UNA METODOLOGÍA DE PROGRAMACIÓN BASADA EN COMPOSICIONES PARALELAS DE ALTO NIVEL (CPANS)

MEMORIA DE TESIS DOCTORAL

Presentada por:
MARIO ROSSAINZ LOPEZ
Para optar al grado de Doctor en Informática

Director de Tesis:
DR. D. MANUEL ISIDORO CAPEL TUÑON
Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

Departamento de Lenguajes y Sistemas Informáticos
Escuela Técnica Superior de Ingeniería Informática
Universidad de Granada
2005

Editor: Editorial de la Universidad de Granada
Autor: Mario Rossainz López
D.L.: Gr. 1613 - 2005
ISBN: 84-338-3581-5

Agradecimientos

A CONACYT-MÉXICO:

Gracias por darme el apoyo moral y económico que me han permitido formarme como un investigador al hacer posible mis estudios doctorales en la Universidad de Granada con la beca otorgada durante los años 1997-2000.

A PROMEP-MÉXICO:

Un agradecimiento muy especial al Programa del Mejoramiento del Profesorado, PROMEP, por su ayuda económica y moral al otorgarme una beca para terminación de tesis doctoral durante el ciclo Enero-Diciembre 2002. Gracias.

A LA BUAP:

A mi Alma Mater, la Benemérita Universidad Autónoma de Puebla donde me formé profesionalmente, dedico este triunfo, agradeciendo de forma especial a la *Academia para el Avance de la Educación* por haberme ayudado a obtener la beca PROMEP, el permiso de superación académica y por la confianza que depositó en mi el *Dr. Rafael Campos Enriquez*, director de la misma. Un agradecimiento especial al *Mtro. Jaime Vázquez López, Vicerrector de docencia*, por el apoyo incondicional que tuve siempre de él y por creer en mí. Agradezco de igual forma todas las facilidades y apoyos otorgados que tuvo para conmigo la *Facultad de Ciencias de la Computación* que es el lugar de trabajo donde llevo a cabo día a día mi quehacer docente y científico.

A LA UNIVERSIDAD DE GRANADA:

Gracias por haberme dado la oportunidad de ser parte integrante de ustedes y de ser ahora, un investigador orgullosamente formado en el Departamento de Lenguajes y Sistemas Informáticos de la E. T. S. de Ingeniería Informática.

A MI DIRECTOR DE TESIS, DR. MANUEL ISIDORO CAPEL TUÑON:

Quien me dio la oportunidad de ver mis estudios doctorales culminados al tenerme la confianza suficiente para depositar en mí un tema de tesis que tomé como una responsabilidad, como un reto y como una meta en mi vida. Mi admiración más profunda y mi gratitud más sincera en reconocimiento a su valiosa ayuda para mi desarrollo científico profesional. Muchísimas gracias Manolo.

A MI ESPOSA:

A ti Marlen, te doy las gracias por ser una esposa admirable, por tu apoyo incondicional, por tus desvelos, sacrificios y por tener mucho que ver en la culminación de este doctorado que a ti dedico.

A MI FAMILIA Y AMIGOS:

Gracias por estar conmigo en los momentos buenos y malos, por el cariño y apoyo recibido y por contagiarme siempre ese deseo de superación. A ustedes dedico este logro.

Índice

ÍNDICE

INTRODUCCIÓN 17

Capítulo I. MARCO TEORICO: Estado del Arte y Antecedentes

I.1. INTRODUCCIÓN 23
I.2. EL PROCESAMIENTO PARALELO 23
I.2.1. Objetivo del Paralelismo 24
I.2.2. El Factor de rendimiento SpeedUp 25
1.2.3. La ley de Amdahl 26
1.2.4. Ciclos Por Instrucción (CPI) 28
I.3. ARQUITECTURAS PARALELAS 28
I.3.1. Arquitecturas SIMD 30
I.3.2. Arquitecturas MISD 32
I.3.3. Arquitecturas MIMD 33
I.4. ALGORITMOS PARALELOS 36
I.5. LENGUAJES DE PROGRAMACION PARALELA 37
I.6. THREADS 38
I.6.1. Características de los Threads 40
I.6.2. Con Threads o sin Threads 40
I.6.3. Programación con Threads 41
I.6.4. La vida de un Thread 42
I.6.5. Los Threads en la Concurrencia 43
I.7. ANTECEDENTES 44
I.7.1. Problemas abiertos 45
I.7.2. El Trabajo previo 47
I.7.3. Requerimientos para el desarrollo de un entorno de programación basado en Objetos Paralelos 49
I.8. OBJETIVOS CIENTIFICOS 50

Capítulo II. COMPOSICIONES PARALELAS DE ALTO NIVEL (CPANS): Análisis y Definición del Modelo

II.1. INTRODUCCIÓN 55

II.1.1. El Paralelismo Estructurado 56
II.1.2. El paradigma de la Orientación a Objetos (OO) 57
II.2. DEFINICIÓN DEL MODELO CPAN 58
II.2.1. Composición del CPAN 60
II.2.2. El CPAN visto como una composición de Objetos Paralelos 61
II.2.2.1. Definición sintáctica de los Objetos Paralelos de un CPAN 62
II.2.2.1.1. INSTANCE_STATE 62
II.2.2.1.2. INSTANCE_METHOD 62
II.2.2.1.3. PRIVATE_METHOD 63
II.2.2.1.4. SCHEDULING_PART 63
II.2.2.2. Los tipos de Comunicación en los Objetos Paralelos 63
II.2.2.2.1. El modo de Comunicación Futuro Asíncrono 64
II.2.2.2.2. Gramática libre del contexto para la definición de clases PO 66
II.3. DEFINICIÓN SINTÁCTICA Y SEMÁNTICA DE LAS CLASES BASE DE UN CPAN 67
II.3.1. La clase abstracta ComponentManager 68
II.3.2. La clase Abstracta ComponentStage 70
II.3.3. La clase Concreta ComponentCollector 71
II.3.4. Las restricciones de sincronización MaxPar, Mutex y Sync 72
II.3.4.1. MaxPar 72
II.3.4.2. Mutex 73
II.3.4.3. Sync 74
II.4. CARACTERÍSTICAS IMPORTANTES DEL MODELO CPAN 75

Capítulo III. DISEÑO DE LOS CPANS: Construcción de los patrones Farm, Pipe y TreeDV como CPANS

III.1. INTRODUCCIÓN 79
III.2. EL CPAN PIPE 80
III.2.1. La técnica del PipeLine 80
III.2.2. Representación del Pipeline como un CPAN 82
III.2.3. Definición Sintáctica y Semántica del CPAN Pipe 83
III.2.4. Uso del CPAN Pipe 84
III.3. EL CPAN FARM 87
III.3.1. La técnica del Farm 88

III.3.2. Representación del Farm como un CPAN 90
III.3.3. Definición Sintáctica y Semántica del CPAN Farm 90
III.3.4. Uso del CPAN Farm 91
III.4. EL CPAN TREEDV 94
III.4.1. La técnica de Divide y Vencerás 94
III.4.2. Representación de la técnica Divide y Vencerás como un CPAN 97
III.4.3. Definición Sintáctica y Semántica del CPAN TreeDV 98
III.4.4. Uso del CPAN TreeDV 100
III.5. RESUMEN METODOLÓGICO DEL USO DE UN CPAN 102

Capítulo IV. CASO DE ESTUDIO: Paralelización de la técnica de Ramificación y Poda como un CPAN

IV.1. INTRODUCCIÓN 107
IV.2. LA TECNICA DE RAMIFICACIÓN Y PODA 108
IV.2.1. El Algoritmo 110
IV.3. EL PROBLEMA DEL AGENTE VIAJERO (TSP) 114
IV.3.1. Solución del TSP usando la técnica de Ramificación y Poda 114
IV.4. PARALELIZACIÓN DE LA TECNICA DE RAMIFICACIÓN Y PODA COMO UN CPAN 116
IV.4.1. Representación de la técnica de Ramificación y Poda como un CPAN117
IV.4.2. Definición Sintáctica y Semántica del Cpan FarmBB 119
IV.2.3. Uso del CPAN FarmBB en la solución del TSP 123

Capítulo V. ANALISIS DEL RENDIMIENTO DE LOS CPANS: Evaluación de los CPANS Farm, Pipe, TreeDV y FarmB&B

V.1. INTRODUCCIÓN 131
V.1.1. Configuración de las colas batch NQE 131
V.2. Ejecución de los CPANS en el sistema de colas de Karnak 132
V.2.1. Metodología de trabajo para la evaluación de los CPANS 133
V.3. EL RENDIMIENTO DE LOS CPANS 134
V.3.1. Resultados de rendimiento del Cpan Farm 134
V.3.1.1. Representación gráfica del rendimiento del Cpan Farm 136
V.3.2. Resultados de rendimiento del Cpan Pipe 137
V.3.2.1. Representación gráfica del rendimiento del Cpan Pipe 139

V.3.3. Resultados de rendimiento del Cpan TreeDV 140
V.3.3.1. Representación gráfica del rendimiento del Cpan TreeDV 142
V.3.4. Resultados de rendimiento del Cpan FarmBB 143
V.3.4.1. Representación gráfica del rendimiento del Cpan FarmBB 145
V.4. Porcentajes de código secuencial y paralelo de los CPANS para la ley de Amdahl 146
V.5. CONCLUSIONES DEL ANÁLISIS DE RENDIMIENTO DE LOS CPANS 147

Capítulo VI. CONCLUSIONES

VI. 1. CONCLUSIONES 151
VI.2. APORTE DE LA TESIS A LA CIENCIA COMPUTACIONAL 152
VI.3. TRABAJOS FUTUROS 153

Apéndice A. LA LIBRERÍA DE CLASES DE LOS CPANS: Descrip´ción de la librería de clases que constituyen los CPANS propuestos

A.1. INTRODUCCIÓN 157
A.2. EL GRUPO DE LAS CLASES BASE DE LA LIBRERÍA DE LOS CPANS 158
A.2.1. La clase base Object 158
A.2.2. El archivo de encabezado Util 159
A.2.3. La clase base CHilo 161
A.2.4. La clase base ComponentCollector 162
A.2.5. La clase base ComponentStage 163
A.2.6. La clase base ComponentManager 164
A.3. EL GRUPO DE LAS CLASES QUE DEFINEN LOS CPANS FARM, PIPE, TREEDV Y FARMBB 165
A.3.1. La clase concreta FarmManager 165
A.3.2. La clase concreta FarmStage 166
A.3.3. La clase concreta PipeManager 166
A.3.4. La clase concreta PipeStage 167
A.3.5. La clase concreta TreeDVManager 167
A.3.6. La clase concreta TreeDVStage 168

A.3.7. La clase concreta FarmBBManager 169
A.3.8. La clase concreta FarmBBStage 169
A.4. EL GRUPO DE LAS CLASES DE LOS OBJETOS ESCLAVOS................ 171
A.5. JERARQUÍA DE HERENCIA DE LA LIBRERÍA DE LOS CPANS.......... 172

Apéndice B. SISTEMA SGI ORIGIN 2000: El Sistema Paralelo del Centro Europeo de Paralelismo de Barcelona

B.1. INTRODUCCIÓN 177
B.2. LOS PROCESADORES MIPS R10000 179
B.3. LA MEMORIA CACHE DEL ORIGIN 2000 180
B.4. EL SISTEMA OPERATIVO DEL ORIGIN 2000 181

BIBLIOGRAFIA Y REFERENCIAS 184

Introducción

INTRODUCCIÓN

En los tiempos actuales la construcción de sistemas paralelos y concurrentes tiene cada vez menos limitantes y su existencia ha hecho posible obtener gran eficiencia en el procesamiento de datos. Tales sistemas se han expandido a muchas áreas de la Ciencia Computacional e Ingeniería. Como es sabido, existen infinidad de aplicaciones que, utilizando máquinas con un solo procesador, tratan de obtener el máximo rendimiento del sistema al resolver un problema. Sin embargo cuando el sistema no puede proporcionar el rendimiento esperado, una posible solución es optar por aplicaciones, arquitecturas y estructuras de procesamiento paralelo y concurrente. El procesamiento paralelo es por tanto una alternativa al procesamiento secuencial. En la computación secuencial un procesador realiza una operación a la vez en contra de la computación paralela donde varios procesadores cooperan para resolver un determinado problema, lo cual reduce el tiempo de cálculo ya que varias operaciones pueden llevarse a cabo simultáneamente. Desde el punto de vista práctico, hoy en día existe suficiente justificación para llevar a cabo investigaciones dentro del procesamiento paralelo y áreas a fines (concurrencia, sistemas distribuidos, sistemas de tiempo real, etc.). Parte importante de estas investigaciones son los algoritmos paralelos, metodologías y modelos de programación paralela que se están desarrollando. El procesamiento paralelo envuelve muchos factores que van desde las arquitecturas paralelas y algoritmos paralelos hasta los lenguajes de programación paralela y análisis de rendimiento por mencionar algunos.

La presente tesis centra su atención en la programación paralela estructurada, proponiendo una metodología de programación paralela bajo el paradigma de la orientación a objetos para resolver problemas paralelizables con alto grado de rendimiento. Partiendo del trabajo realizado para obtener la suficiencia investigadora presentado en Julio de 1999[1], redefiniendo y actualizando la investigación, el presente documento propone una *"Metodología de Programación Basada en Composiciones Paralelas de Alto Nivel (CPANS)"*. Los *CPANS* son patrones paralelos de comunicación bien definidos y lógicamente estructurados que, una vez identificados en términos de sus componentes y de su comunicación, éstos pueden llevarse a la práctica y estar disponibles como abstracciones de alto nivel en las aplicaciones del usuario dentro de

[1] Suficiencia Investigadora obtenida al presentar el trabajo titulado "Una metodología de Programación Paralela en JAVA"

un entorno o ambiente orientado a objetos. Las estructuras de interconexión de procesadores más comunes son los *pipelines o cauces*, los *farms o granjas* y los *trees o árboles*. La implementación de los *CPANS* se ha llevado a cabo a través de la creación de una librería de clases que ofrece al usuario los tres patrones paralelos de comunicación antes mencionados en su forma de *Composiciones Paralelas de Alto Nivel*, es decir, la librería ofrece al usuario la posibilidad de incorporar a sus aplicaciones los patrones paralelos de comunicación *Cpan Farm*, *Cpan FarmBB* (que implementa la técnica de Ramificación y Poda), *Cpan Pipe* y *Cpan TreeDV* (que implementa la técnica de Divide y Vencerás a través de árboles binarios), para la solución de problemas paralelizables. Para su desarrollo, se utilizó el lenguaje *C++* y el estándar *POSIX Threads* como las herramientas de programación que proporcionaron las características necesarias y suficientes dentro de la orientación a objetos y del paralelismo para hacer realidad la librería. Estas características son:

- La capacidad de invocación de métodos de los objetos que conforman los *CPANS*, para la implementación de los modos de comunicación síncrono, asíncrono y futuro asíncrono.
- La implementación del paralelismo interno de los objetos de los *CPANS* usando el mecanismo de hilos del *Posix Threads*, para permitir a un objeto el atender varias peticiones de servicio de forma concurrente.
- La disponibilidad de mecanismos de sincronización (*Sync, MuTex y MaxPar*), para producir peticiones paralelas de servicio hacia los objetos que conforman los *CPANS,* para que puedan gestionar varios flujos de ejecución concurrentemente y garanticen la consistencia de sus datos.
- La disponibilidad de mecanismos flexibles de control de tipos para asociar tipos de forma dinámica a los parámetros de los métodos de los objetos de los *CPANS* y poder gestionar tipos de datos genéricos.
- La transparencia de la distribución de aplicaciones paralelas garantizada por el lenguaje, para proporcionar la portabilidad de los *CPANS* desde un sistema centralizado (inicialmente en entorno LINUX y WINDOWS), a un sistema distribuido (el sistema paralelo *Origin 2000 Silicon Graphics* del Centro Europeo de Paralelismo de Barcelona CEPBA, con 2, 4, 8, 16 y 32 procesadores, bajo un Sistema Operativo *UNIX IRIX 6.5*) sin necesidad de afectar el código fuente.

- El enfoque basado en *patrones como clases y objetos paralelos* adoptado para implementar los *CPANS* y resolver el problema denominado *PPP* (Programmability, Portability, Performance).

Se propone entonces un método de programación basado en Composiciones Paralelas de Alto Nivel o *CPANS* mediante una librería de clases de utilidad dentro de la Programación Paralela Orientada a Objetos que proporciona al programador los patrones paralelos más comúnmente utilizados para poder explotar los mecanismos de generalización por herencia y parametrización, de forma que pueda definir nuevos patrones según el modelo del *CPAN* tomando como base los ya existentes. Se muestra el modelo de programación desarrollado, el análisis, diseño e implementación de los algoritmos que conforman los *CPANS*, así como de los algoritmos utilizados para probar el funcionamiento y rendimiento de la librería, algoritmos de ordenación, búsqueda y de optimización (analizando el problema del Agente Viajero o TSP, tratado como un caso de estudio particular).

La presente memoria de tesis se conforma de los siguientes capítulos que muestran el desarrollo de la investigación:

- *Capítulo I. MARCO TEORICO: Estado del Arte y Antecedentes.* Capítulo centrado en los conceptos teóricos sobre los cuales se sustenta la tesis, como lo es el procesamiento paralelo, las arquitecturas paralelas de computadoras y los Threads. Se muestran los antecedentes, los problemas abiertos y el trabajo previo o relacionado con el tema en cuestión: las propuestas de Brinch-Hannsen, Rabhi, Corradi, etc.
- *Capítulo II. COMPOSICIONES PARALELAS DE ALTO NIVEL (CPANS): Análisis y Definición del Modelo.* Conformado por temas como el paralelismo estructurado y el paradigma de la orientación a objetos, para dar paso a la definición del modelo *CPAN*, su definición sintáctica a través de Objetos Paralelos (PO), los tipos de comunicación de los PO, la gramática libre de contexto de los PO, los templates que definen los componentes de un *CPAN*, así como su definición sintáctica y semántica de cada uno de ellos (objetos manager, stages, collector y objetos esclavos).
- *Capítulo III. DISEÑO DE LOS CPANS: Construcción de los patrones Farm, Pipe y TreeDV como CPANS.* Se muestran los diseños de los patrones paralelos de

comunicación, *Farm, Pipe y Tree* (técnica de Divide y Vencerás) como Composiciones Paralelas de Alto Nivel. Para cada diseño se explica: la técnica del patrón paralelo, su representación como un *CPAN*, sus definiciones sintácticas y semánticas, así como su uso en la solución de un problema ejemplo.

- *Capítulo IV. CASO DE ESTUDIO: Paralelización de la técnica de Ramificación y Poda como un CPAN.* Capítulo que representa el caso de estudio de la técnica de Ramificación y Poda para ser implementada como un *CPAN*. Se inicia con una explicación de la técnica y de sus algoritmos secuenciales que la implementan. Se explican los tipos de problemas que pueden ser resueltos con ésta técnica, poniendo especial interés en el Problema del Agente Viajero o TSP. Se continúa con la explicación de la paralelización de la técnica bajo el modelo de los *CPANS* mostrando su diseño, su definición sintáctica, semántica y su uso en la solución del problema del TSP.
- *Capítulo V. ANÁLISIS DEL RENDIMIENTO DE LOS CPANS: Evaluación de los Cpans Farm, Pipe, TreeDV y FarmBB.* Este es el capítulo donde se dan a conocer los resultados de las evaluaciones de los *CPANS* propuestos: *Farm, Pipe, TreeDV* (técnica de Divide y Vencerás) y *FarmBB* (técnica de Ramificación y Acotación) en cuanto a aceleración (speedup), eficiencia, costo y uso.
- *Capítulo VI. CONCLUSIONES.* Se presentan las conclusiones a las que se han llegado, los aportes de la tesis a las Ciencias de la Computación y los trabajos futuros por hacer para robustecer el presente trabajo.
- *Apéndice A. LA LIBRERÍA DE CLASES DE LOS CPANS.* Anexo que explica la constitución de los *CPANS* como una librería de clases formada por: el grupo de las clases base de los *CPANS*, el grupo de las clases concretas que representan los *Cpans Farm, Pipe, TreeDV y FarmBB*, el grupo de las clases ejemplos y la jerarquía de clases conformada.
- *Apéndice B. SISTEMA SGI ORIGIN 2000: El Sistema Paralelo del Centro Europeo de Paralelismo de Barcelona.* Anexo que muestra las características principales del sistema paralelo Origin 2000 del CEPBA de la Universidad de Barcelona, que se utilizó para ejecutar los *CPANS* y obtener las medidas de rendimiento y eficiencia mostradas en el capítulo V. Se describe su arquitectura y el uso de su sistema de colas para la ejecución de jobs paralelos.

