye con la descripción del modelo de programación de aplicaciones Grid analizando sus principales ventajas. En primer lugar, el modelo DRMAA-Grid *W*ay facilita el desarrollo de aplicaciones Grid a través de llamadas al API DRMAA dentro del código de la aplicación, de forma similar a llamadas al sistema en UNIX. De esta manera, el desarrollador implementa una aplicación, semejante en estructura a un programa secuencial, que se distribuirá por el Grid de manera transparente al usuario. Esta trasparencia se refiere a la planificación, gestión y control de los recursos del Grid que se realizará por medio del metaplanificador Grid *W*ay.

Sin embargo, la trasparencia ofrecida por el modelo DRMAA-Grid *W*ay no impide que el desarrollador tenga control en todo momento del estado de los trabajos que se han enviado a ejecutar y permite, parar los trabajos, reanudarlos, matarlos y comprobar su estado. Además, debido a las propiedades del metaplanificador Grid *W*ay, el modelo DRMAA-Grid *W*ay dota a las aplicaciones de fiabilidad durante la ejecución, ya que permite que la aplicación continúe su ejecución incluso si alguno de los nodos falla. El modelo DRMAA-Grid *W*ay también incluye la adaptación dinámica de la aplicación, de esta manera, los trabajos pueden migrar a otros recursos que se adapten mejor a las necesidades de la aplicación, y permite la incorporación de nuevos recursos Grid durante la ejecución de la aplicación. Todas estas características permiten definir al modelo DRMAA-Grid *W*ay como un modelo único dentro de la tecnología Grid para el desarrollo de aplicaciones.

Para comprobar la funcionalidad y eficiencia de este modelo de desarrollo para aplicaciones en Grid, en el siguiente capítulo se presentan distintos escenarios donde se prueba dicha funcionalidad a través de la implementación de diferentes perfiles de aplicación.

Capítulo 4

Estudio de Perfiles de Aplicaciones Grid con DRMAA-Grid *W*ay

El objetivo de este capítulo consiste en demostrar la eficiencia y efectividad del modelo de programación de aplicaciones Grid presentado en el capítulo 3. Con este fin, se analizará el comportamiento del modelo en el desarrollo, implementación y ejecución de los perfiles de aplicación más comunes que se pueden encontrar en un entorno Grid heterogéneo. También se comprobará la sencillez en el uso de este modelo, que además de facilitar el desarrollo de aplicaciones Grid, permite una adaptación débilmente acoplada entre la aplicación y el entorno Grid donde se ejecuta.

El capítulo está dividido en tres secciones principales. Las secciónes 4.2 y 4.3 presentarán los experimentos preliminares realizados utilizando este modelo de desarrollo, concretamente muestra su comportamiento en la ejecución de aplicaciones de alta productividad y aplicaciones maestro-esclavo. Posteriormente, en la sección 4.4 se detallarán los experimentos y resultados obtenidos en la ejecución del NAS Grid Bechmark, concretamente sobre los tipos *Helical Chain* (HC) y Visualization Pipe (VP).

La sección 4.5 presentará el desarrollo de un algoritmo genético genérico en Grid utilizando el modelo de desarrollo DRMAA-Grid *W*ay. Para ello, en primer lugar se realizará una breve introducción sobre los algoritmos genéticos haciendo hincapié en los algoritmos genéticos distribuidos. Después, en la sección 4.5.1 se describirá el algoritmo genérico y adaptativo DRMAA-GOGA y se analizarán las principales ventajas que aporta frente a otros algoritmos genéticos distribuidos. Finalmente, se demostrará la eficiencia del algoritmo DRMAA-GOGA con los resultados obtenidos en la ejecución de un conjunto de experimentos dentro de un entorno Grid real.

4.1. Introducción

En el capítulo anterior se presentó el modelo de programación DRMAA-Grid*W*ay constituido por el API DRMAA implementado en los lenguajes C y Java para el metaplanificador Grid*W*ay, el propio metaplanificador Grid*W*ay, y el *middleware* Globus Toolkit. Las principales ventajas que ofrece este modelo de programación de aplicaciones Grid frente al resto de paradigmas que se utilizan en la actualidad son: (i) Facilita el desarrollo y despliegue de aplicaciones Grid debido al uso del API DRMAA; (ii) Desacopla la aplicación de la infraestructura Grid donde se ejecuta, debido principalmente al uso del API DRMAA y del metaplanificador Grid*W*ay; (iii) Permite que tanto el desarrollador como al usuario de la aplicación se desvinculen de las tareas relativas a la gestión, administración y control de los recursos Grid donde se ejecuta la aplicación; (iv) Dota a las aplicaciones de dinamismo y fiabilidad permitiendo en tiempo de ejecución la incorporación de nuevos recursos o la migración de trabajos cuando algún recursos falle.

Para demostrar todas las características de este modelo de implementación el presente capítulo analiza su comportamiento en el desarrollo y ejecución de los siguientes perfiles de aplicación:

- **Aplicaciones de Alta Productividad**. Este tipo de aplicaciones se caracteriza por la ejecución de un número determinado de instancias de un mismo programa pero con distintos parámetros de entrada. El principal objetivo del desarrollo de una aplicación de alta productividad con el modelo DRMAA-Grid*W*ay es demostrar la facilidad con la que el desarrollador puede implementar este tipo de aplicaciones utilizando dicho modelo. Además, el uso del modelo DRMAA-Grid*W*ay permite reducir el tiempo total de ejecución de la aplicación, tal y como se mostrará en los resultados obtenidos durante la ejecución de los experimentos.

- **Aplicaciones Maestro-Esclavo**. La principal característica de este tipo de aplicaciones reside en que la comunicación entre los procesos se realiza a través del proceso maestro, que es el encargado de procesar la información obtenida por los procesos esclavos. El objetivo de implementar este tipo de aplicaciones utilizando el modelo DRMAA-Grid*W*ay consiste en analizar el impacto que presenta la comunicación entre los trabajos en el desarrollo de la aplicación. Esta caracteristica es muy importante ya que en el modelo DRMAA-Grid*W*ay la comunicación entre trabajos se realiza mediante el uso de ficheros almacenados en memoria

secundaria.

- **NAS Grid *Benchmarks***. El perfil de aplicación de los NAS Grid Benchmarks (NGB) permite analizar el comportamiento del modelo DRMAA-Grid *W*ay en la ejecución de aplicaciones con dependencias entre trabajos. Debido a que el API DRMAA no ofrece llamadas concretas para la definición de dependencias es importante analizar el grado de dificultad que conlleva desarrollar este tipo de aplicaciones con el modelo DRMAA-Grid *W*ay. Como se podrá comprobar en la sección 4.4, la flexibilidad del modelo presentado permite el desarrollo de forma sencilla y eficiente de aplicaciones con dependencias entre trabajos.

- **Algoritmo Genético**. Con el objetivo de presentar una nueva propuesta de algoritmos genéticos distribuidos que aprovechen las características del modelo DRMAA-Grid *W*ay, se ha desarrollado un algoritmo genético genérico para su ejecución en un entorno Grid heterogéneo. Este algoritmo aportará características como, fiabiliadad, dinámismo o adaptación dinámica, heredadas del modelo de desarrollo DRMAA-Grid *W*ay.

Los perfiles de aplicación presentados representan un amplio banco de pruebas para analizar la funcionalidad del modelo de desarrollo DRMAA-Grid *W*ay. En las sucesivas secciones se estudiará el comportamiento de este modelo en un conjunto de experimentos realizados en un banco de pruebas Grid de investigación.

4.2. Aplicaciones de Alta Productividad

En esta sección se presentan los primeros experimentos realizados utilizando el modelo de desarrollo de aplicaciones Grid formado por DRMAA, el metaplanificador Grid *W*ay y el *middleware* Globus para el diseño, implementación y ejecución de aplicaciones de alta productividad. Durante estas pruebas se utilizó el *middleware* en su versión 2.2, es decir pre-servicios web. El uso de esta versión en lugar de la versión 4.x se debe a que en la época en la que se realizaron estas pruebas la versión 4.x de Globus aún no había sido desarrollada. Aún así el comportamiento del modelo, respecto a los resultados obtenidos, es el mismo tanto para la versión 2.2 como para la versión 4.x, ya que la funcionalidad básica de Globus se sigue manteniendo.

Las aplicaciones de alta productividad se incluyen dentro del grupo de aplicaciones PSA (*Parametric Swep Applications*). El perfil típico de las aplicaciones de alta

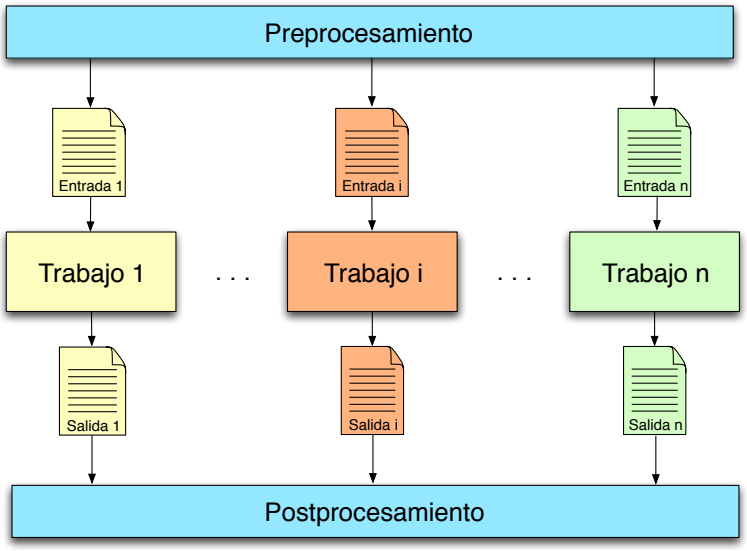

Figura 4.1: Modelo de aplicaciones de alta productividad

productividad consta de múltiples ejecuciones independientes de un mismo programa, pero con distintos parámetros de entrada. Este tipo de aplicaciones es muy frecuente en sistemas Grid, y suele aplicarse a diversos campos científicos como en Bioinformática o en Dinámica de Fluidos.

La figura 4.1 muestra un esquema representativo de este tipo de aplicaciones. Como se puede observar, inicialmente el programa principal realiza diversas tareas de preprocesamiento para dividir la carga computacional entre distintos trabajos o tareas. Dicha carga se traduce en diferentes ficheros de entrada para cada una de la ejecuciones independientes del mismo programa que se van a lanzar al Grid. Cada uno de estos trabajos se ejecutará de manera independiente y en nodos distintos, de esta manera se produce una paralelización de la aplicación que se quiere desarrollar. Cuando finaliza la ejecución de cada trabajo, este almacena el resultado de su ejecución en ficheros de salida. Estos ficheros de salida serán postprocesados por el programa principal para obtener el resultado total de la aplicación. El desarrollo de este tipo de aplicaciones no es trivial ya que el programador debe encargarse de la sincronización entre cada uno de los trabajos que se envían para que el postprocesamiento se realice con éxito. Además debe tener control sobre dichos trabajos ya que en el caso de que uno de ellos falle, el resultado final no sería el esperado. El modelo DRMAA-Grid *W*ay facilta esta labor al desarrollador permitiendo que sea el propio modelo, a través de llamadas DRMAA, el que se encargue de la sincronización de los trabajos, así como de la migración de

trabajos en aquellos nodos que fallen a través del uso del metaplanificador Grid *W*ay.

Para probar la funcionalidad del modelo de desarrollo presentado en el capítulo anterior para la implementación de aplicaciones de alta productividad, se diseñó un programa en lenguaje C que incluía sentencias DRMAA para realizar la ejecución de los trabajos de manera paralela con diferentes ficheros de entrada. La aplicación por tanto crea unos ficheros, uno por cada trabajo a ejecutar, con datos de entrada diferentes, ejecuta de manera remota cada uno de los trabajos con su correspondiente paramentos de entrada, espera por la finalización de estos trabajos, y después almacena todos los resultados en un fichero de salida. Desde el punto de vista del desarrollador, el programa está compuesto por un único fichero en lenguaje C con las sentencias DRMAA necesarias para la ejecución y espera de los trabajos que se envíen al Grid.

Desde el punto de vista del metaplanificador Grid *W*ay, este recibe un ejecutable, su plantilla asociada, y el fichero de entrada, que será diferente en cada caso. Esto es posible ya que Grid *W*ay permite parametrizar determinados campos de la plantilla de los trabajos asociandolos al identificador del trabajo, de esta manera una sola plantilla de trabajo se puede utilizar para lanzar un ejecutable con distintos parametros de entrada y salida, ya que cada instancia del ejecutable, es decir cada trabajo, tendrá un identificador único.

Finalmente desde el punto de vista de Globus, este solo deberá encargase de enviar los trabajos a un recurso, esperar por su finalización y enviar todos los ficheros con los resultados al metaplanificador Grid *W*ay, que a su vez reanudará el programa principal, que se había quedado a la espera de la finalización de los trabajos, para que realice las tareas de postprocesamiento.

La figura 4.2 representa un esquema del código DRMAA para el desarrollo de la aplicación de alta productividad utilizada para realizar los experimentos. Esta aplicación contempla la ejecución de 50 trabajos independientes, cada uno de los cuales, calcula el determinante de una matriz. Tal y como muestra dicha figura, el programa se divide en tres partes. La primera y tercera partes corresponden a las tareas de prepocesamiento y postprocesamiento que se realizan en el nodo local y por lo tanto no es necesario utilizar sentencias DRMAA. La segunda parte corresponde a la ejecución remota de 50 trabajos donde cada uno de ellos calcula el determinante de una matriz. Para ello, en primer lugar se crea una plantilla de trabajo genérica, con los argumentos obtenidos de las tareas de preproceso, indicando los argumentos del ejecutable y los ficheros de entrada y salida, estas tareas se realizan utilizando las llamadas `drmaa_set_attribute()`

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    char                    *error, *attr_value;
    char                    *job_id, *job_id_out;
    char                    **job_ids, **args, **input_files, **output_files;
    int                     stat, start=0, end=49;
    drmaa_job_template_t    *jt;
    drmaa_attr_values_t     *rusage;
    signed long             timeout = 0;

    // Realiza las tareas de preprocesamiento necesarias
    preprocesing_task();

    drmaa_init(NULL, error);

    // Ejecuta n trabajos simultáneamente y espera por su finalización
    drmaa_allocate_job_template(&jt, error);
    drmaa_set_attribute(jt, DRMAA_JOB_NAME, "htc", error);
    drmaa_set_attribute(jt, DRMAA_REMOTE_COMMAND, "determinante", error);
    drmaa_set_vector_attribute(jt, DRMAA_V_ARGV, args, error);
    drmaa_set_vector_attribute(jt, DRMAA_V_GW_INPUT_FILES, input_files, error);
    drmaa_set_vector_attribute(jt, DRMAA_V_GW_OUTPUT_FILES, output_files, error);

    drmaa_run_bulk_jobs(job_ids, jt, start, end, error);
    drmaa_synchronize(job_id, job_id_out, &stat, timeout, &rusage, error);

    drmaa_delete_job_template(&jt, error);
    drmaa_exit(error);

    // Realiza las tareas de postprocesamiento necesarias
    postprocessing_task();

    return 0;
 }
```

Figura 4.2: Ejemplo de código DRMAA para el desarrollo de aplicaciones HTC

y `drmaa_set_vector_attribute()`. Después se ejecutan todos los trabajos a través de una única llamada denominada `drmaa_bulk_jobs()` y el programa espera por la finalización de todos los trabajos utilizando la llamada `drmaa_synchronize()`. Finalmente, se elimina la plantilla asociada a cada uno de los trabajos, y se sale de la sesión DRMAA con el uso de las llamadas `drmaa_delete_job_template()` y `drmaa_exit()`, para terminar el programa con las tareas de postprocesamiento que sean necesarias.

Para el desarrollo de los experimentos, se utilizó 50 matrices cuadradas de 400 elementos almacenadas cada una en un fichero independiente de tamaño 0,5 MB. Dicha aplicación se ejecutó en el banco de pruebas representado en la tabla 4.1. Como se puede observar, el banco de pruebas está compuesto por cinco máquinas, 4 situadas en el laboratorio de la Facultad de Informática de la UCM, y otra en el Centro de Astro-Biologia (CAB), siendo las más rápidas las denominadas hydrus e cygnus, ya que

Tabla 4.1: Características de las máquinas del banco de pruebas experimental UCM-CAB.

Nombre	VO	Modelo	SO	Velocidad	Memoria	Planf. Local
babieca	CAB	5×Alpha DS10	Linux 2.2	466MHz	256MB	PBS
hydrus	UCM	Intel P4	Linux 2.4	2.5GHz	512MB	fork
cygnus	UCM	Intel P4	Linux 2.4	2.5GHz	512MB	fork
cepheus	UCM	Intel PIII	Linux 2.4	662MHz	256MB	fork
aquila	UCM	Intel PIII	Linux 2.4	568MHz	128MB	fork

son las que tienen los procesadores de mayor velocidad, así como la mayor cantidad de memoria. Es importante destacar que en total este banco de pruebas dispone de nueve PE (*Processing Element*) ya que cada máquina dispone de un PE a excepción de la máquina denominada babieca que dispone de cinco PE. Finalmente, solo destacar que la mayoría de las máquinas utilizan el planificador local fork a excepción de babieca que utiliza el planificador local PBS.

La figura 4.3 representa la productividad (trabajos por minuto) del banco de pruebas obtenida durante la ejecución de la aplicación PSA. Como se puede apreciar, el tiempo total de ejecución de la aplicación fue de 40 minutos, con un tiempo medio de ejecución por trabajo de 125 segundos. Gracias a la productividad se puede medir el rendimiento del banco de pruebas utilizado, la fórmula 4.1 representa el cálculo de dicha productividad.

$$P^G(t) = \frac{Trabajos(t)}{t}, \tag{4.1}$$

donde $P^G(t)$ representa la productividad del banco de pruebas G en el instante de tiempo t y $Trabajos(t)$ representa el número de trabajos finalizados en el instante t. La productividad va oscilando de manera ascendente a medida que aumenta el tiempo de ejecución de la aplicación, estas oscilaciones se deben a las velocidades de procesamientos diferentes que existen en cada máquina. Así, si aumenta el tiempo de finalización de un trabajo, debido a que se ejecuta en una máquina lenta, se produce un decremento en la productividad total del banco de pruebas y viceversa. Aún así, la productividad final es de 1,25 trabajos por minuto, mayor que la productividad para un único nodo que es de 0,8. Además, si se compara la ejecución individual del cálculo del determinante de las 50 matrices en la máquina más rápida del banco de pruebas, es decir en hydrus, con la ejecución paralela se aprecia una reducción del 35 %, ya que el tiempo total es de 62

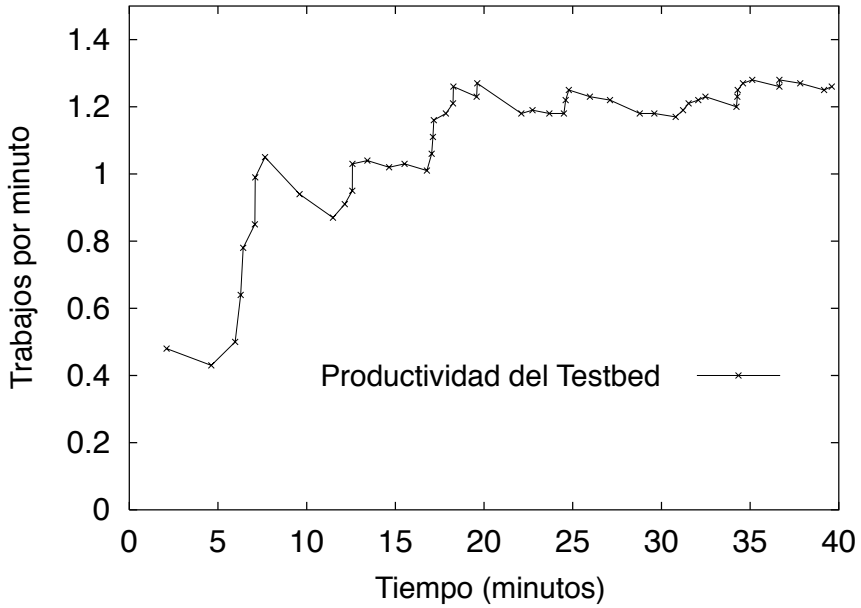

Figura 4.3: Productividad del banco de pruebas durante la ejecución de la aplicación PSA

minutos, respecto al tiempo total de la ejecución distribuida que es de 40 minutos.

A la vista de los resultados obtenidos se ha comprobado que el modelo de programación DRMAA-Grid*W*ay no sólo facilita el desarrollo de aplicaciones PSA sino que también contribuye a aprovechar las ventajas de la distribución de la aplicación como es el decremento del tiempo total de ejecución o incremento de la productividad del Grid.

4.3. Aplicaciones Maestro-Esclavo

Esta sección presenta el comportamiento del modelo DRMAA-Grid*W*ay en la implementación y ejecución de aplicaciones maestro-esclavo. Dicho modelo es utilizado frecuentemente en aplicaciones científicas como los algoritmos genéticos, ver sección 4.5, o simulaciones de tipo Monte Carlo. En este tipo de aplicaciones, el proceso maestro se encarga de la coordinación y sincronización de los procesos esclavos así como de la gestión de la información necesaria para la ejecución de dichos procesos. Además, cuando finaliza la ejecución de los procesos esclavos, el proceso maestro es el encargado de evaluar el criterio de parada con el objetivo de decidir si se termina la ejecución

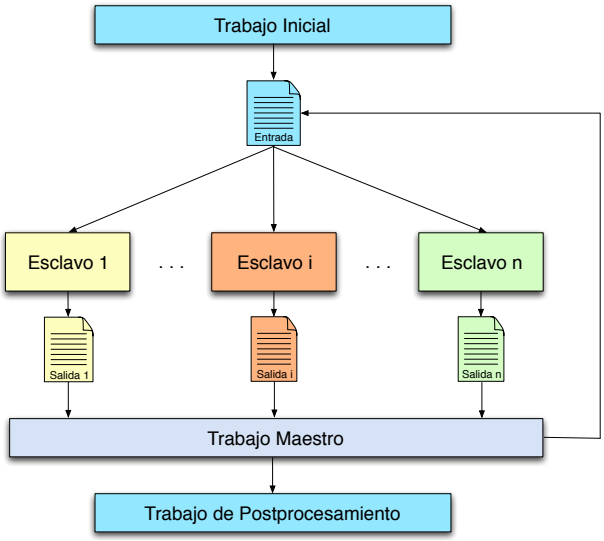

Figura 4.4: Perfil de Aplicación Maestro-Esclavo

de la aplicación o se asignan nuevas tareas a cada proceso esclavo para continuar su ejecución.

Un esquema de este tipo de aplicaciones se muestra en la figura 4.4. Como se puede observar, existe un trabajo inicial encargado de preprocesar toda la información necesaria que se le enviará a los trabajos esclavos. Posteriormente, se ejecutan todos los trabajos esclavos que obtienen el estado actual de la aplicación a través del fichero de entrada. Cuando finaliza la ejecución de todos los trabajos esclavos, el trabajo maestro decide, a través de una función de ajuste, si se ha llegado al objetivo deseado, en ese caso envía toda la información al trabajo de postprocesamiento, en caso contrario, es necesario volver a ejecutar una nueva iteración del problema, así que modifica el fichero de entrada y ejecuta de nuevo los trabajos esclavos. Finalmente, el trabajo de postprocesamiento realizará todas las acciones necesarias para presentar los resultados al usuario final. Este modelo pone de relieve el comportamiento del paradigma DRMAA-Grid *W*ay en las tareas dedicadas al intercambio de información entre trabajos, que en este caso se realiza a través de ficheros. Concretamente, para el modelo maestro-esclavo el intercambio se produce entre el trabajo maestro y el conjunto de trabajos esclavos. El uso del modelo DRMAA-Grid *W*ay permite que este intercambio de información se realice de forma muy sencilla, ya que cada trabajo esclavo tendrá asociado un fichero con los resultados de su ejecución.

La traducción del modelo maestro-esclavo al paradigma de desarrollo DRMAA-

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    drmaa_job_template_t   *jt_master, *jt_slave;
    drmaa_attr_values_t    *rusage;
    .....

    // Realiza las tareas de preprocesamiento necesarias
    preprocesing_task();

    drmaa_init(NULL, error);

    // Definición del trabajo maestro y del trabajo esclavo
    drmaa_allocate_job_template(&jt_master, error);
    drmaa_allocate_job_template(&jt_slave, error);
    .....

    while (exitstatus != 0)
    {
        // Executa los esclavos y espera por ellos
        drmaa_run_bulk_jobs(job_ids, jt_slave, start, end, error);
        drmaa_synchronize(job_id, job_id_out, &stat, timeout, &rusage, error);

        // Ejecuta el trabajo maestro, espera por él y obtiene su estado
        drmaa_run_job(job_id, jt_master, error);
        drmaa_wait(job_id, &stat, timeout, rusage, error);
        drmaa_wexitstatus(&exitstatus, stat, error);
    }

    drmaa_delete_job_template(&jt_slave, error);
    drmaa_delete_job_template(&jt_master, error);
    drmaa_exit(error);

    // Realiza las tareas de postprocesamiento necesarias
    postprocessing_task();

    return 0;
 }
```

Figura 4.5: Ejemplo de código DRMAA para el desarrollo de aplicaciones Maestro-Esclavo

Grid*W*ay se realiza a través de un programa en C con sentencias DRMAA de forma similar a como se realiza para las aplicaciones de alta productividad presentadas en la sección anterior. Por lo tanto el programa principal se encargará de crear el trabajo inicial y final así como el trabajo maestro y cada uno de los trabajos esclavos que se ejecuten. Es importante destacar, que las tareas realizadas por los trabajos inicial y final se pueden encapsular dentro del propio programa principal, por lo que no es necesario ejecutar dichos trabajos de manera distribuida. De esta manera, se dispondrá de tres ejecutables; el programa principal, el programa maestro, y el programa esclavo.

La figura 4.5 muestra un esquema del código empleado para desarrollar la aplicación

maestro-esclavo utilizada en los experimentos. Tal y como muestra la figura 4.4 el programa realiza inicialmente tareas de preprocesamiento de datos. Después genera las plantillas necesarias para ejecutar los trabajos esclavos y el trabajo maestro. A continuación entra en un bucle donde se ejecutan todos los trabajos esclavos y se espera por la finalización de estos. El siguiente paso consiste en ejecutar el trabajo maestro que se encargará de evaluar la función objetivo. Si el resultado de esta evaluación es positivo, el trabajo maestro terminará satisfactoriamente, es decir $exitstatus = 0$, y el bucle terminará, en otro caso continuará la ejecución del programa. Finalmente, como sucedía en la implementación de las aplicaciones PSA, se ejecutan las tareas de postprocesamiento que sean necesarias.

Los experimentos realizados se basan en la ejecución del programa presentado en la figura 4.5 que implementa una aplicación maestro-esclavo donde se ejecutan en cada iteración tres trabajos esclavos y un trabajo maestro. Cada trabajo esclavo realiza operaciones aritméticas básicas sobre un fichero de entrada. A su vez el trabajo maestro comprueba en cada iteración si se alcanza el valor predefinido. Estos experimentos se realizaron en el banco de pruebas representado en la tabla 4.1.

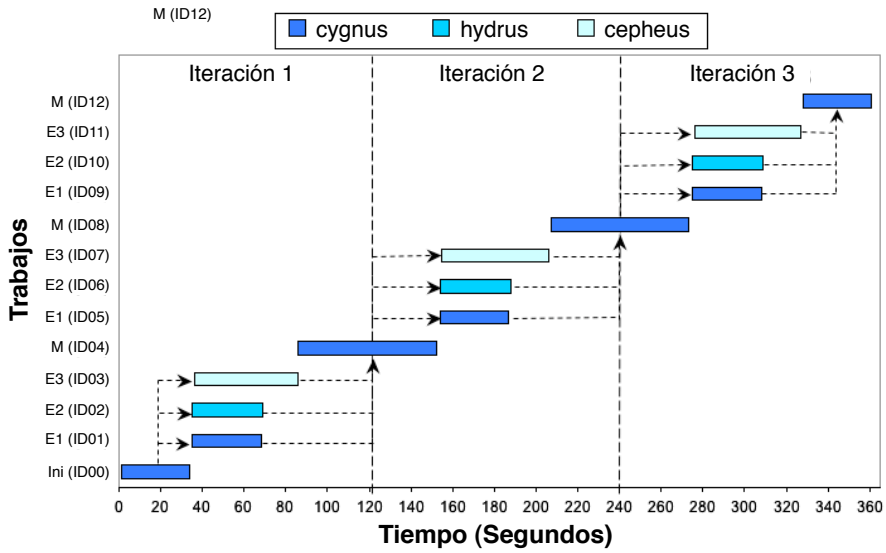

Figura 4.6: Perfil de Ejecución con tres Iteraciones de una Aplicación Maestro-Esclavo

La figura 4.6 representa el comportamiento de la aplicación maestro-esclavo en la que se necesitaron tres iteraciones para alcanzar el objetivo especificado. Así, el trabajo Ini hace referencia a las tareas de preprocesamiento realizadas, los trabajos E1, E2 y E3 se refieren a los trabajos esclavos, mientras que los trabajos M hacen referencia

a los trabajos maestros. En la ejecución se puede comprobar que el último trabajo ejecutado corresponde con un trabajo maestro, ya que no ha sido necesario realizar ninguna tarea de postprocesamiento. Los resultados muestran que efectivamente, el comportamiento del programa corresponde con el representado en la figura 4.4 por lo que queda demostrado la capacidad del modelo DRMAA-Grid *W*ay para desarrollar aplicaciones de este tipo. Además, el modelo DRMAA-Grid *W*ay permite la gestionar de forma sencilla, a través de ficheros, la transferencia de datos entre los distintos trabajos que intervienen en la ejecución de la aplicación. Estas pruebas sirvieron como base para la realización de un algoritmo genético distribuido basado en el modelo DRMAA-Grid *W*ay que se explica al detalle en la sección 4.5.

4.4. NAS Grid Benchmarks

En esta sección se analiza el comportamiento del modelo DRMAA-Grid *W*ay en la ejecución de los *NAS Grid Benchmarks* (NGB). Concretamente, el objetivo de este estudio no consiste en medir el rendimiento hardware del Grid, sino la funcionalidad, fiabilidad y rendimiento del entorno Grid complento con la incorporación del metaplanificador Grid *W*ay.