Capítulo I:

MARCO TEORICO:
Estado del Arte y Antecedentes

I.1. INTRODUCCIÓN

En este capítulo se exponen de manera global los principales conceptos del paralelismo que tienen relación con la presente tesis. Se presenta el marco teórico sobre el cual descansa el trabajo realizado que consta del estado del arte de la paralelización, los antecedentes de la propuesta y los objetivos que se persiguen.

Se hablará del concepto de procesamiento paralelo para exponer posteriormente las arquitecturas paralelas como parte fundamental de dicho concepto. Se abordará el tema de algoritmos paralelos, y se hablará de los lenguajes de programación que nos permiten crear aplicaciones paralelas para continuar con la explicación de un concepto fundamental en el presente trabajo, el concepto de *Thread*, explicando su modelo, sus características, su programación, así como su ciclo de vida y terminar con los antecedentes y objetivos que dan pauta a la investigación descrita en la presente memoria de tesis.

I.2. EL PROCESAMIENTO PARALELO

El procesamiento paralelo ha provocado un tremendo impacto en muchas áreas donde las aplicaciones computacionales tienen cabida. Un gran número de aplicaciones computacionales en la ciencia, ingeniería, comercio o medicina requieren de una alta velocidad de cálculo para resolver problemas de visualización, bases de datos distribuidas, simulaciones, predicciones científicas, etc. Estas aplicaciones envuelven procesamiento de datos o ejecución de un largo número de iteraciones de cálculo. El procesamiento paralelo es uno de los enfoques computacionales que hoy en día puede ayudarnos ha computar éstos cálculos de una manera más viable. Este incluye el estudio de arquitecturas paralelas y algoritmos paralelos.

Las tecnologías del procesamiento paralelo son actualmente explotadas de forma comercial, principalmente para el trabajo en el desarrollo de herramientas y ambientes. El desarrollo de esta área en las redes de computadoras ha dado lugar a la llamada *Computación Heterogénea* [BAC99], que proporciona un ambiente donde la aplicación paralela es ejecutada utilizando diferentes computadoras secuenciales y paralelas en las cuales la comunicación se lleva a cabo a través de una red inteligente. Este enfoque proporciona alto rendimiento.

Existen muchos factores que contribuyen al rendimiento de un sistema paralelo, entre ellos están la integración entre procesos, es decir, el tener múltiples procesos activos simultáneos resolviendo un problema, la arquitectura del hardware (arquitectura superescalar, de vector o pipelinizada, etc.), y enfoques de programación paralela, propuestos para comunicar y sincronizar los procesos.

I.2.1. Objetivo del Paralelismo

El objetivo del paralelismo es conseguir ejecutar un programa en menos tiempo utilizando varios procesadores. La idea central es dividir un problema grande en varios más pequeños y repartirlos entre los procesadores disponibles.

A veces ocurre que no es posible hacer una paralelización de todo el programa, debido a que aparecen elementos que no se pueden paralelizar. Otras veces ocurre que la repartición del programa entre los procesadores no se hace de manera equitativa, lo que se traduce en una ganancia muy pobre de tiempo.

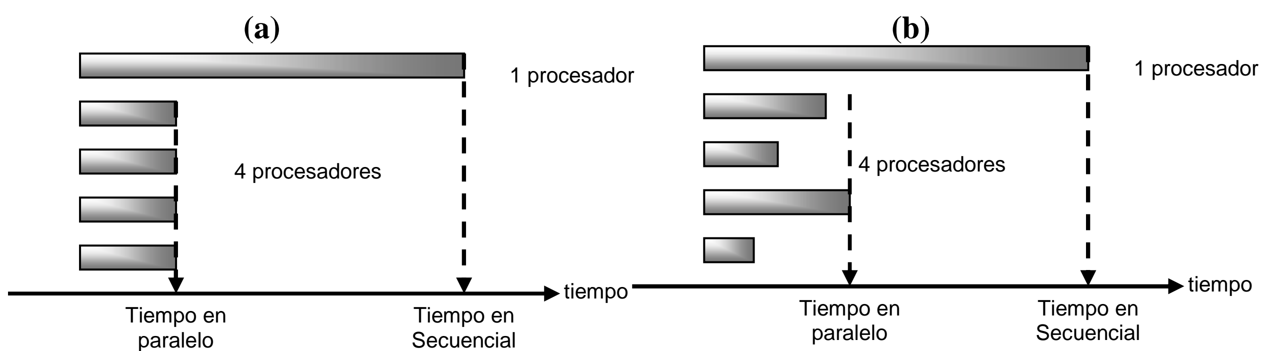


Fig. I.1. (a) Esquema ideal y (b) esquema real, del resultado de la paralelización

Si suponemos que todos los procesadores llevan a cabo el mismo tipo y número de operaciones, idealmente la ganancia de tiempo debería ser la que se muestra en la fig. I.1.(a), sin embargo realmente ocurre con frecuencia lo mostrado en la fig. I.1(b).

Un ejemplo de paralelización puede ser un ciclo de 100 iteraciones que es repartido entre 4 procesadores. Cada uno de ellos ejecutará 25 repeticiones del ciclo original, Fig. I.2.

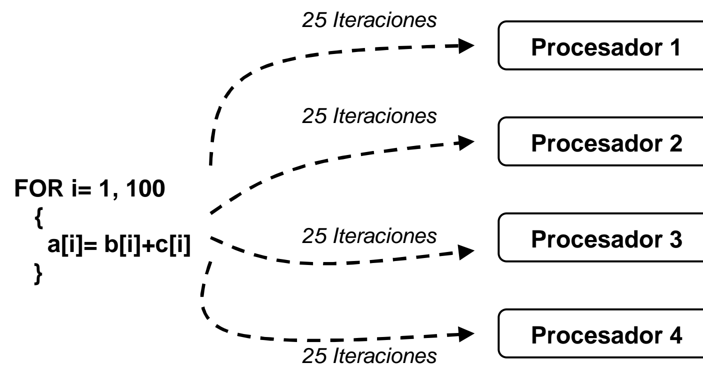


Fig. I.2. Reparticiones de las iteraciones de un ciclo en 4 procesadores

I.2.2. El Factor de rendimiento SpeedUp

También conocido como *aceleración*, el factor *SpeedUp* es una medida del rendimiento entre un sistema multiprocesador y un sistema de procesador simple [WIL99]. Dicho de otra forma, el factor *SpeedUp* mide cuánto efectiva resulta ser la paralelización de un programa fuente, frente a la versión secuencial. Para ello es necesario medir los tiempos de ejecución de cada versión y con ellos calcular la relación entre los tiempos o *SpeedUp,* que se define como:

T_{sec} = Tiempo de ejecución del programa secuencial

T_{par} = Tiempo de ejecución del programa en paralelo

$$SpeedUp = \frac{T_{sec}}{T_{par}}$$

Para hacer la comparación de una solución paralela con una solución secuencial de un problema dado, se debe utilizar el mejor algoritmo secuencial conocido, es decir, el más rápido, que pueda ejecutarse en un simple procesador. El algoritmo utilizado para la implementación paralela debe ser y es usualmente diferente.

Si se tiene en cuenta que la ejecución paralela se ha llevado a cabo con *P* procesadores, el *SpeedUp* ideal es obviamente *P*, y este rendimiento empeora mientras decrece. Por ejemplo, suponiendo que la ejecución paralela de una aplicación se realiza con 4 procesadores donde T_{par} = 3hrs. y la misma aplicación ejecutándose de forma secuencial tarda T_{sec} = 8hrs., el *SpeedUp* calculado es de:

$$SpeedUp = \frac{8h}{3h} = 2.66$$

Pero, ¿qué tan bueno es éste *SpeedUp?* Eso depende. El resultado no se acerca a 4 que sería el mejor *SpeedUp* que se pudiera encontrar, por lo que no parecería aceptable. Sin embargo antes de responder a esta pregunta se deben tener en cuenta otros factores, por ejemplo, el tiempo utilizado en paralelizar el programa, la calidad del código fuente, la naturaleza del algoritmo implementado, la porción del código secuencial, etc.

Entonces, ¿cuál es el máximo *SpeedUp* real que se puede alcanzar? La respuesta esta en la llamada *ley de Amdahl*.

1.2.3. La ley de Amdahl

Si asumimos que podemos tener algunas partes de un programa que son ejecutadas solo en un procesador, es decir, si asumimos que en todo programa hay una fracción no paralelizable, esto supone una cota superior al *speedUp* que pudiera tener el programa ejecutándose en paralelo.

Si la fracción del cómputo que no puede ser dividida en tareas concurrentes se denota como f y el número de procesadores como p, el factor de *speedUp* máximo esta dado por la *ley de Amdahl* denotada como $S(p)$ y definida como :

$$S(p)= \frac{p}{1+(p-1)f}$$

O equivalentemente

$$S(p)= \frac{1}{f+[(1-f)/p]}$$

Por ejemplo, si se tiene una programa donde la fracción del código paralelo equivale a un 75% y la aplicación se ejecuta usando 4 procesadores, la *ley de Amdahl* dice que con $p=4$ *y* $f=0.25$, la cota superior al *SpeedUp* del programa ejecutándose en paralelo es de:

$$S(4)= \frac{4}{1+(4-1)0.25} =2.28$$

O equivalentemente

$$S(4)= \frac{1}{0.25+[(1-0.25)/4]} =2.28$$

Es decir: *SpeedUp* $\leq$ *S(p).*

La fig. I.3. muestra la gráfica de *S(p)* con valores de *p=4,8,12,16 y 20* procesadores y valores para *f=0, 0.05, 0.10 y 0.20* [WIL99].

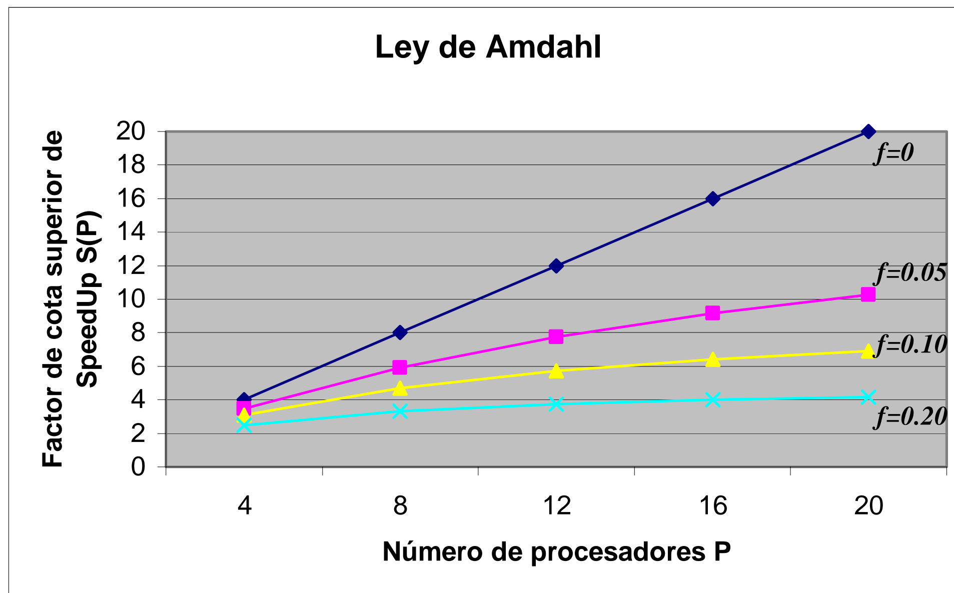


Fig I.3. La ley de Amdahl para varios procesadores y fracciones de código no paralelizado

Se puede observar que de hecho el *speedUp* mejora a medida que se aumenta el número de procesadores. Sin embargo, la fracción del cómputo que es ejecutada por los procesos concurrentes necesita ser una fracción sustancial del cómputo global si se quiere tener un incremento significativo en la velocidad de ejecución. Incluso con un infinito número de procesadores, el máximo *SpeedUp* que se puede alcanzar esta limitado a *1/f*, esto es:

$$\lim_{p \to \infty} S(p) = \frac{1}{f}$$

Por ejemplo, con solo el 0.05 de cómputo serial, el máximo *SpeedUp* es de 20, independientemente del número de procesadores.

La *ley de Amdahl* finalmente nos dice que no se debe descuidar la parte del programa que no ha podido ser paralelizada, ya que limitará la ganancia en tiempo conseguida por la parte que si pudo serlo.

1.2.4. Ciclos Por Instrucción (CPI)

CPI es una medida utilizada para reflejar la calidad de uso del procesador por parte de un programa. *CPI* se define como:

$$CPI = \frac{\#\ ciclos}{\#\ instrucciones}$$

En general, si por cada ciclo se ejecuta una instrucción, el *CPI* que se obtiene es *1*, lo cual se considera bueno. Determinado tipo de procesadores (conocidos como *superescalares)* son capaces de ejecutar más de una instrucción de un programa por ciclo. Esto supone obtener un *CPI* inferior a *1*. Por ejemplo, si se utiliza un procesador capaz de ejecutar hasta 4 instrucciones por ciclo, se podría llegar a conseguir un $CPI \geq 0.25$.

I.3. ARQUITECTURAS PARALELAS

Aun cuando el campo de las computadoras paralelas esta en constante cambio, y aparecen nuevos modelos de paralelismo en la gran diversidad de sistemas que hay hoy en día, se sigue aceptando una división genérica dentro de los multiprocesadores: *Memoria compartida, Memoria distribuida y Memoria compartida distribuida* englobada dentro de la *Memoria compartida*.

- Memoria Distribuida: Se habla de una computadora donde la memoria del sistema está distribuida entre todos los procesadores del sistema. Fig. I.4. En este tipo de computadoras paralelas se trabaja con el modelo de programación de *paso de mensajes*. La comunicación entre procesadores es explícita, es decir, el programador es quien la define [CEPBA].

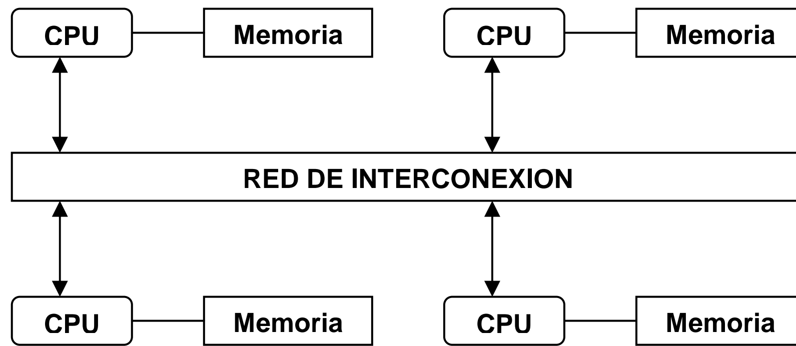


Fig. I.4. Esquema de un multiprocesador de memoria distribuida

- Memoria Compartida: Se habla de una multicomputadora donde todos los procesadores acceden a toda la memoria, es decir, varios procesadores pueden acceder a la misma localidad de memoria y en el mismo instante de tiempo. Según sea uniforme o no el tiempo de acceso a memoria por parte de los procesadores, se tendrán dos tipos de memoria. La memoria *NUMA (Non-Uniform Memory Access)*, donde los procesadores comparten la memoria del sistema de forma que dependiendo de donde esté ubicado el procesador puede tardar más o menos en acceder a la memoria, fig. I.5(a). Y la memoria *UMA (Uniform Memory Access)*, donde los procesadores comparten la memoria que se encuentra distribuida por todo el sistema ofreciendo así un acceso uniforme, fig. I.5(b). [CEPBA]

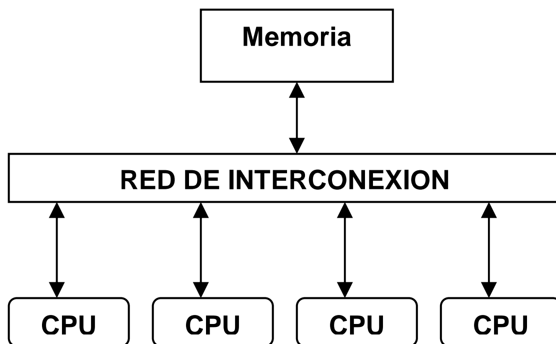


Fig. I.5. (a) Modelo NUMA

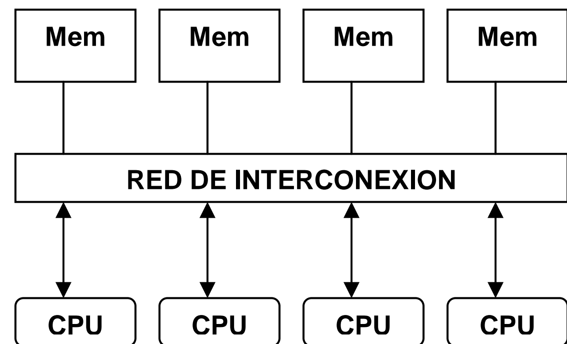


Fig. I.5. (b) Modelo UMA

- Memoria Compartida Distribuida: Se habla de una computadora donde la memoria esta físicamente repartida entre los nodos del sistema (compuestos cada uno por 2 procesadores), de tal forma que solo hay un espacio de direcciones y todos los procesadores pueden acceder a toda la memoria, fig. I.6. Como el acceso a la memoria de un mismo nodo es más rápido que el acceso a la memoria de otro nodo, se debe buscar la manera de aprovechar al máximo las localidades de los datos.

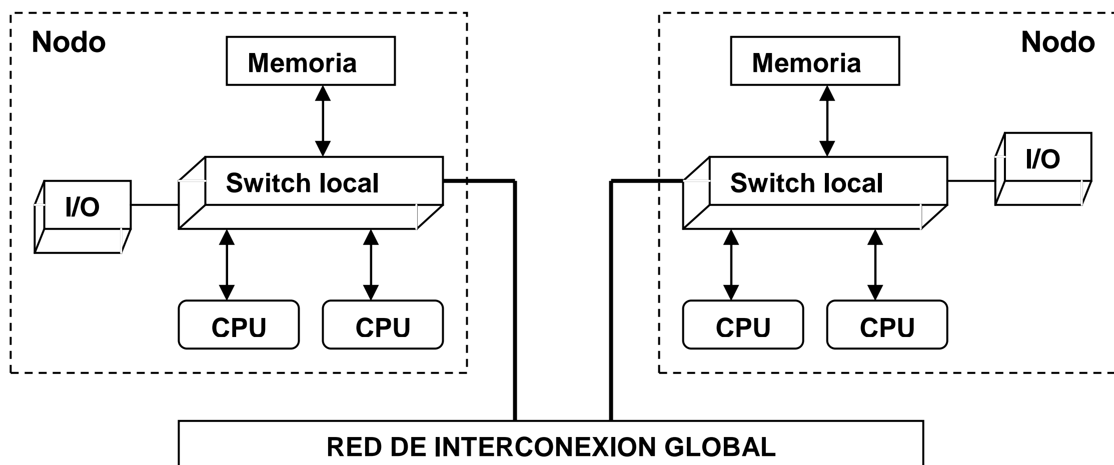


Fig. I.6. Modelo DSM de Memoria Compartida Distribuida

La clasificación de Michael Flynn divide a las arquitecturas paralelas en la familia de las computadoras *SIMD*, *MISD* y *MIMD*. La fig. I.7. muestra su taxonomía.

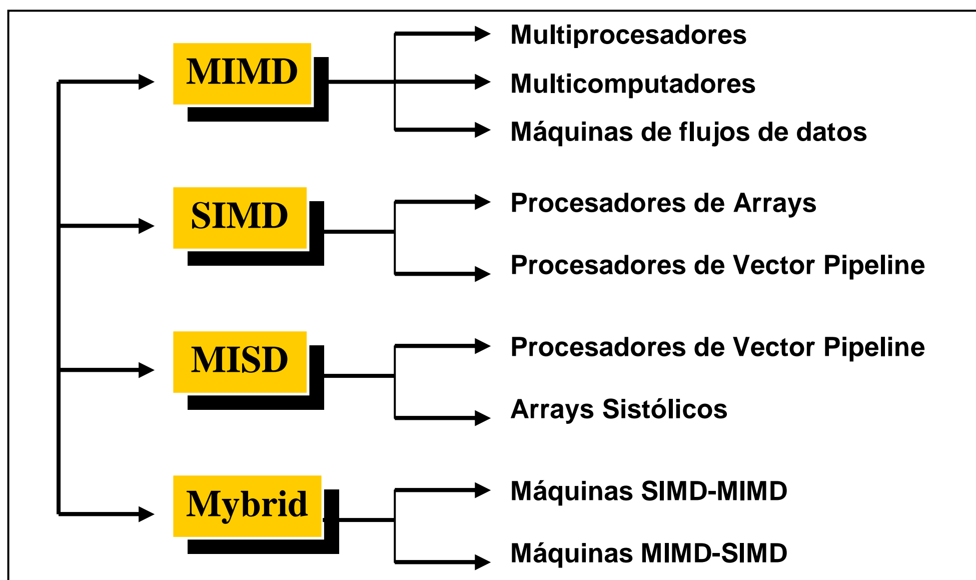


Figura I.7. Taxonomía de las arquitecturas de procesamiento paralelo

I.3.1. Arquitecturas SIMD

SIMD (Single Instruction Stream, Multiple Data Stream): En una máquina *SIMD*, varios elementos procesados se supervisan por una unidad de control. Todas las unidades de procesamiento reciben la misma instrucción desde la unidad de control,

pero operan con diferentes conjuntos de datos, los cuales provienen de distintos flujos de datos, fig. I.8.

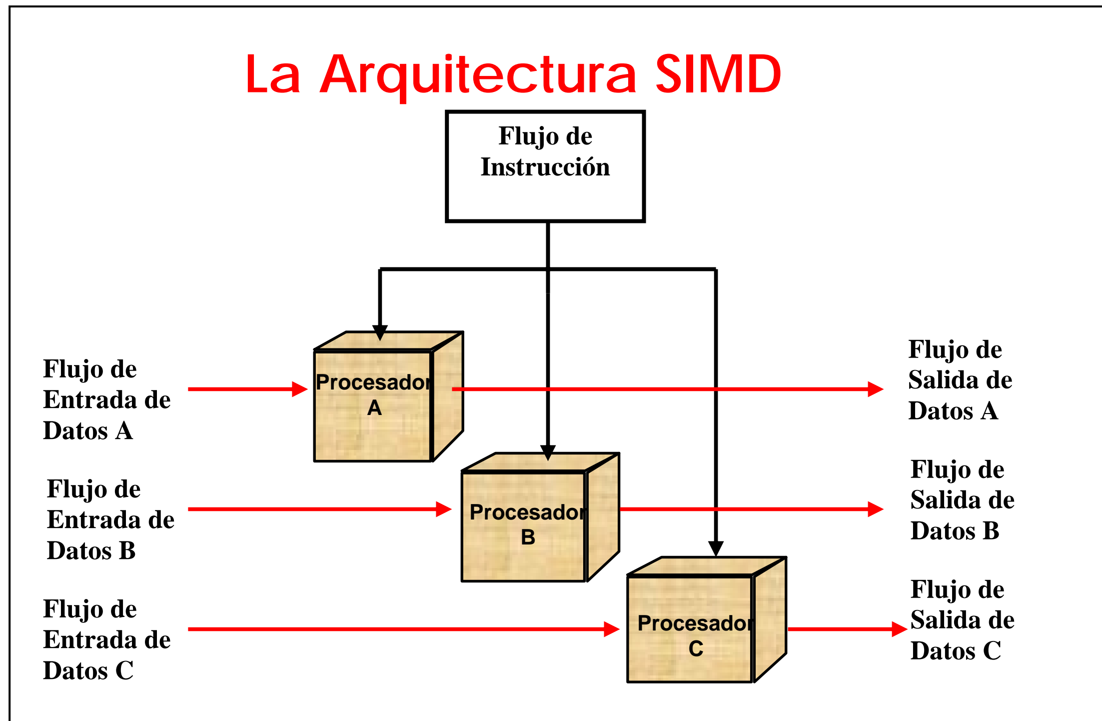


Figura I.8. Modelo de una Arquitectura SIMD

Las características principales de este tipo de máquinas paralelas son que:

- Distribuyen el procesamiento sobre una larga cantidad de hardware.
- Operan concurrentemente con muchos elementos de datos diferentes.
- Realizan el mismo cálculo en todos los elementos de datos.

Cada unidad de procesamiento ejecuta la misma instrucción al mismo tiempo, y los procesadores operan de manera síncrona. El potencial de S*peedUp* de las máquinas *SIMD* es proporcional a la cantidad de hardware disponible. El paralelismo hace que las máquinas *SIMD* desarrollen altas velocidades.

Algunas características generales de las computadoras *SIMD* son:

- Menos hardware, debido a que utilizan solo una unidad global de control.
- Menos memoria, ya que solo se necesita una copia de las instrucciones colocada en la memoria del sistema.

- Flujo de instrucciones simples y sincronización implícita del procesamiento de elementos, lo que hace que una aplicación *SIMD* sea entendible, fácil de programar y fácil de trazar.
- Instrucciones de control de flujo y operaciones escalares que son comunes a todos los elementos procesados que pueden ser ejecutados en la unidad de control, mientras los procesadores están ejecutando otras instrucciones.
- Necesidad de mecanismos de sincronización sobre los procesadores, después de cada ciclo de ejecución de una instrucción.
- Menos costo ya que solo se necesita un decodificador de instrucción simple en la unidad de control.

Ejemplos de computadoras *SIMD* incluyen la ILLIAC IV, MPP, DAP, CM-2, MasPar MP-1 y MasPar MP-2 [SEL99].

I.3.2. Arquitecturas MISD

MISD (Multiple Instruction Stream, Single Data Stream): Las máquinas de esta categoría pueden ejecutar varios programas distintos con el mismo *item* de datos. La arquitectura puede ser ilustrada en dos categorías:

1. Una clase de máquinas que requieren de distintas unidades de procesamiento que pueden recibir distintas instrucciones para ser ejecutadas con los mismos datos. Sin embargo, este tipo de arquitecturas es más un ejercicio intelectual que una configuración práctica.
2. Una clase de máquinas tales que el flujo de datos circula sobre una serie de elementos de procesamiento. Las arquitecturas *pipeline* tales como los *arrays sistólicos* entran dentro de este grupo de máquinas.

La fig. I.9. representa la estructura general de una arquitectura *MISD*.

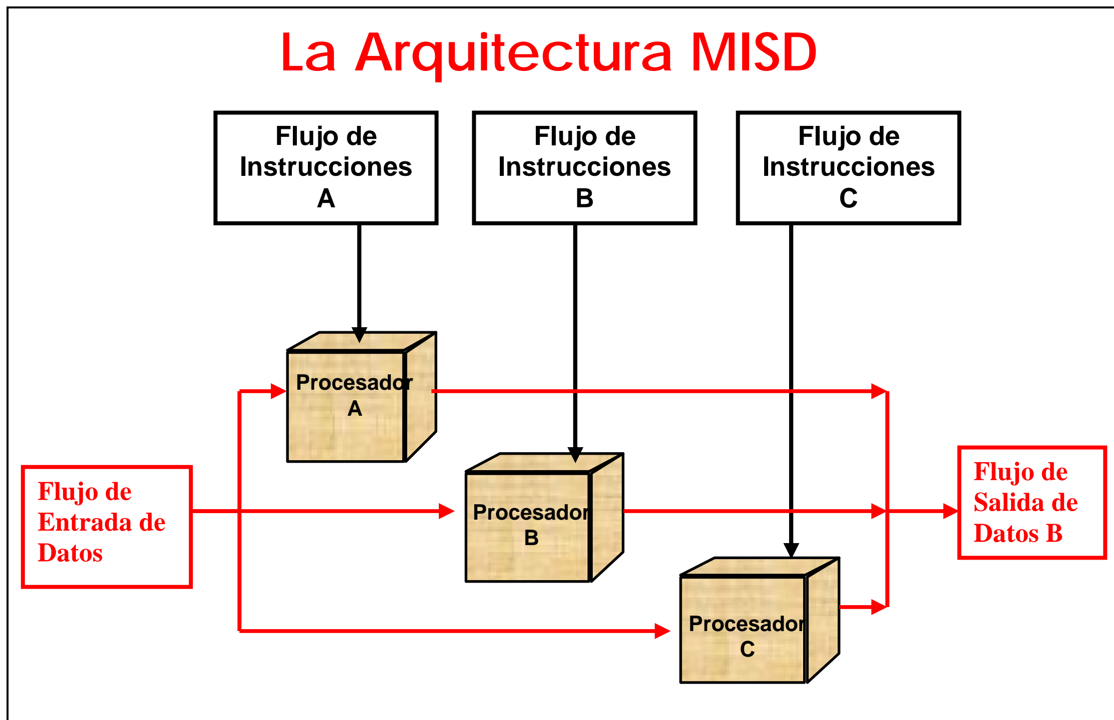


Figura I.9. Modelo de una Arquitectura MISD

I.3.3. Arquitecturas MIMD

MIMD (Multiple Instruction Stream, Multiple Data Stream): Un sistema *MIMD* es un sistema multiprocesador o una multicomputadora en donde cada procesador individual tiene su unidad de control y ejecuta su propio programa, fig. I.10.

Las computadoras *MIMD* tienen las siguientes características:

- Distribuyen el procesamiento sobre un número independiente de procesadores.
- Comparten fuentes, incluyendo el sistema de memoria principal, sobre los procesadores.
- Cada procesador opera independientemente y concurrentemente.
- Cada procesador ejecuta su propio programa.

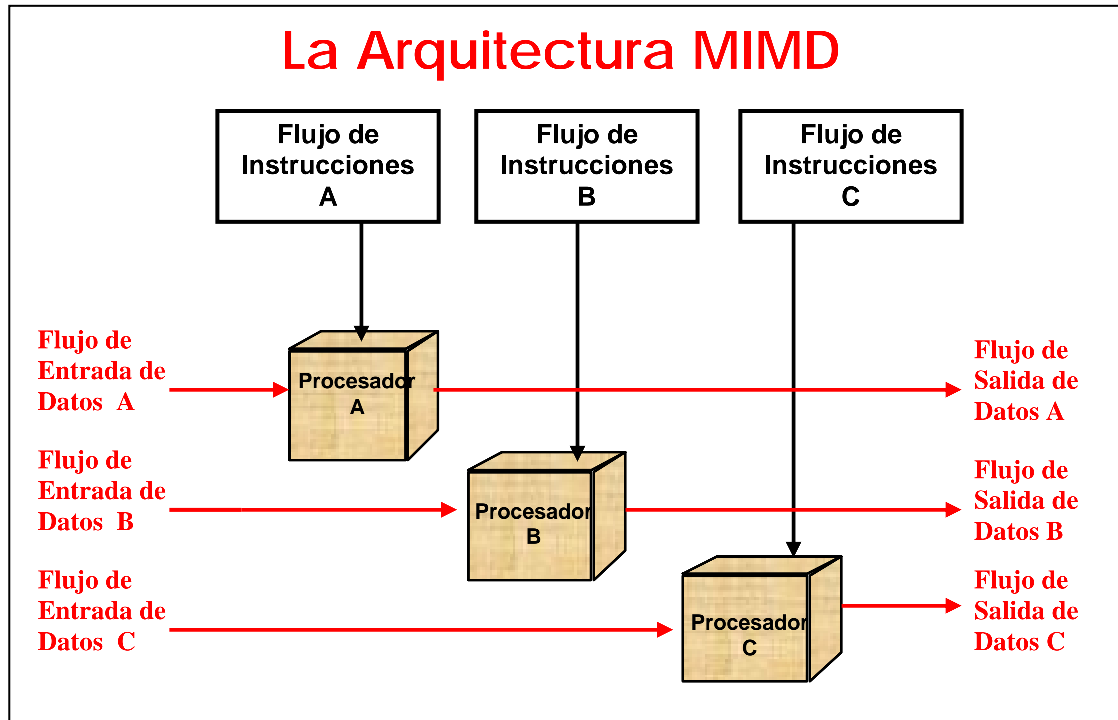


Figura I.10. Modelo de una arquitectura MIMD

Esto indica que los sistemas *MIMD* ejecutan operaciones en paralelo de manera asíncrona; los nodos activos cooperan pero operan independientemente. Las arquitecturas *MIMD* difieren con la interconexión de redes, procesadores, técnicas de direccionamiento de memoria, sincronización y estructuras de control. La interconexión de redes hace que los procesadores se comuniquen e interactúen unos con otros. Ejemplos de computadoras *MIMD* incluyen la Cosmic Cube, nCUBE2, iPSC, Symmetry, FX-8, FX-2800, TC-2000, CM-5, KSR-1 y la Paragon XP/s [SEL99].

Las computadoras *MIMD* se pueden categorizar en *sistemas fuertemente acoplados* y *sistemas débilmente acoplados*, dependiendo de cómo los procesadores accedan a la memoria.

Los procesadores en un *sistema multiprocesador fuertemente acoplado* generalmente comparten un sistema de memoria global, estos sistemas son conocidos como *sistemas de memoria compartida*. Aquellos *sistemas MIMD débilmente acoplados* pueden compartir un sistema de memoria, pero cada procesador tiene su propia memoria local. A estos sistemas se les conoce como *sistemas de paso de mensajes*. Las computadoras fuertemente acopladas y débilmente acopladas corresponden a los sistemas *MIMD* de Memoria Global (GM-MIMD) y *MIMD* de Memoria Local (LM-MIMD) respectivamente.

Las computadoras MIMD de paso de mensajes, Fig. I.11. se refieren a multicomputadoras en donde cada procesador tiene su propia memoria llamada memoria local o privada, y es accesible solo por su propio procesador. Las arquitecturas *MIMD* de paso de mensajes están referidas tanto a arquitecturas de memoria distribuida como a arquitecturas de memoria privada.

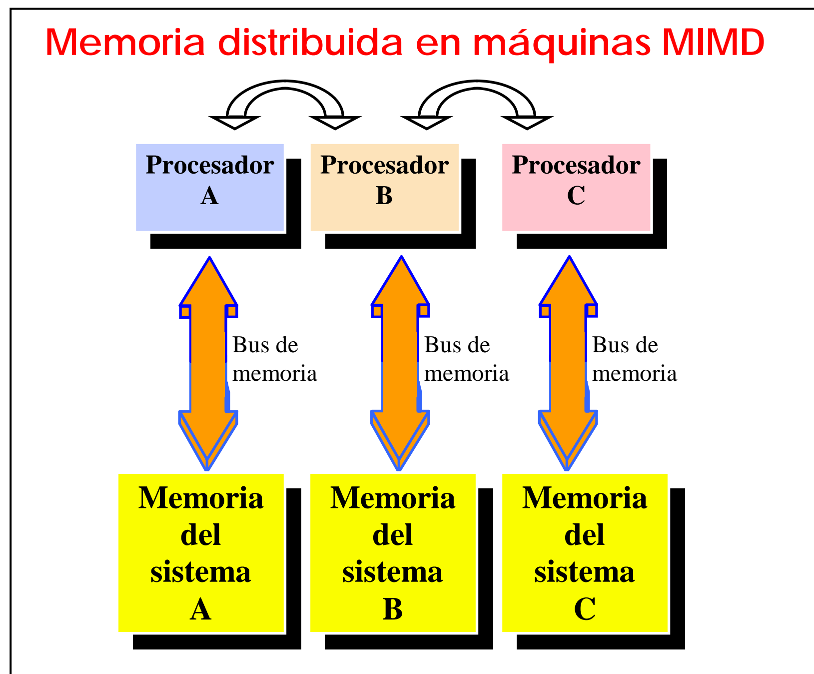


Figura I.11. Modelo de memoria distribuida en la arquitectura MIMD

Un sistema MIMD de memoria compartida, Fig. I.12. es llamado *Sistema de Acceso Uniforme a Memoria (UMA, Uniform Memory Access)*, ya que el tiempo de acceso a memoria es el mismo para todos los procesadores que la comparten. En este modelo, los procesadores pueden comunicarse sin restricciones y de una manera simple compartiendo datos utilizando un mismo espacio de direccionamiento. Los datos que se comparten pueden protegerse mediante el uso de métodos de ocultamiento de datos que los modernos lenguajes de programación ofrecen. Debido a esta capacidad de soportar una gran variedad de modelos de programación eficientes, un sistema multiprocesador de memoria compartida es siempre la primera elección de los usuarios de programación paralela.

Esto entra en contraste con los sistemas de paso de mensajes que son más fáciles de diseñar pero más difíciles de implementar o programar. En general, los sistemas

MIMD fuertemente acoplados proporcionan mayor rapidez en el intercambio de datos entre los procesadores que los sistemas *MIMD* débilmente acoplados.

Ejemplos de computadoras GM-MIMD son la serie CDC 6600 y la Cray XM-P. Ejemplos de computadoras LM-MIMD son la Carnegie-Mellon Cm* y la Tandom/16 [SEL99].

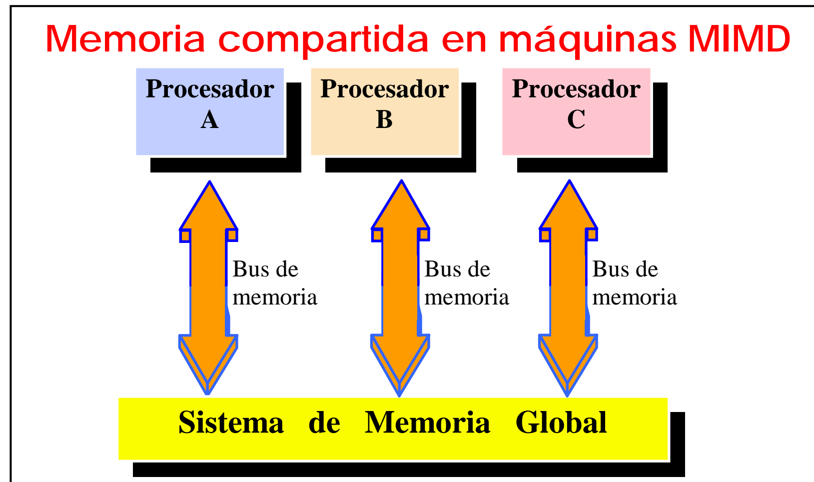


Figura I.12. Modelo de memoria compartida en la arquitectura MIMD

I.4. ALGORITMOS PARALELOS

Uno de los ingredientes más importantes para el procesamiento paralelo son sin duda los algoritmos paralelos que tienen un considerable interés en su desarrollo. Dado un problema a resolver en paralelo, el algoritmo describe como puede resolverse el problema pensando en una determinada arquitectura paralela, mediante la división del problema en subproblemas, comunicando los procesadores y posteriormente uniendo las soluciones parciales para obtener la solución final. Por ejemplo, los algoritmos basados en la estrategia de divide y vencerás usualmente tienen una naturaleza inherentemente paralela. Existen dos enfoques en el diseño de algoritmos paralelos con respecto al número de procesadores disponibles.

El primero, es el diseño de un algoritmo en el cual el número de procesadores utilizados por el algoritmo es un parámetro de entrada, lo que hace que el número de procesadores no dependa del tamaño de entrada del problema.

El segundo enfoque, permite que el número de procesadores utilizados por el algoritmo paralelo crezca con el tamaño de la entrada, de tal forma que el número de procesadores no es un parámetro de entrada, pero si una función del tamaño de entrada

del problema. Utilizando el esquema "división de labor", un algoritmo diseñado con el segundo enfoque puede siempre ser convertido a un algoritmo del primer enfoque.

Gran importancia dentro de los algoritmos paralelos toma la forma en que los distintos procesadores pueden llevar a cabo su propia comunicación. Los patrones de comunicación en los procesadores son rara vez arbitrarios y no estructurados. En cambio, las aplicaciones paralelas tienden a emplear patrones de comunicación predeterminados entre sus componentes. Si los patrones de comunicación más comúnmente utilizados son identificados en términos de sus componentes y de sus comunicaciones, es posible entonces crear un entorno o ambiente que este disponible mediante abstracciones de alto nivel para utilizarse en la escritura de aplicaciones. Esto proporciona un enfoque estructurado que puede ser acomodado dentro de un lenguaje orientado a objetos utilizado como lenguaje de programación paralela.

I.5. LENGUAJES DE PROGRAMACION PARALELA

Los lenguajes de programación paralela se basan en dos categorías, en abstracciones de programación paralela basadas en exclusión mutua de accesos a una memoria individual, y en abstracciones de procesos que se comunican mediante el envío de mensajes unos con otros. El envío de mensajes es una acción de alto nivel que puede ser implementada físicamente mediante procesadores distribuidos.

Cada enfoque de programación paralela sugiere una configuración de hardware particular. La mejor será aquella que empate con las primitivas del lenguaje utilizado en la programación paralela. Actualmente existen muchos lenguajes de programación que ya tienen diseñadas primitivas para el manejo de procesos de manera asíncrona o síncrona. La programación asíncrona se utiliza para la programación de multiprocesadores o sistemas distribuidos, mientras que las soluciones paralelas síncronas son propias del uso de arrays o vectores de procesadores.

En el caso de la programación orientada a objetos, ésta puede ser utilizada como un buen enfoque de programación paralela ya que puede encapsular y abstraer patrones comunes de comunicación paralela y llevar dicha abstracción hacia un estilo estructurado de programación paralela.

El lenguaje de programación C++ es un ejemplo de un excelente lenguaje de programación orientado a objetos con el que se pueden implementar aplicaciones

paralelas mediante el uso de *threads* para el manejo de procesos ligeros en un ambiente de memoria compartida.

I.6. THREADS

Un método para lograr el paralelismo consiste en hacer que varios procesos cooperen y se sincronicen mediante la compartición de memoria. Una alternativa para hacer esto viable es el emplear múltiples *threads* (hilos) de ejecución en un solo espacio de direcciones. Un *thread* es una secuencia de instrucciones ejecutada dentro de un programa. En otras palabras, cuando se ejecuta un programa, el CPU utiliza el contador de programa del proceso para determinar que instrucción debe ejecutar a continuación. El flujo de instrucciones resultante se denomina *hilo de ejecución del programa* y es el flujo de control para el proceso representado por la secuencia de direcciones de instrucciones indicadas por el contador de programa durante la ejecución del código de éste [BUT97].