NGB [Rob02] engloba un conjunto de grafos de flujo de datos (DFG,*Data Flow Graph*) que se encapsulan en instancias de código NPB (*NAS Parallel Benchmarks*) en cada nodo del grafo, que se comunica con el resto de nodos enviando/recibiendo datos de inicialización. El conjunto de esquemas NGB sirve como modelo para apliaciones en Grid, constituyendo un excelente caso de estudio para comprobar la funcionalidad del modelo DRMAA-Grid *W*ay.

NGB está enfocado a Grids computacionales, utilizados principalmente para ejecutar trabajos de computación intensiva que procesan grandes conjuntos de datos. Cada *benchmark* conlleva la ejecución de varios NPB, que representan distintos tipos de computación científica. NGB propone un conjunto de perfiles de ejecución, cada uno de los cuales está estructurado en forma de grafo donde cada nodo es un NPB. La estructura del grafo así como los NPB utilizados varía con cada perfil de ejecución propuesto. Generalmente NGB utiliza cinco NPB distintos como nodos del grafo del perfil de ejecución [BBB+91]:

- ***Scalar Pentadiagonal Benchmark* (SP)**. Se engloba dentro de los problemas que buscan solución a sistemas de ecuaciones lineales. En este caso busca solución

a sistemas independientes de ecuaciones pentadiagonales.

- ***Block Tridiagonal Benchmark*** **(BT)**. Al igual que el benchmark SP obtiene soluciones de sistemas independientes de ecuaciones. Concretamente busca solución a sistemas de ecuaciones tridiagonales de bloques de tamaño (5 X 5).

- ***Lower and Upper Benchmark*** **(LU)**. Busca solución a sistemas lineales formado por matrices de bloques tridiagonales superiores e inferiores. Este problema presenta una menor fuente de paralelismo en comparación con los bechmarks BT y SP.

- ***Multigrid Benchmark*** **(MG)**. Se encarga de realizar diversas tareas de postprocesamiento de datos.

- ***Fourier Transform Benchmark*** **(FT)**. Calcula soluciones de ecuaciones diferenciales parcialmente tridimiensionales en el uso del analisis espectrales.

Además, NGB propone cuatro tipos distintos de familias de perfiles de ejecución. Cada familia está representada por un grafo con diferente estructura de comunicación entre los nodos [Rob02]:

- ***Embarrassingly Distributed*** **(ED)**. Agrupa a las aplicaciones de alta productividad, cuya estructura e implementación se ha discutido en la seccion 4.2. Por tanto, representan ejecuciones independientes de un mismo programa donde no existe comunicación entre cada una de las instancias de ejecución.

- ***Helical Chain*** **(HC)**. Representa grandes cadenas de procesos repetitivos cuya ejecución se va realizando de forma consecutiva. Este tipo de NGB se estudia con mas detalle en la sección 4.4.1.

- ***Visualization Pipe*** **(VP)**. Representa cadenas de procesos compuestos, como los que se pueden encontrar en la visualización de flujos de soluciones o en las simulaciones progresivas. Al igual que en el caso de *Helical Chain* el NGB *Visualization Pipe* se analiza con más detalle en la sección 4.4.2.

- ***Mixed Bag*** **(MB)**. Engloba también aplicaciones de flujos de datos, postproceso, y visualización, pero el énfasis se centra en introducir asimetría. De esta manera, diferentes cantidades de datos son trasferidas entre diferentes tareas. Además, algunas tareas requieren más procesamiento que otras, por lo que la

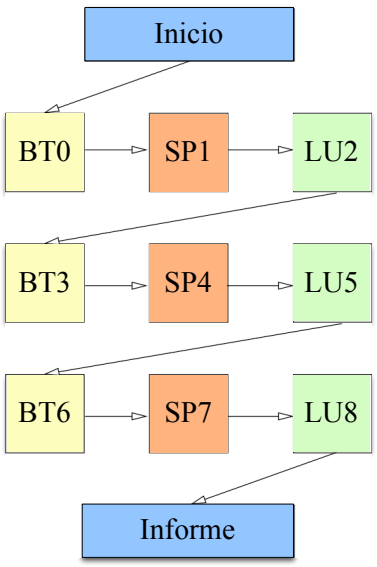

Figura 4.7: Esquema del NAS GRID Benchmark Helical Chain.

planificación de dichas tareas de forma eficiente dentro de un entorno Grid requiere de un esfuerzo adicional.

De entre todas las familias NGB presentadas a continuación se analiza con detalle dos de los cuatro tipos, concretamente son los tipos, Helical Chain (HC) y Visualization Pipe (VP).

4.4.1. Helical Chain (HC)

El *benchmark* HC consiste en una secuencia de trabajos que modelan una tubería computacional, es decir, esta constituida por simulaciones que conllevan el procesamiento de grandes cantidades de datos. Estas simulaciones se dividen en distintas tareas. Cada trabajo de la secuencia utiliza la solución obtenida por su predecesor para inicializarse. Teniendo en cuenta estas dependencias, cada trabajo de la cadena puede ser planificado por Grid*W*ay una vez que el trabajo anterior haya finalizado.

Un esquema de este tipo de aplicaciones se muestra en la figura 4.7. Como se puede observar, inicialmente existe un conjunto de tareas de preprocesamiento que son realizadas por el programa inicial. Este se encargará de ejecutar de forma distribuida cada uno de las tares del *benchmark Helical Chain* que se dividen a su vez en tres subtareas o trabajos. Cada trabajo corresponde a distintas aplicaciones que resuelven ecuaciones de flujo que se irán ejecutando de forma secuencial. Finalmente los resultados del último

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    drmaa_job_template_t   **jt;
    drmaa_attr_values_t    *rusage;
    .....

    drmaa_init(NULL, error);

    // Definición de los trabajos. Cada trabajo es un NPB
    drmaa_allocate_job_template(&jt[0], error);
    .....
    drmaa_allocate_job_template(&jt[8], error);
    drmaa_set_attribute(jt[0], DRMAA_REMOTE_COMMAND, "bt", error);
    .....
    drmaa_set_attribute(jt[8], DRMAA_REMOTE_COMMAND, "lu", error);

    for (i=0; i<9;i++)
    {
        // Ejecuta cada uno de los NPB y espera por su terminación
        drmaa_run_job(job_id, jt[i], error);
        drmaa_wait(job_id, &stat, timeout,&rusage, error);
    }

    drmaa_delete_job_template(&jt[0], error);
    ....
    drmaa_delete_job_template(&jt[8], error);
    drmaa_exit(error);
    return 0;
 }
```

Figura 4.8: Código DRMAA para el desarrollo del *benchmark* HC

trabajo, serán recogidos de nuevo por el programa principal para realizar un informe. Es importante destacar, que el arco del grafo en cada NGB representa transferencia de datos entre los nodos del grafo que une dicho arco.

La figura 4.8 representa el código del *benchmark* HC con el modelo DRMAA-Grid*W*ay. Debido a que este *benchmark* ejecuta cada NPB de forma secuencial, es necesario crear una plantilla por cada uno de los NPB que se vayan a ejecutar, ya que no se puede utilizar la llamada `drmaa_bulk_jobs()` que permite la ejecución concurrente de los trabajos. Por tanto el código ejecuta cada NPB en un nodo, espera por su finalización y ejecuta el siguiente NPB hasta completar la tubería. El *benchmark* HC, por tanto, permite analizar el comportamiento del modelo DRMAA-Grid*W*ay para aquellas aplicaciones que son estrictamente secuenciales. La figura 4.9 muestra los resultados obtenidos de la ejecución este programa.

Como se puede observar, el tiempo de respuesta total es de 17,56 minutos muy superior a 6,3 minutos que es el tiempo de respuesta obtenido en la ejecución secuencial del programa en el nodo con mayor capacidad de procesamiento del banco de pruebas.

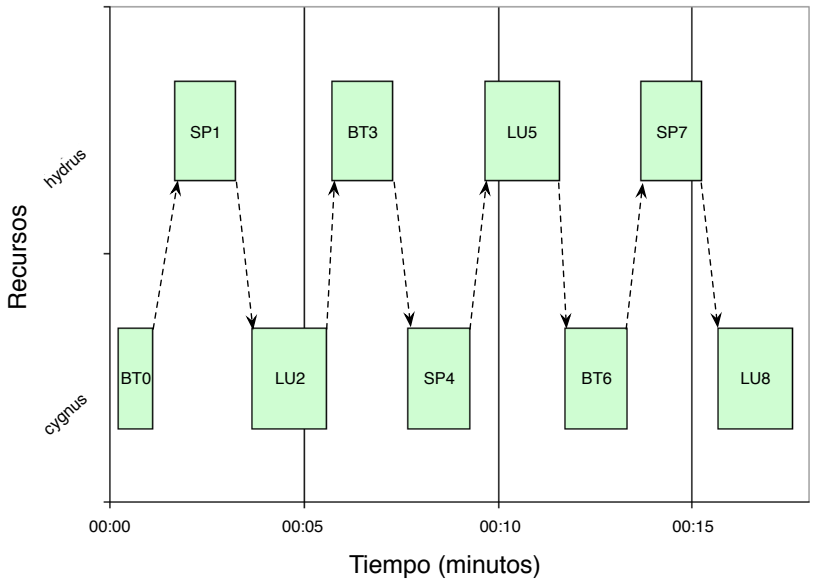

Figura 4.9: Perfil de ejecución del *bechmark* HC

Esto es debido a que existe una planificación oscilante de trabajos causado principalmente por el retardo del MDS en publicar la información del estado de los recursos. Esta planificación, por tanto, reduce claramente el rendimiento obtenido en comparación con la ejecución secuencial. Además, el uso medio de cada recurso es de solo un 20,21 %, con lo que tampoco se aprovecha la potencia computacional del banco de pruebas. El uso medio nos permite evaluar el rendimiento de la aplicación en relación a las capacidades computacionales de cada nodo del Grid. La formula 4.2 representa el cálculo del uso medio:

$$\overline{U_G} = \frac{\sum_{r \in G} \frac{T_r^{cpu}}{T_{wall}}}{|G|}, \tag{4.2}$$

dónde G representa al conjunto de máquinas que forman el banco de pruebas utilizado, en este caso es el que se presenta en la tabla 4.1, T_r^{cpu} es el tiempo total de CPU de todos los trabajos ejecutados en el recurso r y T_{wall} es el tiempo total de ejecución.

Sin embargo, este tipo de aplicaciones pueden ejecutar todas sus tareas encapsuladas en un único trabajo a través del modelo propuesto con el metaplanificador Grid*Way*. Así, los ficheros de salida de cada tarea de la cadena pueden ser gestionados por Grid*Way* como si fueran ficheros de reinicio. De esta manera, si el nodo falla durante la ejecución de dicha aplicación, esta puede migrar a otro nodo del grid y reanudar el trabajo a partir del último fichero de reinicio aprovechando las capacidades

proporcionadas por Grid*Way* para le ejecución de aplicaciones auto-adaptativas.

Utilizando este último método, el uso medio de recurso aumenta hasta un 91 %, ya que las nueve tareas de la misma cadena se planifican al mismo recurso. En este caso, el tiempo de respuesta es de 7 minutos, y el tiempo medio de ejecución se reduce a 6,4 minutos. Esto supone un decremento en el tiempo de respuesta del 60 % comparado con la primera estrategia de planificación, y un incremento de sólo el 11 % comparado con el caso óptimo. Por lo tanto, este modelo permite desarrollar el *benchmark Helical Chain* de manera adaptativa, y así es posible que continué la ejecución del *benchmark Helical Chain* aunque el nodo donde se ejecute falle.

4.4.2. Visualization Pipe (VP)

El *benchmark* VP está constituido por cadenas de procesos complejos cada uno de los cuales esta formado por tres tipos distintos de *NAS Parallel Benchmark*, a saber: una aplicación de resolución de ecuaciones de flujo, concretamente el *benchmark* BT; una aplicación de postprocesamiento, el *benchmark* MG; y finalmente, una aplicación de visualización, el *benchmark* FT.
Este *benchmark* es el resultado de la combinación de los *benchmarks* ED (*Embarrasingly Distributed*), *benchmark* totalmente paralelo, y del *benchmark* HC que es totalmente secuencial tal y como se ha descrito en la sección 4.4.1. Aunque el *benchmark* VP permite cierto paralelismo este se encuentra muy limitado debido a las dependencias entre los trabajos. Además, esta limitación se acentúa debido a que el tamaño de cada cadena del *benchmark* se compone únicamente de tres trabajos.

Como se observa en la figura 4.10 los *benchmarks* BT, MG y FT están localmente unidos formado una tubería, de esta manera se permite la ejecución concurrente de la aplicación de resolución de ecuaciones (BT), mientras se computa el postprocesamiento y visualización de las soluciones obtenidas previamente. Esto es posible porque el intercambio de datos se realiza como sigue. En primer lugar, cuando el *bechmark* termina de ejecutar, envía su resultado al siguiente nodo BT y al nodo MG que se encuentra en su mismo nivel. Este a su vez enviará el resultado de su ejecución al FT de su mismo nivel. Por lo tanto, el FT de un nivel dado recibirá datos del MG de su mismo nivel, y del FT del nivel inmediatamente superior. Cuando el *benchmark* FT termine su ejecución enviará su resultado al FT de nivel inmediatamente inferior o generará un informe en el caso de que se encuentre en el último nivel del NGB.

Aunque este tipo de aplicaciones podría serializarse y ejecutarse adaptativamente

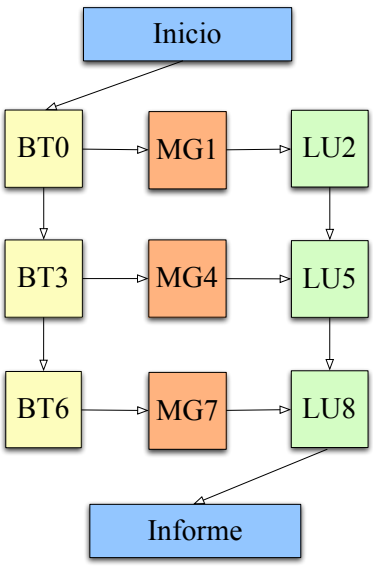

Figura 4.10: Esquema del *benchmark* VP.

como en el anterior caso, lo más apropiado es implementarlas como una aplicación de flujo de trabajo (*workflow*) para explotar el paralelismo que claramente representan.

Dado que Grid*W*ay no soporta directamente la ejecución de flujos de trabajos, se ha desarrollado un motor de flujo de trabajos (*workflow engine*), aprovechando el interfaz de programación DRMAA. Básicamente este motor de flujo de trabajos etiqueta cada uno de los trabajos asignándole una dependencia concreta, según el tipo de *benchmark* que se vaya a ejecutar. Posteriormente, el programa principal se encargará de ejecutar cada *benchmark* si y solo si sus dependencias han sido satisfechas. Debido a la flexibilidad del modelo DRMAA-Grid*W*ay este motor de flujos de trabajos se puede implementar a través de un programa en C con sentencias DRMAA. Este algoritmo sigue una distribución codiciosa, aunque se puede utilizar otras políticas de ejecución de trabajos, como por ejemplo, ejecutar primero los trabajos con requisitos más estrictos, o aquellos con mayor carga computacional, o con mayor número de dependencias.

La figura 4.11 representa el pseudo-código que desarrolla el *benchmark* VP. Para implementar las dependencias se ha utilizado un vector que almacena por cada trabajo el identificador de los trabajos con los que comparte dependencias. De esta manera, cuando se ejecuta un trabajo antes se comprueba que sus dependencias hayan sido satisfechas, sólo en ese caso se realiza la ejecución. Finalmente cuando un trabajo finaliza su ejecución se actualiza el vector de dependencias.

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    drmaa_job_template_t   **jt;
    drmaa_attr_values_t    *rusage;
    char                   **dependences;
    .....

    drmaa_init(NULL, error);

    // Definición de los trabajos. Cada trabajo es un NPB
    drmaa_allocate_job_template(&jt[0], error);
    drmaa_set_attribute(jt[0], DRMAA_REMOTE_COMMAND, "bt", error);
    .....

    dependences[0]=null;
    dependences[1]="0";
    ....
    dependences[8]="5 7";

    while(jobs_left())
    {
        // Ejecuta cada uno de los NPB en el orden preestablecido
        for (i=0; i<9;i++)
            if (dependences[i]==null)
                drmaa_run_job(job_id, jt[i], error);

       // Espera hasta que termine la ejecución de algún trabajo
       drmaa_wait("DRMAA_JOBS_IDS_SESSION_ANY", &stat, timeout, &rusage, error);
       dependences_update(dependences)
    }

    drmaa_delete_job_template(&jt[0], error);
    ....
    drmaa_exit(error);
    return 0;
 }
```

Figura 4.11: Código DRMAA para el desarrollo del *benchmark* VP

La figura 4.12 muestra los resultados del *benchmark* VP obtenidos de la ejecución del experimento en el banco de pruebas presentado en la tabla 4.1. Las líneas discontinuas representan dependencias entre los trabajos mientras que las líneas continuas representan el camino crítico que determina el tiempo de respuesta del *benchmark* completo. En este caso, el tiempo de respuesta es de 21,68 minutos, con un uso medio de recursos del 35,25 %. Los tiempos de ejecución y transferencia son de 22,93 y de 8,1 minutos, respectivamente.

Como se puede observar, el modelo DRMAA-Grid*W*ay facilita el desarrollo de este *benchmark* ya que el programador solo se ocupa de implementar el algoritmo que define a la aplicación. Además, aunque el API DRMAA no soporta el desarrollo de aplicaciones con dependencias de trabajos, se ha comprobado que el modelo DRMAA-Grid*W*ay permite su implementación de una manera sencilla haciendo posible la transferencia de

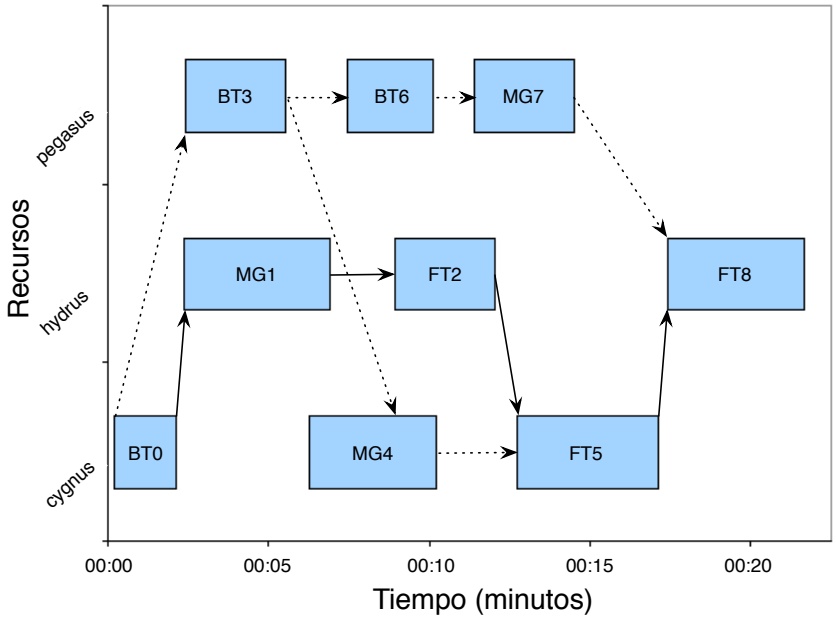

Figura 4.12: Perfil de ejecución del *bechmark* VP

información eficiente entre los distintos trabajos que componen el *benchmark*.

4.5. Algoritmo Genético

El objetivo de esta sección en analizar el comportamiento del modelo DRMAA-Grid*Way* en el desarrollo de un algoritmo genético distribuido para demostrar su viabilidad dentro de un modelo real. Para ello no solo se pretende desarrollar un algoritmo genético con este modelo, sino incorporar al algoritmo genético las características representativas del modelo como fiabilidad y adaptación dinámica.

El concepto de algoritmo genético tal y como se conoce hoy en día, fue concebido por el profesor John Henry Holland de la Universidad de Michigan y apareció por primera vez en su libro *"Adaptation in Natural and Artificial Systems"* [Hol92]. Básicamente el autor buscaba obtener un algoritmo para diseñar sistemas artificiales basados en los mecanismos más importantes de los sistemas naturales. En este sentido, el autor David E. Goldberg propone en [Gol89] la siguiente definición para los algoritmos genéticos:

> *Algoritmos de búsqueda basados en mecanismos de selección natural y genética natural, que combinan la supervivencia de los individuos más compatibles entre las estructuras de cadenas, con una estructura de información*

ya aleatorizada, intercambiada para construir un algoritmo de búsqueda con algunas de las capacidades de innovación de la búsqueda humana.

Es decir, el comportamiento de un algoritmo genético se basa principalmente en la creación de un conjunto nuevo de soluciones utilizando la información obtenida a partir de sus progenitores. De esta manera, los algoritmos genéticos explotan de forma eficiente la información histórica para la obtención de nuevos puntos de búsqueda a la espera de obtener soluciones optimas a un problema dado. Los conceptos más importantes dentro de la teoría de los algoritmos genéticos son: (i) Cromosoma, candidato a solución del problema a estudiar; (ii) Genes, codifican un elemento particular del cromosoma; (iii)Alelo, cada uno de los bits que forma el cromosoma; (iii) Población, conjunto de cromosomas; (iv) Genotipo, estructura del cromosoma dentro de un determinado algoritmo genético.

Todo algoritmo genético dispone de, como mínimo, tres tipos de operadores que utiliza para obtener la solución optima a un problema, estos son selección, cruce y mutación de cromosomas:

- **Selección**. También denominado reproducción, este término hace referencia al operador del algoritmo genético que escoje determinados cromosomas de entre toda la población para realizar la reproducción, dependiendo de las capacidades del cromosoma, este será escogido para reproducirse en más ocasiones.

- **Cruce**. El operador de cruce consiste en el intercambio de material genético entre dos cromosomas de dos padres, para crear una nueva descendencia. Su principal labor es elegir un lugar y cambiar los alelos antes y después de esa posición entre dos cromosomas y así crear dos cromosomas nuevos.

- **Mutación**. La operación de mutación consiste en la permutación en un bit del cromosoma. La elección de bit a permutar se realiza de manera aleatoria. La probabilidad con la que se produce la mutación suele ser muy pequeña ($\simeq 0{,}001$)

Las principales características que diferencian los algoritmos de los métodos tradicionales de búsqueda y optimización son:

- **Conjunto de Trabajo**. El conjunto de trabajo utilizado por los algoritmos genéticos se basa en cadena codificadas del conjunto de trabajo que representan los parámetros necesarios para obtener la solución. Por lo tanto los algoritmos genéticos no trabajan sobre una conjunto de parámetros de un problema sino, sobre

un conjunto de cadenas que representan dichos parámetros codificadas según un alfabeto dado.

- **Espacio de Búsqueda**. El espacio de búsqueda está constituido por una población de muestras en lugar de una única muestra. De esta manera se reduce la posibilidad de obtener una solución invalida.

- **Función Objetivo**. Los algoritmos genéticos hacen uso de una función objetivo para elegir la solución optima de entre todas las soluciones obtenidas, ganando eficiencia frente a los algoritmos de búsqueda clásicos que hacen uso de otras funciones más complejas, como derivadas o integrales, para obtener la mejor solución.

- **Reglas no Deterministas**. Se basan en reglas de transición estocásticas para guiar la búsqueda de los candidatos a la solución de un problema.

Con el objetivo de mejorar la eficiencia y la calidad de la búsqueda de los algoritmos genéticos surgieron los denominados Algoritmos Geneticos Paralelos (PGA, *Parallel Genetic Algorithm*), también denominados Algoritmos Genéticos Distribuidos (DGA, *Distributed Genetic Algorithm*). De entre las principales ventajas que ofrecen estos algoritmos frente a los algoritmos genéticos clásicos se encuentra la disminución del tiempo de convergencia de aquellos problemas que debido a sus dimensiones es necesario el uso de poblaciones numerosas, o de aquellos problemas con múltiples funciones objetivo no triviales. Además los PGA explotan el paralelismo intrínseco que existe en el comportamiento evolutivo de los algoritmos pudiendo trabajar simultáneamente sobre diversas poblaciones semi-independientes para la obtención de la solución optima.

Aunque hubo estudios previos que aplicaron técnicas de paralelismo a los algoritmos genéticos clásicos, fue John J. Grefenstette [Joh81] en 1981 quien propuso una primera taxonomía sobre los PGA. Este autor clasificó los PGA en tres categorías: (i) modelo maestro-esclavo; (ii) modelo de máquinas paralelas de memoria compartida; (iii) modelo distribuido de poblaciones múltiples. Posteriormente, los autores C. Pettey et al. propusieron en [PMG87] un nuevo modelo de PGA basado en subpoblaciones homogéneas cooperativas denominadas “islas"que incluía topología de migración dinámica. Finalmente, en 1989 el autor Reiko Tanese en [Tan89] estudió la influencia en las migraciones en la perdidas de diversidad de soluciones y propuso mecanismos de selección y cruzamiento para cada isla, generando de esta manera PGA heterogéneos.

Los anteriores modelos presentados han ido evolucionando hasta formar ocho categorías identificadas por los autores M. Nowostawski y R. Poli en [NP99]:

- **Modelo Maestro-Esclavo**. Consiste en distribuir funcionalmente el algoritmo, asignando a diferentes procesadores distintas etapas del mecanismo evolutivo, normalmente se suele distribuir la evaluación de la función objetivo.

- **Modelo Distribuido de Subpoblaciones con Migración**. La población total se distribuye en cada nodo donde se ejecuta un algoritmo genético secuencial con cada subpoblacion. Cada cierto tiempo se realizan migraciones entre nodos con el objetivo de optimizar la solución al problema general. El principal inconveniente de este modelo radica en la dificultad de implementar soluciones que sean escalables a grandes problemas, por lo que generalmente se utilizan modelos híbridos conjuntamente con el modelo maestro-esclavo.

- **Modelo de Poblaciones Estáticas Solapadas**. Este modelo no utiliza el operador de migración entre subpoblaciones, y por tanto, el intercambio de información solamente se puede realizar entre las áreas de población que se encuentran solapadas. Estas áreas están formadas por individuos que perteneces al mismo tiempo a distintas “islas".

- **Algoritmos Genéticos Masivamente Paralelos**. Es una variación del modelo anterior en el que se incrementa el número de islas y se decrementa el número de individuos por islas, en la mayoría de los casos cada población está compuesta por un solo individuo. El intercambio de información se realiza a través de la operación de difusión, debido a ello este modelo es adecuado para sistemas masivamente paralelos.

- **Modelo de Poblaciones Dinámicas**. Solución híbrida que combina el modelo maestro-esclavo con el modelo de poblaciones estáticas con solapamiento. Así, en cada generación las islas se van creando de forma dinámica tan pronto como se obtengan los individuos necesarios, produciéndose de esta manera el intercambio entre individuos de distintas poblaciones.

- **PGA Estacionarios**. Utilizan los mecanismos de sección crítica para realizar la paralelización del algoritmo, concretamente los individuos se encuentran en la sección crítica y debido a ello las operaciones de selección y reemplazo deben hacerse de forma secuencial.

- **PGA Desordenados**. Estos algoritmos se dividen en dos fases, una fase principal denominada *primordial phase* en la que se crea la población utilizando una estrategia de distribución parcial por bloques de soluciones, y una fase de yuxtaposición donde se mezclan las soluciones obtenidas en cada bloque con el objetivo de hallar la solución optima. Debido a que la *primordial phase* es la que conlleva mayor tiempo de ejecución, es esta fase la que se paraleliza en los PGA.

- **PGA Híbridos**. Estos algoritmos combinan distintos modelos de PGA, normalmente se suelen distribuir las poblaciones en diferentes niveles jerárquicos, en cada uno de los cuales se aplican distintos modelos de PGA. De esta manera, se combinan las ventajas de los diferentes modelos de PGA utilizados.

Aunque, como se ha visto, existen una gran variedad de modelos de PGA, desarrollar estos algoritmos dentro de un entorno Grid heterogéneo no es trivial y requiere de un gran esfuerzo por parte del desarrollador. Debido a la facilidad que presenta el modelo DRMAA-Grid *W*ay para el desarrollo de aplicaciones en Grid, se utilizará dicho modelo para desarrollar un algoritmo genético distribuido y adaptativo a un entorno Grid heterogéneo. La siguiente sección explica en detalle el desarrollo de este algoritmo, así como las ventajas que aporta la aplicación del modelo frente a otro algoritmos genéticos distribuidos.

4.5.1. Algoritmo Genético DRMAA

Según los autores H. Imade et al. en [IMO+04] los algoritmos genéticos orientados a entornos Grid se denominan GOGAs (*Grid-Oriented Genetic Algorithm*), debido a que el algoritmo orientado a entornos Grid presentado a continuación ha sido desarrollado bajo el modelo de implementación de DRMAA-Grid *W*ay se denominará a este nuevo algoritmo como DRMAA GOGA.

Para la implementación de este algoritmo se ha utilizado un modelo híbrido de PGA basado en el modelo de islas y en el modelo maestro-esclavo. Debido a que es un modelo híbrido, DRMAA GOGA hereda del modelo de islas la característica que consiste en dividir la población total en subpoblaciones cada una de las cuales se ubicará en una isla o nodo distinto, donde se realizará la evaluación de esa subpoblación a través de la ejecución de un algoritmo secuencial. Los resultados de esta evaluación llegarán al nodo maestro que será el encargado de evaluar las distintas soluciones, realizar la migración entre las distintas subpoblaciones, generar una nueva población global para

la siguiente generación y volver a distribuir la población entre los nodos esclavos. Con el objetivo de optimizar el equilibrio de carga entre todos las islas, la conectividad entre ellas es total por lo que cada máquina o procesador intercambia individuos con el resto de máquinas o procesadores en cada generación del algoritmo. Esta comunicación además se aprovecha para realizar los *checkpoints* necesarios en cada una de las generaciones.