Desde el punto de vista del programa, la secuencia de instrucciones de un hilo de ejecución es un flujo ininterrumpido de direcciones. En cambio desde el punto de vista del procesador, los hilos de ejecución de diferentes procesos están entremezclados y el punto en que la ejecución cambia de un proceso a otro se denomina *Conmutación de Contexto*.

Una extensión natural del modelo de proceso es permitir la ejecución de varios *threads* dentro del mismo proceso a lo que se le da el nombre de *multithreading*, fig. I.13. El *multithreading* proporciona un mecanismo eficiente para controlar hilos de ejecución que comparten tanto código como datos, con lo cual se evitan las conmutaciones de contexto. Esta estrategia también mejora el rendimiento pues en una máquina multiprocesador, los procesadores pueden ejecutar múltiples *threads* simultáneamente.

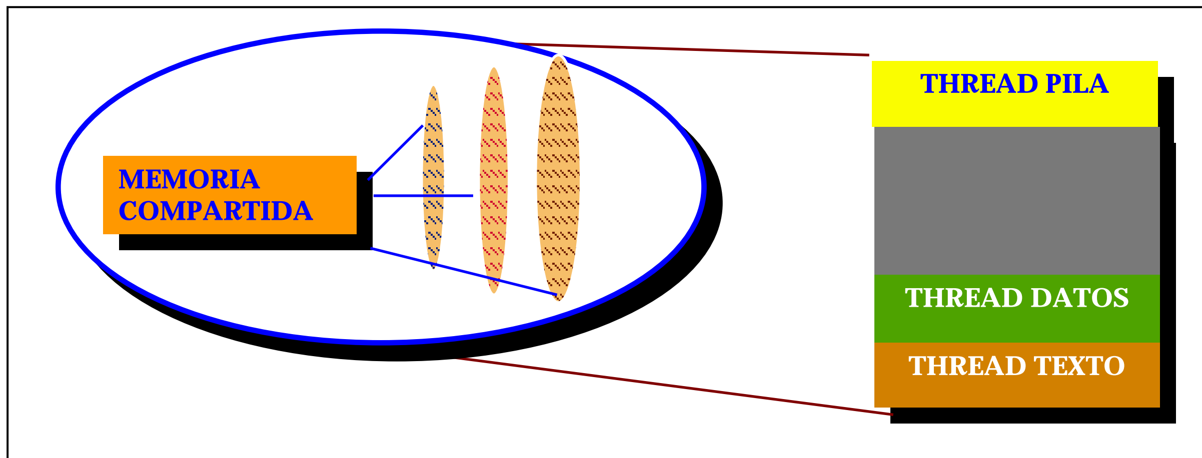


Figura I.13. Threads dentro de un proceso (todos los threads son parte de un proceso)

Cada hilo de ejecución se asocia con un *thread*, un tipo de datos abstracto que representa el flujo de control dentro de un proceso. Cada *thread* tiene su propia pila de ejecución, valor de contador de programa, conjunto de registros y estado, fig. I.14.

Fig. I.14. Modelo genérico de un Thread

Al declarar muchos *threads* dentro de los confines de un solo proceso, el programador puede lograr paralelismo con un bajo costo, sin embargo los *threads* también tienen ciertas complicaciones en cuanto a la necesidad de sincronización. Los *Threads* son frecuentemente llamados *procesos ligeros* y puede decirse que son primos de los procesos *UNIX*. En *UNIX* cualquier proceso tiene:

1. Un espacio de direcciones (pila, datos y segmento de código).
2. Una tabla de descriptores de archivos (archivos abiertos).
3. Un hilo o programa de ejecución.

La figura I.15 muestra el modelo básico de un proceso con estas características.

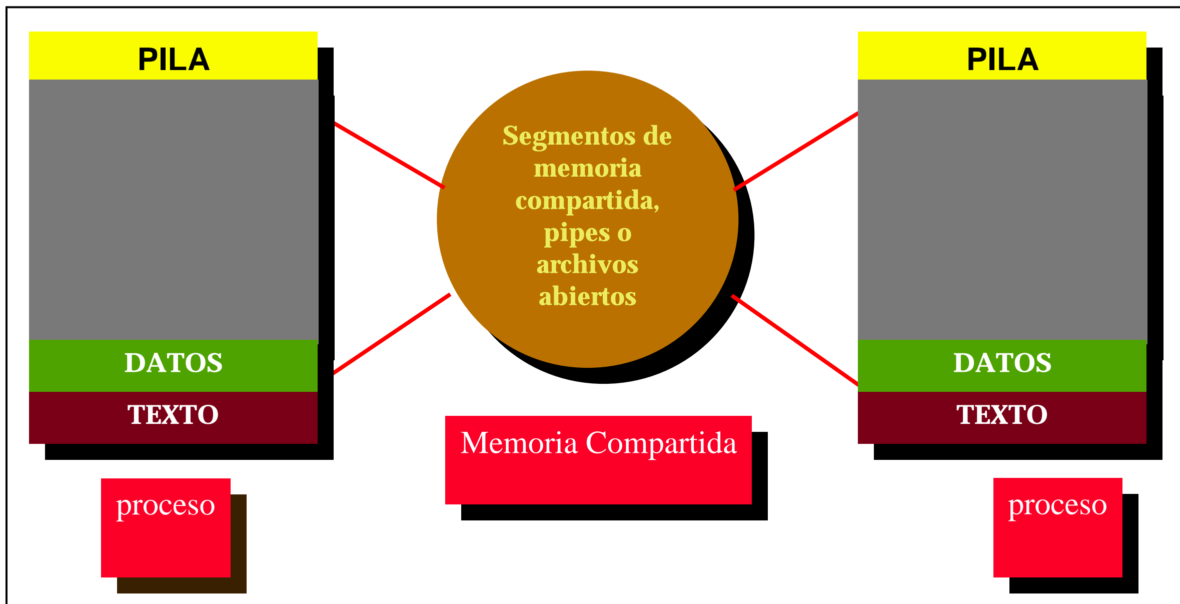


Figura I.15. Modelo básico de un proceso

I.6.1. Características de los Threads

- Los *threads* son más pequeños comparados con los procesos.
- La creación de un *thread* es relativamente menos costosa.
- Los *threads* comparten los recursos mientras que los procesos requieren su propio conjunto de recursos.
- Los *threads* ocupan menos memoria (es decir, son más económicos).
- Los *threads* proporcionan a los programadores la posibilidad de escribir aplicaciones concurrentes que se pueden ejecutar tanto en sistemas monoprocesador como en sistemas multiprocesador de forma transparente.
- Los *threads* pueden incrementar el rendimiento en entornos monoprocesador.

I.6.2. Con Threads o sin Threads

Los *threads* no necesariamente proporcionan la mejor solución a cualquier problema de programación. No siempre son fáciles de usar y no siempre proporcionan el mejor rendimiento. Existen muchos problemas que son inherentemente no concurrentes y añadirles *threads* puede alentar la programación y complicarla. Si

cualquier paso o secuencia en un programa depende directamente del resultado del paso previo, entonces el usar *threads* probablemente no ayudará en nada. Cada *thread* tendría que esperar a otro *thread* para ser completado. Los candidatos más obvios para codificar *threads* son aplicaciones que contemplen:

- Realización de gran cantidad de cálculos que puedan ser paralelizables (o descompuestos) en múltiples *threads*, en donde dichos cálculos puedan intentar ejecutarse en multiprocesadores de hardware, o bien
- Realizar I/O sustanciales las cuales puedan ser superpuestas (es decir, donde muchos *threads* puedan esperar para diferentes peticiones de I/O al mismo tiempo). Servidores o aplicaciones distribuidas son buenos candidatos para ello.

I.6.3. Programación con Threads

Un paquete o librería de *threads* proporcionada por algún lenguaje de programación específico permite escribir programas con varios puntos simultáneos de ejecución, sincronizados a través de memoria compartida. Sin embargo la programación con *threads* introduce nuevas dificultades. La programación concurrente y paralela tienen técnicas y problemas que no ocurren en la programación secuencial. Existen problemas simples (por ejemplo, el bloqueo mutuo o ínter bloqueo), pero otros problemas penalizan el rendimiento de las aplicaciones.

Un *thread* es un concepto sencillo: *un simple flujo de control secuencial* [KLE96]. Con un único *thread* existe en cualquier instante un único punto de ejecución. El programador no necesita aprender nada nuevo para emplear un único *thread*. Cuando se tienen múltiples *threads* en un programa, significa que en cualquier instante el programa tiene múltiples puntos de ejecución. Uno en cada uno de sus *threads*. El programador decide cuando y donde crear múltiples *threads*, mediante la ayuda de un paquete de librería o de un sistema computacional.

En un lenguaje de alto nivel, las variables globales son compartidas por todos los *threads* del programa, es decir, leen y escriben en las mismas posiciones de memoria. El programador es el responsable de emplear los mecanismos de sincronización del paquete de *threads* para garantizar que la memoria compartida se acceda de manera

correcta. Las facilidades proporcionadas por el paquete son conocidas como *primitivas ligeras*, lo que significa que las primitivas de:

- Creación,
- Mantenimiento,
- Sincronización y
- Destrucción,

son suficientemente económicas en esfuerzo para las necesidades de concurrencia del programador.

I.6.4. La vida de un Thread

En cualquier instante un *thread* (una vez creado) se encuentra en cualquiera de los cuatro estados básicos [BUT97] descritos en la tabla I.1.

ESTADO	SIGNIFICADO
LISTO (Ready)	El *thread* esta listo para ejecutarse, pero esta esperando por un procesador.
EJECUTANDOSE (Running)	El *thread* se esta ejecutando corrientemente. En un multiprocesador puede haber más de un thread ejecutándose en un proceso.
BLOQUEADO (Blocked)	El *thread* no esta disponible para ejecutarse debido a que esta esperando para hacer algo. Por ejemplo, puede estar esperando a que se cumpla una condición, o puede estar esperando un bloqueo o esperar a que se complete una operación de I/O
TERMINADO (Terminated)	El *thread* ha terminado o bien ha sido cancelado.

Tabla I.1. Los estados de un Thread

La fig. I.16. [BUT97], ilustra el diagrama de transición de estados del *thread*, y los eventos que causan que los hilos cambien de un estado a otro.

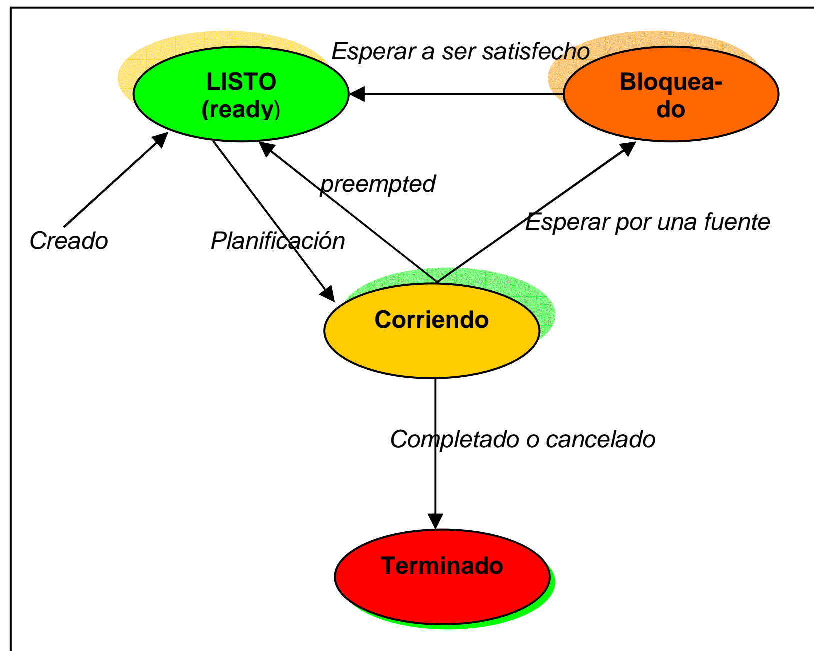


Figura I.16. Transiciones de estado de un Thread

I.6.5. Los Threads en la Concurrencia

- ***En el uso de un sistema multiprocesador:*** Los *threads* son una herramienta atractiva para permitir a un programa aprovecharse de este tipo de hardware. Empleando un paquete de *threads*, el programador puede utilizar los procesadores de forma económica. Esto parece funcionar bien en sistemas que van de 10 a 1000 procesadores.
- ***En la gestión de dispositivos de entrada/salida:*** Estos dispositivos pueden ser programados fácilmente mediante *threads*, de tal forma que las peticiones a un servicio sean secuenciales y el *thread* que realiza la petición se suspenda hasta que la solicitud se completa, mientras el programa principal realiza otro trabajo con otros *threads* solapando la ejecución de varios hilos.
- ***Con los usuarios como fuente de concurrencia:*** A veces un usuario necesita realizar dos o tres tareas simultáneamente. Los *Threads* son una buena forma de programar esta necesidad.
- ***En la constitución de un sistema distribuido:*** Servidores de red compartidos, donde el servidor se encarga de recibir peticiones de múltiples clientes. El uso de múltiples *threads* permite al servidor gestionar las solicitudes de los clientes en paralelo, en

vez de procesarlas en serie o crear un proceso de servicio para cada cliente, lo cual supone un enorme gasto.

- ***En la reducción de la latencia[2] de las operaciones de un programa:*** Emplear *threads* para diferir el trabajo es una técnica potente. La reducción de la latencia puede mejorar los tiempos de respuesta de un programa.

Finalmente, uno de los problemas de utilizar múltiples *threads* de ejecución es que hasta hace poco no existía un estándar. Algunos modelos populares de *threads* son:

- Mach C threads, *CMU*
- Sun OS LWP threads, *Sun Microsystems*
- PARAS CORE threads, *C-DAC*
- Java-Threads, *Sun Microsystems*
- Chorus threads, *Paris*
- OS/2 threads, *IBM*
- Windows NT/95 threads, *Microsoft*
- ***POSIX, ISO/IEEE standard***

La extensión POSIX.1c se aprobó en Junio de 1995. El estándar *POSIX-Thread* significa técnicamente el *API-Thread* especificado por el estándar formal internacional POSIX[3] 1003.1c-1995. Con la adopción de un estándar *POSIX* para los *threads*, las aplicaciones comerciales actuales se han ido construyendo con este *API* y seguramente en un futuro no muy lejano dicha estandarización será adoptada por todos.

I.7. ANTECEDENTES

Actualmente dentro del ámbito de la programación paralela uno de los problemas de mayor interés por parte de los investigadores y que pese a las propuestas y enfoques que se han definido, el problema sigue abierto a la investigación, es la falta de aceptación de entornos de programación paralela estructurada para desarrollar aplicaciones paralelas por parte de los usuarios. Para ello se han definido y actualmente existen varios trabajos previos a esta tesis donde se hace patente la necesidad de contar

[2] La latencia es el tiempo empleado entre la llamada a un procedimiento y la finalización de éste.
[3] POSIX significa *Portable Operating System Interface*.

con patrones paralelos de comunicación. En la literatura existen varias propuestas, pero todas ellas coinciden en la importancia de determinar un conjunto completo de patrones y sobre todo tratar de definir una semántica para ellos. La tendencia para esto es en la actualidad el uso de enfoques de programación orientados a objetos pues se ha visto que el definir objetos paralelos para el desarrollo de nuevas propuestas de metodologías, modelos y patrones de comunicación de programación paralela ha dado buenos resultados.

I.7.1. Problemas abiertos

1. *La falta de aceptación de entornos de programación paralela estructurada para desarrollar aplicaciones:* el paralelismo estructurado es un tipo de programación paralela basada en el uso de patrones de comunicación/interacción (*pipelines, farms, trees*, etc.) predefinidos entre los procesos de una aplicación de usuario. Los patrones también encapsulan las partes paralelas de dicha aplicación, de tal forma que al usuario solo le resta programar el código secuencial de ésta. Existen muchas propuestas de entornos para el desarrollo de aplicaciones y programas paralelos estructurados, pero hasta el momento, solo son utilizados por un círculo muy limitado de programadores expertos. Actualmente, en HPC, existe un gran interés en la investigación de entornos como los anteriormente mencionados.
2. *La necesidad de contar con patrones o composiciones paralelas de alto nivel:* una composición paralela de alto nivel o CPAN, como también se le denomina [COR95], se ha de poder definir y utilizar dentro de una *infraestructura* (lenguaje o entorno[4] de programación) *orientada a objetos*. Los componentes de una aplicación paralela no interactúan de forma arbitraria, sino que siguen patrones básicos regulares [BRI93]. Un entorno de programación paralela ha de ofrecer a sus usuarios un conjunto de componentes que implementen los *patrones* más utilizados en algoritmos y aplicaciones paralelos y distribuidos, tales como *trees, farms, pipes,* etc. El usuario, a su vez, ha de poder componer y anidar *CPANS* para desarrollar programas y aplicaciones. El usuario no ha de estar limitado a un conjunto de *CPANS* predefinidos, sino que, mediante el uso del mecanismo de *herencia*, ha de

[4] Se refiere al concepto de *HPC programming environment*: entorno de programación paralela “amigable” a sus usuarios que proporciona facilidades para el desarrollo de aplicaciones, abstrayendo detalles de bajo nivel como los referidos a la *creación, asignación, coordinación y comunicación* de procesos en un sistema distribuido y paralelo.

poder adaptarlos a sus necesidades. El entorno de desarrollo ha de contemplar, por tanto, el concepto de *clase* de *objetos paralelos.*

3. Existe interés en explorar la línea de investigación relacionada con la definición de conjuntos completos de patrones, así como en su definición semántica, para clases concretas de aplicaciones paralelas.
4. *Determinación de un conjunto completo de patrones así como de su semántica:* en éste punto, la comunidad científica no parece aceptar de forma totalmente satisfactoria y con la suficiente generalidad ninguna de las soluciones que hasta hoy se han obtenido para resolver éste problema. No parece, por tanto, fácil el que se pueda encontrar un conjunto lo suficientemente útil y general, como por ejemplo una librería de patrones o conjunto de constructos de un lenguaje de programación, para ser utilizado en el desarrollo, de una manera estructurada, de una aplicación paralela no específica.
5. *Adopción de un enfoque orientado a objetos:* El integrar un conjunto de clases dentro de una infraestructura orientada a objetos es una posible solución al problema descrito en el punto anterior, ya que permitiría el añadir nuevos patrones a un conjunto inicial incompleto mediante la definición de subclases. Por lo tanto, una de las líneas de investigación seguidas ha sido el encontrar representaciones de patrones paralelos como *clases*, a partir de los cuales se pueden instanciar objetos paralelos que son, a su vez, ejecutados como consecuencia de una petición de servicio externa a dichos objetos y procedente de la aplicación de usuario. Por ejemplo, el patrón derivado de la ejecución *por etapas* de los procesos vendría definido por la clase de patrón denominado *pipeline*, el número de etapas y el código secuencial de cada etapa específica no sería establecido hasta la creación de un objeto paralelo de dicha clase, los datos para procesar y los resultados se obtendrían de la aplicación de usuario. El almacenamiento interno en las etapas podría adaptarse en una *subclase* que herede de *pipeline*. Se obtienen varias ventajas al seguir un enfoque orientado a objetos [COR91], [COR95], respecto de un enfoque únicamente basado en esqueletos algorítmicos y programas modelo [BRI93], cabe reseñar, por ejemplo, las siguientes mejoras:
   5.1. *Uniformidad*: Todas las entidades dentro del entorno de programación son objetos. Como consecuencia de ello, todas las entidades de nivel más bajo que un objeto, como son por ejemplo los canales de comunicación, utilizados en el

modelo de programación distribuida y paralela propuesto en [BRI93], dejan de existir.

5.2. *Genericidad*: la especificación de las partes paralelas de una aplicación quedan claramente separadas de la especificación de cómo funcionan los componentes secuenciales.

5.3. *Reusabilidad:* el uso del mecanismo de herencia simplifica la definición de nuevos patrones, definiendo subclases de patrones ya existentes.

Estos problemas abiertos han sido la motivación principal para investigar sobre patrones de comunicación susceptibles de ser definidos, en particular, con el modelo de las Composiciones Paralelas de Alto Nivel o CPANS, con el objetivo de ofrecer una librería de clases que implementen los *CPANS* más representativos y útiles que puedan ser utilizados en una aplicación paralela dentro de un entorno de programación basado en objetos paralelos.

I.7.2. El Trabajo previo

Existe ya una propuesta dentro del enfoque consistente en considerar a los patrones como clases, las cuales se abstraen en módulos o *agregaciones*, de tal forma que pueden ser reutilizados y/o adaptados en aplicaciones de usuario, como se puede ver en [COR95].