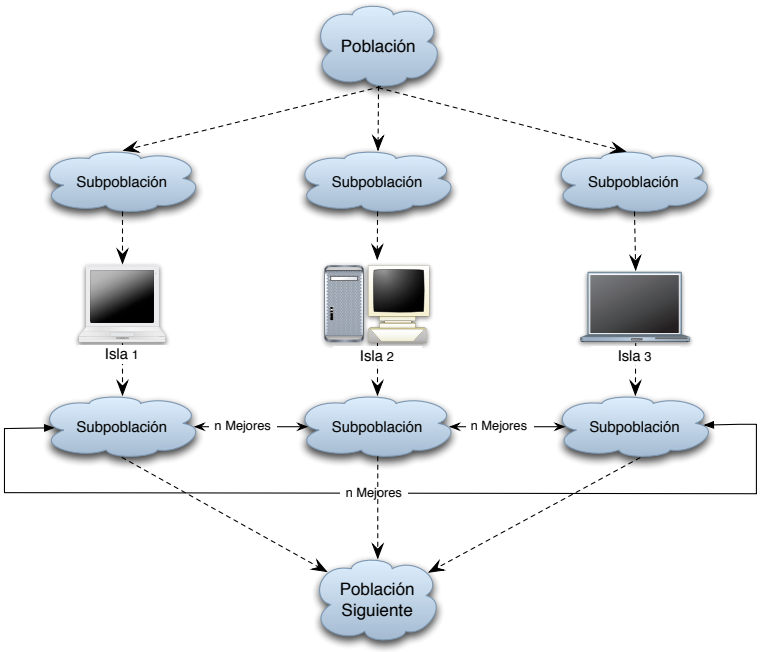

Figura 4.13: Esquema del algoritmo DRMAA GOGA con tres islas

La figura 4.13 representa un esquema del algoritmo DRMAA GOGA cuando se dispone de tres islas. Como se puede observar en dicha figura, la población inicial se divide en subpoblaciones que se almacenan en cada nodo (isla). Posteriormente, se procede a la ejecución de un algoritmo genético secuencial tomado como base la subpoblación de cada isla. Con los resultados obtenidos de la ejecución del algoritmo secuencial en cada nodo, se escogen los mejores individuos que se intercambian con el resto de mejores individuos de cada una de las restantes islas. Seguidamente el algoritmo DRMAA GOGA genera una nueva población que se utilizará en la siguiente generación.

La principal desventaja que tiene la solución propuesta radica en la heterogeneidad de los nodos, es decir, para realizar el proceso de migración y de posterior creación de la nueva población, es necesario que terminen de ejecutar todos los nodos involucrados

en el proceso y por lo tanto el tiempo de finalización de cada generación quedará determinado por el nodo con velocidad de procesamiento menor.

Con el objetivo de eliminar esta desventaja, el algoritmo DRMAA GOGA implementa una nueva característica dentro del campo de los algoritmos genéticos denominada "conectividad dinámica", que consiste en permitir al usuario de la aplicación que determine el número máximo de trabajos por los que el algoritmo genético esperará su terminación, para analizar los resultados de la generación actual y preparar la siguiente. De esta manera, el algoritmo DRMAA GOGA permite la conexión dinámica de los nodos del Grid, reduciendo la influencia de los nodos más lentos sobre el tiempo total de ejecución del algoritmo.

Al igual que en los casos anteriores el desarrollo del algoritmo DRMAA GOGA se ha simplificado gracias al uso de la especificación DRMAA y del metaplanificador Grid*Way*. En este caso el programa está realizado bajo el lenguaje C++ y consta de un programa principal, que será el encargado de realizar las funciones de programa maestro, y un conjunto de procesos esclavos, cada uno de los cuales ejecutará un algoritmo secuencial. De esta manera, el programa generará tantos procesos esclavos como nodos de los que se disponga para distribuir la población global. Así, cada uno de los trabajos que se lanzará al Grid estará compuesto por uno de estos procesos esclavos. De esta manera, la creación, manipulación y espera de estos trabajos se realiza a traves de sentencias DRMAA dentro del código del programa maestro y el metaplanificador Grid*Way* se encargará de todo el proceso de planificación y control de dichos trabajos.

La figura 4.14 representa el pseudo-código de la implementación del algoritmo DRMAA GOGA. La metodología es muy similar a la de los anteriores experimentos. En primer lugar se definen tantas plantillas como trabajos se necesitan para cada una de las generaciones del algoritmo. Después se ejecutan todos los trabajos, comprobando previamente que no estén en ejecución, pero solo se espera por la finalización de un número de ellos determinado por el usuario. Al finalizar cada generación se comprueba si se ha llegado al objetivo deseado, si es así termina la ejecución, en otro caso, se actualizan los parámetros del algoritmo para su ejecución en la siguiente generación y se continúa con la ejecución.

4.5.2. Descripción del Experimento y Resultados

Con el objetivo de mostrar la funcionalidad del algoritmo genético DRMAA GOGA se ha ejecutado dicho algoritmo en búsqueda de solución al problema de 1's máximos.

```
#include <drmaa.h>
int main(int argc, char **argv)
{
    drmaa_job_template_t   **jt;
    int                    *job_ids;
    .....

    drmaa_init(NULL, error);

    // Definición de los trabajos. Cada trabajo es un isla.
    drmaa_allocate_job_template(&jt[0], error);
    .....

    while(!ga_done())
    {
        // Ejecuta cada uno de los trabajos
        for (i=0; i<num_task;i++)
        {
            drmaa_job_ps(job_ids[i], &status, error)
            if(status != DRMAA_ERRNO_SUCCESS)
               drmaa_run_job(job_ids[i], jt[i], error);
        }
        // Espera solo por un número determinado de ellos
        for (i=0; i<num_jobs; i++)
            drmaa_wait("DRMAA_JOBS_IDS_SESSION_ANY", &stat, timeout, &rusage, error);
        algorithm_update();
    }

    store_final_result();
    drmaa_delete_job_template(&jt, error);
    drmaa_exit(error);
    return 0;
 }
```

Figura 4.14: Código DRMAA para el desarrollo del algoritmo genético

El problema de 1's maximos (*One-max problem* [SE91]) es un test utilizado tradicionalmente en la programación de algoritmos genéticos, en el que se dispone inicialmente de una matrix de NxM ceros con el objetivo de obtener una matriz de NxM unos. En el caso concreto que se describe en esta sección la matriz inicial está compuesta por $20x100$ ceros, es decir la matriz contiene 2000 elementos, y el objetivo es llegar a obtener una matriz de $20x100$ unos. El conjunto de experimentos realizados se ha llevado a cabo utilizando el banco de pruebas denominado UCM-UPC que se muestra en la tabla 4.2.

Como se puede observar el banco de pruebas o *testbed* está formado por cinco máquinas, tres pertenecientes a la Universidad Complutense de Madrid (hydrus, cygnus, cepheus), una perteneciente a la Universidad Politécnica de Cataluña (khafre), y finalmente una perteneciente al Centro de Astro-Biologia (babieca). Todas las máquinas constan de un único procesador, a excepción de babieca que dispone de 5 procesadores, por lo tanto se dispone de un total de 9 procesadores para realizar los experimentos. Finalmente es importante destacar que las máquinas más veloces corresponden a hydrus

Nombre	OV	Modelo	Velocidad	SO	Memoria	DRMS
hydrus	UCM	Intel P4	2.5GHz	Linux 2.4	512MB	fork
cygnus	UCM	Intel P4	2.5GHz	Linux 2.4	512MB	fork
aquila	UCM	Intel PIII	700MHz	Linux 2.4	128MB	fork
khafre	UPC	Intel PIII	700MHz	Linux 2.2	512MB	fork
babieca	CAB	5×Alpha DS10	450MHz	Linux 2.2	256MB	pbs

Tabla 4.2: Banco de pruebas UCM-UPC para el algoritmo genético.

y cygnus, mientras que la máquina más lenta se corresponde con el *cluster* babieca.

Para el primer experimento realizado en este banco de pruebas se ha escogido una población de 1000 individuos, cada uno de los cuales está compuesto por una matriz de $20x100$ ceros. Además en cada isla se ha ejecutado un número total de 50 generaciones con una probabilidad de mutación y cruce de 0,1 % y de un 60 % respectivamente. El número de individuos que se intercambian en cada generación es de un 10 %. Estos valores son los típicamente utilizados en la implementación de algoritmos genéticos. Así mismo, se ha utilizado una conectividad dinámica de grado 5, es decir se espera a la finalización de la ejecución del algoritmo secuencial en 5 islas o nodos. Finalmente, el número de generaciones del algoritmo DRMAA GOGA ha sido de 4. Con este experimento se quiere comprobar la funcionalidad del uso de la conectividad dinámica para la ejecución del algoritmo DRMAA GOGA.

La figura 4.15 muestra el tiempo de ejecución en cada uno de los nodos y en cada generación del algoritmo DRMAA GOGA para este experimento. Como se puede observar, debido al uso de la conectividad dinámica el algoritmo DRMAA GOGA asigna menos trabajos a los nodos más lentos, aprovechando de esta manera la capacidad computacional de los nodos más rápidos y por lo tanto ajustando el equilibro de carga computacional del banco de pruebas. Además, como se comentó previamente, la conectividad dinámica también permite utilizar la capacidad de los nodos más lentos sin que su tiempo de procesamiento diriga el comportamiento del algoritmo genetico DRMAA GOGA. Finalmente, se puede comprobar que gracias al uso del modelo de programación presentado en la sección 3.5, el algoritmo es auto-adaptativo, es decir, si en algún momento un nodo deja de estar disponible, el propio algoritmo distribuye el resto de carga entre los nodos que sí lo estén. Esta característica se ve reflejada en el caso de la máquina cygnus. Como muestra esta figura la máquina cygnus solo interviene en la primera generación del algoritmo, ya que después pasó al estado no-diponible,

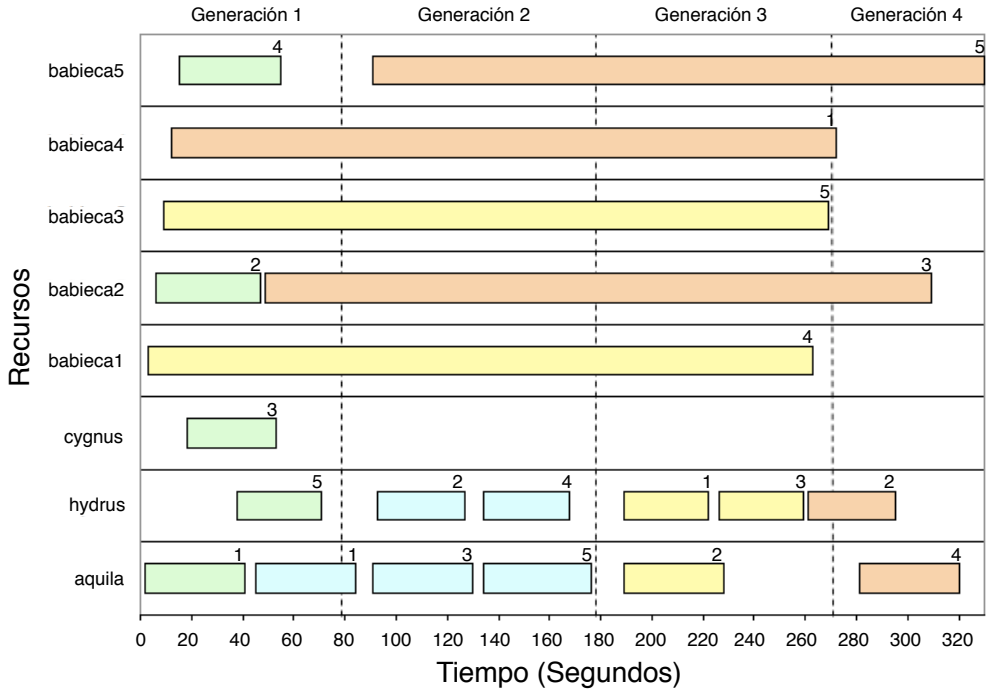

Figura 4.15: Tiempo de ejecución frente a recursos y generaciones

obligando al algoritmo DRMAA GOGA a prescindir de este nodo y a distribuir el resto de trabajos entre los nodos disponibles. Como se puede observar, esta situación no entorpece la ejecución del algoritmo genético, permitiendo que continúe la ejecución sin que el usuario tenga que intervenir en ningún momento.

El siguiente experimento está compuesto por cinco ejecuciones del algoritmo DRMAA GOGA, cuyo grado de conectividad dinámica es diferente en cada ejecución. Concretamente se ha utilizado una conectividad dinámica de grado 2, 3, 5, 6 y 8. Con este experimento se pretende estudiar el comportamiento del algoritmo dependiendo del grado de conectividad dinámica en el cálculo del valor optimo para el problema *One´s Max Problem*. Los valores referidos al tamaño de la población, al número de elementos de de cada individuo, a la probabilidad de cruce o mutación son los mismos que para el experimento anterior, solo difiere en el número de generaciones del algoritmo DRMAA GOGA que en este caso corresponde a 10 generaciones.

La figura 4.16 muestra los resultado obtenidos de la ejecución de este experimento. Concretamente, presenta la relación existente entre los resultados obtenidos en cada generación (*score*), y el tiempo requerido para obtener el valor máximo. En el caso de estos experimentos se ha escogido como valor máximo dentro de la función objetivo,

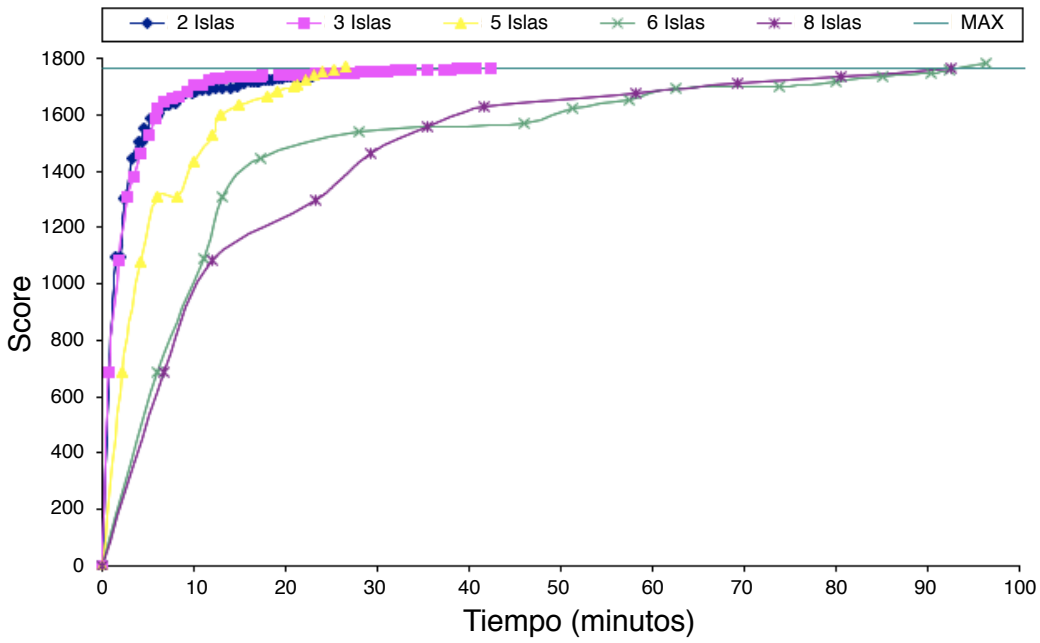

Figura 4.16: Resultado por tiempo de ejecución y número de islas

un número total de 1800 unos.
Tal y como muestra dicha figura, la solución más cercana a la optima es la correspondiente con la ejecución que utiliza una conectividad dinámica de grado cinco (amarillo), para este caso se llega a una solución óptima en un tiempo mucho menor que el resto. Esto se debe a que 5 es la cota superior del *testbed* con respecto a los nodos más rápidos, es decir, a partir de una conectividad dinámica de grado 6 comienzan a incluirse los nodos más lentos dentro de la lista de nodos a espera de su finalización, y por lo tanto, su velocidad de procesamiento influye de manera considerable en el tiempo total de ejecución de cada generación del algoritmo DRMAA GOGA.

Con el objetivo de mostrar la influencia del grado de conectividad dinámica en el tiempo de ejecución de cada generación del algoritmo DRMAA GOGA, se han analizado los tiempos de finalización en cada una de las generaciones del algoritmo con diferente conectividad dinámica, el resultado de ese análisis se presenta en la figura 4.17. Dicha figura representa el tiempo requerido para terminar cada generación de un total de 10 generaciones del algoritmo genético DRMAA GOGA, dependiendo del grado de conectividad dinámica que se utilice. Para este experimento, las mejores propuestas son las soluciones que utilizan conectividad dinámica de grado 5, 3 y 2, ya que, como se comprobó en la figura 4.16, debido a las características del banco de pruebas utilizado,

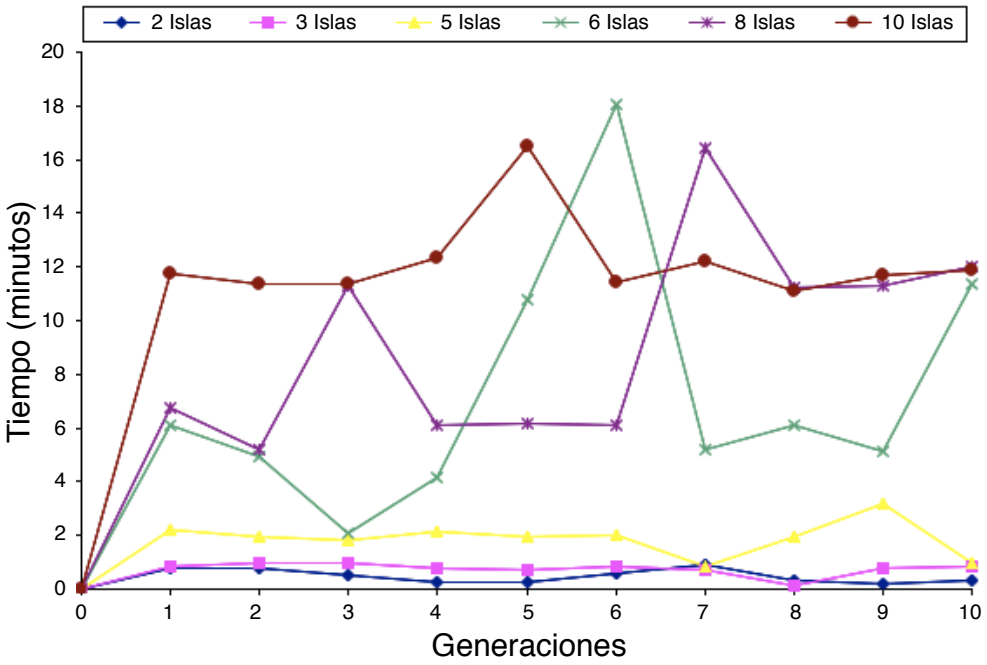

Figura 4.17: Tiempo de ejecución de cada generación por conectividad dinámica

a partir de uso de una conectividad de grado mayor que 5, se incorporan a la lista de nodos de espera nodos con velocidad de procesamiento mucho menor, aumentando de esta manera el tiempo total de ejecución por generación, y por lo tanto, el tiempo total de ejecución del algoritmo DRMAA GOGA.

4.6. Conclusiones

Para comprobar la funcionalidad y eficiencia del modelo de desarrollo para aplicaciones en Grid, este capítulo ha presentado distintos escenarios donde se prueba dicha funcionalidad a través de la implementación de diferentes perfiles de aplicación. El capítulo se divide en tres partes principales, cada una corresponde con un perfil de ejecución determinado. La primera de ellas corresponde con la ejecución de aplicaciones de alta productividad y aplicaciones maestro-esclavo. En los dos casos se comprueba la eficiencia del modelo en el desarrollo y ejecución de este tipo de aplicaciones, ya que además de facilitar su desarrollo, el modelo DRMAA-Grid *W*ay aprovecha la distribución de trabajos para decrementar el tiempo total de ejecución e incrementar la productividad del Grid. Además, la ejecución de estas aplicaciones pone de manifiesto

la gestión de intercambio de información entre los distintos trabajos que conforman la aplicación, que se realiza a través de ficheros almacenados en memoria secundaria.

La segunda parte analiza el comportamiento del modelo en el desarrollo y ejecución de los *NAS Grid Benchmarks* concretamente de los tipos Helical Chain (HC) y Visualization Pipe (VP). En primer lugar, debido a la naturaleza estrictamente secuencial del *bechmark* Helical Chain, ha sido necesario encapasular todas las tareas en un único trabajo de tal manera que se ejecuten en un único nodo. Esto permite desarrollar el *benchmark Helical Chain* de manera adaptativa y así hacer posible que continúe su ejecución aunque el nodo donde se ejecute falle.
Para implementar el *bechmark* Visualization Pipe ha sido necesario desarrollar un motor de flujo de trabajos (*workflow engine*), aprovechando el interfaz de programación DRMAA, ya que el modelo no soporta directamente la ejecución de flujos de trabajos con dependencias. Aún así, el modelo DRMAA-Grid*Way* permite su implementación de una manera sencilla realizando una transferencia de información eficiente entre los distintos trabajos que componen el *benchmark*, reduciendo además el tiempo de ejecución de la aplicación.
Por lo tanto queda demostrado la eficiencia en el modelo DRMAA-Grid*Way* para el desarrollo de estos *benchmarks*, que permite además reducir el tiempo total de ejecución así como dotar a los *benchmarks* de adaptabilidad a entornos Grid heterogéneos.

La última parte estudia el comportamiento del modelo DRMAA-Grid*Way* en el desarrollo y ejecución de un nuevo algoritmo genético distribuido. Para ello, presenta el desarrollo de este nuevo algoritmo basado en el modelo DRMAA-Grid*Way* y denominado DRMAA GOGA (Grid Oriented Genetic Algorithm). La característica fundamental de este algoritmo consiste en lo que se ha denominado como conectividad dinámica. Esta característica permite al usuario del algoritmo especificar el máximo número de nodos que desea que entre en el cómputo de cada generación. De esta manera, el usuario puede estudiar el comportamiento del algoritmo dependiendo del valor de este parámetro. Los resultados demuestran que el algoritmo converge más rápidamente en aquellos casos en los que la conectividad dinámica es igual al número de nodos con mayor capacidad de procesamiento del Grid. Esta conclusión tiene sentido, ya que en cada generación se esperará por la finalización de ejecución de los nodos más rápidos, mientras que los más lentos continúan su ejecución y sus resultados se utilizarán en la siguiente generación. Por lo tanto queda demostrada la eficiencia del modelo DRMAA-Grid*Way* en el desarrollo de algoritmos genéticos distribuidos permitiendo

que su ejecución sea más eficiente y autoadaptativa a entornos Grid heterogéneos.

Como se ha demostrado, el modelo de desarrollo DRMAA-Grid *W*ay ofrece un gran abanico de posibilidades dentro del campo de la programación distribuida ya que permite una adaptación débilmente acoplada de las aplicaciones, al contrario que el paradigma MPI, aprovechando de esta manera la potencia computacional del Grid y facilitando su desarrollo. El siguiente capítulo presenta un nuevo método de paralelización de código basado en el modelo DRMAA-Grid *W*ay.

Capítulo 5

Planificación de Iteraciones en Grid

El objetivo de este capítulo es presentar un nuevo método de planificación de iteraciones en Grid utilizando el modelo de desarrollo DRMAA-Grid *W*ay. Con ello se pretende desarrollar un nuevo algoritmo de distribución de iteraciones que se adapte a la necesidades de un Grid heterogéneo, y permita una distribución uniforme de la carga computacional de los trabajos entre los nodos, disminuyendo de esta manera el tiempo de ejecución de la aplicación. Además este nuevo algoritmo debe obtener un tiempo de planificación despreciable en comparación con el tiempo total de ejecución de la aplicación.

Para justificar las necesidades de un algoritmo de planificación de iteraciones en Grid, en primer lugar se realizará una introducción sobre los bucles paralelos y las principales motivaciones de su uso. En la sección 5.2 se explicará las principales características de los algoritmos de distribución estáticos. Posteriormente, en la sección 5.3 se describirán al detalle los distintos algoritmos de planificación dinámicos así como su modelo Maestro-Esclavo. A continuación, se explicarán los principales algoritmos de planificación distribuidos.

Este capítulo finalizará con la sección correspondiente al nuevo algoritmo de planificación Grid Trapezoid Self-Scheculer. La sección comenzará describiendo las principales aportaciones de este nuevo algoritmo para posteriormente explicar los dos elementos fundamentales en los que se basa: el modelo Benchmark de Grid y el factor de relación R_i^{min}. Finalmente este capítulo concluirá con la descripción exhaustiva del nuevo algoritmo, y con los resultados obtenidos de su ejecución respecto a los algoritmos de distribución clásicos.

5.1. Introducción

Con la llegada de los sistemas distribuidos, y concretamente los sistemas distribuidos heterogéneos como la tecnología Grid, cobra cada vez más importancia la necesidad de asignar tareas a cada uno de los nodos con el objetivo de maximizar el equilibro de carga. El problema principal consiste en la dificultad de asignar las diferentes partes de una aplicación paralela a cada uno de los recursos computacionales, y de esta manera, minimizar el tiempo total de ejecución de la aplicación obteniendo a su vez un uso eficiente de dichos recursos. Una de las técnicas más eficientes que permite explotar el paralelismo intrínseco de las aplicaciones es la distribución de las iteraciones o bucles que componen dicha aplicación. Concretamente, esta técnica se encarga de distribuir bucles paralelos, que son aquellos en los que no existen dependencias entre iteraciones y que por tanto cada iteración se puede ejecutar de manera independiente, incluso al mismo tiempo. Si se define carga computacional [Lil94] como el conjunto de instrucciones que se ejecutan dentro de un bucle paralelo, se pueden clasificar estos bucles en cuatro tipos distintos, a saber:

- *Bucles con carga computacional uniforme*. Denominados también bucles uniformemente distribuidos, son aquellos en los que la carga computacional de cada instrucción dentro del bucle es la misma.

- *Bucles con carga computacional ascendente*. Pertenece al grupo de los bucles linealmente distribuidos, y son aquellos cuya carga computacional crece con cada iteración del bucle.

- *Bucles con carga computacional descendente*. Al igual que los anteriores son bucles linealmente distribuidos, pero en esta ocasión la carga computacional decrece en cada iteración. Los bucles linealmente distribuidos se suelen encontrar en aplicaciones científicas más comunes, como la multiplicación de matrices.

- *Bucles con carga computacional irregular*. Son aquellos en los que la carga computacional varía en cada iteración de forma aleatoria. Este tipo de bucles se suele encontrar en el cálculo de fractales, como ocurre cuando se computa el conjunto de Mandelbrot [Man88].

Para distribuir el código existente en cada uno de estos tipos de bucles, se utilizan distintas soluciones. La más común consiste en utilizar los algoritmos de planificación

que se clasifican en algoritmos de planificación estáticos y algoritmos de planificación dinámicos. Estás técnicas se han utilizado sobretodo en sistemas de memoria compartida [PD97] o en *clusters* utilizando el paradigma de programación MPI [CBGA01]. El objetivo de este capítulo es presentar un nuevo algoritmo de planificación que se adapte a un entorno Grid heterogéneo y que pueda aprovechar todas las características del modelo de programación DRMAA-Grid *W*ay.

5.2. Algoritmos Estáticos

Denominados también algoritmos preplanificados, son aquellos en los que el cálculo del número de trabajos o iteraciones por nodo, se realiza en tiempo de compilación, dentro de este tipo de algoritmos se encuentran:

- *Planificador por Bloques (BS)* [Lil94]. Este planificador estático divide la carga total, es decir el número de bucles, entre todos los procesadores de manera equitativa. Por tanto, si se dispone de N iteraciones y p procesadores, a cada nodo se le adjudicará un bloque de $\lceil \frac{N}{p} \rceil$ iteraciones de manera consecutiva. La principal desventaja de este algoritmo de planificación reside en el desequilibrio de carga computacional cuando dicha carga difiere en cada iteración. Supóngase un escenario en el que la carga computacional de cada iteración se incrementase linealmente, como ocurre en los bucles linealmente distribuidos. En este caso, los primeros nodos acabarían antes que el resto, disminuyendo de está manera el rendimiento total del sistema. Si en lugar de incrementarse la carga computacional se decrementase, serían los últimos nodos los que terminarían antes que los primeros, produciendose de la misma manera una disminución en el rendimiento total.

- *Planificador Cíclico(CS)* [Lil94]. Al igual que el planificador por bloques, a cada nodo se le asigna $\lceil \frac{N}{p} \rceil$ iteraciones. Sin embargo, en este algoritmo la asignación no se produce de manera consecutiva, sino de forma cíclica, de este modo se resuelve el problema de desequilibrio de carga que generaba el anterior algoritmo. Si se dispone de p procesadores en el sistema, y se encuentra ejecutando el procesador p_x, este procesador ejecutaría la secuencia de iteraciones $x, x + p, 2x + p$ y así sucesivamente.

Los algoritmos estáticos realizan la planificación en tiempo de compilación, por ello, es necesario conocer de antemano, el número de iteraciones totales de la aplicación y el número de nodos o procesadores disponibles durante toda la ejecución. Obtener dicha información en un entorno heterogéneo y distribuido como el Grid, es una tarea bastante complicada, más aún si unimos a esas características el dinamismo existente en la aplicaciones actuales.

5.3. Algoritmos Dinámicos

Surgieron como contrapartida a los algoritmos de planificación estáticos y a diferencia de estos, realizan la planificación en tiempo de ejecución en lugar de en tiempo de compilación. Por ello, es recomendable su uso en aquellos problemas de paralelización donde se desconoce de antemano el tiempo de ejecución de cada iteración o donde el número de iteraciones no es fijo. Todos ellos denominan como *chunk* o pedazo al número de iteraciones que se van a realizar en cada nodo. Dependiendo de cómo se calcule este *chunk* o pedazo, se obtienen distintos tipos de algoritmos dinámicos, a saber:

- *PSS (Pure Self-Scheduling)* o Algoritmo de Planificación Puro [TY86].
- *CSS (Chunk Self-Scheduling)* o Algoritmo de Planificación por Pedazos [TY86].
- *GSS (Guided Self-Scheduling)* o Algoritmo de Planificación Guiado [PK87].
- *TSS (Trapezoid Self-Scheduling)* o Algoritmo de Planificación Trapezoidal [TN93].
- *FSS (Factoring Self-Scheduling)* o Algoritmo de Planificación Factorizado [HSF92].
- *FISS (Fixed Increase Self-Scheduling)* o Algoritmo de Planificación Ajustado Incremental [PD97].

Estos algoritmos planifican las iteraciones siguiendo un modelo Maestro-Esclavo, se explicará en primer lugar este modelo para posteriormente detallar cada uno de los algoritmos de planificación dinámicos de forma individual.