Otro enfoque alternativo sería considerar que los patrones pueden incluirse como *constructos* dentro un lenguaje de programación, como se hace en el lenguaje de programación P3L [BAC99].

El enfoque más antiguo consiste en considerar a los patrones como *esqueletos algorítmicos* para programas paralelos [BRI93], [BRI94], [DAR93]. En esta aproximación se mezclan entidades de bajo nivel con estructuras de programación paralela de alto nivel, el resultado suele ser la pérdida de transportabilidad de las aplicaciones programadas con este esquema y de otros beneficios del desarrollo modular del software.

Los *esqueletos* son estructuras comunes de procesamiento paralelo expresadas de forma *genérica* (en función de parámetros que, una vez sustituidos por sus actuales, permiten adaptar la estructura a una aplicación particular). Algunos entornos de

programación paralela, como SklECL, están basados en *esqueletos* y *envoltorios* (wrappers) que constituyen los *constructos* básicos de un *lenguaje de coordinación*[5]*,*el cual define módulos que encapsulan código escrito en un lenguaje secuencial y 3 clases de esqueletos: *control, stream parallel, y data parallel*, que pueden ser compuestos entre sí o anidados para construir aplicaciones paralelas complejas.

SklECL es un entorno de desarrollo integrado del proyecto PQE2000 que permite la programación rápida de aplicaciones paralelas complejas en varias arquitecturas (memoria distribuida y compartida MPP, SMP, *clusters* de SMP, NOWs, metacomputadores, etc.). Permite utilizar notaciones de programación paralela (MPI, HPF, PVM, etc.) que se han convertido en estándares de la industria, así como herramientas para depurar, visualizar la ejecución y evaluar el rendimiento de aplicaciones paralelas. El usuario puede reutilizar *trozos grandes* de código secuencial escrito en su lenguaje favorito (C, C++, F77, F90, Java, etc.), encapsulándolo en módulos. También, el código paralelo preprogramado puede ser encapsulado en estructuras paralelas más grandes (utilizando MPI y bibliotecas especializadas). El entorno de programación facilita al usuario el diseño de la estructura global de su aplicación, mediante una colección de estructuras paralelas típicas, denominadas *esqueletos*, que pueden ser instanciadas y combinadas para definir aplicaciones paralelas complejas. Los *esqueletos* se pueden utilizar para coordinar y conectar entre sí módulos secuenciales o paralelos encapsulados en *envoltorios*. Estos últimos sirven para asegurar que el paso de parámetros y la representación de datos sean consistentes entre todos los módulos que componen una aplicación paralela. Ejemplos de *esqueletos* comúnmente utilizados son los *farms* (una reserva de *procesos trabajadores* lleva a cabo un conjunto de tareas de cómputo), *pipelines* (que se utilizan para explotar el paralelismo derivado de ejecutar simultáneamente las diferentes fases de un cálculo) y *trees* (donde se pueden aplicar técnicas de Divide y Vencerás en paralelo). Dichos esqueletos son actualmente proporcionados por varias bibliotecas y entornos de programación paralela. La ventaja de SklE, respecto de éstos últimos, consiste en que permite componer libremente los esqueletos, para así construir estructuras más complejas, y además es capaz de generar código optimizado para arquitecturas

[5] El compilador de dicho lenguaje, a partir de la estructura global, definida como una grafo que anida varios esqueletos, y el código secuencial escrito por el usuario en C, F90, Java, ..., es capaz de generar código optimizado para una determinada clase de arquitectura.

concretas[6]. El desarrollo de aplicaciones paralelas en SklE se lleva a cabo utilizando VisualSklE, que es un sistema gráfico de ventanas. Un usuario puede definir la estructura global de su aplicación de una forma interactiva. Puede editar nuevas partes secuenciales de su aplicación, utilizando un editor integrado en el entorno, y encapsular código secuencial, ya programado, dentro del envoltorio adecuado. También, la estructura paralela de la aplicación puede ser definida explícitamente utilizando, C y Fortran, además de MPI, o bien utilizando los esqueletos incorporados en SklE.

I.7.3. Requerimientos para el desarrollo de un entorno de programación basado en Objetos Paralelos

Dado que el problema a resolver es la definición de un *Método de Programación Basado en Composiciones Paralelas de Alto Nivel (CPANS)*, se han planteado los siguientes requerimientos indispensables que deben ser solventados para el buen desarrollo de la tesis presentada en ésta memoria. Se requiere, en principio, un entorno de Programación Orientado a Objetos que proporcione:

1. Capacidad de invocación de métodos de los objetos que contemple los modos de comunicación *asíncrono* y *futuro asíncrono.* El modo asíncrono no fuerza a esperar el resultado del cliente que invoca un método de un objeto,. El modo de comunicación *futuro asíncrono* hace que espere el cliente sólo cuando necesite el resultado en un instante futuro de su ejecución. Ambos modos de comunicación permiten que un cliente pueda seguir ejecutándose concurrentemente con la ejecución del método (paralelismo entre objetos).
2. Los objetos han de poder tener *paralelismo interno.* Un mecanismo de *threads* ha de permitir a un objeto servir varias invocaciones de sus métodos concurrentemente (paralelismo intra-objetos).
3. Disponibilidad de *mecanismos de sincronización* cuando se producen peticiones paralelas de servicio. Esto es necesario para que los objetos puedan gestionar varios flujos de ejecución concurrentemente y, al mismo tiempo, garantizar la consistencia de sus datos.

[6] SklE no solo es capaz de generar automáticamente el código necesario para implementar una construcción paralela de procesos, sino que también optimiza los recursos asignados a cada esqueleto, decide el tamaño óptimo de los procesos de bajo nivel y localiza las posibles causas de ineficiencia de la estructura resultante.

4. Disponibilidad de *mecanismos flexibles de control de tipos*. Se ha de tener la capacidad de asociar tipos dinámicamente a los parámetros de los métodos de los objetos. Esto es, se necesita que el sistema pueda gestionar tipos de datos genéricos, ya que los *CPANS* solo definen la parte paralela de un patrón de interacción, por tanto, han de poder adaptarse a las diferentes clases de posibles componentes del patrón.
5. *Transparencia de distribución de aplicaciones paralelas*. Ha de proporcionar el transporte de las aplicaciones desde un sistema centralizado a un sistema distribuido sin que se vea afectado el código del usuario. Las clases han de mantener sus propiedades, independientemente del entorno de ejecución de los objetos de las aplicaciones.
6. Rendimiento (*performance*). Éste es siempre el parámetro más importante a considerar cuando se hace una nueva propuesta de entorno de desarrollo para aplicaciones paralelas. Un enfoque basado en *patrones como clases* y *objetos paralelos* ha de resolver satisfactoriamente el problema denominado PPP (*Programmability, Portability, Performance*) para que se considere una aproximación relevante en la búsqueda de soluciones a los problemas planteados.

El entorno de programación Orientado a Objetos que se ha considerado como idóneo para cubrir los 6 requerimientos anteriormente citados es el lenguaje de programación C++, junto con el uso del estándar POSIX Thread, teniendo como base el sistema operativo Linux, en particular Red Hat 7.0 para el desarrollo de la propuesta en un sistema centralizado y un sistema operativo Unix, en particular el IRIX 6.5 del sistema paralelo Origin 2000 Silicon Graphics, para el desarrollo de la misma en un sistema distribuido.

I.8. OBJETIVOS CIENTIFICOS

Se pretende el desarrollo de un método de programación basado en *Composiciones Paralelas de Alto Nivel o CPANS* que implementen una librería de clases de utilidad en la Programación Concurrente/Paralela Orientada a Objetos. El método ha de proporcionar al programador los patrones paralelos de comunicación más comúnmente utilizados, de tal forma que éste pueda explotar los mecanismos de

generalización por herencia y parametrización para definir nuevos patrones según el modelo del *CPAN*. Los objetivos específicos a alcanzar en este trabajo son:

- Desarrollar un método de programación basada en Composiciones Paralelas de Alto Nivel o *CPANS*
- Desarrollar una librería de clases de Objetos Paralelos que proporcione al usuario los patrones (bajo el modelo del *CPAN*) más comúnmente utilizados para la programación paralela.
- Ofrecer esta librería al programador para que, con conocimientos mínimos de Paralelismo y Concurrencia, pueda explotarlos, mediante el uso de diferentes mecanismos de reusabilidad, bajo el paradigma de la Orientación a Objetos y pueda, asimismo, definir patrones propios, adaptados a la estructura de comunicación entre los procesos de sus aplicaciones.
- Transformar algoritmos conocidos que resuelven problemas secuenciales y que pueden ser fácilmente paralelizables en su versión paralela/concurrente para probar la metodología y los componentes software desarrollados en el presente trabajo.

Capítulo II:

COMPOSICIONES PARALELAS DE ALTO NIVEL (CPANS): Análisis y Definición del Modelo

II.1. INTRODUCCIÓN

Uno de los problemas con que se encuentran los entornos de programación paralela es el de su aceptación por los usuarios, que depende de que puedan ofrecer expresiones completas del comportamiento de los programas paralelos que se construyen con dichos entornos [COR95]. Actualmente sucede que, entre los sistemas orientados a objetos, los entornos de programación basados en objetos paralelos sólo son conocidos por la comunidad científica dedicada al estudio de la Concurrencia.

Un primer enfoque que intenta atacar este problema es tratar de hacer que el usuario desarrolle sus programas según un estilo de programación secuencial y, ayudado de un sistema o entorno específico, éste pueda producir su contraparte paralela. Sin embargo, existen dificultades de implementación intrínsecas a la definición de la semántica formal de los lenguajes de programación que impiden la paralelización automática sin la participación del usuario, por lo que el problema de generar paralelismo de manera automática para una aplicación general sigue abierto.

Un enfoque alternativo prometedor, que es el que se adopta en la presente investigación para alcanzar los objetivos planteados, es el denominado *paralelismo estructurado*. En general las aplicaciones paralelas siguen patrones predeterminados de ejecución. Estos patrones de comunicación son rara vez arbitrarios y no estructurados en su lógica [BRI93]. Las *Composiciones Paralelas de Alto Nivel o CPANS* son patrones paralelos de comunicación bien definidos y lógicamente estructurados que, una vez identificados en términos de sus componentes y de su esquema de comunicación, pueden llevarse a la práctica y estar disponibles como abstracciones de alto nivel en las aplicaciones del usuario dentro de un entorno o ambiente de programación, en este caso el de la orientación a objetos. Las estructuras de interconexión de procesadores más comunes como los *pipelines* o cauces, los *farm* o granjas y los *trees* o árboles pueden ser construidas utilizando *CPANS*, dentro del entorno de trabajo de los *Objetos Paralelos*, que es el utilizado para detallar la estructura de la implementación de un *CPAN*.

II.1.1. El Paralelismo Estructurado

Un *enfoque estructurado* para la programación paralela se basa en el uso de patrones de comunicación/interacción (*pipelines*, *farms*, *trees*, etc.) predefinidos entre los procesos de una aplicación de usuario. En tal situación, el enfoque del paralelismo estructurado proporciona la abstracción del patrón de interacción y describe aplicaciones a través de *CPANS* capaces de implementar los patrones ya mencionados.

La encapsulación de un *CPAN* debe seguir el principio de modularidad y debe proporcionar una base para obtener una reusabillidad efectiva del comportamiento paralelo que se implementa. Cuando se logra hacer esto, se crea un *patrón paralelo genérico*, el cual proporciona una posible implementación de la interacción entre los procesos, independiente de la funcionalidad de éstos. El enfoque estructurado para la programación paralela en los últimos años ha seguido básicamente dos caminos:

1. El enriquecimiento de entornos paralelos tradicionales con *librerías de "esqueletos" (skeletons)* [DAR93] de programas que representan patrones de comunicación concretos.
2. La definición de *lenguajes paralelos restrictivos y cerrados* que proporcionan comunicación en términos de los patrones que ya están definidos en el sistema [BAC99].

El enfoque presentado aquí atiende al primer camino por considerarlo más genérico y abierto. Lo que se plantea ahora es que, en lugar de programar una aplicación concurrente desde el principio y de controlar tanto la creación de los procesos como la de las comunicaciones entre ellos, el usuario simplemente identifique los *CPANS* que implementan los patrones adecuados para las necesidades de comunicación de su aplicación y los utilice junto con el código secuencial que implementa los cómputos que individualmente realizan sus procesos.

Pueden identificarse de manera informal varios patrones paralelos de interconexión significativos y reutilizables en múltiples aplicaciones y algoritmos paralelos, pero no existe un acuerdo lo suficientemente general que permita definir formalmente sus semánticas [COR95]. Por ejemplo, el patrón *farm* es un concepto que puede ser entendido por la mayoría de sus posibles usuarios de una formal general, pero su

concreción en una aplicación particular obliga a éstos a elegir entre diferentes estrategias para su implementación.

II.1.2. El paradigma de la Orientación a Objetos (OO)

La orientación a objetos parece ser una posible solución al problema de la necesidad de encontrar nuevos conceptos, métodos, herramientas y algoritmos que hagan frente a las dificultades inherentes de la programación paralela y concurrente. En particular, el paradigma de la OO se utiliza en la presente tesis para encapsular y abstraer patrones comunes de interacciones paralelas hacia un estilo de paralelismo estructurado. La idea básica es definir a los *CPANS* como objetos. Más aun, un *CPAN* debe ser definido como un objeto encargado de controlar y coordinar la ejecución de sus componentes internos, objetos en alguna fase que siguen un patrón definido.

Bajo esta premisa se logra definir un ambiente de *CPANS* extensible que tiende a sobrepasar los límites disponibles en la actualidad proporcionando características tan importantes como:

- ***Uniformidad:*** Todas las entidades dentro del ambiente (incluyendo los *CPANS* en sí mismos) son objetos y los canales de comunicación dejan de existir. Los patrones paralelos de comunicación son descritos en un nivel de "clase"[7] como cualquier otro componente de la aplicación y cualquier nuevo patrón paralelo puede ser definido en el mismo nivel en que se definieron los primeros mediante la definición de una nueva clase.
- ***Generalidad:*** La capacidad de generar referencias dinámicas en un ambiente orientado a objetos hace posible crear clases de *CPANS* genéricas, mediante la definición de sus componentes siempre como referencias genéricas a objetos. La especificación de la parte paralela puede ser separada de cualquier especificación funcional. Por ejemplo, un *pipeline* puede ser definido sin especificar qué harán los nodos que lo conforman.
- ***Reusabilidad:*** El mecanismo de herencia simplifica la definición de patrones paralelos especializados. En principio, para definir un nuevo patrón el usuario tiene que explicitar el trato con el paralelismo, sin embargo, la herencia aplicada

[7] Una clase es una colección de datos y métodos que operan sobre dichos datos.

al comportamiento concurrente del objeto ayuda en la especificación del comportamiento paralelo de un *CPAN*. Por ejemplo, si el entorno inicialmente proporciona un patrón *farm* cuyo objeto controlador espera sólo para la primera petición de servicio, es muy simple para el usuario especializar el comportamiento definiendo un nuevo *farm* cuyo objeto controlador opere con todas las peticiones de servicio en paralelo.

II.2. DEFINICIÓN DEL MODELO CPAN

La idea central es la de implementar cualquier tipo de patrones paralelos de comunicación entre los procesos de una aplicación o algoritmo paralelo y distribuido como clases, siguiendo el paradigma de la Orientación a Objetos. A partir de dichas clases, un objeto puede ser instanciado y la ejecución de un método del objeto en cuestión se puede llevar a cabo a través de una petición de servicio. Un *CPAN* proviene de la composición de un conjunto de objetos de tres tipos:

- Un objeto manager (Fig. II.1) que representa al *CPAN* en sí mismo y hace de él una abstracción encapsulada que oculta su estructura interna. El manager controla las referencias de un conjunto de objetos (un objeto denominado *Collector* y varios objetos denominados *Stage*), que representan los componentes del *CPAN* y cuya ejecución se lleva a cabo en paralelo y debe ser coordinada por el propio *manager*.

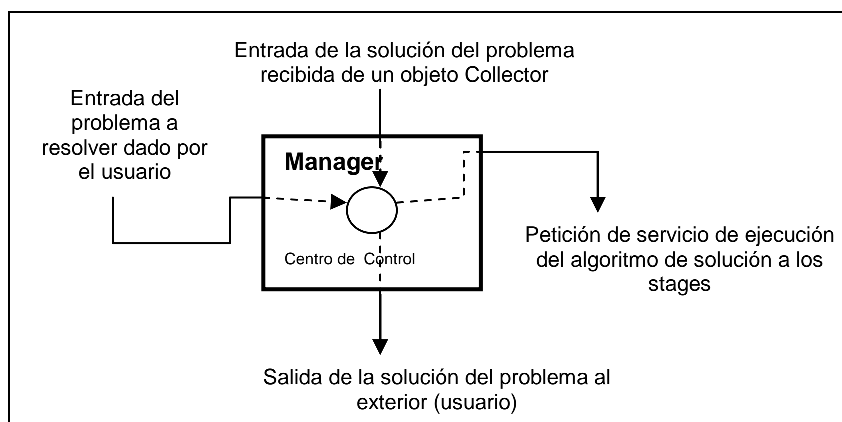


Fig. II.1. Componente MANAGER del modelo CPAN

- Los objetos Stage (Fig. II.2), que son objetos de propósito específico, encargados de encapsular una interfaz tipo *cliente-servidor* que se establece entre el manager y los *objetos esclavos* (objetos que no son activamente participativos en la composición del

CPAN, sino que se consideran entidades externas que contienen el algoritmo secuencial que constituye la solución de un problema dado), así como el proporcionar la conexión necesaria entre ellos para implementar la semántica del patrón de comunicación que se pretende definir. En otras palabras, cada *stage* debe actuar en paralelo como un nodo del grafo que representa al patrón. Un *stage* puede estar directamente conectado al *manager* y/o a otros componentes *stage* dependiendo del patrón particular del *CPAN* implementado.

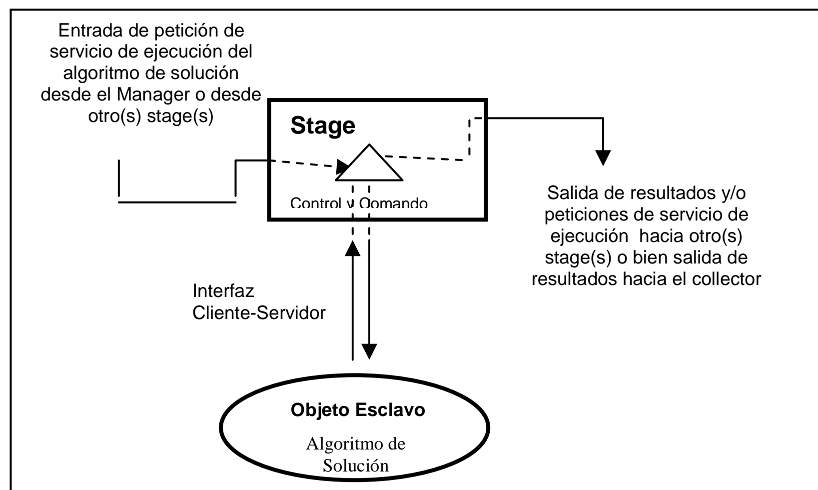


Fig. II.2. Componente Stage del Modelo CPAN y su Objeto Esclavo asociado

- y un objeto Collector (Fig. II.3), que es un objeto encargado de almacenar en paralelo los resultados que le lleguen de los objetos *stage* que tenga conectados. Es decir, durante el servicio de una petición, el flujo de control dentro de los *stages* de un *CPAN* depende del patrón de comunicación implementado. Cuando la composición finaliza su ejecución, el resultado no se regresa directamente al *manager*, sino que una instancia de la clase *Collector* se encarga de almacenar dichos resultados y de enviarlos al *manager*, el cual enviará al exterior los resultados, (que a su vez, le envíe un objeto colector), tan pronto como le vayan llegando, sin necesidad de esperar a que hayan sido obtenidos todos los resultados .