5.3.1. Modelo Maestro-Esclavo

En el modelo Maestro-Esclavo se etiqueta un nodo o procesador de sistema como nodo Maestro (*Master*), y al resto de nodos o procesadores como nodos Esclavos (*Slaves*). En este modelo solo existe comunicación entre el nodo maestro y los esclavos,

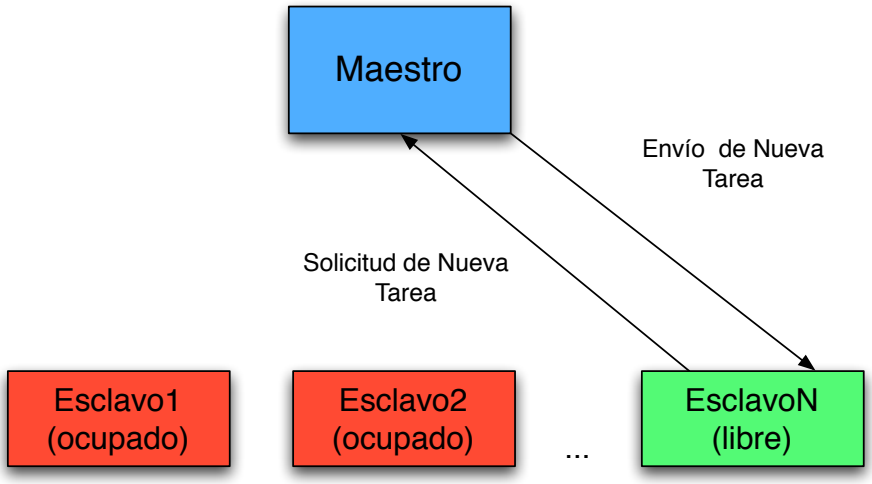

Figura 5.1: Modelo Maestro-Esclavo.

no existe comunicación esclavo-esclavo. En un primer paso, el nodo maestro calcula el número de iteraciones a procesar por un nodo determinado, y envia dichas iteraciones a un nodo esclavo libre. El resto del tiempo se queda en espera de recibir peticiones por parte del resto de nodos esclavos. Cuando un nodo esclavo se queda libre, notifica al nodo maestro su estado y se queda a la espera de recibir nuevas iteraciones. La figura 5.1 muestra uno de los posibles escenarios de este modelo, en ella aparece un nodo maestro, y el proceso de comunicación cuando uno de los nodos esclavos está libre.

Por tanto el nodo o procesador maestro se encargará de obtener el tamaño del *chunk* y distribuirlo por el resto de los nodos esclavos. De esta manera, cada nodo esclavo ejecutará el número de iteraciones que le indique el nodo maestro, que vendrá determinado por el algoritmo de planificación que se utilice. Dicho algoritmo de planificación deberá minimizar el tiempo de planificación, así como conseguir un buen equilibro de carga computacional entre todos los nodos esclavos.

A continuación se definen las elementos matemáticos que se utilizarán a lo largo de los siguientes subsecciones : (i) I, número total de iteraciones a distribuir; (ii) C_i, número de iteraciones (tamaño del *chunk*) que se computará en la iésima etapa de planificación; (iii) R_i, número de iteraciones que quedan por computar; (iv) p, número de procesadores de los que se dispone. Por tanto como regla general para estos algoritmos, se pueden definir:

$$R_0 = I, \;\; C_i = f(R_{i-1}, p), \;\; R_i = R_{i-1} - Ci, \tag{5.1}$$

donde f es una función que varia dependiendo del algoritmo de planificación que se vaya a utilizar.

5.3.2. Algoritmo PSS (Pure Self-Scheduling)

Los algoritmos de planificación estáticos, surgieron como contrapartida a los algoritmos de planificación dinámicos, para resolver los problemas de paralelización en máquinas de memoria compartida. Los autores Tang y Yew propusieron en [TY86] nuevos algoritmos para distribuir las iteraciones en este tipo sistemas, entre ellos se encuentra el algoritmo PSS, en el que cada procesador libre se le asigna una única iteración. Es decir, el tamaño del *chunk* es siempre de una unidad. Si se ajusta la formula 5.1 a este algoritmo, se obtiene:

$$R_0 = I, \;\; C_i = 1, \;\; R_i = R_{i-1} - 1. \tag{5.2}$$

La principal característica de este algoritmo de planificación, consiste en que se mantiene un buen equilibrio de carga computacional en cada procesador, ya que todos ejecutan el mismo número de iteraciones. Sin enbargo, disminuye mucho el rendimiento total de la aplicación, debido a que cada procesador, sólo ejecuta una única iteración en cada fase de planificación.

5.3.3. Algoritmo CSS (Chunk Self-Scheduling)

Para aumentar el rendimiento del algoritmo PSS, Tang y Yew propusieron [TY86] este algoritmo en el que el tamaño del *chunk* es asignado por el usuario. Dependiendo de este tamaño el comportamiento del algoritmo difiere, a saber: (i) si $C_i = 1$, el algoritmo se convierte en el algoritmo planificación anterior (Pure Self-Scheduling); (ii) cuando $C_i = \left\lfloor \frac{I}{p} \right\rfloor$, se obtiene un algoritmo de planificación estático;(iii) en el caso de que $C_i >> \left\lfloor \frac{I}{p} \right\rfloor$, aumenta el rendimiento del sistema pero disminuye el equilibrio de carga computacional; (iv) finalmente si $C_i << \left\lfloor \frac{I}{p} \right\rfloor$, disminuye el rendimiento del sistema, y la carga computacional se equilibra. Adaptando la formula general 5.1 a este algoritmo se obtiene la siguiente formula:

$$C_i = k \;\; \text{con} \;\; k \geq 1 \;\; \text{y} \;\; R_i = R_{i-1} - k, \tag{5.3}$$

donde k es el tamaño del *chunk* o pedazo definido por el usuario.

5.3.4. Algoritmo GSS (Guided Self-Scheduling)

El algoritmo de planificación guiado, propuesto por Polychronopoulos y Kuck [PK87], debe su denominación a la utilización de una función matemática que le sirve como guía para calcular el tamaño del *chunk* en cada uno de los nodos. A diferencia de los algoritmos presentados en las secciones anteriores, el tamaño del *chunk* varía dependiendo del nodo y de la etapa de planificación. La ecuación 5.4 representa el comportamiento de este algoritmo.

$$C_i = \left\lceil \frac{R_{i-1}}{p} \right\rceil \quad \text{con} \quad R_i = R_{i-1} - C_i. \tag{5.4}$$

En esta ecuación se puede observar que la asignación del tamaño del *chunk* se realiza de manera decreciente, es decir durante las primeras fases de la planificación la longitud del *chunk* en cada nodo será mayor que en el resto de fases. De esta manera se reduce el coste de planificación y se obtiene un buen equilibro de carga. Sin embargo, debido a la naturaleza del algoritmo, existe un tamaño mínimo, correspondiente a una iteración, que se repite al menos $(p-1)$ veces, siendo p el número de procesadores. Este es uno de los mayores inconvenientes del algoritmo GSS, ya que si existen muchas fases de planificación en las que el tamaño del *chunk* es de una unidad, se produce un incremento en el coste de dicha planificación, disminuyendo de esta manera el rendimiento total de la aplicación.

Como solución al problema anterior, Polychronopoulos y Kuck propusieron en [PK87], una variante del algoritmo denominada GSS(k). Donde k es el tamaño de *chunk* mínimo definido por el usuario. De esta manera el algoritmo GSS es un caso particular del GSS(k), sustituyendo k por el valor de una unidad (GSS(1)). Esta solución aunque es efectiva, a la larga produce los mismos efectos que el algoritmo CSS, ya que el valor de k tiene que ser elegido previamente a la ejecución del programa y por lo tanto no puede variar dinámicamente.

5.3.5. Algoritmo TSS (Trapezoid Self-Scheduling)

Los autores Tzen y Ni [TN93] propusieron este algoritmo basándose en los algoritmos de planificación vistos anteriormente. La idea general del algoritmo TSS, consiste en desarrollar una función capaz de obtener las ventajas de los algoritmos CSS(k) y

GSS(k), y al mismo tiempo, resolver todos sus inconvenientes. La principal ventaja del algoritmo CSS(k), consiste en que sigue una esquema lineal, y debido a ello, si el valor de k está ajustado, el algoritmo ofrece un coste de planificación bastante óptimo. Por el contrario, su principal desventaja radica en el desequilibrio de carga entre los procesadores existentes. El algoritmo GSS(k), sin embargo, mantiene un buen equilibrio de carga a costa de un incremento en el tiempo de planificación, debido principalmente a que dicha planificación se realiza siguiendo una función decreciente no lineal.

El algoritmo TSS(F,L), los parámetros F y L provienen de los términos *first* (primero) y *last* (último) respectivamente, pretende mantener un equilibrio entre el coste de planificación y la carga computacional en cada nodo. Para ello asigna un tamaño de *chunk* grande durante las primeras fases de planificación, que va decrementandose linealmente en el resto de fases, hasta alcanzar el número de iteraciones totales. La ecuación 5.7 representa el comportamiento del algoritmo TSS(F,L) basándose en los parámetros explicados en la función 5.1:

$$C_i = C_{i-1} - D \;\text{ con }\; D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor \;\text{ y }\; N = \left\lceil \frac{2I}{F+L} \right\rceil \;\text{ donde }\; C_0 = F. \tag{5.5}$$

Los parámetros F y L pueden ser definidos por el usuario o se pueden calcular mediante las siguientes funciones:

$$F = \frac{I}{2p} \;\text{ y }\; L = 1. \tag{5.6}$$

Debido a la naturaleza linealmente decreciente de este algoritmo, cuando no se definen los parámetros F y L, es decir F y L toman los valores que aparecen en la función 5.6, el tamaño del último *chunk* puede variar entre 1 y $\frac{I}{2P}$.

En la fórmula 5.7 aparecen dos nuevos términos, D y N. El primero de ellos (D) se refiere al decremento del tamaño del *chunk*, es decir al número de iteraciones que se irán decrementando linealmente en cada fase de la planificación, como se puede observar es un valor fijo. Del mismo modo, el término N se refiere al número de fases en las que se va a dividir la planificación. Como muestra dicha fórmula, en este algoritmo el número de iteraciones restantes (R_i) no interviene en el cálculo del tamaño del *chunk*, pudiéndose calcular fácilmente a través de la siguiente formula:

$$R_i = R_{i-1} - C_i \;\text{ donde }\; R_0 = I. \tag{5.7}$$

Tal y como sus autores comentan [TN93], este algoritmo logra obtener un buen

equilibrio de carga computacional entre los nodos, sin que se produzca un incremento en el tiempo total de planificación. Sin embargo, la principal desventaja de este algoritmo reside en la importancia del parámetro L ya que dependiendo de su valor el algoritmo puede desequilibrarse, llegando a resultar ineficiente.

5.3.6. Algoritmo FSS (Factoring Self-Scheduling)

El algoritmo FSS propuesto por S. Flynn, E. Schonberg y L. E. Flynn [HSF92] incorpora la novedad de agrupar distintas fases de planificación en una misma etapa de planificación. Se define etapa de planificación aquella en la cual el tamaño del *chunk* a distribuir es el mismo en todos los procesadores. Cuando un procesador termina su ejecución, pasa a la siguiente etapa. De este modo, se distribuye equitativamente la mitad de las iteraciones restantes entre todos los procesadores, logrando que todos ellos en un instante de tiempo determinado, ejecuten el mismo número de iteraciones y equilibrando de esta manera la carga computacional. La siguiente formula expresa el comportamiento de este algoritmo:

$$C_i = \left\lceil \frac{R_{i-1}}{\alpha p} \right\rceil \quad \text{con} \quad R_i = R_{i-1} - pC_i. \tag{5.8}$$

En la fórmula anterior el término α hace referencia al parámetro óptimo que se obtiene a través de una distribución probabilistica del problema, o es su defecto se toma como $\alpha = 2$, para obtener una distribución de la carga lo más equilibrada posible.

La principal ventaja que aporta este algoritmo con respecto a los anteriores, consiste en el buen equilibro de carga computacional que se puede obtener, ya que la planificación se puede adaptar al problema ajustando el valor de α, consiguiendo de esta manera distribuir equitativamente el tiempo de ejecución de cada *chunk*. La principal desventaja radica, precisamente, en la optimización del parámetro α, ya que un valor desajustado puede desequilibrar la carga computacional. Además, en el caso de que existan pocas iteraciones, el tiempo de planificación se incrementa, si se compara con otros algoritmos de planificación como por ejemplo el algoritmo CSS.

5.3.7. Algoritmo FISS (Fixed Increase Self-Scheduling)

El autor Teebu Philip [PD97] propuso el algoritmo FISS el cual agrupa distintas fases de planificación en etapas, al igual que algoritmo FSS. Cada etapa, como se

ha comentado en la sección 5.3.6, está compuesta por un número determinado de fases de planificación, que se distribuyen equitativamente entre todos los procesadores del sistema. A diferencia de todos los algoritmos anteriores el algoritmo FSS, es un algoritmo creciente, es decir en las primeras etapas de planificación el tamaño del *chunk* será menor que en el resto. Para obtener el tamaño del *chunk* en cada etapa de planificación el algoritmo sigue la siguiente fórmula:

$$C_i = C_{i-1} + B \;\text{ con }\; C_0 = \left\lfloor \frac{I}{Xp} \right\rfloor \;\text{ y }\; B = \left\lceil \frac{2I(1-\sigma/X)}{p\sigma(\sigma-1)} \right\rceil \;\text{ y }\; X = \sigma + 2 \qquad (5.9)$$

De la anterior formula es importante destacar tres nuevos términos B, X y σ. El parámetro σ hace referencia al número de etapas que va a contener el proceso de planificación, este valor se ajusta por el usuario en tiempo de compilación, y según el autor [PD97] no debe ser mayor que cuatro para minimizar la comunicación entre los procesos. El término X es un valor constante, definido también por el usuario, y necesario tanto para obtener el tamaño inicial del *chunk* (C_0), como el valor del parámetro B, tal y como se muestra en la función 5.10. El autor recomienda en [PD97] que $X = \sigma+2$. Finalmente el valor B hace referencia al incremento del *chunk* (*bump*), como en el resto de casos anteriores es un valor constante, que depende de los parámetros X y σ.

De la misma forma que el algoritmo FSS, este algoritmo tiene la ventaja de obtener un buen equilibrio de carga entre los procesadores, además de reducir la comunicación entre ellos cuando el tamaño del *chunk* es pequeño. En cambio, la principal desventaja consiste en la dificultad de obtener el mejor ajuste para los parámetros X y σ, que además se tiene que determinar en tiempo de compilación.

5.3.8. Algoritmo TFSS (Trapezoid Factoring Self-Scheduling)

Este nuevo algoritmo de planificación propuesto por los autores A.T. Chronopoulos y R. Andonie en [CBGA01], realiza la planificación basándose en los algoritmos TSS (sección 5.3.5) y FSS (sección 5.3.6). La principal semejanza con el algoritmo propuesto por S.Flynn et al. en [HSF92], consite en que el algoritmo TFSS también divide el proceso de planificación en distintas etapas, cada una de las cuales contiene un número de p fases con el mismo tamaño de *chunk* . La principal diferencia radica en el cálculo del tamaño del *chunk* en cada una de las etapas. El tamaño del *chunk* se obtiene como la suma las siguientes p fases de planificación del algoritmo TSS. Posteriormente este tamaño es dividido equitativamente entre los p procesadores, por tanto la formula del

cálculo del *chunk* se puede expresar como:

$$C_i = \left\lfloor \frac{C_i^{FSS}}{p} \right\rfloor \quad \text{con} \quad C_i^{FSS} = \sum_{j=k}^{k+p} C_j^{TSS} \quad \text{y} \quad k = ip. \tag{5.10}$$

La siguiente tabla muestra un ejemplo del proceso de planificación dependiendo del algoritmo utilizado, para un número de iteraciones $I = 2000$ y un número de procesadores $p = 4$.

Algoritmo	Tamaño del *chunk*
PSS	1 1 1 1 1 1 1 1 1 1 ...
CSS	k k k k k k k k k k ...
GSS	500 375 282 211 158 119 89 67 50 38 28 21 16 12 9 7 5 4 3 2 1 1 1 1
GSS(10)	500 375 282 211 158 119 89 67 50 38 28 21 16 12 10 10 10 10
TSS	250 234 218 202 186 170 154 138 122 106 90 74 58
TSS(200, 50)	200 190 180 170 160 150 140 130 120 110 100 90 80 70 60 50
FSS	250 250 250 250 125 125 125 125 63 63 63 63 31 31 31 31 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1
FISS	100 100 100 100 167 167 167 234 234 234 234
TFSS	226 226 226 226 162 162 162 162 98 98 98 98 58

Tabla 5.1: Ejemplo del cálculo del tamaño del *chunk* con 4 procesadores.

Los algoritmos de planificación dinámicos, ofrecen soluciones alternativas a los principales problemas de distribución generados por el uso de algoritmos de planificación estáticos. Sin embargo, estas soluciones están pensadas para la distribución de bucles en máquinas de memoria compartida. Y por tanto, se vuelven ineficientes cuando se trasladan a entornos distribuidos.

5.4. Algoritmos Dinámicos Distribuidos

Los algoritmos de planificación distribuidos, surgieron para dar soporte a los problemas de planificación de iteraciones en entornos distribuidos. Como se ha comentado en las secciones previas, los algoritmos de planificación dinámicos clásicos, no se ajustan a la problemática surgida con la aparición de los sistemas distribuidos, ya que en estos sistemas, cada nodo difiere del otro en elementos como la velocidad del procesador, la cantidad de memoria disponible o la velocidad de conexión entre los nodos. Por lo

tanto, es necesario tener en cuenta estos elementos durante el proceso de planificación. De esta manera, Chronopoulos et al. proponen en [CBGA01] un conjunto de algoritmos distribuidos compuesto por las versiones distribuidas de los algoritmos de planificación dinámicos clásicos. En estos nuevos algoritmos de planificación el conjunto de iteraciones se distribuye dependiendo de la velocidad de procesamiento de cada CPU del cluster, así como, de la carga actual del nodo esclavo donde se realiza la ejecución de las iteraciones. Para llevar acabo esta tarea, es necesario incluir en los algoritmos un nuevo conjunto de parámetros:

- Se define V_i como la capacidad virtual del procesador P_i. Por ejemplo, para el procesador más lento $V_i = 1$.
- El parámetro V es la capacidad virtual total del cluster. Se define como: $V = \sum_{i=1}^{p} V_i$.
- El parámetro Q_i identifica el número de procesos que se encuentran en la cola de ejecución del procesador P_i, por tanto refleja la carga total del procesador P_i.
- Se define $Ai = \left\lfloor \frac{V_i}{Q_i} \right\rfloor$ como la capacidad disponible del procesador P_i.
- Finalmente, $A = \sum_{i=1}^{p} A_i$ es la capacidad total disponible del cluster.

Tomando como referencia estos parámetros Chronopoulos et al. proponen un conjunto de algoritmos de planificación distribuidos basados en algoritmos de planificación dinámicos. El más importante de estos algoritmos es el denominado algoritmo de planificación distribuido trapezoidal (*Distributed Trapezoid SelfScheduler*, DTSS) que adapta el modelo maestro-esclavo, utilizado por los algoritmos de planificación dinámicos, al comportamiento de un sistema distribuido. A continuación se detallan los elementos más importantes de este nuevo modelo.

- **Nodo Maestro:**

  1. *a*) Espera por todos los esclavos con $A_i > 0$ a que estos envíen su capacidad disponible. Después ordena los A_i en orden decreciente y los almacena en un array. Almacena las peticiones en orden de llegada y calcula A.
     b) Usa $p = A$ para obtener los principales parámetros del algoritmo TSS.
  2. *a*) Mientras tenga iteraciones sin asignar y reciba peticiones de los esclavos, almacena la petición y actualiza el valor de A_i.

b) Obtiene una petición almacenada y le asigna un nuevo *chunk* $C_i = A_i * (F - D * (S_{i-1} + (A_i - 1/2))$, donde $S_{i-1} = \sum_{k=1}^{k=i-1} A_k$.

c) Si más de la mitad de los valores A_i almacenados han cambiado desde la ultima vez, actualiza el array y vuelve al paso 1, con el número total de iteraciones igual al número de iteraciones que quedan por planificar.

- **Nodo Esclavo:**

1. Obtiene el número de procesos en encolados Q_i y recalcula A_i. Si Ai es mayor que 0 va al paso dos, en caso contrario se queda en este paso.
2. Envia la petición al nodo maestro.
3. Espera por la respuesta del nodo maestro: Si llegan más tareas vuelve al paso 1 sí no, termina.

Este algoritmo obtiene el tamaño del *chunk* realizando los mismos cálculos que el algoritmo TSS explicado en la sección 5.3.5, pero en lugar de utilizar el parámetro p para indicar el número de procesadores utiliza el parámetro A. Los autores Chronopoulos et al. han utilizado este mismo modelo para realizar versiones distribuidas de los algoritmos FSS, FISS, y TFSS. Como se explicó en la sección 5.3 estos algoritmos dividen el proceso de planificación en etapas. Debido a ello, la versión distribuida de estos algoritmos debe modificar los apartados 1.a y 2.b del modelo maestro-esclavo distribuido. Además incorpora dos nuevos parámetros, a saber: SC_k es el sumatorio de los tamaños de los *chunks* en la etapa k; $C_i^k = SC_k * (A_j/A)$ es el tamaño del *chunk* en la etapa k. A continuación se especifican las modificaciones que se realizan en cada uno de los algoritmos:

- *DFTSS*. Apartado 1.b igual que para el algoritmo DTSS, para el apartado 2.b se calculan $SC_k = \sum_{j=1}^{p} C_j^{TSS}$ y C_j^k.
- *DFSS*. No es necesario el apartado 1.b, para el apartado 2.b se calculan $SC_k = \left\lfloor \frac{2Ri-1}{A} \right\rfloor$ y C_j^k.
- *DFISS*. En el apartado 1.b se calculan $SC_0 = \left\lfloor \frac{I}{X} \right\rfloor$ y $B = \left\lceil \frac{2I(1-\sigma/X)}{\sigma(\sigma-1)} \right\rceil$. En el apartado 2.b se calculan $SC_k = SC_{k-1} + B$ y C_j^k.

Aunque estos algoritmos se adaptan mejor a entornos distribuidos, ya que mejoran el equilibro de carga computacional respecto a los algoritmos dinámicos, son difícilmente aplicables a un entorno tan heterogéneo y dinámico como el Grid. En un entorno

Grid es muy difícil saber de antemano el número de procesadores asignados a una determinada aplicación así como el estado de los mismos. Además, en un entono Grid no solo es importante la velocidad de la CPU, sino también, otros factores como la red de interconexión entre los distintos nodos o el grado de saturación de estos. Así mismo, la mayoría de las implementaciones de estos algoritmos se realizan utilizando el paradigma de programación MPI, con todas las desventajas que conlleva el uso de este modelo de programación. En la siguiente sección se presenta un nuevo algoritmo de planificación dinámico y distribuido que permite realizar una planificación de iteraciones adaptada a un entorno Grid heterogéneo. De esta manera, el nuevo algoritmo aprovecha todas las características del modelo DRMAA-Grid *W*ay y permite una planificación de iteraciones totalmente trasparente desde el punto de vista del desarrollador. Además, este algoritmo permite una planificación eficiente y dinámica, manteniendo un buen equilibrio de carga entre los diferentes nodos que componen el Grid y reduciendo el tiempo de ejecución de la aplicación.

5.5. Grid Trapezoid Self-Scheduler (GTSS)

A lo largo de este capítulo se han presentado distintas técnicas de planificación de iteraciones. Se ha comenzado con lo algoritmos de planificación estáticos, que resuelven el problema de planificación, pero para ello, se debe conocer en tiempo de compilación el número total de iteraciones así como el número total de procesadores disponibles. Para resolver estas carencias, aparecieron los algoritmos de planificación dinámicos. Estos algoritmos obtienen el tamaño del *chunk* en tiempo de ejecución, pero no son eficientes en un entorno heterogéneo y distribuido como el Grid, ya que estos algoritmos están diseñados para su ejecución en máquinas de memoria compartida. Finalmente se presentó un conjunto de algoritmos dinámicos distribuidos, que adaptan los algoritmos dinámicos clásicos a un entorno distribuido y heterogéneo. Estos algoritmos basan su proceso de planificación en la velocidad de los procesadores y en la capacidad disponible de estos. Esta solución ofrece varias desventajas, ya que la obtención de la información necesaria incrementa el tiempo dedicado a realizar la planificación, y además, en un entorno Grid distribuido y heterogéneo interviene muchos más factores como la velocidad de interconexión de los nodos o el grado de saturación de los nodos esclavos. También es importante destacar que estos algoritmos están diseñados para clusters y por lo tanto no tienen en cuenta las características intrínsecas de un Grid

computacional (alto grado de heterogeneidad, alta tasa de fallos o disponibilidad de recursos dinámicas), para realizar la planificación.

Por todo lo expuesto, se ve en la necesidad de obtener un algoritmo de planificación que se adapte a un entorno Grid computacional y heterogéneo, y que además tenga en cuenta las características expuestas anteriormente. Este nuevo algoritmo denominado Grid Trapezoid Self-Scheduler (GTSS) modifica el el algoritmo TSS para adaptarlo a un entorno Grid. Básicamente el algoritmo GTSS se basa en tres elementos fundamentales: (i) Un modelo benchmark de Grid; (ii) El factor de relación R_i^{min}; y (iii) El algoritmo dinámico TSS. A continuación se explican cada uno de los elementos que componen el algoritmo GTSS, exceptuando el algoritmo TSS que se explicó en la sección 5.3.5, para en el último apartado de esta sección describir exhaustivamente el comportamiento del nuevo algoritmo de planificación.

5.5.1. Modelo Grid

Con el objetivo de plantear un modelo que evalúe el comportamiento de un Grid en aplicaciones de alta productividad, los autores R.S. Montero et al. proponen en [MHL06] un conjunto de métricas y metodologías necesarias para modelar dicho comportamiento. De entre todas estas técnicas se destacan dos, la caracterización de la carga y las metodologías bechmarks.

Caracterización de la Carga Computacional

Si se toma como referencia una aplicación de alta productividad, en inglés *High Throughput Computing (HTC) Application*, que determine un conjunto de tareas independientes, se puede considerar a un sistema Grid como un vector de procesadores heterogéneos. De esta manera, el número de trabajos ejecutados en un instante de tiempo se puede expresar como:

$$\sum_{i \in G} N_i \left\lfloor \frac{t}{T_i} \right\rfloor, \tag{5.11}$$

donde N_i es el número de procesadores en el Grid (G) que ejecuta una tarea en T_i segundos, incluyendo el tiempo de transferencia de los ficheros así como tiempos de latencia de la red.

Por otro lado, los autores Hockney y Jesshope proponen en [HJ83] un método para caracterizar el rendimiento de un sistema homogéneo. Aplicando este modelo, un

sistema Grid se puede representar a través de la siguiente función lineal:

$$n(t) = mt + b, \tag{5.12}$$

donde n es el número de trabajos ejecutados en el instante de tiempo t. Si utilizamos los parámetros definidos por Hockney y Jesshope obtenemos la siguiente fórmula;

$$n(t) = r_\infty t - n_{1/2} \text{ con } m = r_\infty \text{ y } b = -n_{1/2}. \tag{5.13}$$

Estos parámetros se denominan:

- Rendimiento Asintótico(r_∞): máxima tasa de rendimiento en tareas ejecutadas por segundo. En el caso de un sistema homogéneo con N procesadores y un tiempo de ejecución por tarea T, se obtiene $r_\infty = N/T$.

- Longitud de mitad-rendimiento($n_{1/2}$): el número de tareas necesarias para obtener la mitad del rendimiento asintótico. Este parámetro también mide el paralelismo del sistema desde el punto de vista de la aplicación. En el caso homogéneo, se obtiene $n_{1/2} = N/2$.

La ecuación 5.13 se puede interpretar como una representación ideal de un Grid, equivalente a un sistema homogéneo de $2n_{1/2}$ procesadores con un tiempo de ejecución por tarea de $2n_{1/2}/r_\infty$.
La longitud de mitad-rendimiento ($n_{1/2}$), permite obtener la heterogeneidad de un sistema Grid. Esto es debido, a que los nodos más rápidos contribuyen en mayor grado al rendimiento total del sistema. Además, se puede definir como $2n_{1/2}$ al número aparente de procesadores de un Grid, desde el punto de vista de la aplicación. Este número, en general, será menor que él número total de procesadores del Grid (N). Por tanto, se puede definir el grado de heterogeneidad como:

$$v = \frac{2n_{1/2}}{N}. \tag{5.14}$$

Este parámetro varía desde $v = 1$ en el caso homogéneo hasta $v \approx 0$ cuando el número de procesadores es mucho mayor que el número de procesadores aparente, es decir, existe un alto grado de heterogeneidad.

Metodologías Benchmark

En la sección anterior, se ha descrito un modelo para obtener caracterización del rendimiento de un sistema Grid en la ejecución de aplicaciones de alta productividad. Concretamente se han definidos los parámetros $n_{1/2}$ y r_∞. Estos parámetros se pueden obtener a través de dos métodos:

- Método Benchmark intrusivo. Los parámetros del sistema se obtienen a través de un ajuste lineal de los resultados obtenidos durante la ejecución de aplicaciones de alta productividad. Esté método permite obtener empíricamente los parámetros $n_{1/2}$ y r_∞ muy ajustados al entorno real.

- Método Benchmark no-intrusivo. En general, este método no es tan eficiente como el método anterior, ya que los parámetros no se obtienen de manera empírica sino a través de la formula 5.11, asumiendo que el tiempo medio de ejecución por tarea es T_i. El parámetro T_i se puede representar como:

$$T_i = T_i^{xfr} + T_i^{exe} + T_i^{sch}, \tag{5.15}$$

donde T_i^{xfr} y T_i^{exe} es el tiempo de transferencia y el tiempo de ejecución en el nodo i, incluyendo los tiempos de cola locales. El parámetro T_i^{sch} representa el tiempo utilizado en la planificación, es decir, el tiempo dedicado a seleccionar el recurso donde se va a ejecutar la tarea. Si se sustituye en la fórmula 5.11 T_i por su definición en la fórmula 5.17 se obtiene:

$$n(t) = \sum_{i \in G} N_i \left\lfloor \frac{t}{T_i^{xfr} + T_i^{exe} + T_i^{sch}} \right\rfloor. \tag{5.16}$$

Realizando un ajuste lineal de la fórmula anterior, se obtienen los valores $n_{1/2}$ y r_∞ de manera no-intrusiva.