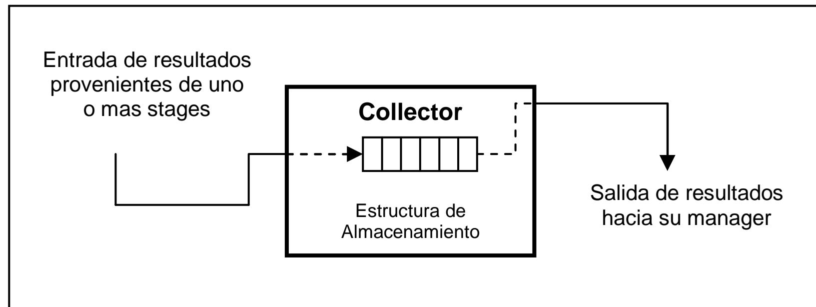


Fig. II.3. Componente Collector del modelo CPAN

II.2.1. Composición del CPAN

Si observamos el esquema como una encapsulación, es decir, como una caja negra, el diagrama gráfico de la representación de un *CPAN* sería el que se muestra en la fig. II.4:

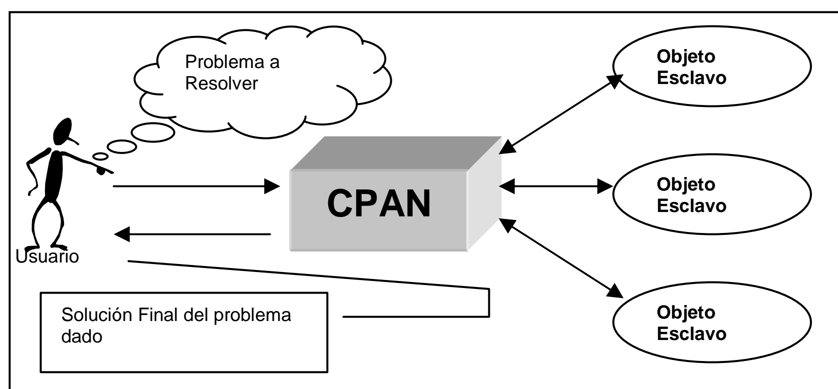


Fig. II.4 Representación gráfica de un CPAN como caja negra

Por otro lado, si observamos el interior de la caja negra del *CPAN* de la figura anterior, veremos que su estructura interna esta constituida por la composición de un conjunto de objetos de los tres tipos ya citados. En particular, cada *CPAN* esta compuesto por un objeto *manager*, que representa el *CPAN* en sí mismo, algunos objetos *stage* interconectados entre sí de acuerdo con el patrón implementado y un objeto *collector* para cada petición a tratar dentro del *CPAN* conectado al *manager* y conectado por lo menos a un *stage*. Además, para cada *stage* del *CPAN*, un *objeto esclavo* esta encargado de la implementación de las funcionalidades necesarias (ver fig. II.5).

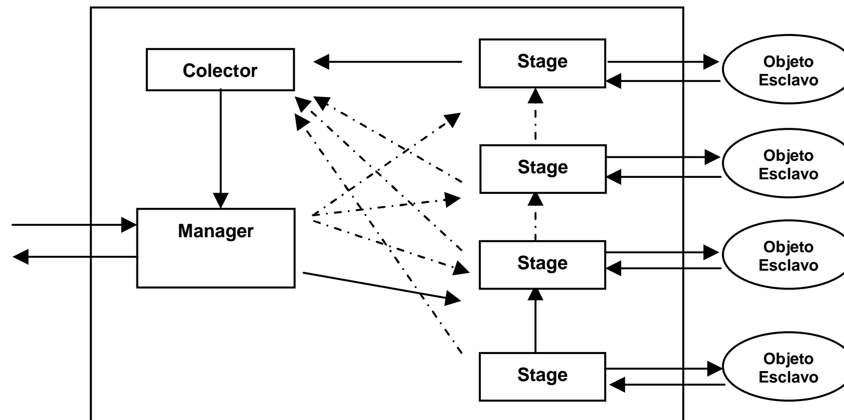


Fig. II.5 Estructura Interna de un CPAN (Composición de sus Componentes)

La figura II.5 muestra el modelo *CPAN* en general sin definir ningún patrón de comunicación paralelo explícito. La caja que engloba a los componentes, representa el *CPAN* encapsulado. Las cajas internas representan objetos compuestos (*collector*, *manager* y objetos *stages*), en tanto que los círculos son los objetos esclavos asociados a los *stages*. Las líneas continuas dentro del *CPAN* suponen que por lo menos debe existir una conexión por ejemplo entre el *manager* y alguno de los componentes *stage*. Lo mismo sucede entre los *stages* y el objeto *collector*. Las líneas punteadas significan que puede haber más de una conexión entre los componentes.

II.2.2. El CPAN visto como una composición de Objetos Paralelos

Los objetos *manager*, *collector* y *stages* se engloban dentro de la definición de *Objeto Paralelo (PO)* [COR95]. Los *Objetos Paralelos* son objetos activos, es decir, objetos que tienen capacidad de ejecución en sí mismos. Las aplicaciones dentro del modelo *PO* pueden explotar tanto el paralelismo entre objetos (*inter-object*) como el paralelismo interno de ellos (*intra-object*). Un objeto *PO* tiene una estructura similar a la de un objeto en Smalltalk[8], pero además incluye una política de planificación determinada a priori, que especifica la forma de sincronizar una o más operaciones de una clase en paralelo. Las políticas de sincronización se expresan en términos de restricciones, por ejemplo, la exclusión mutua en procesos lectores/escritores, el paralelismo máximo en procesos escritores o la sincronización entre procesos que acceden a recursos compartidos. Todos los objetos paralelos derivan entonces de la

[8] Un objeto en Smalltalk al igual que en C++ se compone de un estado y de un comportamiento.

definición clásica de "clase" más la incorporación de la política de planificación de procesos (restricciones de sincronización, exclusión mutua y paralelismo máximo). Los objetos de una misma clase comparten la misma especificación contenida en ella de la cual son instanciados. La herencia permite derivar una nueva especificación de una que ya existe. Los objetos paralelos soportan herencia múltiple.

II.2.2.1. Definición sintáctica de los Objetos Paralelos de un CPAN

Desde el punto de vista sintáctico, la definición de una clase que define un *Objeto Paralelo* sigue el siguiente template:

```
CLASS: <nombre de la clase>

       INHERITS_FROM: <lista de los nombres de clases padres directas>

       INSTANCE_STATE: <lista de los nombres y tipos de las variables de
                        instancia>

       INSTANCE_METHODS: <lista de métodos públicos>

       PRIVATE_METHODS: <lista de métodos privados>

       SCHEDULING_PART: <lista de restricciones de sincronización>

END_CLASS <nombre de la clase>
```

II.2.2.1.1. INSTANCE_STATE

Contiene la definición de las variables de instancia de la clase. En *PO*, todas las variables son especificadas mediante un tipo. El tipo utilizado puede ser cualquier tipo primitivo especificado en el lenguaje C o el nombre de una clase para definir un tipo de dato abstracto. Todas las variables, incluyendo los arreglos, tienen una semántica por valor. Sólo en el caso de que el tipo represente el nombre de una clase, la definición de la variable tiene una semántica por referencia. Por ejemplo, una referencia a una instancia de la clase *Object* puede ser utilizada para direccionar a objetos de cualquier clase que hereden de *Object*.

II.2.2.1.2. INSTANCE_METHOD

Contiene la definición de los métodos públicos de la clase. Existe un método importante en la definición de una clase *PO*: el método *init( )* que es implementado en

esta parte del template y que es considerado como el método de inicialización y encargado de crear la instanciación de la clase de la cual forma parte. El método *init( )* es siempre ejecutado antes que cualquier otro servicio de petición y en exclusión mutua con ellos. Más adelante se hablará de éste método.

II.2.2.1.3. PRIVATE_METHOD

Permite la definición de métodos privados que pueden ser invocados sólo por actividades internas del *PO*. Además de la definición e invocación del método, la sintaxis de la especificación del cuerpo de los métodos *PO* siguen la sintaxis del lenguaje C. Las funciones de C pueden ser definidas y llamadas por métodos *PO*.

II.2.2.1.4. SCHEDULING_PART

Se definen las restricciones de planificación de procesos: exclusión mutua, paralelismo máximo y sincronización de procesos [COR91].

II.2.2.2. Los tipos de Comunicación en los Objetos Paralelos

Los *objetos paralelos* definen 3 modos de comunicación: el modo de comunicación síncrono, el asíncrono y el modo de comunicación futuro asíncrono.

1. ***El modo síncrono*** detiene la actividad del cliente hasta que el objeto activo servidor le cede la respuesta. La notación[9]: `ref_obj.name_meth([lista_param])` facilita su utilización en la programación de aplicaciones.
2. ***El modo asíncrono*** no fuerza la espera en la actividad del cliente. El cliente envía simplemente la petición al objeto servidor activo y continúa su ejecución. Su uso en la programación de aplicaciones también resulta fácil. Sólo hay que crear un hilo y lanzarlo para su ejecución[10]. Usaremos la siguiente notación para

---

[9] Las notaciones utilizadas en esta sección se basan en la gramática de Objetos Paralelos descrita en la sección II.2.2.2. de este capítulo.

[10] El POSIX Thread proporciona la instrucción *pthread_create(. . .)* junto con el tipo *pthread_t* para la creación y uso de hebras

referirnos a este modo de comunicación: **`Thread`** `ref_obj.name_meth([lista_param])`, donde `Thread` denota un hilo que es lanzado para ejecutar el método `name_meth([lista_param])` de un objeto `ref_obj`.

3. ***El modo futuro asíncrono*** hace esperar la actividad del cliente sólo cuando, dentro de su código, se necesita el resultado del método para evaluar una expresión, [LAV], [ARJ96], [LIE87]. Su uso también es sencillo, aunque su implementación requiere de un cuidado especial para conseguir un constructo con la semántica deseada. La notación utilizada para ello será: **`FutureType`** `futureVar= ref_Obj.name_meth([lista_param])` que expresa la generación y asignación futura del resultado de una función invocada a través de una referencia a un objeto. Donde `FutureType` es el tipo que define el *futuro* y **`Anytype`** `ResulVar= ref_Obj.futureVar`, se utiliza para la conversión de tipo del *futuro* que devuelve la función cuando se ejecute, a un tipo `Anytype`**.** La palabra `Anytype` se utiliza para sugerir el uso de "cualquier tipo", el que sea de interés para el usuario.

Los modos de comunicación asíncrono y futuro asíncrono llevan a cabo el paralelismo *inter-objetos* ejecutando los objetos cliente y servidor al mismo tiempo.

II.2.2.2.1. El modo de Comunicación Futuro Asíncrono

El término *"futuro"* se utiliza para identificar los mecanismos relacionados con la generación de, y acceso a, un valor retornado por una función a la cual se accede a través de la referencia a un objeto. La palabra "*futuro*" se utiliza para sugerir que el valor retornado, si no está disponible inmediatamente, pueda estarlo algún tiempo después, en el futuro. Debido a que el elemento "*tiempo*" está presente, el manejo del valor de retorno lleva consigo un mecanismo de sincronización añadido a las acciones que se llevan a cabo para el chequeo de tipos. Definimos entonces "*un futuro*" como un mecanismo de sincronización que devuelve un resultado con tipo, el cual se utiliza para representar un valor que puede estar disponible en algún instante posterior, después de la creación del "*futuro*" [LEA96]. Los lenguajes Orientados a Objetos, en este caso C++, junto con el estándar *POSIX Thread* para el manejo de hilos proporcionan las

herramientas necesarias para programar de una manera más o menos fácil la abstracción denominada *"futuros"*. A continuación se muestra la implementación del *tipo futuro* expresado como una clase y referido como un Futuro de tipo *Anytype*, es decir, de cualquier tipo.

```
CLASS FutureType
  {
    ANYTYPE res;
    PUBLIC VOID operator=(ANYTYPE fut)
      {
       MUTEX(this);
       res=fut;
      }

    PUBLIC operator ANYTYPE()
       {
        SYNC(operator=,this);
        RETURN res;
       }
};
```

La sobrecarga del operador de asignación es necesaria para poder asignar el valor de retorno de una función a una variable del tipo *FutureType*, de forma que sólo un proceso productor pueda realizar dicha asignación cada vez, es decir, se lleva a cabo una sincronización de exclusión mutua (*Mutex*[11]) entre procesos productores que se ejecutan en paralelo y que quieren asignar el valor que devuelve la función que están ejecutando a un *futuro*. Además, se bloquea cualquier intento de usar el valor de retorno por parte de procesos consumidores (que es cuando se ejecuta la operación de conversión de tipos) hasta que haya sido asignado el valor a la variable de instancia del *futuro*. Para informar que está disponible, existe un semáforo (*Sync*[12]) que sirve para establecer una restricción de sincronización entre procesos productores y consumidores que intenten utilizar la variable al mismo tiempo.

La operación de conversión de tipos se utiliza para convertir un objeto del tipo *FutureType* a un valor de tipo *Anytype* que pueda ser utilizado en los cálculos realizados por el código de la aplicación del usuario. La declaración de *Anytype* para una variable *futuro* bloquea inicialmente, lo cual se interpreta como que no está disponible actualmente el valor de la variable, pudiéndolo estar en un futuro. Es decir, cuando el valor no se encuentra disponible para su procesamiento inmediato, será un *futuro*. Cuando se encuentre disponible, el tipo del futuro cambiará al tipo del valor que el

[11] Ver sección II.3.4.2 para detalles de definición e implementación de un *Mutex*.
[12] Ver sección II.3.4.3 para detalles de definición e implementación de un *Sync*.

código de la aplicación necesita para procesar, por ejemplo, una expresión que incluye al *futuro* como término o factor.

II.2.2.2.2. Gramática libre del contexto para la definición de clases PO

```
Definición de una clase PO::=CLASS <tipo de clase> <nombre de clase> [herencia]
                                {
                                 [<Definición de variables de instancia>]
                                 [<Definición de métodos públicos>]
                                 [<Definición de métodos privados>]
                                 [<Definición de restricciones de sincronización>]
                                };

tipo de clase::= ABSTRACT | CONCRETE

nombre de clase::= <letra mayúscula>{letra | dígito}*

herencia::= EXTENDS OF <nombre de clase>

letra::= <letra mayúscula> | <letra minúscula>

letra mayúscula::= A| B| C| D| E| F| G| . . . | Y| Z

letra minúscula::= a| b| c| d| e| f| g| . . . | y| z

dígito::= 0| 1| 2| 3| 4| 5| 6| 7| 8 | 9

Definición de variables de instancia::= {<tipo> <nombre variable>{,nombre variable}*;}*

tipo::= <tipo primitivo>|<tipo array>|<tipo clase>| ASOCIACION| ANYTYPE| METHOD|
FUTURETYPE

tipo primitivo::= CHAR | INT | REAL | DOUBLE | LONG | FLOAT | VOID| BOOL| STRING

tipo array::= <tipo>[ ]

tipo clase::= <nombre de clase>

nombre variable::= letra{letra | dígito}*

Definición de métodos públicos::={ PUBLIC <tipo> <Nombre Método>([<parámetros>])
                                      [<declaración de variables locales>]
                                      {
                                       <cuerpo>
                                      } }*


Definición de métodos privados::= { PRIVATE <tipo> <Nombre Método>([<parámetros>])
                                        [<declaración de variables locales>]
                                        {
                                         <cuerpo>
                                         } }*

Nombre Método::= letra{letra}*[{dígito}*] | OPERATOR[Operador]

Operador::= + | - | * | / | = | ++ | -- | += | -= | =+ | =-

Parámetros::= <tipo> <nombre parámetro>{,<tipo> <nombre parámetro>}*

Nombre parámetro::= letra{letra | dígito}*

Declaración de Variables locales::= VAR
                                    <variables locales>{; <variables locales>}*;

variables locales::= <tipo> <nombre variable>{,<nombre variable>}*

cuerpo::=  ABSTRACT; | {<sentencia>}*
```

```
sentencia::=   <asignación>
             | <condición>
             | <ciclo>
             | <ejecución de método>
             | RETURN <expresión>
             | <composición de objetos paralelos>

asignación::=  <nombre de variable> = <expresión>;
             | <definición de objeto> = <creación de un objeto>;

<definición de objeto>::= <tipo clase> <nombre de objeto>

<nombre de objeto>::= <letra>{letra | digito}*

<creación de un objeto>::=<nombre de clase> CREATE ([<parámetros>])

<uso de un objeto>::=  <nombre de objeto>. <nombre variable>;
                     | <nombre de objeto>. <Nombre método>( [argumentos]);

expresión::= HEAD(ASOCIACION[ ])
           | TAIL (ASOCIACION[ ])
           | CONS( ANYTYPE[ ], ANYTYPE)
           | <creación de un objeto>
           | <uso de objeto>
           | <nombre de objeto> EVAL (<nombre método>, argumento)

condicion::= IF (expresión booleana)
               {
                sentencia;{sentencia;}*
               }
             ELSE {
                    sentencia;{sentencia;}*
                  }

ciclo::= <ciclo condicional> | <ciclo no condicional>

ciclo condicional::= WHILE (expresión booleana)
                       {
                        sentencia;{sentencia;}*
                       }

ciclo no condicional::=FOR <nombre variable>=(expresión entera, expresión entera)
                         {
                           sentencia;{sentencia;}*
                         }

ejecución de método::=<nombre método>( [argumentos])

composición de objetos paralelos::= <modo síncrono> | <modo asíncrono>

modo síncrono::= <nombre de objeto>.<nombre método>([argumentos]);

modo asíncrono::=THREAD <nombre de objeto>.<nombre método>([argumentos]);

Definición de restricciones de sincronización::=  MUTEX ( <nombre método> )
                                                 | MUTEX ( <nombre método>, <nombre
método> )
                                                 | SYNC (<nombre método>, <nombre
método>)
                                                 | MAXPAR(<nombre método>)
```

II.3. DEFINICIÓN SINTÁCTICA Y SEMÁNTICA DE LAS CLASES BASE DE UN CPAN

Como ya se ha descrito, un *CPAN* proviene de la composición de un conjunto de objetos de tres tipos. En particular, cada *CPAN* esta compuesto por un objeto *manager*,

algunos objetos *stage* y un objeto *collector* para cada petición realizada por los objetos clientes del *CPAN*. Además, para cada *stage* del *CPAN*, un objeto esclavo estará encargado de la implementación de la parte secuencial del cómputo que se pretende llevar a cabo en la aplicación o en el algoritmo distribuido y paralelo. En *PO* las clases básicas necesarias para definir los objetos *manager, colector, stages* de un *CPAN* son:

- la clase abstracta *ComponentManager*
- la clase abstracta *ComponentStage*
- la clase concreta *ComponentCollector*

Una instancia *PO* de una clase concreta derivada de la clase *ComponentManager* (llamada *manager*) representa un *CPAN* dentro de una aplicación programada según el modelo de objeto paralelo. Las instancias (llamadas stages) de una clase concreta derivada de la clase *ComponentStage* se conectan entre sí para implementar la composición de *stages*. Cada *stage* comanda la ejecución de un objeto *PO*, llamado esclavo (*slave*), que es controlado por el propio *stage*.

La creación de los *stages* y de los *colectores* y sus interacciones posteriores son manejadas transparentemente al código de la aplicación por el *manager*. Desde el punto de vista de un usuario interesado en reutilizar el comportamiento paralelo ya definido en algunas clases *CPAN*, la clase de interés será la del *manager*. Cuando un usuario está interesado en usar un *CPAN* dentro de una aplicación, tiene que crear una instancia de una clase manager concreta, esto es, una que implemente el comportamiento paralelo requerido por la aplicación y que la inicialice con la referencia a los objetos esclavos que van a ser controlados por cada *stage* y el nombre del método solicitado. Las definiciones sintácticas siguientes han sido escritas utilizando la gramática libre de contexto de la sección II.2.2.2 de éste capítulo.

II.3.1. La clase abstracta ComponentManager

Define la estructura genérica del componente *manager* de un *CPAN*, de donde se derivarán todos los *manager* concretos dependiendo del comportamiento paralelo que se contemple para la creación del *CPAN*. Toda instancia concreta de un *manager* acepta como entrada una lista de n-asociaciones. Una asociación es un par de elementos, a saber: un objeto esclavo y el nombre del método que tiene que ser ejecutado por dicho

objeto. Los objetos esclavos son entidades externas que contienen un algoritmo secuencial que ejecutan a través de uno de sus métodos.

Una vez que el *manager* ha capturado la lista de n-asociaciones, éste genera los *stages* concretos, uno para cada asociación y entonces cada *stage* se convierte en responsable de un objeto esclavo, junto con el método de ejecución asociado. A su vez, los *stages* se conectan entre sí, de acuerdo con el patrón paralelo que haya sido implementado en el *CPAN*.

Finalmente el manager lleva a cabo el procesamiento de un cómputo, a través de la ejecución de uno de sus métodos (el método *execution( )*). Para ello es necesario pasarle al método los datos de entrada con los que se quiere trabajar. El *manager* crea entonces un componente *collector* y le pasa a los *stages* la referencia a dicho objeto *collector*, así como los datos de entrada. Los *stages* los procesan y, de acuerdo a como estén conectados unos con otros, se pasarán los resultados que vayan obteniendo. Al final el *collector* recogerá los resultados de los *stages* para regresarlos al *manager*, quien finalmente transmitirá los resultados al exterior del *CPAN* (al código de la aplicación de usuario que lo utilice).

```
CLASS ABSTRACT ComponentManager
    {
      ComponentStage[] stages;

      PUBLIC VOID init (ASOCIACION[ ] list )
            {
             ABSTRACT;
            }

      PUBLIC ANYTYPE execution(ANYTYPE datain)
          VAR
             ComponentCollector res;
          {
             res = ComponentCollector CREATE();
             commandStages(datain,res);
             RETURN res.get();
           }

      PRIVATE VOID commandStages(ANYTYPE datain, ComponentCollector res)
          {
           ABSTRACT;
          }

       MAXPAR (execution);
     };
```

El mismo manager puede ser utilizado para llevar a cabo más cálculos en paralelo. La política de planificación utilizada para ello es la restricción de

sincronización del *"paralelismo máximo"* o *MAXPAR*[13] aplicado al método "*execution()*".

II.3.2. La clase Abstracta ComponentStage

Define la estructura genérica del componente *stage* de un *CPAN*, así como de sus interconexiones, de donde se derivarán todos los *stages* concretos dependiendo del comportamiento paralelo que se contemple en la creación del *CPAN*.

Toda instancia concreta de un *stage* acepta como entrada una lista de asociaciones *objeto_esclavo-método* para trabajar con ella y se conecta o no con el siguiente *stage* de la lista de asociaciones, dependiendo del patrón paralelo que implementen, de forma que cuando el manager comanda a los *stages* en paralelo, cada uno de ellos hace que el objeto esclavo ejecute su método asociado, el *stage* captura los resultados y los envía al siguiente *stage* o al *collector*, dependiendo de la estructura implementada (todo ello dentro de un método llamado *request( )*). A su vez, cada *stage* puede comandar a otros en la ejecución del cómputo iniciado por el *manager*.

```
CLASS ABSTRACT  ComponentStage
  {
   ComponentStage[] otherstages;
   BOOL am_i_last;
   METHOD meth;
   OBJECT obj;

   PUBLIC VOID init (ASOCIACION[ ] list)
      VAR
       ASOCIACION item;
     {
      item = HEAD(list);
      obj = item.obj;
      meth= item.meth;
      if (TAIL(list) == NULL)
      am_i_last = true;
     }

   PUBLIC VOID request (ANYTYPE datain, ComponentCollector res)
      VAR
       ANYTYPE dataout;
     {
      dataout = EVAL (obj,meth, datain);
      IF (am_i_last)
         TREAD res.put(dataout)
      ELSE commandOtherStages (dataout, res);
     }

   PRIVATE VOID commandOtherStages(ANYTYPE dataout, ComponentCollector res)
     {
      ABSTRACT;
```

[13] Su definición e implementación se encuentran en la sección II.3.4.1.

```
    }
   MAXPAR (request);
  };
```

Nuevamente como en el caso anterior puede haber más de una petición de la operación "*request( )*" ejecutándose en paralelo, por lo que la política de planificación que ha de ser utilizada para su correcta implementación es la que proporciona la restricción de sincronización de *"paralelismo máximo"*, aplicado ahora al método "*request( )*".

II.3.3. La clase Concreta ComponentCollector

Define la estructura concreta del componente *collector* de cualquier *CPAN*. Este componente implementa básicamente un buffer *multi-item,* donde se irán almacenando los resultados de los *stages* que hagan referencia al *collector*. De esta forma se puede obtener el resultado del cálculo iniciado por el *manager*.

```
CLASS CONCRETE ComponentCollector
  {
   VAR
    ANYTYPE[] content;

   PUBLIC VOID put (ANYTYPE item)
     {
      CONS(content, item);
     }

   PUBLIC ANYTYPE get()
      VAR
          ANYTYPE result;
     {
      result = HEAD(content[]);
      content = TAIL(content[]);
      RETURN result;
     }
   SYNC(put,get);
   MUTEX(put);
   MUTEX(get);
  };
```

En este caso las restricciones de sincronización utilizadas son *SYNC* y *MUTEX* para poder sincronizar concurrentemente la comunicación entre los métodos *put( )* y *get( )* y proporcionar exclusión mutua.

II.3.4. Las restricciones de sincronización MaxPar, Mutex y Sync

Es necesario el disponer de mecanismos de sincronización, cuando se producen peticiones paralelas de servicio en un *CPAN*, para que los objetos que lo conforman puedan gestionar varios flujos de ejecución concurrentemente y, al mismo tiempo, garanticen la consistencia en los datos que se están procesando. Dentro de cualquier *CPAN* se pueden utilizar las restricciones *MAXPAR, MUTEX y SYNC* para la correcta programación de sus métodos.

II.3.4.1. MaxPar

El *paralelismo máximo o MaxPar* es el número máximo de procesos que pueden ejecutarse al mismo tiempo. Dicho de otra forma, el *MAXPAR* aplicado a una función representa el número máximo de procesos que pueden ejecutar esa función concurrentemente. En el caso de los *CPAN*, el paralelismo máximo se aplica a la función "*execution( )*" de la clase *ComponentManager* y a la función "*request( )*" de la clase *ComponentStage.*

En el primer caso significa que el número máximo de procesos que podrán ejecutar la función "*execution( )*" en paralelo será como mucho el del número de objetos *manager* que se hayan creado. Por ejemplo, si se crean dos objetos *manager* y uno de ellos resuelve tres problemas diferentes, mientras que el otro resuelve cinco, significa que habrá ocho procesos queriendo ejecutar la función "*execution( )*"en paralelo, sin embargo al aplicar el *MAXPAR*, se limitaría a un máximo de dos procesos la ejecución concurrente de "*execution( )*", bloqueándose los restantes y estableciéndose una comunicación de sincronización entre los que se están actualmente ejecutando y los que están en espera de serlo.

En el segundo caso el número máximo de procesos que podrán ejecutar la función "*request( )*" en paralelo será como máximo el número de objetos *stage* que se hayan creado por parte de un objeto *manager*. Por ejemplo, si un objeto *manager* (es decir, un *CPAN*) resuelve dos problemas diferentes y para cada problema se crean tres objetos *stage* eso significa que habrá seis procesos queriendo ejecutar la función "*request( )*"en paralelo para satisfacer la petición de servicio hecha por el manager que los comanda, sin embargo, al aplicar el *MAXPAR*, se limitaría a un máximo de tres procesos la ejecución concurrente de "*request( )*", bloqueándose los restantes y

estableciéndose una comunicación de sincronización entre los procesos en ejecución y los que están en espera de serlo.

El algoritmo implementado de la restricción de sincronización *MAXPAR* es el siguiente:

1. Incorporar una variable de clase en la clase de interés e inicializarla a cero.
2. Cada vez que se genere una instancia de esa clase de interés, habrá que actualizar su variable de clase sumándole uno al valor que tenía anteriormente, de forma que el número de instancias creadas corresponderá al número máximo de procesos que puedan ejecutar una función particular
3. En la función particular de interés implementar un semáforo de la siguiente forma:
   - 3.1. Si n es el número máximo de procesos que pueden ejecutar la función particular de interés y si k es el número de procesos que actualmente se están ejecutando entonces:
   - 3.2. Si $k \leq n$ ejecutar en paralelo los k procesos, sino bloquear los k-n procesos y despertarlos hasta que por lo menos uno de los n procesos que se están ejecutando haya terminado de forma que siempre se mantenga el límite de a lo más n procesos ejecutándose concurrentemente.

II.3.4.2. Mutex

La restricción de sincronización *mutex* lleva a cabo una exclusión mutua entre procesos que quieren acceder a un objeto compartido. El *mutex* preserva secciones críticas de código y obtiene acceso exclusivo a los recursos. En el caso de los *CPANS*, la restricción *mutex* aplicada a una función representa el uso de esa función por parte de un proceso cada vez. En otras palabras, el *mutex* permite que uno y sólo uno de los procesos ejecute la función, bloqueando a todos los demás procesos que quieran hacer uso de su servicio hasta que el que la está ejecutando termine. El *mutex* dentro del *CPAN* se aplica tanto a la función *get( )* como a la función *put( )* de un objeto *collector.*

El algoritmo que implementa la restricción de sincronización *Mutex* es el siguiente:

1. Utilizar una variable de condición que funcione como una bandera y un candado mutex.
2. Inicializar la bandera a falso
3. En la función particular de interés implementar un semáforo de la siguiente forma:
   - 3.1. Aquel proceso que adquiera el candado pondrá la bandera a verdadero y podrá ejecutar la función de interés

3.2. Mientras la bandera sea cierta todos los demás procesos que intenten adquirir el candado para ejecutar la función se bloquearán y esperarán a que la bandera sea falsa para que alguno de ellos pueda tomar el candado

3.3. El proceso que actualmente posee el candado ejecutará lo que tenga que ejecutar

3.4. Al terminar su ejecución, pondrá la bandera a falso y liberará el candado

3.5. Aquel proceso que este al principio en la cola de espera, adquirirá el candado repitiéndose la secuencia desde el paso 3.1.

II.3.4.3. Sync

La restricción *SYNC* no es más que una sincronización del tipo productor/consumidor utilizada, por ejemplo, para programar los métodos *put( )* y *get( )* de la clase *ComponentCollector*. *SYNC* ayuda a sincronizar dichos métodos de forma que el método *get( )* pueda ejecutarse siempre y cuando el método *put( )* le confirme antes que ha colocado por lo menos un elemento en el contenedor compartido, en este caso en *content*, de otra forma, el proceso que ejecuta *get( )* quedará bloqueado hasta la notificación de la próxima ejecución del método *put( )*. Sin embargo, el método *get( )* no necesita confirmarle a *put( )* si ha utilizado o no un elemento del contenedor que comparten, ya que en la implementación que se ha realizado, tal contenedor no tiene un tamaño predefinido, sino que dinámicamente va creciendo conforme se necesite, aún cuando el método *get( )* no haya extraído ningún elemento. En otras palabras, el contenedor “nunca” se satura.

El algoritmo de implementación es el siguiente:

1. Utilizar una variable de condición (digamos, n_items) que contabilice el número de elementos colocados en un contenedor.

2. Inicializar n_items a cero

3. Implementar un semáforo compartido por las funciones productor y consumidor de la siguiente forma:

   3.1. Un proceso consumidor permanecerá bloqueado mientras n_items sea igual a cero, pues con ello se indica que el proceso productor no ha colocado ningún dato en el contenedor que comparten.

   3.2. Un proceso productor colocará un dato en el contendor, adquirirá el candado y aumentará en uno el valor de n_items.

   3.3. Despertará al proceso consumidor y continuará ejecutándose en caso de procesar mas datos

3.4. En el momento en que el n_items sea mayor a cero, el consumidor adquirirá el candado, disminuirá en uno el valor de n_items indicando con ello el procesamiento de un dato del contenedor y sacará el dato del contenedor.

3.5. Mientras el n_items sea mayor a cero el consumidor se estará ejecutando en paralelo con el productor.

II.4. CARACTERÍSTICAS IMPORTANTES DEL MODELO CPAN

El modelo *CPAN* que se presenta es genérico y asume las siguientes características que están disponibles en los ambientes de programación concurrente orientados a objetos:

- La capacidad de solventar los requerimientos de servicios de los objetos en los modos de comunicación síncrono, asíncrono y futuro asíncrono.
- La capacidad de los objetos de tener paralelismo interno, es decir, de gestionar varias peticiones de forma concurrente.
- La disponibilidad de tener mecanismos de sincronización para atender peticiones paralelas de servicio.
- La disponibilidad de contar con un mecanismo flexible de control de tipos, es decir, la capacidad de asociar tipos dinámicamente a los parámetros de los métodos de los objetos.

La primera característica es necesaria para permitir al *manager* y a los *stages* de un *CPAN* implementar cualquier patrón de comunicación sin un control centralizado. En la segunda característica, para que los objetos tengan la capacidad de atender peticiones de forma concurrente, es necesario que el *CPAN* sea capaz de manejar varios flujos de ejecución al mismo tiempo.

La necesidad de la sincronización es un requerimiento obvio para cualquier sistema que soporte paralelismo interno. Es necesario si se quiere garantizar consistencia en el estado del objeto. Dentro de los *CPANS*, la sincronización es necesaria en la implementación de cualquier patrón paralelo.

Finalmente para garantizar la genericidad de los *CPANS*, el sistema deberá suministrar la capacidad del manejo de tipos de datos genéricos. Si un *CPAN* es diseñado para no implementar en sí mismo la parte algorítmica y definir sólo el patrón paralelo de interacción, éste debe ser capaz de adaptarse a su interfaz para manejar

varios tipos de datos diferentes de entrada y salida, dependiendo del algoritmo particular que implementen los objetos esclavos. Para hacer esto se tienen dos soluciones:

1. Tener la capacidad para manejo de tipos de datos genéricos, esto es, datos cuyos tipos son dinámicamente especificados como parámetros en la creación de los objetos.
2. Tener la capacidad de que esos objetos manejen datos sin tipo, o dicho de otra forma, datos que sean tratados como un *stream* (flujo) no definido sin ningún chequeo de tipos.

Capítulo III:

DISEÑO DE LOS CPANS: Construcción de los patrones farm, pipe y treeDV como CPANS

III.1. INTRODUCCIÓN

Con el conjunto básico de clases del modelo de programación de *PO* se pueden construir *CPANS* concretos. Para construir un *CPAN*, primero se debe tener claro el comportamiento paralelo que se necesita implementar, de forma que el *CPAN* en sí mismo sea dicho patrón. Existen varios patrones paralelos de interacción como son los *farms* (o granjas de procesos), los *pipes* (o cauces), los trees (o árboles), los *cubes* (o cubos), los *grids* (o mallas), las *matrices de procesos*, etc.

Una vez identificado el comportamiento paralelo, el segundo paso consiste en elaborar un bosquejo gráfico de su representación como mera técnica de diseño informal de lo que será posteriormente el procesamiento paralelo del sistema objetivo. También sirve para ilustrar sus características generales, y permitirá después definir su representación como *CPANS*, siguiendo el modelo propuesto en el capítulo anterior.

Cuando ya se tiene concretizado el modelo de un *CPAN* que define un patrón paralelo específico digamos, por ejemplo, un *tree*, o alguno de los anteriormente mencionados, el paso siguiente será realizar su definición sintáctica y semántica. Finalmente, se traduce la definición sintáctica a un *CPAN*, programado en el entorno de programación más adecuado para su implementación paralela verificando que la semántica resultante sea la correcta, probando con varios ejemplos distintos para demostrar su genericidad y observando el rendimiento de las aplicaciones que lo incluyan como un componente software.

Los patrones paralelos trabajados en la presente tesis han sido el *pipeline*, el *farm* y el *treeDV*[14], por tratarse de un conjunto significativo y reutilizable de patrones en múltiples aplicaciones y algoritmos, siendo actualmente utilizados con diferentes propósitos, en diferentes áreas y con diferentes aplicaciones según la literatura que hay sobre el tema.

- ***El pipeline***, esta compuesto por un conjunto de estados interconectados uno después de otro. La información sigue un flujo de un estado a otro.
- ***El farm***, se compone de un conjunto de procesos trabajadores y un proceso controlador. Los trabajadores se ejecutan en paralelo hasta alcanzar un objetivo

[14] El patrón treeDV implementa el paradigma de programación de Divide y Vencerás mediante la utilización de árboles binarios.

común. El controlador es el encargado de distribuir el trabajo y de controlar el progreso del cómputo global.

- ***En el treeDV***, la información fluye desde la raíz hacia las hojas o viceversa. Los nodos que se encuentran en el mismo nivel en el árbol se ejecutan en paralelo haciendo uso de la técnica de diseño de algoritmos denominada *Divide y Vencerás* para la solución del problema.

Estos patrones paralelos conforman la librería de clases propuesta bajo el modelo del *CPAN*.

III.2. EL CPAN PIPE

Se presenta la técnica del procesamiento paralelo del *pipeline* como una *Composición Paralela de Alto Nivel*, aplicable a un amplio rango de problemas que son parcialmente secuenciales en su naturaleza. El *CPAN Pipe* garantiza la paralelización de código secuencial utilizando el patrón *PipeLine*.

III.2.1. La técnica del PipeLine

Utilizando la técnica del *Pipeline*, la idea es dividir el problema en una serie de tareas que tienen que ser completadas, una después de otra. En un *pipeline* cada tarea puede ser ejecutada por un proceso, thread o procesador por separado [ROB], (fig.III.1):

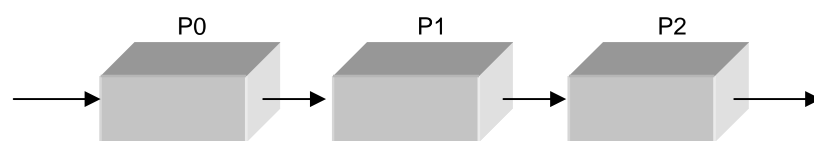


Fig. III.1. PipeLine

Algunas veces los procesos del *pipeline* son llamados *etapas (stages)* del *pipeline* [SEL99]. Cada etapa puede contribuir a la solución del problema total y puede pasar la información que es necesaria a la siguiente etapa del *pipeline*. Este tipo de paralelismo es visto muchas veces como una forma de descomposición funcional.

El problema es dividido en funciones separadas que pueden ser ejecutadas individualmente, pero con esta técnica, las funciones se ejecutan en sucesión. Un problema puede entonces ser formulado como un *pipeline* si se puede dividir en una serie de funciones (descomposición funcional, [ROB]) que podrían ser ejecutadas por las etapas del *pipe*. Por ejemplo, supongamos que queremos ordenar un conjunto de datos en desorden de mayor a menor (en orden descendente) pero se tiene ya implementado un algoritmo de ordenación de menor a mayor (en orden ascendente). Si se utiliza este algoritmo de ordenación será necesario invertir la secuencia de los datos ya ordenados. El *pipeline* se constituiría de dos etapas donde en cada una de ellas habría asignada una función que llevaría a cabo un proceso específico (figura III.2.)

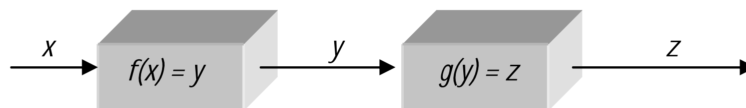


Fig. III.2. Pipeline con descomposición funcional

Donde:

- *x* representa al conjunto de datos inicial que supondremos está en desorden.
- *f(x)* representa la función *"ordena"* que recibe como entrada el conjunto de datos a ordenar y da como salida la ordenación en orden ascendente (de menor a mayor) del conjunto de datos que le llegan.
- *y* representa entonces la salida de la función *f(x)*, es decir, los datos ordenados.
- *g(y)* representa la función *"invierte"* que recibe el resultado de la función *"ordena"* para dar como salida el conjunto de datos previamente ordenados pero invertidos en su secuencia para tener un orden descendente (de mayor a menor)
- *z* es precisamente la salida de la función *g(y)* en la última etapa del *pipeline*.

Suponiendo que el conjunto de datos con los que se trabaja en este ejemplo son números enteros, la secuencia de resultados dentro del *pipeline* sería la mostrada en la fig. III.3.

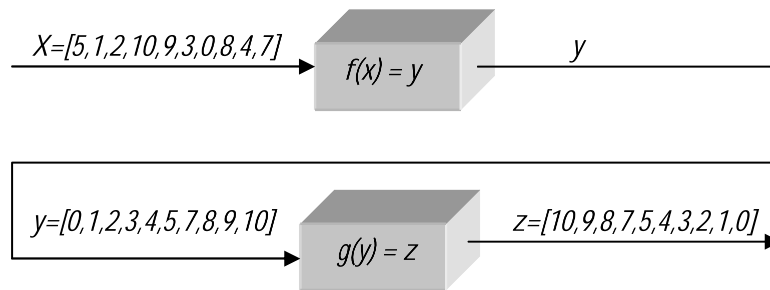


Fig. III.3. Secuencia de resultados en un Pipeline

Dado entonces un problema que puede ser dividido en una serie de tareas secuenciales, el enfoque del *pipeline* puede proporcionar incremento en la velocidad de ejecución de los siguientes tres tipos de cálculos:

1. Si más de una instancia del problema completo puede ser ejecutada.
2. Si una serie de datos pueden ser procesados y cada uno requiere de múltiples operaciones.
3. Si la información que requiere el siguiente proceso para iniciar su cálculo es pasada después de que el proceso corriente haya completado todas sus operaciones internas.

Con esta técnica muchos de los problemas computacionales que se llevan a cabo de forma secuencial pueden ser fácilmente paralelizados como un *pipeline*. Ejemplo de ello son:

- La suma de números
- El ordenamiento de números
- La generación de números primos
- El solucionar un sistema de ecuaciones lineales

III.2.2. Representación del Pipeline como un CPAN

La Fig.III.4. representa el patrón paralelo de comunicación *PipeLine* como un *CPAN*.