Por todo lo expuesto, el algoritmo GTSS calculará los valores $n_{1/2}$ y r_∞ para ajustar la planificación al estado del Grid. Y tomará el valor $2n_{1/2}$ para definir el número de procesadores equivalentes del Grid.

5.5.2. Factor de Relación R_i^{min}

En la actualidad no existe ningún algoritmo de planificación de iteraciones que se adapte al comportamiento del Grid computacional dinámico y heterogéneo. Con el objetivo de obtener la planificación más ajustada posible, se ha diseñado un nuevo

parámetro denominado, Factor de Relación (R_i^{min}). Este nuevo parámetro es un coeficiente que relaciona entre sí los tiempos totales de ejecución de todos los nodos de un Grid. Como tiempo de ejecución total se define:

$$T_i^{wall} = T_i^{xfr} + T_i^{exe}, \tag{5.17}$$

donde T_i^{xfr} y T_i^{exe}, se refieren, respectivamente, a los tiempos de transferencia de ficheros y de ejecución en el nodo i, incluyendo todos los tiempos intermedios.

Estos tiempos de relación se obtienen a través de las estadísticas de usuario que ofrece el metaplanificador Grid*Way*. Utilizando dichas estadísticas se obtienen los tiempos de ejecución totales de experimentos en los que existe la misma carga computacional en cada nodo del Grid. De esta manera, se obtiene una relación de los tiempos de ejecución en cada uno de los nodos. La siguiente fórmula define el cálculo del parámetro R_i^{min}:

$$R_i^{min} = \frac{\overline{T}_{min}}{\overline{T}_{wall}(i)}, \tag{5.18}$$

donde $\overline{T}_{min}(i)$ se refiere al mínimo tiempo medio de ejecución total, y $\overline{T}_{wall}(i)$ es el tiempo medio de ejecución total en el nodo i.

Por tanto, el factor R_i^{min} es el coeficiente entre el menor tiempo medio de ejecución, y el tiempo medio de ejecución en el nodo i. Así, si se obtiene un $R_i^{min} = 1$, entonces el nodo i es aquél con mejor tiempo medio de ejecución que el resto, y por lo tanto es a este nodo al que se le tiene que enviar un tamaño de *chunk* mayor que al resto de nodos del Grid. Por el contrario, si $R_i^{min} \approx 0$, el subindice i se refiere al nodo con mayor tiempo medio de ejecución, y por lo tanto es a este nodo donde se enviarán tareas con el mínimo tamaño de *chunk*.

5.5.3. Descripción del Algoritmo

Una vez presentados los elementos más importantes que definen el algoritmo GTSS, es decir: el algoritmo dinámico TSS, expuesto en la sección 5.3.5; el modelo benchmark de Grid, analizado en la sección 5.5.1; y el factor R_i^{min} descrito en la sección 5.5.2; se continúa con la descripción detallada del funcionamiento y comportamiento del algoritmo Grid Trapezoid Self-Scheduler.

Al igual que ocurría con los algoritmos FSS y FISS el algoritmo GTSS también divide el proceso de planificación en etapas. Durante cada etapa todos los nodos de

un mismo sitio ejecutan el mismo número de iteraciones, es decir, el tamaño del *chunk* se mantiene en cada etapa de planificación. Por lo tanto el tamaño del *chunk* difiere dependiendo de la etapa y el sitio donde se ejecute. A continuación, se define el comportamiento del algoritmo GTSS para un nodo j, utilizando la misma terminología que en la descripción de los algoritmos de planificación dinámicos:

$$C_0^j = \lceil F * R_j^{min} \rceil \text{ con } F = \frac{I}{4n_{1/2}} \text{ y } L = 1. \tag{5.19}$$

$$\begin{aligned} C_i^j &= C_{i-1}^j - D \text{ con } D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor \text{ y} \\ N &= \left\lceil \frac{2I}{F+L} \right\rceil. \end{aligned} \tag{5.20}$$

Es importante destacar, que los parámetros D, N, F y L son constantes durante todo el proceso de planificación, al igual que ocurría en el algoritmo TSS. De esta manera, se disminuye el tiempo planificación, ya que no es necesario volver a recalcular estos valores, como sucedía en los algoritmos de planificación distribuidos.
Tal y como muestra la ecuación 5.19 el *chunk* inicial en un determinado nodo se obtiene del producto entre el parámetro F y el factor de relación $R^{min}(j)$. De esta forma, se asegura que los nodos con tiempo de ejecución mayor computen un menor número de iteraciones, produciendose además, un aumento en el número de iteraciones en los nodos con tiempo menor de ejecución.
Por otro lado, también se ha modificado la manera de calcular el parámetro F, en el algoritmo GTSS el factor F se calcula teniendo en cuenta el número aparente de procesadores, obtenido al realizar el modelo bechmark de Grid. De este modo, se asegura un ajuste óptimo del tamaño del *chunk*.

La tabla 5.2 muestra el comportamiento de este algoritmo para un número de iteraciones $I = 20000$, un número de procesadores aparentes $2n_{1/2} = 25$, 5 sitios distintos y para un conjunto de factores de relación $R_{min}^A = 1$, $R_{min}^B = 0,85$, $R_{min}^C = 0,75$, $R_{min}^D = 0,63$ y $R_{min}^E = 0,48$. Como se puede observar, el tamaño del *chunk* varía dependiendo del sitio donde se vaya a ejecutar, adaptandose de esta manera a la velocidad de ejecución de los nodos que componen cada sitio Grid. En la tabla 5.2 los valores iniciales del *chunk* van desde las 800 iteraciones para el sitio A, hasta las 384 iteraciones para el sitio E. Por lo tanto, el algoritmo GTSS se adapta al comportamiento de cada

Sitio	Tamaño del *chunk*
A	800 800 800 800 800 784 784 784 784 784 ...
B	680 680 680 680 680 664 664 664 664 664 ...
C	600 600 600 600 600 584 584 584 584 584 ...
D	504 504 504 504 504 488 488 488 488 488 ...
E	384 384 384 384 384 368 368 368 368 368 ...

Tabla 5.2: Ejemplo del cálculo del tamaño del *chunk* con el algoritmo GTSS.

sitio, presentado una ajustado equilibro de carga computacional.
También se comprueba, que el tamaño del *chunk* varía de manera linealmente decreciente dependiendo de un valor contaste para todos los sitios Grid. De esta manera, se asegura obtener un tiempo de planificación óptimo sin que esté se vea reflejado en el tiempo total de ejecución.

Si se compara la tabla 5.2 con la tabla 5.1, se pueden observar las principales semejanzas y diferencias entre los algoritmos presentados. En primer lugar todos los algoritmos presentado en la tabla 5.1 son algoritmos no distribuidos, por lo que presentan el mismo número de iteraciones independiente del sitio donde se vayan a ejecutar. Por otro lado, existen semejanzas entre el algoritmo GTSS y los algoritmos FSS y FISS, ya que son algoritmos que dividen la planificación en etapas; y con el algoritmo TSS, ya que es linealmente decreciente. Por todo lo expuesto, es de suponer que la ejecución de experimentos en un entorno Grid será más eficiente cuando se utilice el algoritmo GTSS.

5.6. Conclusiones

La distribución de iteraciones es una de las técnicas más utilizadas para aprovechar las características intrínsecas de las aplicaciones y así reducir el tiempo total ejecución. Estas técnicas se han utilizando clásicamente, en sistemas de memoria compartida y *clusters* a través del uso de los algoritmos de planificación autoadaptativos dinámicos. Sin embargo, en la actualidad, no existe un algoritmo diseñado específicamente para un entorno Grid heterogéneo que aproveche las ventajas que ofrecen el uso de los metaplanificadores y de *middlewares* como Globus.

Este capítulo presenta un nuevo tipo de algoritmo de planificación de código para

entornos Grid, basado en Globus y el metaplanificador Grid *W*ay. Concretamente se ha especificado un nuevo algoritmo denominado GTSS basado en el algoritmo dinámico TSS que aprovecha todas las ventajas que incorpora la tecnología Grid para realizar una planificación de código en este tipo de entornos de manera eficiente. Para ello se basa en dos pilares fundamentales, un modelo Grid que permite la definición de un Grid a través de dos parámetros $n_{1/2}$ y r_∞, y el factor de relación R_i^{min}. Estos parámetros permiten realizar una planificación dinámica y eficiente optimizando además el equilibro de carga entre todos los nodos que componen el entorno Grid donde se esté trabajando. Además, el uso del modelo DRMAA-Grid *W*ay en el desarrollo de aplicaciones paralelas con este algoritmo permite desacoplar el proceso de planificación del programa, aportando transparencia desde el punto de vista del desarrollador de la aplicación.

El siguiente capítulo presenta los experimentos realizados para comprobar la funcionalidad y eficiencia de este nuevo algoritmo en la planificación de tareas en un entorno Grid heterogéneo. Para ello se realizarán diferentes experimentos tanto en un entorno real, como en un simulador Grid adaptado al metaplanificador Grid *W*ay. Por tanto, se describirá el conjunto de experimentos realizados, así como los resultados obtenidos de su ejecución, mostrando de esta manera el comportamiento del algoritmo GTSS frente a los algoritmos de planificación clásicos.

Capítulo 6

Análisis del Algoritmo GTSS

Este capítulo analiza los resultados obtenidos en distintos experimentos con el nuevo algoritmo de planificación Grid Trapezoid Self-Scheduler. El objetivo principal del capítulo es justificar el uso de dicho algoritmo frente al resto de algoritmos de planificación existentes en la actualidad. Para ello se realizarán diversos experimentos en dos plataformas diferentes. Por un lado se comprobará la eficiencia del algoritmo en un simulador de Grid adaptado al modelo de desarrollo DRMAA-Grid *W*ay, para después comprobar su efectividad en un entorno real Grid concretamente sobre la plataforma EGEE y la aplicación MaRaTra.

El capítulo comienza con una pequeña introducción que presenta las principales motivaciones para desarrollar un simulador Grid adaptado al modelo DRMAA-Grid *W*ay y su uso en el análisis del algoritmo GTSS. También se realizará una breve descripción de la aplicación MaRaTra y se analizarán las principales causas de su elección para la ejecución de los experimentos en un entorno real Grid. A continuación, se explicarán los principales simuladores de GRID que existen en la actualidad, realizando una descripción mas detallada del simulador GridSim, explicando su arquitectura y su modelo de aplicación.
La sección 6.3 describirá el simulador Grid *W*aySim y explicará sus principales aportaciones, incluyendo un análisis de los resultados preliminares de la ejecución de este nuevo simulador. La sección 6.4 presentará los experimentos y resultados obtenidos en la ejecución del algoritmo GTSS dentro del simulador Grid *W*aySim, se detallarán los experimentos realizados, así como el banco de pruebas utilizado, y se realizará un análisis exhaustivo de los resultados obtenidos.
Posteriormente, en la sección 6.5 se describirán los experimentos y resultados obtenidos

durante la ejecución del algoritmo en un entorno real. Concretamente se abordará la ejecución sobre el proyecto de fusión MaRaTra, se realizará una descripción de este proyecto, del banco de pruebas utilizado y de los resultados obtenidos. Finalmente, este capítulo concluirá con un apartado dedicado a las conclusiones obtenidas como resultado del análisis de los experimentos realizados.

6.1. Introducción

En el capítulo 3 se presentó un nuevo paradigma de programación que utiliza la potencia del metaplanificador Grid *W*ay y el API de programación DRMAA para el desarrollo de programas en entornos Grid denominado modelo DRMAA-Grid *W*ay. Este nuevo modelo de implementación permite al usuario realizar programas distribuidos relegando todas las tareas de planificación y administración de los recursos y procesos al metaplanificador Grid *W*ay. En el capítulo 4 se demostró la eficiencia y efectividad de este nuevo paradigma en el desarrollo y posterior ejecución de diferentes tipos de aplicaciones comunes dentro de los sistemas Grid. En ese mismo capítulo se presentó un nuevo tipo de algoritmo genético distribuido y adaptado al Grid denominado DRMAA GOGA, su implementación se basa en el modelo propuesto en el capitulo 4 y por tanto hereda todas sus características, haciendo de este un algoritmo distribuido y adaptativo a entornos Grid heterogéneos. Finalmente, en el capítulo 5, se propuso un nuevo algoritmo de distribución de iteraciones adaptado para su ejecución en sistemas Grid. Este nuevo algoritmo de planificación de iteraciones se basa en un modelo de Grid, en el factor de relación R_i^{min} y en el algoritmo de planificación dinámico TSS. Las principales ventajas que presenta este algoritmo respecto a los algoritmos de planificación clásicos radican en la adaptabilidad a un entorno Grid heterogéneo, así como un buen equilibro de carga entre los distintos nodos del Grid, y un tiempo de planificación despreciable en comparación con el tiempo total de ejecución de la aplicación.

Con el objetivo de demostrar dichas características, el presente capítulo muestra un conjunto de experimentos realizados bajo dos plataformas de investigación. Una de ellas corresponde con el simulador Grid *W*aySim que es una ampliación del simulador GridSim incorporando las funcionalidades del modelo DRMAA-Grid *W*ay. Y la otra corresponde con la plataforma Grid involucrada en el proyecto fusión de EGEE MaRaTra, donde se realizaron diferentes experimentos para comprobar la funcionalidad de este algoritmo en un entorno real.

Los simuladores Grid permiten al usuario diseñar una gran cantidad de infraestructuras Grid virtuales y por lo tanto son muy útiles como banco de pruebas para la ejecución de experimentos. El desarrollo de un simulador Grid adaptado al modelo DRMAA-Grid*W*ay se basa en dos objetivos principales. En primer lugar, hace posible la ejecución más rápida de experimentos que comprueben la funcionalidad de dicho modelo en un entorno Grid repetible. De esta manera se eliminan los efectos debidos al dinámismo del Grid permitiendo obtener un análisis más exhaustivo del comportamiento del modelo DRMAA-Grid*W*ay. Y por otro lado, permite la obtención de los resultados preliminares del algoritmo de planificación GTSS, que servirán como base para analizar la efectividad de dicho algoritmo en comparación con un entorno real de ejecución.

Dicho entorno real, corresponde con el proyecto de fusión de EGEE denominado MaRaTra. El proyecto MaRaTra (*Massive Ray Tracing*) es una aplicación de trazado de rayos masivo que calcula la trayectoria de un haz de microondas en un plasma. El objetivo de las pruebas con esta aplicación es la de aplicar el algoritmo GTSS para distribuir tareas, cada una de ellas compuesta por un conjunto de rayos, y así comprobar la funcionalidad del algoritmo en un entorno Grid real. Por lo tanto el capítulo se divide en dos partes fundamentales, la primera se encargará de analizar los resultados obtenidos en la ejecución de diversos experimentos utilizando el algoritmo GTSS en en simulador Grid*W*aySim. La segunda parte presentará los resultados obtenidos en la ejecución del algoritmo GTSS con el objetivo de distribuir las diferentes tareas que componen la aplicación MaRaTra.

6.2. Simuladores de Entornos GRID

Esta sección realiza una descripción de los principales simuladores Grid que existen en la actualidad, con el objetivo de justificar la elección del simulador GridSim como base para el desarrollo del simulador Grid*W*aySim.
En los últimos años han aparecido herramientas que permiten la simulación de entornos Grid cuyo principal objetivo es facilitar el estudio del comportamiento de ciertas infraestructuras Grid. Además, estos simuladores permiten al desarrollador diseñar infraestructuras Grid virtuales adaptadas a sus necesidades y hacen posible la creación de entornos Grid muy diversos que ofrecen la posibilidad de obtener un gran número de bancos de pruebas donde se pueden conseguir resultados fiables con un coste de tiempo

mínimo. Sin embargo existen muy pocas herramientas que implementen un simulador eficiente en un entorno Grid computacional, la más importantes son Bricks [TMA+99], MicroGrid [SLJ+00], SimGrid [Cas01] y GridSim [BM02].

6.2.1. Bricks

El simulador Bricks ha sido desarrollado en el Instituto de Tecnología de Tokio en Japón por Aitda Kento et al [TMA+99] y está constituido por un sistema de evaluación del rendimiento, desarrollado en Java, que permite analizar y comparar diversas estrategias de planificación en un entorno típico de computación global de alto rendimiento. De esta manera, Bricks puede simular el comportamiento de varios sistemas globales de computación, concretamente simula el comportamiento de las redes y los algoritmos de planificación de recursos de dichos sistemas, basándose en el modelo canónico de sistemas globales de computación de alto rendimiento propuesto por los mismos autores en [ATN+00]. Además, al estar desarrollado de forma modular, permite incorporar diferentes algoritmos de planificación, así como diversos componentes de sistemas de computación existentes a través de su interfaz externa que permiten simular otros entornos de ejecución.

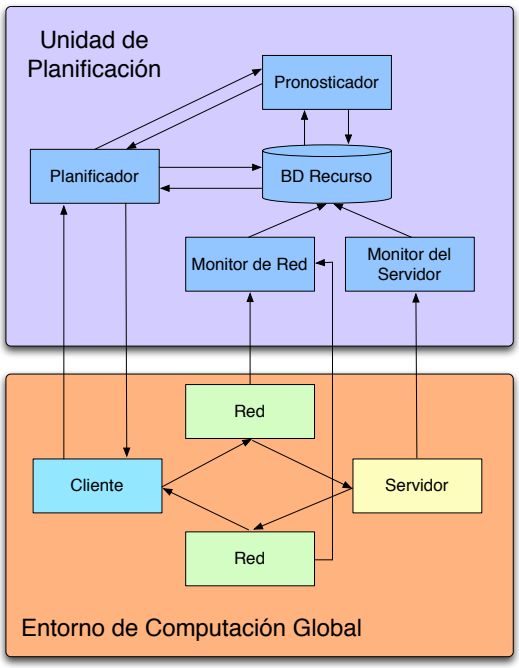

Figura 6.1: Arquitectura Bricks

La figura 6.1 representa un esquema de la arquitectura del simulador Bricks para el análisis del sistema NWS (*Network Weather Service*) [WSP97]. Como se puede observar, Bricks se compone de dos subsistemas, el primero se denomina Simulador de Entornos de Computación y está constituido por los siguientes módulos:

- **Cliente**. Representa la máquina del usuario que lanzará las tareas de computación global.
- **Red**. Representa la red de interconexión entre el cliente y el servidor.
- **Servidor**. Agrupa los recursos computacionales de un sistema de computación global determinado.

El otro subsistema se denomina Unidad de Planificación y está formado por los siguientes módulos:

- **Módulo Base de Datos del Recurso**. Está compuesto por una base de datos específica para el proceso de planificación, donde se almacenan los valores de medida que utilizarán el Pronosticador y el Planificador para realizar las tareas dedicadas a la predicción y decisión de planificación.
- **Monitor de Red**. Se encarga de medir el ancho de banda y la latencia de la red de un entorno de computación global que almacenará en el módulo Base de Datos del Recurso.
- **Monitor del Servidor**. Mide el rendimiento, la carga y la disponibilidad de las máquinas servidoras que también almacenará en el módulo Base de Datos del Recurso.
- **Pronosticador**. Se basa en la información almacenada en el módulo Base de Datos del Recurso, que realiza una predicción de los recursos disponibles para la planificación de nuevas tareas de ejecución
- **Planificador**. Envía las nuevas tareas invocadas por el cliente a la máquina servidora apropiada, basándose en la información almacenada en el módulo Base de Datos del Recurso y en el Pronosticador.

Estos subsistemas permiten la simulación del comportamiento de varios algoritmos de planificación de recursos, de módulos de programación de planificadores, de topologías

de red formadas por clientes y servidores en sistemas globales de computación y de estrategias de procesamiento de redes y servidores.
Con el objetivo de obtener de manera sistemática información sobre los recursos de computación global para los algoritmos de planificación de recursos, Bricks contiene un conjunto de componentes que monitorizan los recursos del entorno global de computación permitiendo la monitorización, predicción y planificación de trabajos en la red simulada, además, estos componentes pueden ser reemplazados por otros desarrollados por el programador a través del SPI (*Service Provider Interface*) de la unidad de planificación de Bricks, permitiendo de esta manera la simulación de nuevos algoritmos de planificación.

Aunque Bricks presenta muchas facilidades para la simulación de ejecución de aplicaciones en Grid, sigue una metodología de planificación centralizada, cuando lo deseable es que cada aplicación desarrollada para su ejecución en el simulador pueda disponer de una planificación determinada, permitiendo al desarrollador tener control total del proceso de planificación, sin centrarse en aspectos concretos de la topología de red. Además, no ofrece un API lo suficientemente completa, para definir Grid heterogéneos de manera sencilla. El desarrollo del simulador Grid*W*aySim requiere de un API robusta, de tal manera que permita ejecutar diferentes algoritmos de planificación en bancos de prueba muy diferentes, soportar el diseño de un gran número de Grids heterogéneos y que todas estas tareas se puedan realizar fácilmente desde el punto de vista del desarrollador. Por todo ello, esta herramienta no ha sido la escogida para desarrollar el simulador Grid*W*aySim.

6.2.2. MicroGrid

El simulador MicroGrid desarrollado por H. J. Song et al [SLJ+00], basado en MPI es en realidad un emulador Grid, que permite la ejecución flexible de aplicaciones en entornos Grid heterogéneos y virtuales, y soporta aplicaciones desarrolladas utilizando el *middleware* Globus Toolkit. El conjunto de herramientas de emulación MicroGrid permite la ejecución controlada y repetible de experimentos, soportando algoritmos de gestión de recursos en una gran variedad de configuraciones Grid.

La figura 6.2 representa un esquema de la arquitectura del emulador MicroGrid. Cada uno de los elementos de la figura constituyen cada uno de los componentes que forman la arquitectura del sistema MicroGrid, estos son: Virtualización, Coordinación Global y Simulación de Recursos.

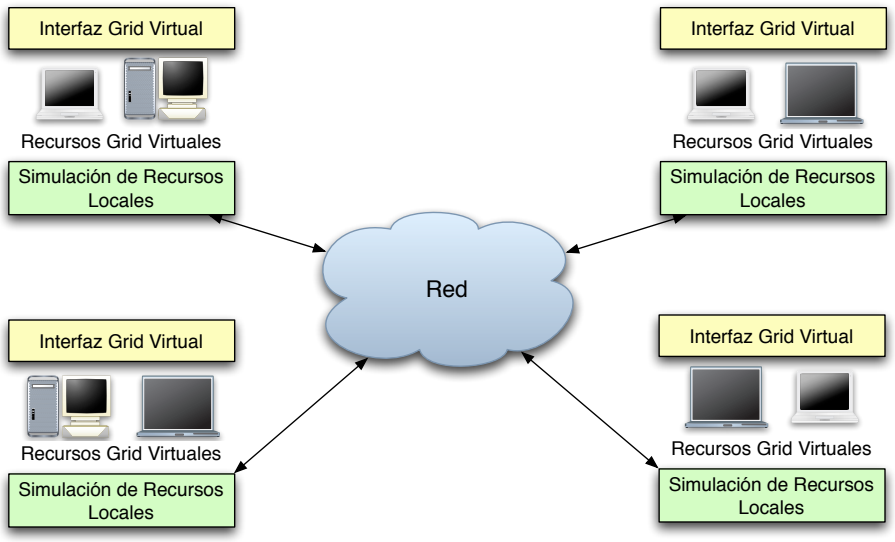

Figura 6.2: Arquitectura de MicroGrid

- **Virtualización**. El objetivo del proceso de virtualización consiste en engañar a la aplicación de tal manera que solo utilice los recursos virtuales, independientemente de los recursos físicos de los que se disponga. Para ello MicroGrid intercepta la comunicación entre la aplicación, los recursos y los servicios de información. De esta manera MicroGrid hace de intermediario entre la aplicación y los recursos, tanto para su uso como para obtener información sobre ellos. Para realizar dicha función es necesario virtualizar los recursos y los servicios de información. En MicroGrid cada recurso virtual es *mapeado* a un recurso físico a través de una tabla que relaciona cada dirección IP virtual con una dirección IP física, así todas las llamadas realizadas desde la aplicación son interceptadas por MicroGrid que utiliza esta tabla para enviarlas al recurso apropiado. Para realizar la virtualización de los servicios de información, MicroGrid extiende los registros del GIS LDAP con campos que contienen información específica de virtualización.

- **Coordinación Global**. Con el objetivo de conseguir una simulación equilibrada entre los distintos recursos es necesario realizar una coordinación global basada en las necesidades de los recursos virtuales y en la disponibilidad de los recursos físicos. De esta manera se controlan parámetros arquitectónicos como a capacidad de procesamiento o el ancho de banda y latencia de la red. Para ello MicroGrid utiliza un módulo de tiempo virtual encargado de determinar el rango máximo de simulación factible. Este parámetro indica la velocidad máxima a la que debe ir

un recurso para que no se produzcan errores, y por lo tanto, garantiza un análisis de rendimiento preciso de todos los procesos que se ejecutan en el simulador MicroGrid.

- **Simulación de Recursos**. Cada uno de los elementos que componen un recurso determinado (discos, procesadores o redes) debe ser simulado de manera efectiva para garantizar el correcto funcionamiento del emulador. Por ello MicroGrid separa la simulación de recursos en dos tipos, simulación de recursos computacionales y simulación de redes. Para la simulación de recursos computacionales MicroGrid se centra en modelar la velocidad de la CPU de los recursos virtuales. De esta manera, una tarea enviada a un recurso virtual puede que sea ejecutada en varios recursos físicos y viceversa, para ello utiliza un planificador local que ajusta la tarea MicroGrid a la capacidad de CPU de los recursos físicos. Para la simulación de redes MicroGrid utiliza el simulador VINT/NSE [Fal99] para configurar y crear las redes virtuales necesarias. Así, el simulador NSE se conectará con la infraestructura de comunicación virtual ejerciendo de intermediario entre la red virtual y el emulador MicroGrid.

MicroGrid permite la emulación de entornos Grid y la ejecución de aplicaciones desarrolladas en Globus. Sin embargo, al ser un emulador, la ejecución de aplicaciones se basa en los recursos físicos disponibles, ya que MicroGrid realiza traducciones de direcciones IP virtuales a direcciones físicas. Por lo tanto, MicroGrid no se adapta a las necesidades del simulador que se desea desarrollar, debido a que el simulador debe ser totalmente software, de tal manera, que su ejecución sea como cualquier otro programa secuencial. Además, MicroGrid no dispone de un API que permita al programador adaptar el emulador a sus necesidades, y por lo tanto impide el desarrollo de un simulador alternativo.

6.2.3. SimGrid

El autor Henri Casanova propone en [Cas01] un conjunto de herramientas, basadas en los lenguajes C y Java, que permiten el desarrollo de simuladores para el estudio de la planificación de aplicaciones en entornos distribuidos. Además, con SimGrid el usuario puede definir tareas con diferentes tiempo de ejecución y realizar la simulación de recursos basados en máquinas estándar. Las principales características del simulador SimGrid son: (i) Ofrece un modelo de programación y un nivel de abstracción

ajustado a las necesidades del desarrollador; (ii) Permite modelar y evaluar de manera eficiente algoritmos de planificación para sistemas distribuidos; y (iii) Hace posible una simulación más realista en comparación con los simuladores presentados previamente.

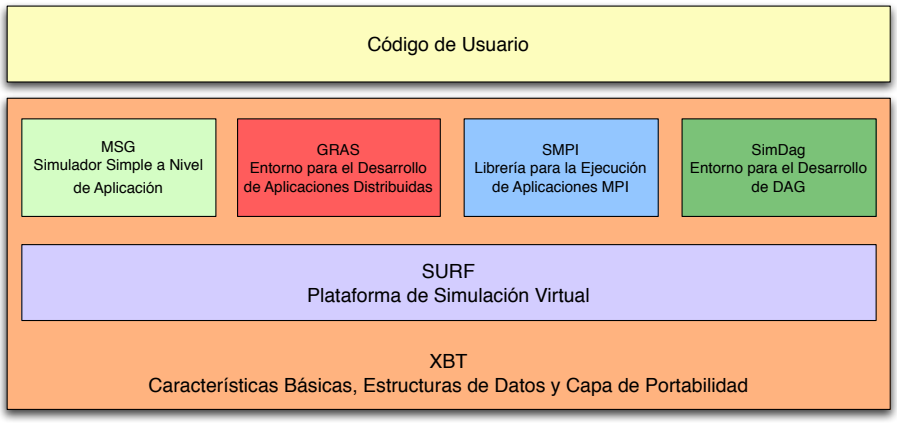

Figura 6.3: Arquitectura del Simulador SimGrid

La figura 6.3 muestra la arquitectura del simulador SimGrid siguiendo una estructura basada en capas. Básicamente, se compone de tres capas principales, la capa de entornos de programación, la capa de simulación y la capa base:

- **Capa de Entornos de Programación**. SimGrid permite la simulación de un conjunto de paradigmas y entornos de programación construidos sobre un único núcleo de simulación. SimGrid ofrece un conjunto de componentes especializados para cada tipo de paradigma de programación que se desee desarrollar. Entre estos componentes se pueden encontrar:

  – *MSG*. Constituye la primera herramienta realizada para el proyecto SimGrid y está compuesta por un conjunto de APIs para la simulación simple de aplicaciones. El objetivo de este componente consiste en el estudio de las heurísticas de un determinado problema, pero sin llegar a desarrollarlo, ya que las interfaces de programación que presenta no permite la implementación y simulación de problemas complejos.

  – *GRAS*. El componente GRAS (*Grid Reality And Simulation*) ofrece una interfaz de programación completa que permite el desarrollo de aplicaciones distribuidas en sistemas heterogéneos, basados en *sockets* y llamadas a procedimientos remotos. GRAS, además de simular las aplicaciones desa-

rrolladas a través de su interfaz, también permite la ejecución de dichas aplicaciones un entorno real, sin necesidad de modificarlas previamente.

  - *SMPI*. Permite la simulación de aplicaciones MPI ya desarrolladas, es decir no ofrece un interfaz para el desarrollo de este tipo de aplicaciones.
  - *SimDag*. Es el homólogo al entorno de programación MSG, pero con el objetivo de simular tareas cuya planificación se realiza siguiendo el modelo DAG (*Direct Aciclic Graphs*).

- **Capa de Simulación**. También denominada SURF constituye el núcleo de simulación de SimGrid y ofrece un conjunto de herramientas para la simulación de entornos virtuales. SURF está definido a muy bajo nivel, y por tanto no está pensado inicialmente para su uso directo por parte de los desarrolladores, sino como base para la simulación de los componentes que forman la capa de entornos de programación. Si alguno de estos componentes no se ajusta a las necesidades del desarrollador, este deberá utilizar SURF para implementar su propio simulador. Sin embargo, debido a la estructura tan compleja de SURF el desarrollo de un nuevo simulador requiere un gran esfuerzo por parte del desarrollador.

- **Capa Base**. La capa base está constituida por el módulo XBT (*eXtended Bundle of Tools*) y consiste en el conjunto de herramientas (tipos de datos y soporte de portabilidad) necesarias para el desarrollo de los componentes que se encuentran ubicados en las capas superiores.

Por tanto SimGrid ofrece un conjunto de herramientas para el desarrollo y simulación de aplicaciones distribuidas. Sin embargo, ninguno de los componentes de la capa de entornos de programación se adapta al modelo de desarrollo DRMAA-Grid *W*ay. Por lo tanto, sería necesario desarrollar el simulador programando directamente sobre SURF con la dificultad que este proceso conlleva ya que además, se tendría que desarrollar el simulador completamente.

6.2.4. GridSim

El simulador GridSim, propuesto por los autores Rajkumar Buyya y M. Manzur Murshed en [BM02], está constituido por un conjunto de herramientas basadas en el lenguaje de programación Java que permite diseñar y simular recursos Grid heterogéneos. GridSim hace posible la simulación de recursos con diferentes políticas de

ubicación, incluso permite al usuario el desarrollo de sus propias políticas de ubicación a través de su API en Java. De esta manera, el desarrollador puede simular un entorno Grid completo incluyendo diferentes, usuarios, recursos, aplicaciones, gestores de recursos y planificadores. Entre las principales características de GridSim se pueden encontrar: (i) Permite definir de manera exhaustiva cada uno de los elementos que forman el recurso como el número de procesadores y máquinas, la velocidad de la CPU, políticas de ubicación, zona horaria de localización o velocidad de red entre recursos; (ii) Puede simular aplicaciones heterogéneas cada una de ellas basadas en diferentes modelos de desarrollo; (iii) Permite la ejecución simultánea de tareas pertenecientes a múltiples usuarios en un mismo recurso, cuya ejecución vendrá definida por la política de ubicación asignada al recurso; (iv) Soporta la simulación de planificadores estáticos y dinámicos; (v) Permite obtener estadísticas de todas las operaciones realizadas en el proceso de simulación; y (vi) Ofrece una completa API en Java, para el desarrollo de simulaciones, que permite de manera sencilla la simulación de un entorno Grid complejo y heterogéneo, así como multitud de escenarios de ejecución.

Todas estas características hacen del simulador GridSim una herramienta completa para el desarrollo de simulaciones basadas en Grid heterogéneos. Por esta razón, se ha escogido a esta herramienta para el desarrollo del simulador Grid*W*aySim. Con el objetivo de presentar una visión más concreta del simulador GridSim, en las siguientes secciones se especifica su arquitectura y su modelo de aplicación.

Arquitectura GridSim

La arquitectura del simulador GridSim sigue una estructura basada en capas y módulos con el objetivo de facilitar la gestión de las tecnologías existentes como componentes individuales. La figura 6.4 representa dicha arquitectura junto con los módulos que constituyen el desarrollo del simulador. El primer nivel de la arquitectura corresponde con la máquina virtual de Java (*Java Virtual Machine*, JVM) cuya implementación se encuentra disponible tanto para sistemas monoprocesadores, como multiprocesadores y *clusters*. El segundo nivel corresponde con herramientas de simulación basadas en eventos que utiliza las interfaces ubicadas en el primer nivel para el desarrollo del simulador. GridSim utiliza uno de los simuladores más populares disponible en Java denominado SimJava [HM98] que además dispone de una versión distribuida. El tercer nivel agrupa todos los componentes que intervienen en el modelado y simulación de las entidades, recursos y servicios de información del simulador para el desarrollo de entidades de

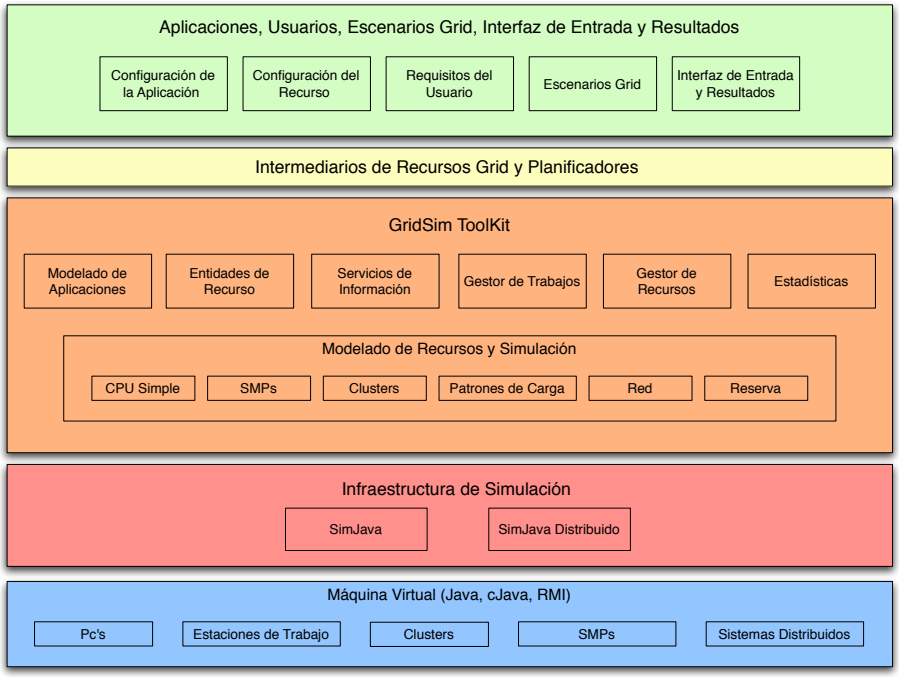

Figura 6.4: Arquitectura del Simulador GridSim

alto nivel. Para ello utiliza los servicios ofrecidos por el simulador SimJava. El cuarto nivel representa la simulación de gestores de recursos y planificadores. Finalmente, el quinto nivel representa el modelado de recursos y aplicaciones en diferentes escenarios utilizando los servicios ofrecidos por las dos capas inmediatamente inferiores, con el objetivo de evaluar las políticas, heurísticas y algoritmos de gestión de recursos y de planificación.

Un elemento fundamental de la arquitectura del simulador GridSim es el correspondiente a las entidades, que se utilizan para realizar tanto la simulación de recursos monoprocesadores, multiprocesadores y distribuidos como la simulación de la interconexión entre estos recursos a través de entidades de red. Durante el proceso de simulación, GridSim crea un determinado número de entidades multi-hilo que se ejecutarán en paralelo. Estas entidades corresponden con los usuarios, recursos, gestores de recursos, servicios de información, estadísticas y redes definidas por el desarrollador. Debido a la importancia de las entidades dentro de la arquitectura GridSim, a continuación se analizan por separado las entidades más importantes que la componen:

- **Usuario**. Cada una de las instancias de la entidad Usuario representa a un usuario Grid, que se diferencia en por ejemplo, el tipo de trabajos que crea, la estra-

tegia de planificación que sigue o la zona horaria donde se encuentren.

- **Intermediario de Recursos**. Cada uno de los usuarios creados se conecta con una instancia de la entidad *Resource Broker*. De esta manera, cada trabajo del usuario se envía en primer lugar al *resource broker* que se encargará de planificar dichos trabajos según la política de planificación definida por el usuario. Además, el *resource broker* obtiene de forma dinámica los recursos que se encuentran disponibles, antes de enviar los trabajos a ejecutar. La eficiencia de los algoritmos de planificación utilizados por el *resource broker* depende de las necesidades del desarrollador, ya que GridSim permite al programador implementar cualquier algoritmos de planificación de trabajos.

- **Recurso**. La entidad Recurso representa a un determinado recurso Grid, cada uno de los recursos difiere en número de procesadores, coste de procesamiento, velocidad de procesamiento o política de planificación interna (tiempo compartido o espacio compartido).

- **Servicio de Información Grid**. Esta entidad ofrece servicios de registro de recursos y se ocupa del mantenimiento de la lista de recursos disponibles en el Grid. De esta manera, el *resource broker* consulta esta entidad para obtener información acerca del estado o configuración de cada registro.

- **Entidades de Entrada y Salida**. Estas entidades son las encargadas de controlar todo el flujo de información que ocurre entre el resto de entidades del simulador GridSim. Este flujo de información es posible gracias a los puertos de entrada/salida que contienen cada una de las entidades Grid, y que se utilizan para establecer la conexión entre estas entidades y las de entrada y salida.

Modelo de Aplicación del Simulador GridSim

El simulador GridSim no define explícitamente ningún modelo de aplicación específico, de esta manera los desarrolladores serán los encargados de definir el modelo de aplicación que se adapte a sus necesidades. Esto hace de GridSim un simulador muy flexible ya que permite la simulación de multitud de aplicaciones.

En GridSim, cada una de las tareas independientes que intervienen en la ejecución del simulador varía tanto en tiempo de procesamiento como en tamaño de ficheros de

entrada y salida. Cada una de estas tareas se define a través de unos objetos denominados Gridlets, que contienen toda la información relacionada con el trabajo y todos los detalles referidos a la gestión de su ejecución. Además, las Gridlets almacenan otras características relacionadas con el trabajo en ejecución, como su tamaño expresado en MIPS, operaciones de entrada/salida o el tamaño de los ficheros de entrada y salida. Estos elementos permiten obtener de una manera sencilla el tiempo de ejecución de cada trabajo, así como el tiempo de trasferencia de los ficheros al recurso remoto donde se ejecuten. Finalmente, el conjunto de herramientas GridSim ofrece un conjunto de servicios y protocolos para la gestión de Gridlets que permiten el desarrollo de algoritmos de planificación adaptados a las necesidades del usuario.

Por lo tanto GridSim permite: (i) El desarrollo de multitud de aplicaciones con diferentes modelos de programación, debido a que no dispone de un modelo de aplicación propio; (ii) El diseño de entornos distribuidos como Grids heterogéneos, permitiendo la definición exhaustiva de cada uno de los componentes que constituyen dicho entorno, de esta manera se pueden simular un extenso conjunto de bancos de pruebas; (iii) La creación de diferentes políticas de ubicación que se adapten a las necesidades del desarrollador; (iv) La implementación de algoritmos de planificación de trabajos de manera sencillas; y (v) Facilidad en el desarrollo de un nuevo simulador, debido principalmente a su arquitectura basada en niveles y a una extensa API en lenguaje Java. Todas estas características permiten desarrollar un simulador que se adapta al modelo DRMAA-Grid *W*ay con una política de planificación de trabajos definida a través del algoritmo GTSS.

6.3. Simulador Grid *W*aySim

Esta sección presenta un simulador de entornos Grid adaptado al modelo de desarrollo de aplicaciones en Grid presentado en esta tesis denominado DRMAA-Grid *W*ay y basado en el simulador GridSim. El objetivo de esta sección es analizar el desarrollo, implementación y comportamiento de este nuevo simulador que se utilizará en la sección 6.4 para comprobar la funcionalidad del algoritmo de planificación GTSS. En primer lugar se muestra la estructura general del simulador, analizando exhaustivamente cada uno de los componentes que lo constituyen. A continuación se detalla su funcionalidad y se analiza su comportamiento en la ejecución del producto de dos matrices comparando sus resultados con los obtenidos en la ejecución del mismo experimento

en una plataforma Grid de investigación real.

6.3.1. Arquitectura del Simulador

La figura 6.5 representa un esquema de la arquitectura del simulador Grid *W*aySim. Como se puede observar, dicha arquitectura se basa en la arquitectura el simulador GridSim con la incorporación de nuevos módulos, marcados en rojo, que constituyen la adaptación de este simulador al modelo de desarrollo DRMAA-Grid *W*ay. Estos nuevos módulos representan a clases, o en su mayoría, a jerarquías de clases en Java, diseñadas para que el comportamiento de Grid *W*aySim sea lo más parecido posible al del modelo DRMAA-Grid *W*ay. A continuación se explican cada uno de los módulos desarrollados para la implementación del simulador Grid *W*aySim.

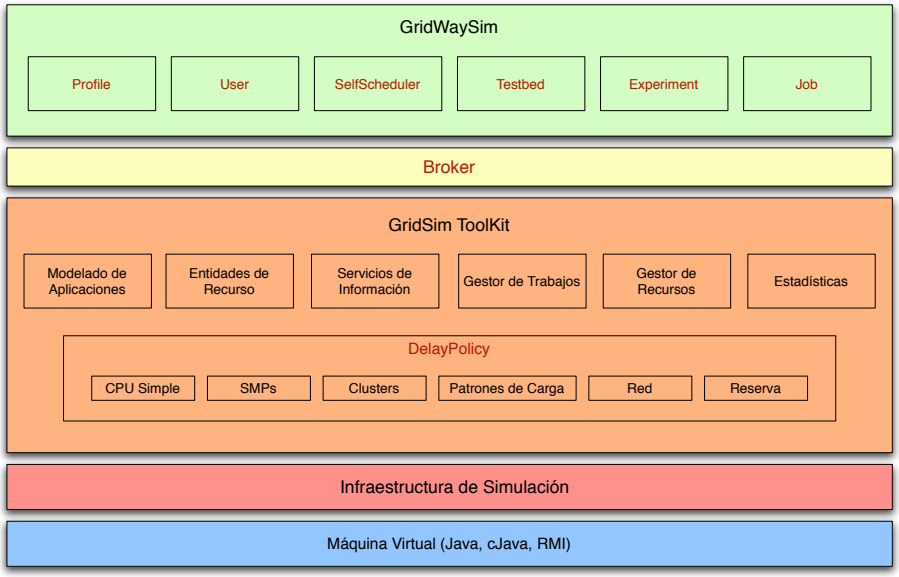

Figura 6.5: Arquitectura del Simulador GridWaySim

Módulo Profile

Gestiona la creación de diferentes tipos de usuario, esta distinción radica en las diferencias de implementación que constituye la aplicación de los algoritmos de planificación clásicos (CSS, GSS, TSS, FSS y FISS) respecto al algoritmo de planificación adaptado al Grid, GTSS. Debido a que el algoritmo GTSS es mucho más dinámico que los algoritmos de planificación clásicos, el comportamiento del usuario en cada caso

varia de manera significativa. Por lo tanto es necesario diferenciar a los usuarios que utilizan los algoritmos de planificación clásicos (*SelfSchedulerUser*) de aquellos que utilizan el algoritmo GTSS (*DistributedSelfchedulerUser*) y en consecuencia la gestión de la creación de dichos usuarios será distinta obteniendo de esta manera dos clases diferentes (*SelfSchedulerProfile* y *DistributedSelfchedulerProfile*). La figura 6.6 representa a esta jerarquía de clases.

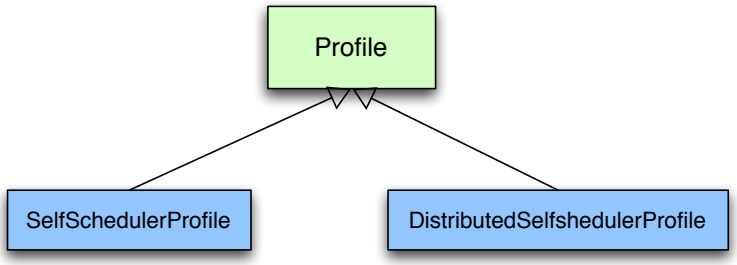

Figura 6.6: Jerarquía de clases Profile

Módulo User

El módulo User esta constituido por una jerarquía de clases formada por las clases *User*, *SelfSchedulerUser* y *DistributedSelfchedulerUser*, tal y como muestra la figura 6.7. Este módulo representa a cada uno de los usuarios que ejecutará los diferentes trabajos en el entorno Grid virtual. Debido a las diferencias de implementación de los algoritmos de planificación clásicos respecto al algoritmo GTSS, es necesario definir dos tipos distintos de usuarios (*SelfSchedulerUser* y *DistributedSelfchedulerUser*) Cada uno de los usuarios estará definido por un algoritmo de planificación, un número de trabajos, y un gestor de recursos (*DelayerBroker* para la clase *SelfSchedulerUser* y *DistributedDelayerBroker* para la clase *DistributedSelfchedulerUser*). El objetivo de cada usuario es, primer lugar, obtener el tamaño del *chunk* apropiado en cada uno de los casos, agrupar los trabajos en tareas dependiendo del tamaño del *chunk*, y enviar todas estas tareas al gestor de recursos apropiado. Básicamente estas entidades hacen una función muy semejante a la que haría el programa principal en el caso de su ejecución en un Grid real. Por ello, dentro del modelo DRMAA-Grid *W*ay estas entidades corresponden con el programa principal implementado con llamadas al API DRMAA.

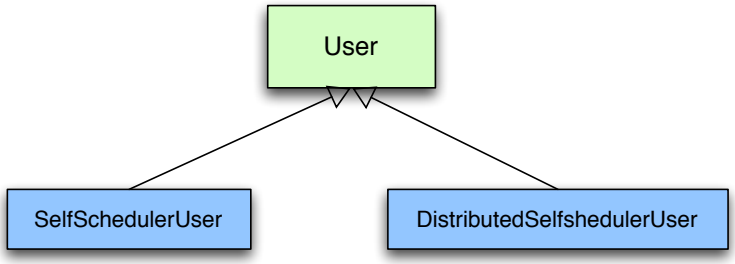

Figura 6.7: Jerarquía de clases User

Módulo Self-Scheduler

Esta constituido por una jerarquía de clases con el objetivo de incorporar los algoritmos de planificación dinámicos al simulador GridWaySim. Cada una de las clases representa un tipo de algoritmo de planificación dinámico clásico, de esta manera el usuario obtiene el tamaño del *chunk* apropiado que le permite agrupar los diferentes trabajos en tareas. Todas estas clases están relacionadas entre sí a través de la clase base *SelfScheduler*, la figura 6.8 muestra un esquema de esta jerarquía. Como se

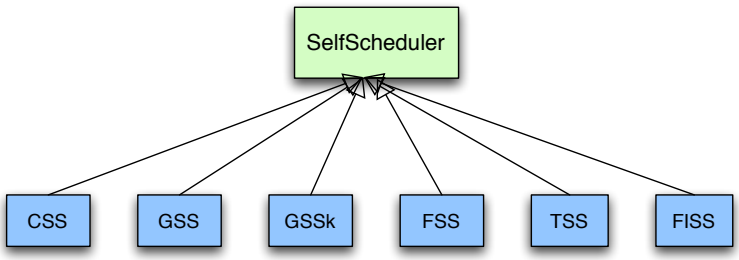

Figura 6.8: Jerarquía de clases SelfScheduler

puede observar, en esta jerarquía no se incluye el algoritmo de planificación GTSS ya que, debido a sus características de implementación, se incluye dentro de la clase *DistributedDelayerBrocker*.

Módulo Testbed

Este módulo esta formado por una jerarquía de clases que definen los diferentes bancos de pruebas que se utilizarán durante los experimentos. El objetivo principal de cada uno de estas clases consiste en crear un banco de pruebas virtual con las mismas características de un banco de pruebas real. Esto es posible ya que GridSim permite definir un amplio conjunto de elementos que caractericen a cada recurso, como el número

de máquinas, el número de procesadores, el coste de procesamiento o la velocidad de procesamiento. La figura 6.9 representa la jerarquía de clases utilizada para los experimentos que se mostrarán en este capítulo. Está compuesta por la clase ASDTestbed, que modela las máquinas del laboratorio del grupo de Arquitectura de Sistemas Distribuidos y por las clases *HeterogeneousTestebed*, *HomogeneousTestbed* e *HibridTestebed* que modelan bancos de pruebas esencialmente Heterogéneos, Homogéneos e Híbridos.

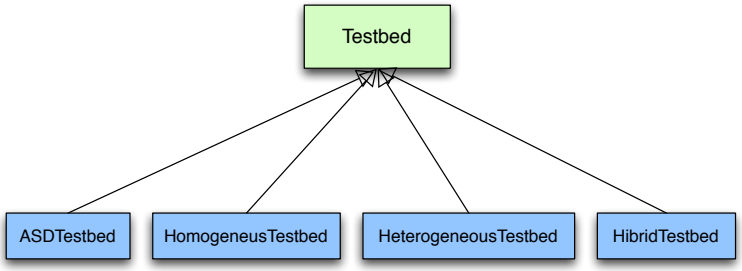

Figura 6.9: Jerarquía de clases Testbed

Módulos Job y Experiment

Estos módulos están constituidos por la clase *Job* y *Experiment* respectivamente. La clase *Job* encapsula a cada una de las tareas que se ejecutarán en los experimentos. Por tanto incorporará los atributos necesarios para su ejecución en el modelo de desarrollo DRMAA-Grid *W*ay como por ejemplo el tamaño del *chunk*, el conjunto de Gridlets a ejecutar, o atributos relacionados con los tiempo de ejecución y trasferencia. La clase *Experiment* define los experimentos que se ejecutarán en el simulador, básicamente es un contenedor de instancias de la clase *Job* con la incorporación de algunos atributos adicionales como por ejemplo el identificador del usuario que envía el experimento.

Módulo Broker

El módulo Broker está formado por una jerarquía de clases que hace de intermediario entre la aplicación y los recursos. Dentro del modelo DRMAA-Grid *W*ay corresponde con las tareas asociadas al metaplanificador Grid *W*ay. El objetivo de estas clases es planificar cada una de las Gridlets según el algoritmos de planificación utilizado, y enviarla a un recurso determinado. Este módulo esta estrechamente relacionado con el módulo DelayPolicy, ya que la ejecución de cada Grilet variará dependiendo de la política de ubicación utilizada. La figura 6.10 representa la jerarquía de clases que

constituyen el módulo Broker. La principal diferencia entre las clases *DelayerBroker* y

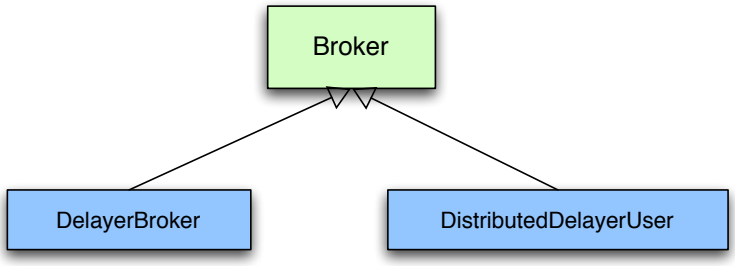

Figura 6.10: Jerarquía de clases Broker

DistributedDelayerBroker radica en la utilización de diferentes algoritmos de planificación. La clase *DelayerBroker* utiliza los algoritmos de planificación clásicos (CSS, GSS, TSS, FSS y FISS), mientras que la clase *DistributedDelayerBroker* utiliza el nuevo algoritmo GTSS. La necesidad de utilizad dos clases diferentes se debe a las diferencias de implementación entre los algoritmos clásicos y el algoritmo GTSS.

Módulo DelayPolicy

Este módulo está compuesto por la clase *DelayPolicy* que hereda de la clase *AllocPolicy* de GridSim. El objetivo de esta clase es implementar una nueva política de ubicación que consiste en retrasar la ejecución del trabajo en el recurso, para que el comportamiento del experimento sea lo más parecido a una situación real, en la que los nodos no están dedicados a la ejecución de un solo experimento, si no que son compartidos por multitud de usuarios y experimentos diferentes. De esta manera el comportamiento del simulador Grid*W*aySim se ajusta mucho más a una situación real, ya que el tiempo de espera en la cola del recurso se verá reflejado en el tiempo total de ejecución. Este tiempo de espera, ha sido modelado a través de la ejecución del mismo experimento en un entorno real, estos experimentos se explican con más detalle en la sección 6.3.3.

6.3.2. Secuencia de Ejecución del Simulador

Con el objetivo de analizar el comportamiento del simulador Grid*W*aySim, esta sección describe su secuencia de ejecución. La figura 6.11 muestra un esquema de la secuencia de ejecución del simulador Grid*W*aySim donde se simula el comportamiento

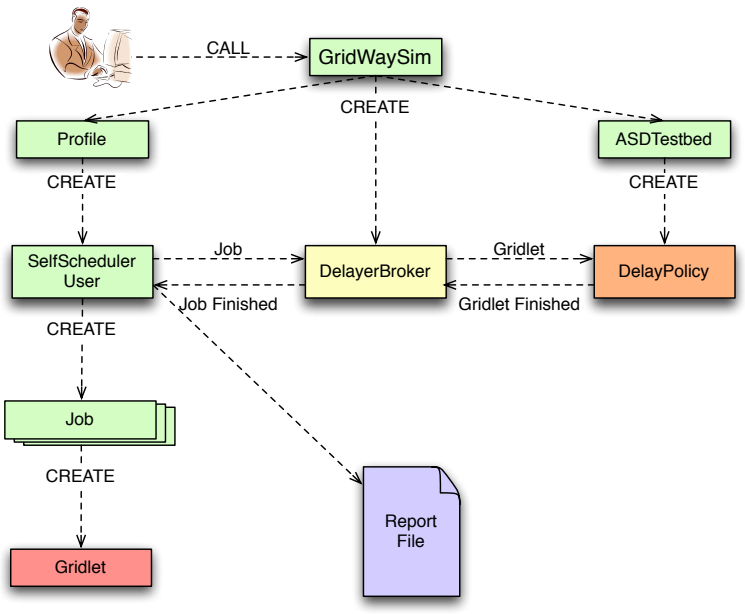

Figura 6.11: Secuencia de Ejecución en Grid *W*aySim

de un algoritmo de planificación clásico para la ejecución de una determinada aplicación. Como se puede observar, inicialmente el usuario ejecuta el simulador utilizando el interfaz de entrada apropiado. Después Grid *W*aySim crea instancias de las clases *SelfSchedulerProfile*, *DelayerBroker* y *ASDTestbed* para obtener los objetos que definen el perfil de ejecución de la aplicación, el *broker* encargado de realizar la planificación, dependiendo del algoritmo de planificación escogido por el usuario, y el banco de pruebas donde se ejecutarán cada uno de los trabajos. Además, la clase *SelfSchedulerProfile* se encargará de crear un objeto de la clase *SelfSchedulerUser* y un conjunto de instancias de la clase *Job* que almacenará en un objeto de la clase *Experiment*. El número de trabajos dependerá del algoritmo de planificación utilizado. A continuación, el objeto *DelayerBroker* se encarga de enviar todos los trabajos a la instancia de la clase *DelayerBroker* ya creada, y espera hasta recibir los resultados por parte de esta. Por su parte, la entidad *DelayerBroker* se comunica con una instancia de la clase *DelayPolicy* para enviarle la Gridlet a ejecutar indicando también el tiempo de medio espera en cola determinado por el recurso. La elección de un determinado recurso se realiza a través del planificador de recursos, que se encuentra implementado en la clase *Broker* y cuyo comportamiento es idéntico al del metaplanificador Grid *W*ay. El objeto *DelayPolicy* se encargará de ejecutar la Gridlet en el recurso y enviará los resultados al *DelayerBroker* que a su vez se los enviará al objeto de la clase *SelfSchedulerUser*. Finalmente,

esta clase realizará todas las tareas de postprocesamiento necesarias para generar un informe con los resultados obtenidos. Debido a la naturaleza del simulador, todas la entidades anteriores pertenecen a hilos independientes y por lo tanto se ejecutarán de manera concurrente.

6.3.3. Resultados Experimentales

El objetivo de esta sección consiste en analizar los resultados obtenidos durante la simulación de la ejecución del producto de dos matrices, para comprobar la funcionalidad y eficiencia del simulador GridWaySim en comparación con un banco de pruebas Grid real. Para ello, en primer lugar se presentarán los experimentos realizados sobre un *testbed* real, para después comparar dichos resultados con los obtenidos en la ejecución de la misma aplicación en el simulador GridWaySim. El motivo de utilizar el producto de matrices es que estás se utilizan en muchos problemas para la resolución de ecuaciones lineales, diferenciales y de derivadas parciales. Además, debido a las características del producto de matrices, como su estructura simple y un conjunto de propiedades bien definido, es utilizado como benchmark en sistemas de arquitecturas paralelas [rMrs92].

Para la ejecución de los experimentos en un banco de pruebas real, se utilizó el banco de pruebas descrito en la tabla 6.1 basado en el modelo de desarrollo DRMAA-GridWay. Cada uno de los experimentos realizados ejecuta un algoritmo de planifica-

Nombre	Modelo	SO	Velocidad	Memoria	Nodos
hydrus	Intel P4	Linux 2.6	3.2GHz	2GB	2
ursa	Intel P4	Linux 2.6	3.2GHz	2GB	2
cygnus	Intel P4	Linux 2.6	3.2GHz	2GB	1
aquila	Intel P4	Linux 2.6	3.2GHz	2GB	2

Tabla 6.1: Características de las máquinas del banco de pruebas UCM

ción dinámico clásico con el objetivo de calcular el producto de dos matrices. Concretamente, se definen dos matrices cuadradas NxN denominadas A y B con $N = 2^{14}$ elementos cada una. Esta aplicación se encuentra dentro del tipo de aplicaciones denominadas PSA (*Parametric Swep Applications*) y que fueron analizas en la sección 4.2. Por lo tanto, inicialmente se realizan una tareas de preprocesamiento de datos, que corresponde con la división de cada una de las matrices en matrices más pequeñas,

denominadas submatrices. Cada una de estas submatrices es de tamaño MxM con $M = 2^{10}$, y permiten obtener el producto de general de las matrices A y B a través del cálculo de productos de las submatrices utilizando el algoritmo Divide y Vencerás. Por lo tanto, lo que se distribuirá por el Grid será el cálculo de los productos de las submatrices que se denominan productos parciales. La ecuación 6.1 muestra el cálculo del tamaño del conjunto formado por estos productos parciales.

$$PP = \left(\frac{M}{N}\right)^3 \tag{6.1}$$

Donde N es el tamaño de cada matriz, en este caso $N = 2^{14}$, y M es el tamaño de cada una de las submatrices, $M = 2^{10}$. Por lo tanto el número total de productos parciales que se obtienen es de 4096. Cada uno de estos productos parciales se agrupan formando tareas independientes cuyo tamaño viene determinado por el algoritmo de planificación utilizado. Estas tareas se ejecutarán en el Grid de manera distribuida, y al final el usuario obtendrá un conjunto de resultados parciales sobre los que se realizará un postprocesamiento para obtener el resultado final.

Experimento	T_{exec}	$\overline{T_{cpu}}$	$\overline{T_{xfr}}$	$\overline{T_{queue}}$
CSS(32)	8:54:00	0:18:10	0:00:20	0:01:44
CSS(128)	7:32:00	1:13:35	0:05:00	0:06:12
GSS	6:40:00	0:43:53	0:01:00	0:04:15
GSSk(128)	7:13:00	2:00:00	0:02:30	0:09:11
TSS	6:29:00	1:08:12	0:00:52	0:04:48
FISS	7:47:00	1:17:50	0:00:40	0:07:15

Tabla 6.2: Resultados del experimento en un entorno Grid real

La tabla 6.2 muestra el resultado de dichos experimentos, haciendo hincapié en el tiempo total de ejecución, el tiempo medio de CPU ($\overline{T_{cpu}}$), el tiempo medio de trasferencia ($\overline{T_{xfr}}$) y el tiempo medio en el que el trabajo ha estado encolado en el recurso ($\overline{T_{queue}}$), el formato de todos estos tiempos sigue el estándar horas, minutos y segundos. Como se puede observar, el algoritmo que se adapta mejor a este banco de pruebas corresponde con el algoritmo TSS, ya que es el que obtiene un menor tiempo de ejecución. En contrapartida, el algoritmo con peores resultados corresponde con CSS(32), esto es debido a que el algoritmo CSS envía siempre la misma cantidad de productos parciales, en este caso 32, generando una cantidad mayor de trabajos que el

resto de algoritmos y por lo tanto incrementando el tiempo de ejecución total, debido principalmente a los tiempos T_{xfr} y T_{queue}.

La tabla 6.3 muestra los resultados obtenidos en la ejecución de los mismos experimentos con el simulador Grid*W*aySim. Como se puede observar, los tiempos de

Experimento	T_{exec}	$\overline{T_{cpu}}$	$\overline{T_{xfr}}$	$\overline{T_{queue}}$
CSS(32)	5:45:28	0:16:01	0:00:19	0:01:46
CSS(128)	6:03:34	1:04:02	0:01:19	0:06:22
GSS	5:55:30	0:33:52	0:00:46	0:04:18
GSSk(128)	6:39:00	1:38:40	0:02:10	0:09:11
TSS	5:47:10	1:01:40	0:01:35	0:05:50
FISS	6:29:08	1:08:33	0:01:24	0:07:25

Tabla 6.3: Resultados del experimento en un entorno Grid real

ejecución con el simulador Grid*W*aySim son inferiores a los tiempos de ejecución real, esto es debido a que los simuladores trabajan en un entorno ideal sin ningún tipo de errores de memoria o de comunicación. Además los tiempos medios de trasferencia, CPU y cola están ajustados para modelar el banco de pruebas presentado en la tabla 6.1. En la simulación también se comprueba que el algoritmo TSS es el que dispone de un mejor comportamiento desde el punto de vista de un entorno Grid heterogéneo en concordancia con los resultados obtenidos durante la ejecución en un entorno Grid real. Por ello, la figura 6.12 representa el tiempo de ejecución frente al número de trabajos ejecutados del algoritmo TSS en la ejecución del experimento en el banco de pruebas real y en el simulador Grid*W*aySim. Esta figura pone en relevancia la similitud de comportamiento entre Grid*W*aySim y el banco de pruebas Grid utilizado, con las diferencias debidas al tiempo de ejecución total. El motivo de estas diferencias reside en un menor tiempo de ejecución de los trabajos por parte del simulador. Este comportamiento es el esperado, ya que el simulador ejecuta los trabajos en un entorno ideal, y por lo tanto no existe, por ejemplo, errores de comunicación entre los distintos nodos, o durante la ejecución de cada trabajo en el recurso asignado. Por lo tanto, Grid*W*aySim demuestra su efectividad y eficiencia en la simulación de ejecución de aplicaciones Grid con el modelo de programación DRMAA-Grid*W*ay, convirtiéndose en una herramienta de gran utilidad en el análisis de algoritmos de planificación en entornos Grid heterogéneos.

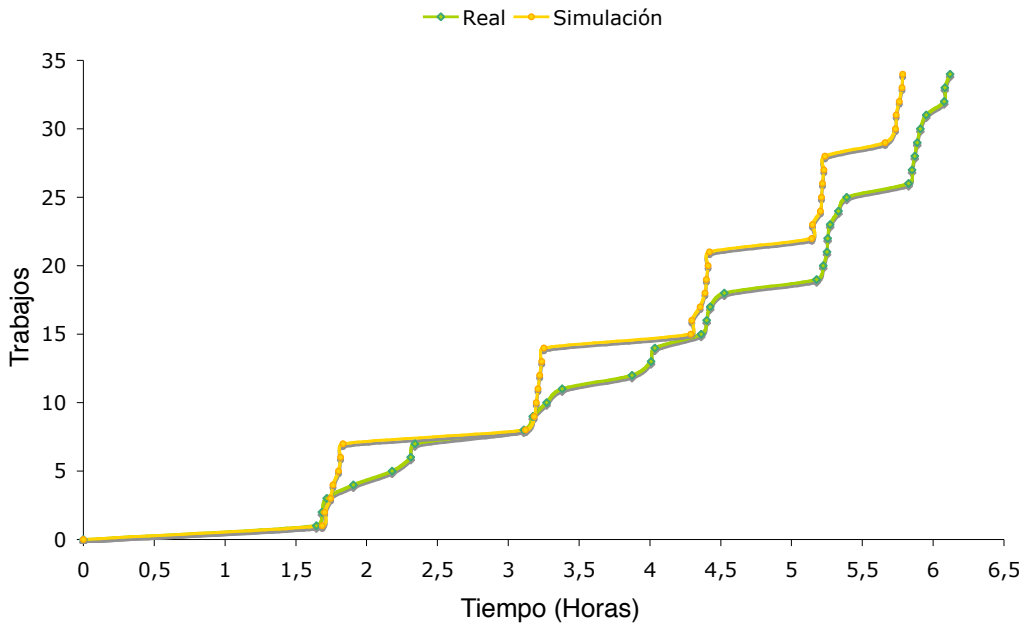

Figura 6.12: Algoritmo TSS y su ejecución en GridWaySim y en un entorno Grid real

6.4. Experimentos y Resultados en Grid *W*aySim

Una vez demostrada la efectividad de Grid *W*aySim para la simulación de ejecución de aplicaciones Grid según el modelo de programación DRMAA-Grid *W*ay, esta sección presenta los resultados obtenidos al utilizar dicho simulador como herramienta para comprobar la eficiencia del algoritmo de planificación GTSS frente a los algoritmos de planificación clásicos. El objetivo de esta sección es presentar el conjunto de experimentos realizados bajo esta plataforma, y analizar los resultados obtenidos que demuestren la adaptabilidad del algoritmo GTSS a un entorno Grid heterogéneo, frente al resto de algoritmos que existen en la actualidad. Para ello, se realizará una descripción de los experimentos ejecutados, después se presentarán los diferentes bancos de pruebas donde se han realizado dichos experimentos, para finalmente, analizar los resultados obtenidos.

6.4.1. Descripción de los Experimentos

Con el objetivo de comprobar el comportamiento del algoritmo GTSS frente al resto de algoritmos de paralización en la planificación de tareas en Grid heterogéneos, se desarrolló un conjunto de experimentos basados en el problema del producto de

matrices de grande dimensiones al igual que se realizó para los experimentos de la sección 6.3.3. Concretamente se escogió el rango de productos parciales $[2000, 10000]$ con un incremento en cada experimento de 100 productos parciales. De esta manera se obtuvo un total de 98 experimentos para cada uno de los algoritmos de planificación utilizados (CSS, GSS, TSS, FSS, FISS y GTSS). Por lo tanto, el número total de experimentos realizados con el simulador Grid$W$aySim asciende hasta un total de 588 experimentos.

Previamente a la ejecución de estos experimentos, se ejecutó un único experimento con el algoritmo CSS para el cálculo de de 4096 productos parciales en el mismo banco de pruebas donde se realizaron el resto de experimentos. El objetivo de esta ejecución consiste en calcular los parámetros $n_{1/2}$ and $R_{max}(i)$ para ajustar el algoritmo GTSS al banco de pruebas utilizado.

6.4.2. Banco de Pruebas de Ejecución

Para la ejecución de estos experimentos se modeló un banco de pruebas meramente heterogéneo constituido por las máquinas que muestran la tabla 6.4. El diseño de este

Name	Modelo	SO	MIPS	Nodos
libra	ADL	Linux 2.6	12765	2
ursa	ADL	Linux 2.6	10872	2
cygnus	ADL	Linux 2.6	9787	1
aquila	ADL	Linux 2.6	8851	2
hydrus	ADL	Linux 2.6	7829	2
orion	ADL	Linux 2.6	2893	2
cepheus	ADL	Linux 2.6	1914	2
pisces	ADL	Linux 2.6	936	2
leo	ADL	Linux 2.6	857	2
gemini	ADL	Linux 2.6	779	2

Tabla 6.4: Características de las máquinas del banco de pruebas heterogéneo

banco de pruebas virtual se realizó basándose en el conjunto de máquinas existentes en laboratorio del grupo ASD (Arquitectura de Sistemas Distribuidos), ubicado en la facultad de informática de la Universidad Complutense de Madrid. Como muestra dicha tabla, el número total de nodos de los que se dispone es de 19, repartidos entre un conjunto de máquinas con velocidades de procesamiento muy heterogéneas, siendo

la máquina más rápida libra con 12765 MIPS y la más lenta gemini con 779 MIPS. Este banco de pruebas permite comprobar determinados aspectos del algoritmo de planificación GTSS, como el equilibrio de carga computacional y la reducción en tiempo de ejecución de cada experimento.

6.4.3. Análisis de los Resultados

Esta sección presenta un análisis de los resultados obtenidos en la ejecución de los 588 experimentos realizados en el simulador Grid*W*aySim. Todos los resultados obtenidos se han resumido en dos gráficos principales, el primero de ellos se muestra en la figura 6.13 y representa el porcentaje de uso medio de cada nodo del banco de pruebas tal y como se describió en la ecuación 4.2. El porcentaje del uso medio de cada nodo es un parámetro muy importante dentro de la planificación de tareas, ya que, permite medir el equilibro de carga computacional entre todos los nodos que intervienen en el experimento. El gráfico muestra el grado de utilización para cada

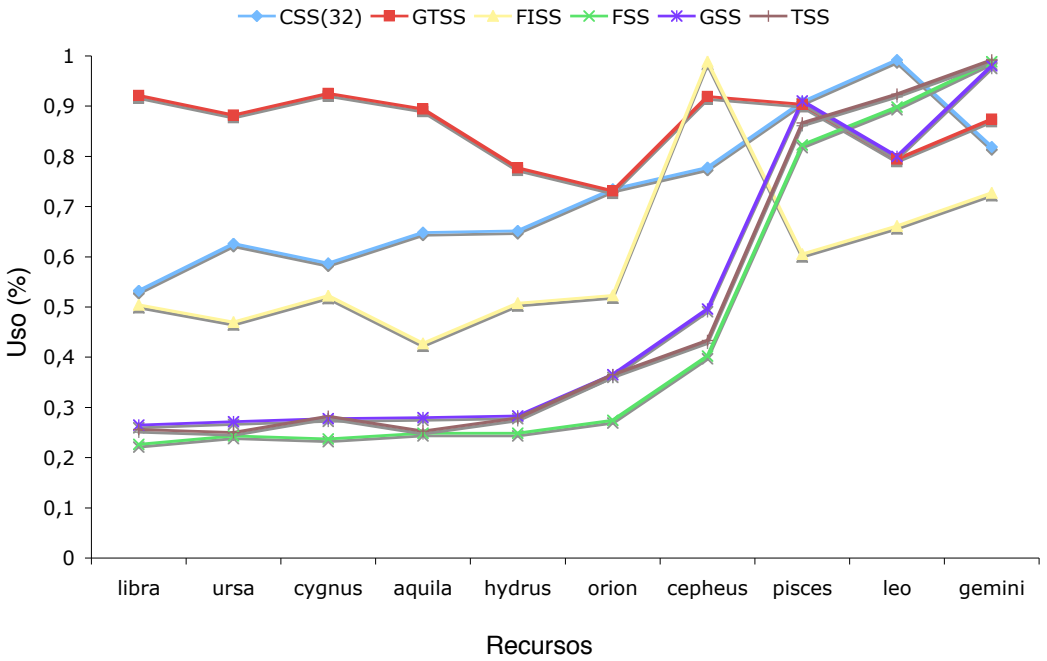

Figura 6.13: Porcentaje de utilización de las máquinas del banco de pruebas

una de las máquinas que constituyen el banco de pruebas virtual, ordenadas según su velocidad de procesamiento. Siendo la máquina más cercana al eje de ordenadas aquella con velocidad de procesamiento mayor, es decir libra, y la máquina más alejada de este

eje aquella con velocidad de procesamiento menor, gemini. La primera conclusión que se puede obtener a la vista de esta figura es que los algoritmos de planificación GSS, TSS y FSS no consiguen obtener un buen equilibro de carga entre los diferentes nodos, ya que el porcentaje de utilización obtenido en cada caso es inversamente proporcional a la velocidad de procesamiento de los nodos. Consiguiendo así un porcentaje de utilización muy bajo para los nodos más rápidos, y saturando los nodos con menor velocidad de procesamiento desequilibrando de esta manera la carga computacional entre los nodos del banco de pruebas.
Un caso especial es el que ocurre con el algoritmo CSS(32), ya que aunque la tendencia es la misma que con los algoritmos GSS, TSS y FSS, el porcentaje de utilización en los nodos mas rapidos es mayor que el obtenido en los casos anteriores. Sin embargo su comportamiento presenta un desequilibro de carga notable, con una diferencia de un 46 % entre el nodo con menor porcentaje de uso, correspondiente al nodo con mayor capacidad de procesamiento (libra), y el nodo con mayor grado de utilización, que corresponde con uno de los nodos más lentos del banco de pruebas (leo).

Los algoritmos que presentan una distribución de carga computacional más equilibrada para estos experimentos corresponden con los algoritmos FISS y GTSS. En este caso el algoritmo GTSS es el que presenta un equilibro de carga más ajustado, ya que no llega a saturar totalmente ningún nodo durante los experimentos, y además es el que presenta un mayor grado de utilización, en general, entre todos los nodos del banco de pruebas. Con una cota inferior del 73 % en la máquina orion y con una cota superior del 92 % en la máquina con mayor capacidad de procesamiento (libra). Por lo tanto el algoritmo GTSS obtiene unos resultados que permiten confirmar una óptima distribución de la carga computacional entre todos los nodos que intervienen en los experimentos, obteniendo un buen equilibro de carga entre ellos.

El segundo gráfico, representado en la figura 6.14 corresponde con el estudio del tiempo de ejecución en cada uno de los experimentos realizados para los algoritmos CSS, TSS, FSS, FISS y GTSS. Según los resultados obtenidos, el algoritmo que presenta un tiempo de ejecución menor que el resto es el correspondiente al algoritmo GTSS. Comportándose en todos los experimentos como cota inferior del resto de los algoritmos utilizados. En los resultados obtenidos, a excepción de los algoritmos CSS y TSS, el tiempo de ejecución sigue una tendencia creciente proporcional al número de trabajos en ejecución. Por lo tanto el comportamiento de estos algoritmos sigue una distribución lineal, y permite, de esta manera, pronosticar su comportamiento en

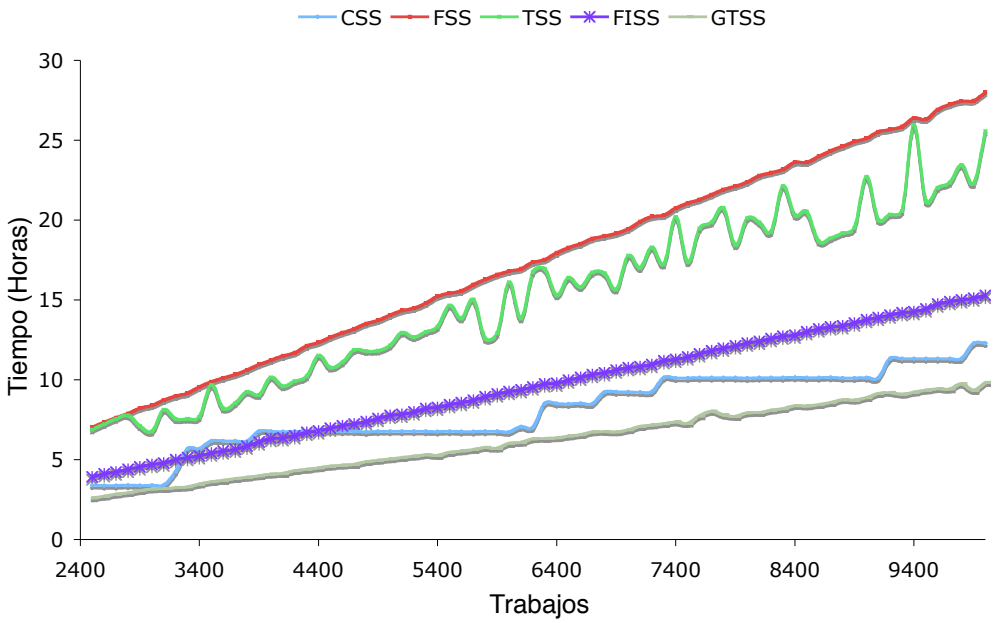

Figura 6.14: Tiempo de ejecución de cada experimento

experimentos futuros. Sin embargo, los algoritmos CSS y TSS siguen una distribución escalonada y oscilante respectivamente, debido principalmente a la influencia del nodo más lento en la planificación de los trabajos. De esta manera el trabajo ubicado en el nodo más lento realizará la función de cota superior respecto al tiempo total de ejecución del experimento. Es decir, el tiempo total del experimento vendrá determinado por el nodo más lento y la carga de trabajo asignada a este nodo. En estos resultados, se ve la importancia de una distribución de carga eficiente que además permita disminuir el tiempo total de ejecución de la aplicación. El único algoritmo que realiza una distribución de carga eficiente y obtiene una disminución en el tiempo total de ejecución, es el correspondiente al algoritmo adaptado a entornos Grid GTSS. Es importante destacar que todos los resultados obtenidos con los algoritmos de planificación clásicos, se han realizado utilizando los valores óptimos de los parámetros de cada algoritmo. Dichos valores los debe saber el usuario a priori, a diferencia del algoritmo GTSS donde el usuario no necesita calcular ningún parámetro para obtener los resultados presentados, ya que se realizan automáticamente. De esta manera queda comprobada la eficiencia y efectividad del algoritmo GTSS en su ejecución en un entorno virtual generado por el simulador Grid*W*aySim. La siguiente sección analizará el comportamiento de este algoritmo en un entorno Grid real.

6.5. Experimentos y Resultados en Grid *W*ay

En la sección anterior se analizó el comportamiento del algoritmo GTSS en la planificación de tareas sobre un simulador de sistemas Grid adaptado al modelo de programación DRMAA-Grid *W*ay. Esta sección presenta y analiza los resultados obtenidos de la ejecución de un conjunto de experimentos realizados en bancos de pruebas Grid reales. Concretamente en el proyecto de fusión MaRaTra. En primer lugar se presentará el proyecto MaRaTra así como el conjunto de experimentos diseñados para su ejecución en un banco de pruebas de la plataforma EGEE. A continuación se mostrará el banco de pruebas utilizado para la ejecución de los experimentos. Finalmente, se analizarán los resultados obtenidos de la ejecución de estos experimentos, destacando aquellas características que hacen del algoritmo GTSS una buena alternativa en la planificación de tareas Grid.

6.5.1. Descripción de los Experimentos: Proyecto MaRaTra

Con el objetivo de demostrar la eficiencia del algoritmo de planificación GTSS, se ha diseñado un conjunto de experimentos para su ejecución en la plataforma MaRaTra (*Massive Ray Tracing*) dentro del proyecto EGEE [EGE08]. La aplicación MaRaTra calcula la trayectoria de un haz de microondas en un plasma. Este haz de microondas se utiliza para calentar el plasma y de esta manera obtener energía. Normalmente el haz se simula utilizando 105 rayos, sin embargo, en estos experimentos se han aumentado hasta un total de 2×10^3 rayos obteniendo así unos resultados con mayor precisión. Para realizar esta simulación, el programa denominado TRUBA calcula la trayectoria y la absorción de cada rayo independiente en plasmas complejos. De esta manera, se obtienen 2×10^3 ejecuciones del programa TRUBA que se distribuirán a través del Grid agrupadas en tareas. El número de ejecuciones TRUBA que formen cada tarea vendrá determinado por el tamaño del *chunk* obtenido de la ejecución de los algoritmos de planificación CSS, GSS, TSS, FSS, FISS, GTSS. Así, se analizarán los resultados obtenidos, comprobando de esta manera que el algoritmo GTSS se adapta mucho mejor a un entorno Grid que el resto de algoritmos, obteniendo un buen equilibrio de carga entre los nodos y reduciendo el tiempo de ejecución total. Es importante remarcar, que al igual que ocurría en los experimentos realizados con el simulador, en estos experimentos se han utilizado los parámetros óptimos de cada uno de los algoritmos de planificación clásicos, estos parámetros los debe conocer el usuario de la aplicación

a priori.

Además, estos experimentos demostrarán la trasparencia y flexibilidad del modelo DRMAA-Grid*W*ay ya que se realizan bajo un *middleware* diferente al resto de experimentos presentado en esta tesis, concretamente se trata del *middleware* gLite [gLi08] del EGEE. El *middleware* gLite se utiliza en el proyecto EGEE como alternativa a Globus, y está constituido por un software mucho más ligero que permite a los usuarios implementar diferentes servicios según sus requerimientos, sin necesidad de utilizar un sistema más completo, como Globus o Condor. Con esto se pretende que cada usuario adapte el sistema a su situación particular. Además gLite es compatible con los estándares *Web Service Resource Framework* (WSRF) de OASIS y *Open Grid Service Architecture* (OGSA) del Global Grid Forum.

6.5.2. Banco de Pruebas de Ejecución

El banco de pruebas utilizado para este experimento se muestra en la tabla 6.5. Como se observa, el número total de nodos de los que se dispone es de 50, es decir

Name	Modelo	SO	MHZ	Nodos
t2ce02.physics.ox.ac.uk	INTEL	Scientific Linux 3	2800	10
grid002.jet.efda.org	INTEL	ScientificCERNSLC Beryllium	2614	10
ce2.egee.cesga.es	INTEL	ScientificSL Beryllium	3000	10
ce1.egee.fr.cgg.com	INTEL	Scientific Linux SL	1266	10
ce.hep.ntua.gs	INTEL	Scientific Linux SL	2800	10

Tabla 6.5: Características de las máquinas del banco de pruebas EGEE

se utilizarán 10 nodos de cada uno de los *clusters* del EGEE presentados en la tabla, constituyendo un banco de pruebas heterogéneo. Según la información que muestra esta tabla, la máquina con mayor velocidad de procesamiento (ce2.egee.cesga.es a 3000 MHZ) debería ejecutar un mayor número de trabajos que el resto, de la misma forma la máquina con menos velocidad de procesamiento (ce1.egee.fr.cgg.com a 1266 MHZ) debería ejecutar un número de trabajos mucho menor. Como se comprobará en los resultados, esta predicción no se cumple, ya que en un entorno Grid heterogéneo no solo hay que tener en cuenta la velocidad de procesamiento, sino también otros aspectos como el tiempo de espera en cola de los recursos, la memoria asignada a cada tarea, la velocidad de red, la velocidad de interconexión de los nodos o el porcentaje de

saturación de estos. Un planificador adaptado al Grid debería tener en cuenta todos estos aspectos a la hora de planificar las tareas, con el objetivo de disminuir el tiempo total de ejecución y obtener un buen equilibrio de carga.

6.5.3. Análisis de los Resultados

Esta sección presenta el análisis de los resultados obtenidos en la ejecución de los experimentos realizados de la aplicación MaRaTra sobre la plataforma EGEE. La figura 6.15 representa el porcentaje medio de uso de cada nodo del banco de pruebas del EGEE. Los nodos están ordenados por velocidades de ejecución, siendo el más lento

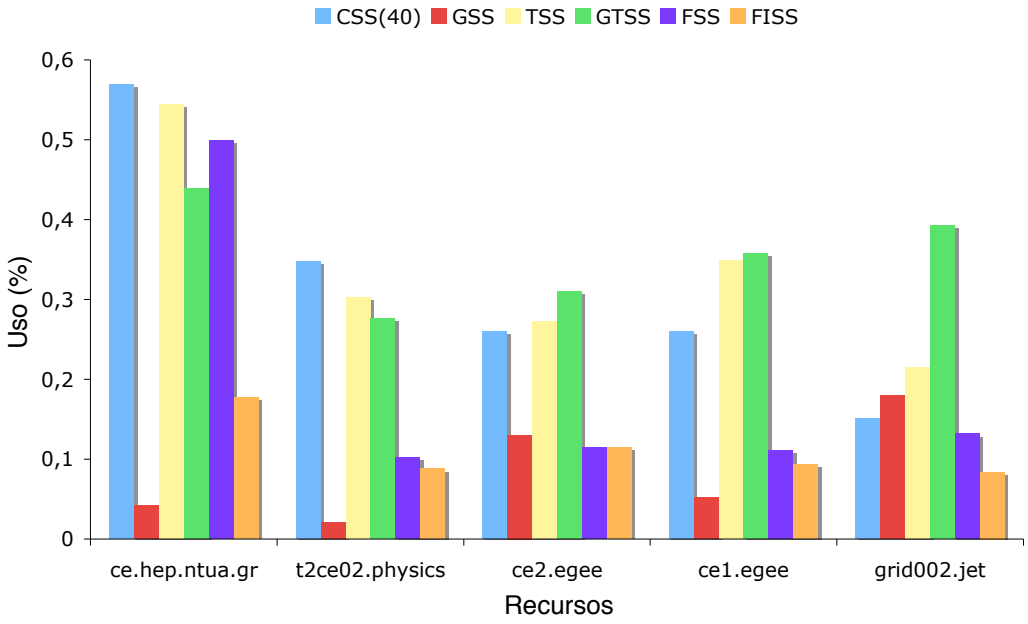

Figura 6.15: Porcentaje medio de uso de las máquinas del banco de pruebas

el nodo ce.hep.ntua.gs, y el nodo grid002.jet.efda.org, el que presenta un mejor tiempo de ejecución. Como se puede observar en la tabla 6.5 la velocidad de procesamiento de la máquina grid002.jet.efda.org es menor que la máquina ce.hep.ntua.gs, sin embargo, la máquina grid002.jet.efda.org obtiene un tiempo de ejecución mucho menor que la máquina ce.hep.ntua.gs, debido a los diferentes factores que intervienen durante la ejecución de los trabajos como la velocidades de red o el grado de saturación de los nodos. El único algoritmo que utiliza esta información para realizar una planificación eficiente de las tareas es el algoritmo GTSS. Además, el grado de utilización, en el

caso del algoritmo GTSS, se encuentra siempre en un rango muy ajustado entre el 30 % y el 45 %, obteniendo de esta manera un buen equilibrio de carga entre todos los nodos que intervienen en la ejecución. Un dato importante a tener en cuenta es el porcentaje de utilización en las máquinas con velocidad de ejecución mayor. En todos los algoritmos de planificación clásicos el porcentaje de uso tiende a decrementarse en los nodos con velocidad de ejecución mayor. De esta manera desaprovechan la capacidad de computación de estos nodos a favor de los nodos con velocidad de ejecución menor, debido a ello obtienen un reparto de carga más desequilibrado y un incremento en el tiempo total de ejecución, tal y como se muestra en la figura 6.16.

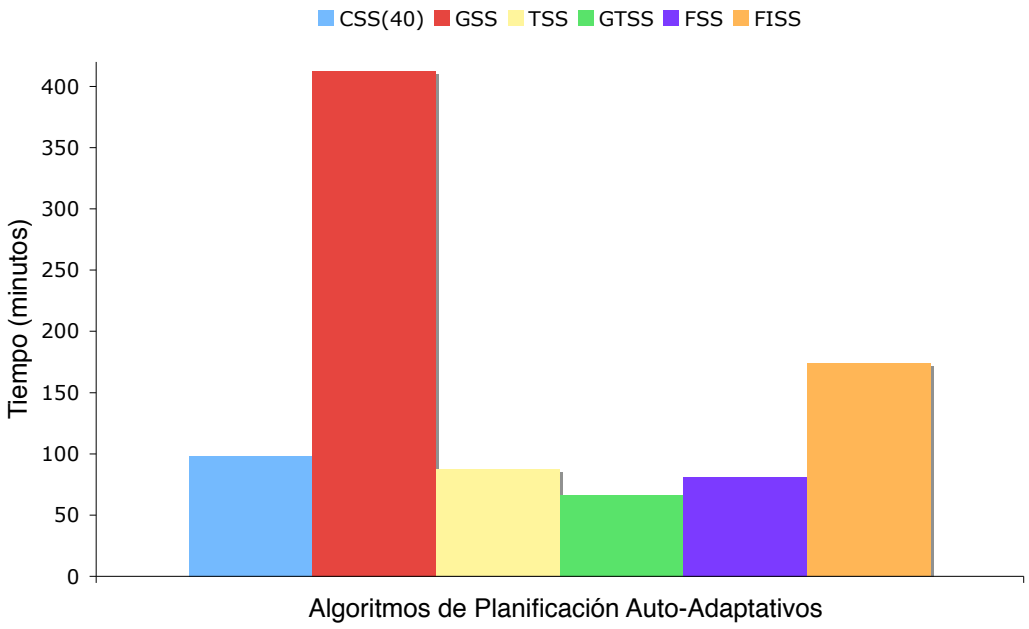

Figura 6.16: Tiempo de ejecución del experimento con cada planificador.

La figura 6.16 representa el tiempo de ejecución total de la aplicación frente a los distintos tipos de algoritmos de planificación utilizados. De todos ellos, el algoritmo que presenta un menor tiempo de ejecución es el correspondiente al algoritmo GTSS con un tiempo de ejecución de 66,20 minutos, que representa una reducción del 83,94 % respecto al tiempo de ejecución con el algoritmo más lento (GSS), y un 18,22 % con respecto al algoritmo de planificación clásico que obtiene un mejor tiempo de ejecución total (FSS). Esto es debido a la adaptabilidad del algoritmo de planificación GTSS a cualquier entorno Grid de ejecución.

Por lo tanto queda demostrada la efectividad del algoritmo GTSS para la ejecución

de aplicaciones distribuidas en un entorno Grid heterogéneo. Ya que, además de disminuir el tiempo total de la aplicación y obtener un buen equilibrio de carga entre los nodos del Grid, GTSS dota a todas estas aplicaciones de las características del modelo DRMAA-Grid*W*ay, como son flexibilidad, trasparencia y adaptación dinámica.

6.6. Conclusiones

En este capítulo se han presentado los experimentos realizados para comprobar la funcionalidad y eficiencia del algoritmo GTSS para la planificación de tareas en Grid. Estos experimentos se han ejecutado en dos plataformas distintas. Por un lado, se han realizado un conjunto de experimentos sobre una plataforma de simulación adaptada al modelo DRMAA-Grid*W*ay. Y por otro, se ha ejecutado un conjunto de experimentos en un entorno real de investigación Grid dentro del proyecto EGEE.

En primer lugar, este capitulo realiza una descripción de los principales simuladores Grid que existen en la actualidad (Bricks, MicroGrid, SimGrid y GridSim), y determina que el conjunto de herramientas GridSim es el más apropiado para desarrollar el simulador Grid*W*aySim. Esto es debido a que GridSim ofrece un conjunto de características (modelado de recursos, políticas de ubicación, gestión de usuarios o estadísticas de resultados), que no se encuentran disponibles en resto de simuladores Grid analizados.

Para demostrar la efectividad del simulador Grid*W*aySim en el modelado de entornos Grid heterogéneos se han ejecutado el mismo conjunto de experimentos en un banco de pruebas real de investigación y en el propio simulador Grid*W*aySim. Los resultados obtenidos en las dos plataformas demuestran la eficiencia del simulador Grid*W*aySim para el modelado de aplicaciones y entornos Grid adaptado al modelo de desarrollo DRMAA-Grid*W*ay.

Una vez demostrada la funcionalidad del nuevo simulador, el capítulo presenta los resultados obtenidos en los experimentos realizados utilizando el algoritmo GTSS para la planificación de trabajos en un entorno Grid virtual heterogéneo. El conjunto de resultados obtenidos de la ejecución de los experimentos realizados en esta plataforma permiten pronosticar la funcionalidad y efectividad en la planificación de tareas por parte del nuevo algoritmo de planificación GTSS. Los resultados obtenidos demuestran una disminución en el tiempo total de ejecución de la aplicación, así como un reparto eficiente de la carga computacional que permite obtener un ajustado equilibrio de carga

entre los nodos virtuales del simulador.

El último conjunto de experimentos demuestran la efectividad de este algoritmo en la ejecución de una aplicación real sobre el proyecto EGEE. Los resultados obtenidos permiten ratificar las conclusiones realizadas sobre los experimentos ejecutados en la plataforma virtual Grid*W*aySim. Además, ponen de relieve la adaptabilidad del modelo DRMAA-Grid*W*ay a diferentes *middlewares* para la ejecución de aplicaciones, sin necesidad de realizar cambios en su código.

Por otra parte, la ejecución de estos experimentos demuestran que el algoritmo GTSS se adapta a cualquier tipo de entorno Grid heterogéneo, ya que realiza una planificación basada en un *ranking*, que permite clasificar los nodos según el tiempo de ejecución de las tareas en lugar de tener en cuenta parámetros arquitectónicos como ocurre en la mayoría de los planificadores actuales. Esta es la principal diferencia de un entorno real frente al simulador, ya que en el simulador aquel nodo con mejores prestaciones también dispondrá de un tiempo de ejecución mejor que el resto. Sin embargo, como se ha comprobado con estos experimentos, en una situación real esta afirmación no se cumple, ya que existen muchos factores que hacen variar considerablemente el tiempo de ejecución, como el tiempo de espera en cola de los recursos, la memoria asignada a cada tarea, la velocidad de red, la velocidad de interconexión de los nodos o el porcentaje de saturación de estos. Un algoritmo de planificación adaptativo debería tener todos estos aspectos en cuenta a la hora de realizar la planificación, como ocurre con el algoritmo de planificación desarrollado en esta tesis.

El conjunto de experimentos realizados demuestran por tanto, la eficiencia y efectividad del nuevo algoritmo de planificación adaptado a entornos Grid heterogéneos, GTSS (Grid Trapezoid Self-Scheduler), en comparación con las versiones óptimas de los algoritmos de planificación clásicos.

Capítulo 7

Principales Aportaciones y Trabajo Futuro

Este capítulo resume las principales aportaciones presentadas en esta tesis doctoral. Así mismo, se muestran las publicaciones realizadas que justifican cada una estas aportaciones, y el trabajo futuro que originan. Para ello, el capítulo se divide en dos secciones, la sección 7.1 realiza un resumen de las aportaciones de la tesis doctoral incluyendo sus principales publicaciones. Y la sección 7.2, describe el trabajo futuro que se origina a partir de las investigaciones y aportaciones realizadas durante el desarrollo de la presente tesis doctoral.

7.1. Principales Aportaciones

Esta sección realiza una descripción de las principales aportaciones presentadas en esta tesis doctoral. En primer lugar se analizará el modelo de desarrollo DRMAA-Grid *W*ay, después se presentará el algoritmo de planificación Grid Trapezoid Self-Scheduler (GTSS) y finalmente se realizará una descripción del simulador para entornos Grid heterogéneos Grid *W*aySim.

7.1.1. Modelo de Desarrollo DRMAA-Grid *W*ay

En los últimos años han aparecido una gran cantidad de herramientas que facilitan el desarrollo y ejecución de aplicaciones Grid. Sin embargo, no existe un modelo diseñado específicamente para el desarrollo de aplicaciones en entornos Grid heterogéneos, que

además dote a las aplicaciones de trasparencia, fiabilidad y adaptación dinámica. Estas características permiten el desacople por parte de las aplicaciones de la infraestructura Grid donde se ejecuten.

Esta tesis presenta un nuevo modelo de programación, al que se ha denominado como DRMAA-Grid *W*ay, que cumple con todas estas características. DRMAA-Grid *W*ay debe su denominación a los principales componentes que lo constituyen, estos son, el metaplanificador Grid *W*ay y el paradigma de programación de aplicaciones distribuidas DRMAA.

El metaplanificador Grid *W*ay permite la ejecución de trabajos de manera fiable y además ofrece una extensa CLI (*Command Line Interface*) que facilita la ejecución y gestión de los trabajos a través del Grid. Grid *W*ay ofrece, además, otras características adicionales como la migración de trabajos, o la incorporación dinámica de recursos. Estas características en el modelo DRMAA-Grid *W*ay se traducen como, fiabilidad en la ejecución de las aplicaciones, y adaptación dinámica de estas a los cambios que se produzcan en el Grid durante su ejecución.

Por su parte el API de programación distribuida DRMAA (*Distributed Resource Management Application API*) es una especificación que permite el desarrollo de programas utilizando un estándar que facilita el despliegue de aplicaciones en Grid. Además, este API permite el desarrollo de aplicaciones Grid independientemente del gestor de recursos utilizado, en el caso del modelo DRMAA-Grid *W*ay la gestión de recursos la realiza el metaplanficador Grid *W*ay, y proporciona al desarrollador las llamadas necesarias para realizar una completa gestión de los trabajos enviados para su ejecución. Por lo tanto, el API DRMAA permite al usuario del modelo DRMAA-Grid *W*ay el desarrollo de aplicaciones Grid de manera sencilla, facilitando su despliegue a otros *middlewares* Grid como se demuestra en el capítulo 6, al utilizar dicho modelo con el *middleware* gLite.

El modelo de programación DRMAA-Grid *W*ay facilita el desarrollo de aplicaciones Grid en entornos distribuidos e incluye otras características que le hacen único respecto al resto de modelos de programación Grid existentes. De entre todas las características que aporta, la más importante consiste en la capacidad para desarrollar aplicaciones cuya ejecución esté totalmente desacoplada del entorno Grid donde se desarrolle. Esto permite al desarrollador implementar una aplicación Grid en un determinado entorno, y utilizarla en cualquier otro sin necesidad de realizar ningún cambio. También es muy importante destacar que dentro del modelo de programación DRMAA-Grid *W*ay

está incluida la primera implementación del API DRMAA tanto para el lenguaje C como para el lenguaje Java, ya que antes de incorporarse al modelo no existía ninguna implementación desarrollada de este API.

A continuación se presentan las publicaciones más importantes realizadas durante la investigación del modelo de desarrollo DRMAA-Grid *W*ay:

- J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente. Execution of typical scientific applications on globus-based grids. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 177–183, Washington, DC, USA, 2004. IEEE Computer Society.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Developing grid-aware applications with drmaa on globus-based grids. In *Euro-Par*, pages 429–435, 2004.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Embarrassingly distributed and master-worker paradigms on the grid. In Pilar Herrero, María S. Pérez, and Víctor Robles, editors, *SAG*, volume 3458 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2004.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Drmaa implementation within the gridway framework. Technical report, Global Grid Forum (GGF), September 2004.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Porting of scientific applications to grid computing on gridway. *Scientific Programming*, 13(4):317–331, 2005.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. A grid-oriented genetic algorithm. In Peter M. A. Sloot, Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *EGC*, volume 3470 of *Lecture Notes in Computer Science*, pages 315–322. Springer, 2005.

7.1.2. Algoritmo Grid Trapezoid Self-Scheduler

Uno de los mayores retos a la hora de ejecutar aplicaciones en Grid reside en los algoritmo utilizados para la planificación de código. Hoy en día existen multitud de algoritmos para la distribución de código, desde los estáticos, como el planificador de bloques [Lil94], hasta los dinámicos distribuidos como *Distributed Trapezoid SelfScheduler* (DTSS) [CBGA01]. Sin embargo, ninguno de estos algoritmos está diseñado especialmente para su ejecución en entornos Grid heterogéneos.

El nuevo algoritmo de planificación GTSS (*Grid Trapezoid Self-Scheduler*) se adapta a las necesidades de un entorno Grid heterogéneo y permite a su vez una distribución uniforme de la carga computacional de los trabajos entre todos los nodos del Grid, reduciendo de esta manera el tiempo total de ejecución respecto al resto de algoritmos de planificación existentes. Todas estas características se obtienen por la combinación de los tres elementos que constituyen el algoritmo, a saber:

- **Modelo benchmark de Grid**. Permite modelar cualquier sistema Grid heterogéneo a partir de dos parámetros r_∞ y $n_{1/2}$ que caracterizan la carga computacional del sistema.

- **El factor de relación** R_i^{min}. Es el coeficiente entre el menor tiempo medio de ejecución, y el tiempo medio de ejecución en el nodo i. Este parámetro permite realizar un ranking entre todos los nodos del Grid ordenado según velocidad de ejecución. De esta manera el algoritmo se ajustará mejor al comportamiento del Grid en cada momento.

- **El algoritmo dinámico TSS**. Es un algoritmo de planificación de iteraciones cuyo comportamiento es el que mejor se adapta a un entorno Grid de todos los algoritmos clásicos estudiados.

Todos estos elementos constituyen un algoritmo dinámico y distribuido que permite una distribución de la carga computacional adaptada al comportamiento específico del Grid donde se vaya a realizar la ejecución. Este algoritmo, además se puede utilizar para realizar tareas de grano fino, como es la planificación de iteraciones, o tareas de grano grueso, como por ejemplo la planificación de trabajos.

A continuación se presentan las publicaciones más importantes realizadas durante la investigación del algoritmo de planificación GTSS:

- J. Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Loosely-coupled loop scheduling in computational grids. In *IPDPS*. IEEE, 2006.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Benchmarking of a joint irisgrid/egee testbed with a bioinformatics application. *Scalable Computing: Practice and Experience*, (7):25–32, 2006.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Ejecución distribuida de bucles en grids computacionales. *RedIRIS: boletín de la Red Nacional de I+D RedIRIS*, (80):52–56, april 2007.

- Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. *Distributed Task Scheduling for Physics Fusion Applications*, page 27. March 2008.

7.1.3. Simulador Grid *W*aySim

En los últimos años han surgido un conjunto de herramientas que permiten la ejecución de aplicaciones en un entorno de simulación Grid. Estos simuladores consiguen obtener resultados de manera casi instantánea evitando las largas esperas producidas durante las ejecuciones en un entorno Grid real. Por lo tanto, los simuladores Grid se perfilan como herramientas muy útiles para la investigación de nuevas aplicaciones, paradigmas y planificadores dentro del universo Grid.

Con el objetivo de desarrollar un simulador adaptado al modelo de programación DRMAA-Grid *W*ay, se han aprovechado las características del conjunto de herramientas GridSim [BM02], para diseñar un simulador, denominado Grid *W*aySim, que permite investigar el comportamiento del algoritmo de planificación GTSS. La idea principal de investigación del algoritmo GTSS consiste en probar su eficiencia en el simulador Grid *W*aySim, para después comprobar su funcionalidad en un entorno real. Desde este punto de vista, el simulador Grid *W*aySim permite generar una gran cantidad de bancos de pruebas y de esta manera estudiar el comportamiento del algoritmo GTSS en diferentes entornos.

Para ajustar el simulador Grid *W*aySim a un entorno Grid real, se ha incorporado una nueva política de ubicación que permite modelar el comportamiento de los planificadores locales de cada uno de los recursos que pertenecen al banco de pruebas virtual. De esta manera, el desarrollador podrá incluir un tiempo medio de espera en la cola

del planificador local del recurso, y así modelar el *testbed* según las necesidades del usuario.

Por lo tanto, Grid*W*aySim permite simular una gran cantidad de escenarios distintos donde analizar el comportamiento de aplicaciones y planificadores dentro del modelo de programación DRMAA-Grid*W*ay. Además, debido a las características del simulador GridSim, Grid*W*aySim obtiene un conjunto muy amplio de estadísticas de resultados que facilitan la investigación de los experimentos realizados.

7.2. Trabajo Futuro

El trabajo futuro relacionado con la línea de investigación compuesta por los paradigma de programación en Grid, consiste en adaptar este modelo de programación para su uso con otros paradigmas como la especificación SAGA, u otras tecnologías de computación Grid como ocurre con UNICORE. De esta manera, el modelo DRMAA-Grid*W*ay seria aún más flexible y adaptativo, permitiendo la ejecución de sus aplicaciones en entornos Grid muy diversos, pero manteniendo todas las características mencionadas como trasparencia, adaptabilidad dinámica o facilidad de implementación.

Como trabajo futuro relacionado con los algoritmos de planificación, consiste en el desarrollo de nuevos algoritmos de planificación Grid basados en otros algoritmos de planificación dinámicos clásicos, como CSS, GSS, FSS o FISS. De esta manera, se obtiene una familia de algoritmos de planificación Grid distribuidos que permite analizar el comportamiento de cada algoritmo en diferentes entornos Grid, y así, comprobar cual de estos nuevos algoritmos se adapta mejor a un entorno Grid determinado. Otro aspecto importante que se incluye dentro del trabajo futuro consiste en incorporar el algoritmo GTSS en el kernel del metaplanificador Grid*W*ay. De esta manera toda la planificación se realizaría de manera automática, facilitando así el desarrollo de la aplicación por parte del programador.

Finalmente el trabajo futuro relacionado con la aportación definida por el simulador Grid*W*aySim, consiste en incorporar a este simulador toda la funcionalidad que ofrece en la actualidad el metaplanificador Grid*W*ay, así como adaptar el interfaz del usuario para facilitar el desarrollo de las simulaciones.

Bibliografía

[AAB+02] D. ARNOLD, S. AGRAWAL, S. BLACKFORD, J. DONGARRA, M. MILLER, K. SAGI, Z. SHI, and S. VADHIYAR. Users' guide to netsolve v. Technical Report cs-01-467, University of Tennessee, University of Tennessee, Knoxville, TN, July 2002.

[ABB+02] W. Allock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, and S. Tuecke. GridFTP Protocol Specification. *GridFTP Working Group - The Global Grid Forum*, september 2002.

[ABF+02] W. Allock, J. Bresnahan, I. Foster, L. Liming, J. Link, and P. Plaszczac. GridFTP Update January 2002. *GridFTP Working Group (The Global Grid Forum)*, january 2002.

[ACC+05] Sumalatha Adabala, Vineet Chadha, Puneet Chawla, Renato Figueiredo, José Fortes, Ivan Krsul, Andrea Matsunaga, Mauricio Tsugawa, Jian Zhang, Ming Zhao, Liping Zhu, and Xiaomin Zhu. From virtualized resources to virtual computing grids: the in-vigo system. *Future Gener. Comput. Syst.*, 21(6):896–909, 2005.

[AEH+04] S R Amendolia, F Estrella, W Hassan, T Hauer, D Manset, R McClatchey, D Rogulin, and T Solomonides. Mammogrid: A service oriented architecture based medical grid application. 2004.

[AIS08] Grid Technology Research Center. http://www.aist.go.jp/aist_e/research_units/research_center/grid/grid_main.html, 2008.

[ATN+00] Kento Aida, Atsuko Takefusa, Hidemoto Nakada, Satoshi Matsuoka, Satoshi Sekiguchi, and Umpei Nagashima. Performance evaluation model for scheduling in global computing systems. *Int. J. High Perform. Comput. Appl.*, 14(3):268–279, 2000.

[BBB+91] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The

NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[BLS+03] Rosa M. Badia, Jesús Labarta, Raúl Sirvent, Josep M. Pérez, José M. Cela, and Rogeli Grima. Programming grid applications with grid superscalar. *J. Grid Comput.*, 1(2):151–170, 2003.

[BM02] Rajkumar Buyya and M. Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *CoRR*, cs.DC/0203019, 2002.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BOI08] The BOINC Project. http://boinc.berkeley.edu/index.php, 2008.

[caB08] caBIG: Cancer Biomedical Informatics Grid. https://cabig.nci.nih.gov, 2008.

[Cas01] Henri Casanova. Simgrid: A toolkit for the simulation of application scheduling. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430, Washington, DC, USA, 2001. IEEE Computer Society.

[CBGA01] Anthony T. Chronopoulos, Manuel Benche, Daniel Grosu, and Razvan Andonie. A class of loop self-scheduling for heterogeneous clusters. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 282, Washington, DC, USA, 2001. IEEE Computer Society.

[CFK+98] I. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. *Proceedings of the IPPS/SPDP'98 Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[CFK99] Karl Czajkowski, Ian Foster, and Carl Kesselman. Resource Co-Allocation in Computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 219–228, Redondo Beach, CA, August 1999. IEEE Computer Society Press.

[CON08] Condor. http://www.cs.wisc.edu/condor, 2008.

[CSF08] Community Scheduler Framework. http://www.globus.org/toolkit/docs/4.0/contributions/csf/, 2008.

[CW03] Wahid Chrabakh and Rich Wolski. Gridsat: A chaff-based distributed sat solver for the grid. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 37, Washington, DC, USA, 2003. IEEE Computer Society.

[DRM07] DRMAA WORK GROUP. http://forge.ogf.org/sf/projects/drmaa-wg/, 2007.

[DRM08a] Distributed Resource Management Application API. https://forge.gridforum.org/projects/drmaa-wg, 2008.

[DRM08b] Drmaa implementations. http://www.drmaa.net/w/DrmaaImplementations, 2008.

[EGE08] Enabling Grids for E-sciencE . http://www.eu-egee.org/, 2008.

[EnK+07] M. Ellert, M. Grønager, A. Konstantinov, B. Kónya, J. Lindemann, I. Livenson, J. L. Nielsen, M.Ñiinimäki, O. Smirnova, and A. Wäänänen. Advanced resource connector middleware for lightweight computational grids. *Future Gener. Comput. Syst.*, 23(2):219–240, 2007.

[Fal99] Kevin Fall. Network emulation in the vint/ns simulator. In *ISCC '99: Proceedings of the The Fourth IEEE Symposium on Computers and Communications*, page 244, Washington, DC, USA, 1999. IEEE Computer Society.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of SuperComputer Applications*, 11(2):115–128, 1997.

[FK99] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, 1999.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Intl. J. Supercomputer Applications*, 15(3), 2001.

[FKTT98] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A Security Architecture for Computational Grids. *Proceedings of a 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[Fos02] I. Foster. What is the Grid? A Three Point Checklist. *GRIDtoday*, 1(6), 2002.

[Fos05] Ian Foster. A Globus Toolkit Primer. http://www.globus.org/toolkit/docs/4.0/key/, April 2005. (Draft).

[GGF08] Global Grid Forum. http://www.ggf.org, 2008.

[GL06] GridLab Project. http://www.gridlab.org/, 2006.

[gLi08] gLite: Lightweight Middleware for Grid Computing. http://glite.web.cern.ch/glite/default.asp, 2008.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[GRA01] GRADS. http://www.hipersoft.rice.edu/grads/, 2001.

[GRI00] griphyn. http://www.griphyn.org/, 2000.

[GRM08] GridLab Resource Management System. http://www.gridlab.org/WorkPackages/wp-9/index.html, 2008.

[GSB08] Grid Service Broker. http://www.gridbus.org/broker/, 2008.

[GT08] The globus project. http://www.globus.org, 2008.

[GW08] Grid *W*ay Metascheduler. http://www.gridway.org, 2008.

[HHML04a] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente. Execution of typical scientific applications on globus-based grids. In *ISPDC '04: Proceedings of the Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pages 177–183, Washington, DC, USA, 2004. IEEE Computer Society.

[HHML04b] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Developing grid-aware applications with drmaa on globus-based grids. In *Euro-Par*, pages 429–435, 2004.

[HHML04c] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Drmaa implementation within the gridway framework. Technical report, Global Grid Forum (GGF), September 2004.

[HHML04d] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Embarrassingly distributed and master-worker paradigms on the grid. In Pilar Herrero, María S. Pérez, and Víctor Robles, editors, *SAG*, volume 3458 of *Lecture Notes in Computer Science*, pages 108–119. Springer, 2004.

[HHML05a] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. A grid-oriented genetic algorithm. In Peter M. A. Sloot, Alfons G. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *EGC*, volume 3470 of *Lecture Notes in Computer Science*, pages 315–322. Springer, 2005.

[HHML05b] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Porting of scientific applications to grid computing on gridway. *Scientific Programming*, 13(4):317–331, 2005.

[HHML06a] J. Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Loosely-coupled loop scheduling in computational grids. In *IPDPS*. IEEE, 2006.

[HHML06b] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Benchmarking of a joint irisgrid/egee testbed with a bioinformatics application. *Scalable Computing: Practice and Experience*, (7):25–32, 2006.

[HHML07] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. Ejecución distribuida de bucles en grids computacionales. *RedIRIS: boletín de la Red Nacional de I+D RedIRIS*, (80):52–56, april 2007.

[HHML08] Jose Herrera, Eduardo Huedo, Rubén S. Montero, and Ignacio Martín Llorente. *Distributed Task Scheduling for Physics Fusion Applications*, page 27. March 2008.

[HJ83] R. W. Hockney and C. R. Jesshope. *Parallel Computers: Architecture, Programming, and Algorithms*. pub-ADAM-HILGER, pub-ADAM-HILGER:adr, 1981, 1983.

[HM98] F. W. Howell and R. McNab. Simjava: a discrete event simulation package for Java with applications in computer systems modelling. In *Proceedings of the First International Conference on Web-based Modelling and Simulation*, San Diego, CA, 1998. The Society for Computer Simulation.

[Hol92] John H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[HOL+07] Shannon Hastings, Scott Oster, Stephen Langella, David Ervin, Tahsin M. Kurç, and Joel H. Saltz. Introduce: An open source toolkit for rapid development of strongly typed grid services. *J. Grid Comput.*, 5(4):407–427, 2007.

[HSF92] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn. Factoring: a method for scheduling parallel loops. *Commun. ACM*, 35(8):90–101, 1992.

[IMO+04] H. Imade, R. Morishita, I. Ono, N. Ono, and M. Okamoto. A Grid-oriented Genetic Algorithm Framework for Bioinformatics. *New Generation Computing*, 22(2):177–186, 2004.

[Joh81] John J. Grefenstette. Parallel adaptive algorithms for function optimization. Technical Report CS-81-19, Vanderbilt University, Computer Science Department, Nashville, TN, 1981.

[JSD08] Job Submission Description Language. http://forge.ogf.org/sf/projects/jsdl-wg/, 2008.

[KMHA06] Hartmut Kaiser, Andre Merzky, Stephen Hirmer, and Gabrielle Allen. The SAGA C++ Reference Implementation. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'06) - Library-Centric Software Design (LCSD'06)*, Portland, OR, October, 22-26 2006.

[KTF02] N. T. Karonis, B. Toonen, and I. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. 2002.

[LAM08] LAM/MPI. http://www.lam-mpi.org, 2008.

[Lil94] David J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, 27(2):13–26, 1994.

[LMT+01] C. Lee, S. Matsuoka, D. Talia, A. Sussmann, M. uller, G. Allen, and J. Saltz. A grid programming primer. Global Grid Forum, August 2001.

[LSF01] Load Sharing Facility. http://www.platform.com/products/wm/LSF/, 2001.

[Man88] B. B. Mandelbrot. *Fractal Geometry of Nature*. W.H. Freeman & Co., August 1988.

[MHL06] R. S. Montero, E. Huedo, and I. M. Llorente. Benchmarking of high throughput computing applications on grids. *Parallel Comput.*, 32(4):267–279, 2006.

[MOX99] MOSIX. www.mosix.cs.huji.ac.il, 1999.

[MPI98] MPI 2.0. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, 1998.

[MPI05] MPICH. http://www-unix.mcs.anl.gov/mpi/mpich1/, 2005.

[MPI08] MPI Forum. http://www.mpi-forum.org/, 2008.

[NG08] NorduGrid. http://www.nordugrid.org/, 2008.

[NP99] M. Nowostawski and R. Poli. Parallel Genetic Algorithm Taxonomy. In *Proceedings of the Third International Conference on Knowledge-based Intelligent Information Engineering Systems KES'99*, pages 82–92. IEEE Computer Society, 1999.

[NSJ04] Bill Nitzberg, Jennifer M. Schopf, and James Patton Jones. Pbs pro: Grid computing and scheduling attributes. pages 183–190, 2004.

[NUG00] nug30. http://www-unix.mcs.anl.gov/metaneos/nug30/, 2000.

[OGF08] Open Grid Forum. http://www.gridforum.org/, 2008.

[OSG08] Open Science Grid. http://www.opensciencegrid.org/, 2008.

[PBS08] Portable Batch System. http://www.openpbs.org, 2008.

[PD97] Teebu Philip and C.R. Das. Evaluation of loop scheduling algorithms on distributed memory systems. *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, 1997.

[PK87] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[PMG87] C. Pettey, M.Leuze, and J. Grefenstette. Genetic Algorithms and their Implementation. In *Proceedings of the Second International Conference on Genetic Algorithms*, pages 155–161. Morgan Kauffmann Publishers, 1987.

[REA07] reacciun2. http://www.reacciun2.edu.ve/, 2007.

[Res03] Sensima Research. The End of EAI As We Knew It. *GRIDtoday*, Diciembre 2003.

[rMrs92] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic. Efficient matrix multiplication on simd computers. *SIAM J. Matrix Anal. Appl.*, 13(1):386–401, 1992.

[Rob02] Rob F. Van der Wijngaart and Michael Frumkin. NAS Grid Benchmarks Version 1.0. Technical Report NAS-02-005, NASA, NASA Ames Research Center, Moffet Field, CA 94035-100, July 2002.

[Rom02] Mathilde Romberg. The unicore grid infrastructure. *Sci. Program.*, 10(2):149–157, 2002.

[SAG07] SAGA, A Simple API for Grid Applications. http://saga.cct.lsu.edu/index.php, 2007.

[SAM08] Security Association Markup Language (SAML) Specification v.1.0. http://www.oasis-open.org/committees/security/, 2008.

[SC05] Borja Sotomayor and Lisa Childers. *Globus® Toolkit 4: Programming Java Services*. Morgan Kaufmann, December 2005.

[SE91] J.D. Schaffer and L.J. Eshelman. On Crossover as an Evolutionary Viable Strategy. In R.K. Belew and L.B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 61–68. Morgan Kaufmann, 1991.

[SET03] The SETI@home Project. http://setiathome.ssl.berkeley.edu, 2003.

[SGE94] Sun Grid Engine. http://www.sun.com/gridware, 1994.

[SID07] Babel Project. Scientific Interface Definition Language (SIDL). https://computation.llnl.gov/casc/components/babel.html, 2007.

[SLJ+00] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, Kenjiro Taura, and Andrew A. Chien. The microgrid: a scientific tool for modeling computational grids. In *Supercomputing*, 2000.

[Smi03] C. Smith. Open source metascheduling for virtual organizations with the community scheduler framework (CSF). Technical report, Platform Computing, Inc, 2003.

[SNM+02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of gridrpc: A remote procedure call api for grid computing. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, London, UK, 2002. Springer-Verlag.

[Sot03] Borja Sotomayor. Un nuevo paradigma de computación distribuida. abril 2003.

[STRZ05] Mike Surridge, Steve Taylor, David De Roure, and Ed Zaluska. Experiences with GRIA; Industrial Applications on a Web Services Grid. In *E-SCIENCE '05: Proceedings of the First International Conference on e-Science and Grid Computing*, pages 98–105, Washington, DC, USA, 2005. IEEE Computer Society.

[Tan89] Reiko Tanese. Distributed genetic algorithms. In *International Conference on Genetic Algorithms*, pages 434–439, 1989.

[TG08] Enabling Grids for E-sciencE . http://www.teragrid.org/, 2008.

[TMA+99] Atsuko Takefusa, Satoshi Matsuoka, Kento Aida, Hidemoto Nakada, and Umpei Nagashima. Overview of a performance evaluation system for global computing scheduling algorithms. In *HPDC '99: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, page 11, Washington, DC, USA, 1999. IEEE Computer Society.

[TN93] Ten H. Tzen and Lionel M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[TNS+03] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-g: A reference implementation of rpc-based programming middleware for grid computing. *J. Grid Comput.*, 1(1):41–51, 2003.

[TY86] Peiyi Tang and Pen-Chung Yew. Processor self-scheduling for multiple-nested parallel loops. *Proceedings of the 1986 International Conference on Parallel Processing (ICPP'86)*, pages 528–535, 1986.

[VBW04] Srikumar Venugopal, Rajkumar Buyya, and Lyle Winton. A grid service broker for scheduling distributed data-oriented applications on global grids. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 75–80, New York, NY, USA, 2004. ACM.

[WSP97] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: the network weather service. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–19, New York, NY, USA, 1997. ACM.

Yo ya he terminado.....he puesto 88.