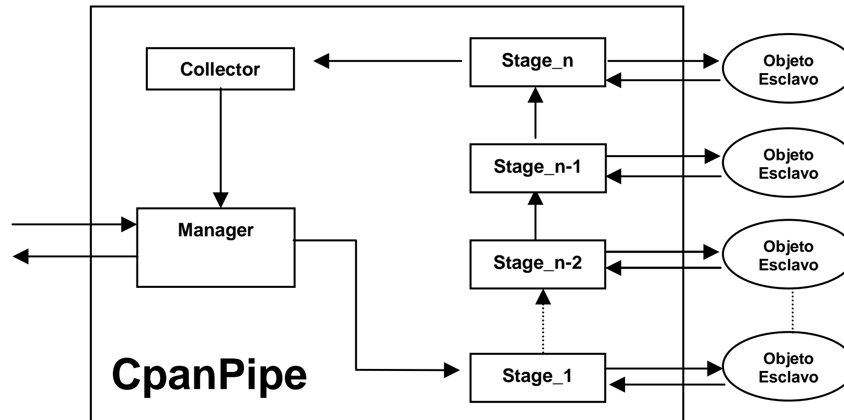


Fig. III.4. El CPAN de un PipeLine

Los objetos *Manager* y *stage_i* del modelo gráfico del *CpanPipe* son instancias de clases concretas digamos *PipeManager* y *PipeStage* que heredan las características de las clases base *ComponentManager* y *ComponentStage* respectivamente y poder así redefinir los métodos abstractos de las superclases. El objeto *Collector* es el único que será una instancia de la clase base *ComponentCollector* ya que ésta es una clase concreta. Una vez creados los objetos y conectados de manera apropiada de acuerdo al patrón paralelo *Pipeline*, se tendrá entonces un *CPAN* para un tipo de patrón paralelo específico y poder así resolver el problema de los objetos esclavos asociados a los *stages*.

III.2.3. Definición Sintáctica y Semántica del CPAN Pipe

El *Cpan Pipe* esta representado por la clase *PipeManager*, que hereda de *ComponentManager*, e implementa un patrón de comunicación *pipeline*, cuyos *stages* son instancias de la clase *PipeStage*, que hereda de *ComponentStage*. Cualquier objeto *PipeManager* se encarga sólo del primer *stage* del *pipeline* en su inicialización. Durante la ejecución de una petición de servicio, solamente es comandado el primer *stage* .

```
CLASS CONCRETE PipeManager EXTENDS OF ComponentManager
  {
   PUBLIC VOID init(ASOCIACION[] list)
     {
      stages[0] = PipeStage CREATE( list );
     }

   PRIVATE VOID commandStages(ANYTYPE datain, ComponentCollector res)
     {
      THREAD stages[0].request(datain,res);
     }
};
```

Los objetos de la clase *PipeStage* crean el siguiente *stage* del *pipeline* durante su fase de inicialización. En la ejecución de su operación *request( )*, un objeto *stage* comanda directamente al siguiente y es el último el que envía el resultado al objeto *Collector* (instancia de la clase *ComponentCollector*), cuya referencia se transmite dinámicamente *stage* por *stage*.

```
CLASS CONCRETE  PipeStage EXTENDS OF ComponentStage
  {
   PUBLIC VOID init (ASOCIACION[] list)
     {
      stage.init(list);
      IF (! am_i_last)
        {
         otherstages[0] = PipeStage CREATE(TAIL(list));
        }
      }
   PRIVATE VOID commandOtherStages(ANYTYPE datain, ComponentCollector res)
      {
       THREAD otherstages[0].request(datain, res);
      }
   };
```

Las operaciones *execution( ) y request( )* se heredan de sus respectivas superclases y las operaciones privadas *commandStages(),* de la clase *PipeManager,* y *commandOtherStages( ),* de la clase *PipeStage*, junto con la operación *init( )* se redefinen. Sin embargo no hay problemas de sincronización en sus definiciones ya que las restricciones de sincronización son heredadas de sus superclases.

III.2.4. Uso del CPAN Pipe

Supongamos que el problema a resolver es el que se describió en la sección III.2.1, referente al uso del *Pipeline* como descomposición funcional, es decir, supongamos que queremos ordenar un conjunto de datos en desorden de mayor a menor (en orden descendente) pero se tiene ya implementado un algoritmo de ordenación de menor a mayor (en orden ascendente). Si se utiliza este algoritmo de ordenación será necesario invertir la secuencia de los datos ya ordenados.

El patrón óptimo para poder ejecutar diferentes algoritmos como una descomposición de funciones es el patrón *Pipeline* o *Cauce*. Dado que ya se tiene implementado este patrón como *CPAN*, se procederían a crear los objetos esclavos que serán instancias de las diferentes clases que implementan los algoritmos de ordenamiento y de cambio de orden en la secuencia de los datos, que el usuario tendría

que haber implementado previamente. En dichas clases el usuario deberá definir un método de ejecución del algoritmo de solución (Fig. III.5).

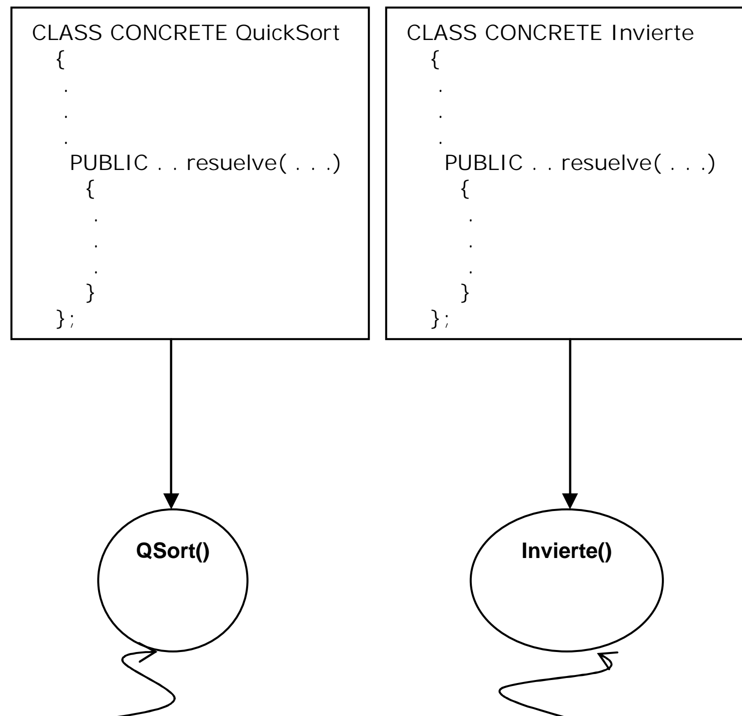


Fig. III.5. Objetos Esclavos y sus métodos de ejecución para el CPAN Pipe

De forma que:

1. Se crean los objetos esclavos que representan las instancias de las clases *QuickSort* e *Invierte* de forma respectiva y se almacenan en un array de referencias a *Object*:

```
Object obj[]= {
               QuickSort CREATE(. . .);
               Invierte CREATE(. . .);
              }
```

2. Se crean los objetos que representan los métodos asociados a los objetos esclavos y se almacenan en un array de referencias a *method*.

```
method meth[]= {
                method CREATE( . . .resuelve),
                method CREATE( . . .resuelve)
               };
```

3. Se crea una lista de asociaciones (objeto_esclavo, método asociado)

```
asociación pareja= crea_asociacion(obj, meth, 2);
```

4. Se crean las instancias de la clase *PipeManager* que se inicializarán con la lista de asociaciones del punto anterior.

```
PipeManager CpanPipe[] = {
                          PipeManager CREATE(pareja);
                          PipeManager CREATE(pareja);
                          }
```

En este momento, se crean automáticamente los *stages* internos necesarios para cada *CPAN PipeManager* y se les pasa a cada uno de ellos, es decir, a cada *stage*, una asociación. Los stages en cada *CPAN* se interconectan entre sí formando el *PipeLine* y se consideran listos para trabajar.

5. Se especifican los datos iniciales a procesar mediante la creación del tipo y datos definidos por el usuario como un objeto, en este ejemplo, *MyType*. Cada objeto *MyType* representa un problema a resolver (el conjunto de datos a ordenar y a invertir).

```
int nums1[tam1]={-11, -14, -6, . . ., -1, -15};
int nums2[tam2]={5, 1, 9, . . ., 11, 4, 8};

ANYTYPE data[]= {
                 MyType CREATE(nums1);
                 MyType CREATE(nums2);
                };
```

6. Se imprimen los datos iniciales en pantalla

```
FOR i=(0,num_problems)
   {
    imprime_datos(data[i]);
   }
```

7. Las instancias *PipeManager* se encuentran listas para trabajar, es decir, se pide la ejecución de la operación *execution( )* de cada instancia, en paralelo, para ordenar e invertir los arrays de datos que cada *CpanPipe* acepta en su operación *execution( )*. La ejecución paralela se logra usando los modos de comunicación síncrono, asíncrono y

futuro asíncrono[15], de manera transparente desde el programa principal del usuario tal como lo muestran las siguientes líneas de código

```
FUTURETYPE resul[num_problems];
FOR i=(0,num_problems)
    {
     resul[i]= THREAD CpanPipe[i].execution(data[i]);
    }
```

Es en este punto donde cada *CpanPipe* que es en sí mismo un objeto *Pipemanager*, crea su objeto *Collector*, y es pasado como referencia a su primer *stage* que tiene conectado junto con una copia del conjunto de datos a procesar. El *stage* inicial trabaja con los datos ejecutando por medio de su objeto esclavo asociado el algoritmo de ordenación, en este caso *QuickSort* a través de su método resuelve y el resultado es pasado al siguiente *stage* que es creado y comandado por el *stage* corriente. El segundo *stage* ejecuta de la misma forma su método *Invierte* a través de su objeto esclavo y el resultado que envía como salida son los datos ordenados pero invertidos en su secuencia. Este resultado es recibido por el objeto *collector* quien lo recopila y lo envía a su *PipeManager*, el cual a su vez envía la solución al exterior del *CPAN* quedando el resultado final en la variable *futura result* del ejemplo.

8. Finalmente se imprimen los resultados finales en pantalla

```
ANYTYPE resultados[num_problems];
FOR i=(0,num_problems)
    {
     resultados[i]=resul[i];
     imprime_datos(resultados[i]);
    }
```

III.3. EL CPAN FARM

Se muestra la técnica del procesamiento paralelo del patrón *Farm* como una *Composición Paralela de Alto Nivel* o *CPAN*.

[15] Para más detalles respecto remítase a la gramática libre del contexto definida en el capítulo II.

III.3.1. La técnica del Farm

El patrón paralelo de interacción *Farm*, se forma de un conjunto de procesos independientes entre sí, llamados procesos trabajadores, y un proceso que los controla, llamado el proceso controlador [ROB], [SEL99]. Los procesos trabajadores se ejecutan en paralelo hasta alcanzar un objetivo común para todos ellos. El proceso controlador es el encargado de distribuir el trabajo y de controlar el progreso del *farm* hasta hallar la solución del problema. La fig. III.6. muestra el modelo del *Farm*.

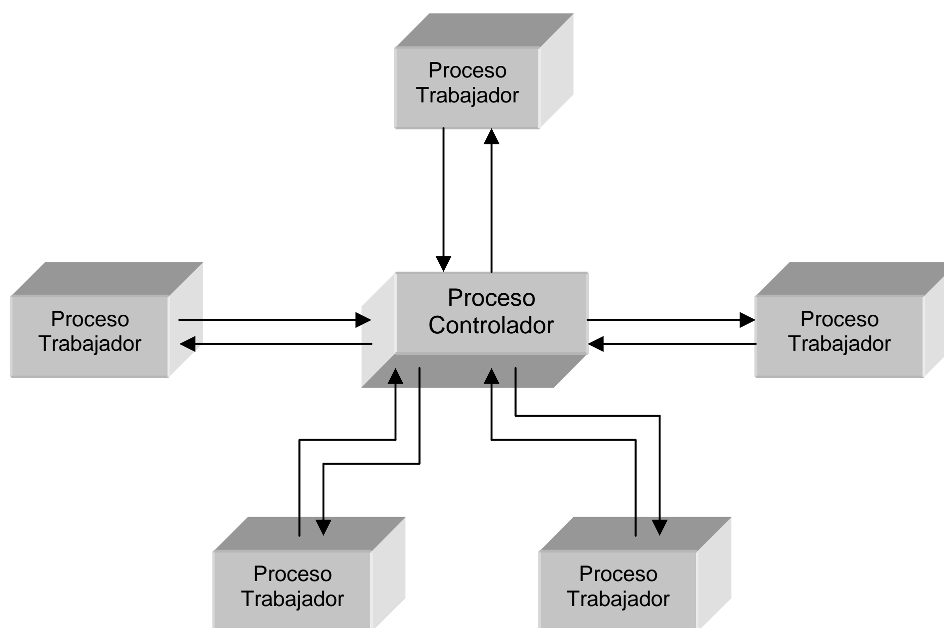


Fig. III.6. Farm con un proceso controlador y 5 procesos trabajadores

Con este modelo podría ser interesante observar el rendimiento de la ejecución en paralelo de varios algoritmos de ordenación con un mismo conjunto de datos para todos ellos. Si utilizamos como patrón un *Farm* para resolver el problema planteado, el diseño del modelo sería más o menos el que se muestra en la figura III.7.

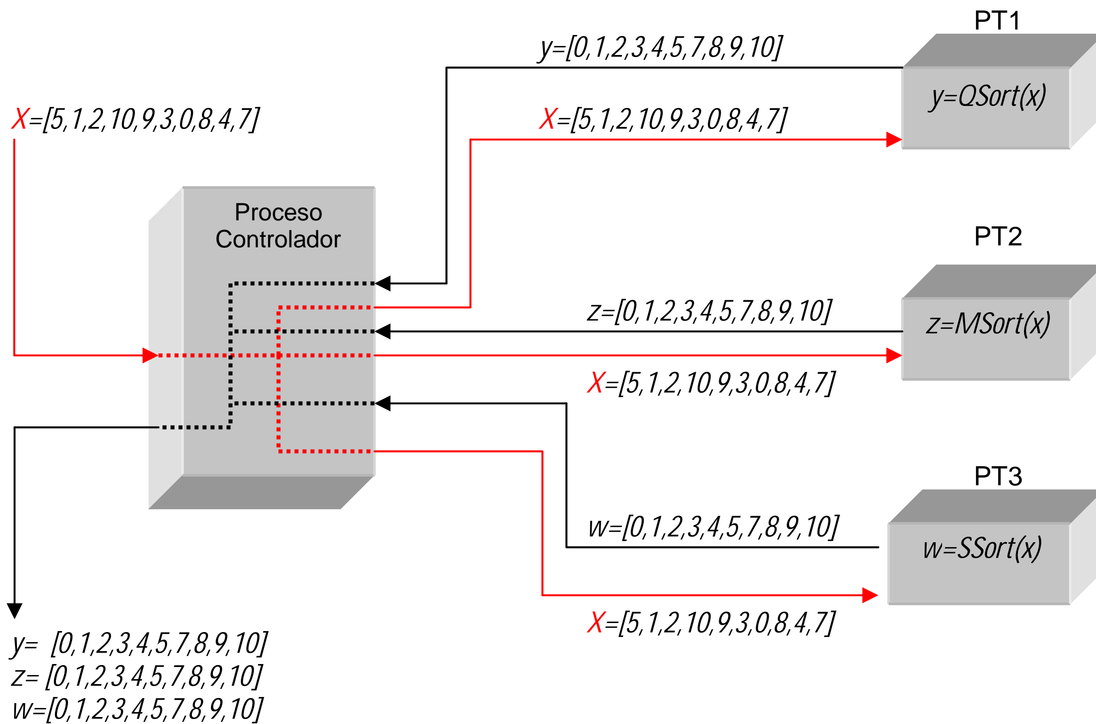


Fig. III.7. Farm que ejecuta 3 algoritmos de ordenación en paralelo

El *Farm* se forma de tres procesos trabajadores (*PT1, PT2, PT3*) y un proceso controlador. Cada proceso trabajador tiene asociado un algoritmo de ordenación diferente. En el caso del ejemplo, *PT1* tiene asociado el algoritmo *QuickSort*, a *PT2* se le asocia el algoritmo *MergeSort* y finalmente *PT3* tiene asociado el algoritmo *ShellSort*.

El proceso controlador recibe como entrada el conjunto de datos a ordenar (definido como el arreglo *X* en el ejemplo). Éste distribuye a sus procesos trabajadores la información y en paralelo se lleva a cabo la ejecución de los tres procesos *PT1, PT2 y PT3*. El resultado (*y, z, w*) es devuelto al proceso controlador quien finalmente los retornará como resultados finales también en forma paralela.

Con esta técnica muchos problemas computacionales pueden ser fácilmente paralelizados como un *farm* de procesos. Ejemplo de ello son:

- El cómputo de matrices.
- El ordenamiento de números.
- La solución de sistemas de ecuaciones.
- Solución al problema de caminos en grafos dirigidos a través de matrices de adyacencia.

III.3.2. Representación del Farm como un CPAN

La representación del patrón paralelo *FARM* como un *CPAN* se muestra en la fig.III.8.

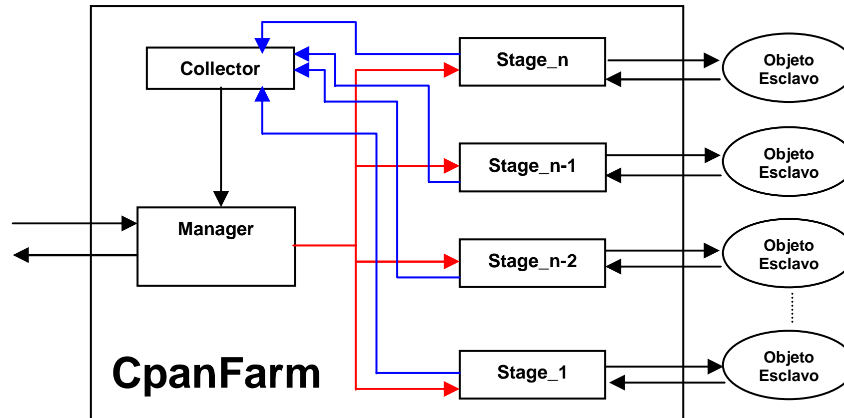


Fig. III.8. El Cpan de un Farm

Al igual que en el modelo anterior, los objetos *Manager y stage_i* son instancias de las clases concretas digamos *FarmManager y FarmStage*, que heredan de las clases base denominadas *ComponenManager* y *ComponentStage*, respectivamente. El objeto *Collector* es aquí también una instancia de la clase base *ComponentCollector*.

III.3.3. Definición Sintáctica y Semántica del CPAN Farm

Una primera política para la composición del *Farm* es que el *manager* espere sólo el primer resultado disponible dado por cualquiera de los *stages*, los cuales responden a una petición de servicio de forma asíncrona.

```
CLASS CONCRETE FarmManager EXTENDS OF ComponentManager
  {
 VAR
    INT nWorker;

   PUBLIC VOID init (ASOCIACION[] list)
      VAR
       asociacion[] newlist, INT i = 0;
     {
      WHILE (! (newlist = TAIL(list)))
        {
         stages[i++] = FarmStage CREATE (CONS(HEAD(list),NULL));
         list = newlist;
        }
      nWorker = i;
     }

   PRIVATE VOID  commandStages(ANYTYPE datain, ComponentCollector res)
      VAR
```

```
        INT i;
     {
      FOR i =(0,nWorker)
        {
         THREAD stages[i].request(datain, res);
        }
     }
   };
```

La clase concreta *FarmManager* hereda de *ComponentManager*. La operación *init()* crea todos los *stages* necesarios, mientras que la operación *execution( )* es lanzada en paralelo de forma asíncrona, distribuyendo datos a todos los *stages*, esperando el primer resultado disponible en el objeto *collector*. Como en el caso del *pipeline*, las restricciones de sincronización son heredadas de la clase abstracta *ComponentManager* sin ningún problema. Los stages del *Farm* son objetos de *FarmStage*, que heredan de *ComponentStage*:

```
CLASS CONCRETE FarmStage EXTENDS OF ComponentStage
   {
   };
```

Los *stages* del *Farm* no se conectan unos con otros[16]. El manager los comanda a todos en su ejecución y el resultado de cada uno es enviado al objeto *collector* y puesto a disposición del *manager*. Como los *stages* son ejecutados en paralelo de forma asíncrona, la política de planificación heredada de la superclase garantiza que el acceso al *collector*, necesario para retornar los resultados, se hará de una forma sincronizada por parte de dichos objetos.

III.3.4. Uso del CPAN Farm

Suponiendo que queremos procesar un arreglo que contiene datos que han de ser ordenados. Es conocido que, dependiendo de los datos, diferentes algoritmos de ordenación pueden mostrar diferentes rendimientos. Puede ser interesante ordenar los mismos datos mediante la ejecución de diferentes algoritmos en paralelo y esperar los resultados. El patrón paralelo identificado para resolver el problema es el *Farm*. Suponiendo que ya se tiene implementado este patrón como *CPAN*, Se procederían a crear los objetos esclavos que serán instancias de las diferentes clases de ordenamiento que el usuario tendría que haber implementado previamente. En dichas clases el usuario

[16] *am_i_last* es siempre *true* y la operación *CommandOtherStages*, como en la clase abstracta esta vacía.

deberá definir un método de ejecución del algoritmo de ordenación en cuestión (Fig. III.9).

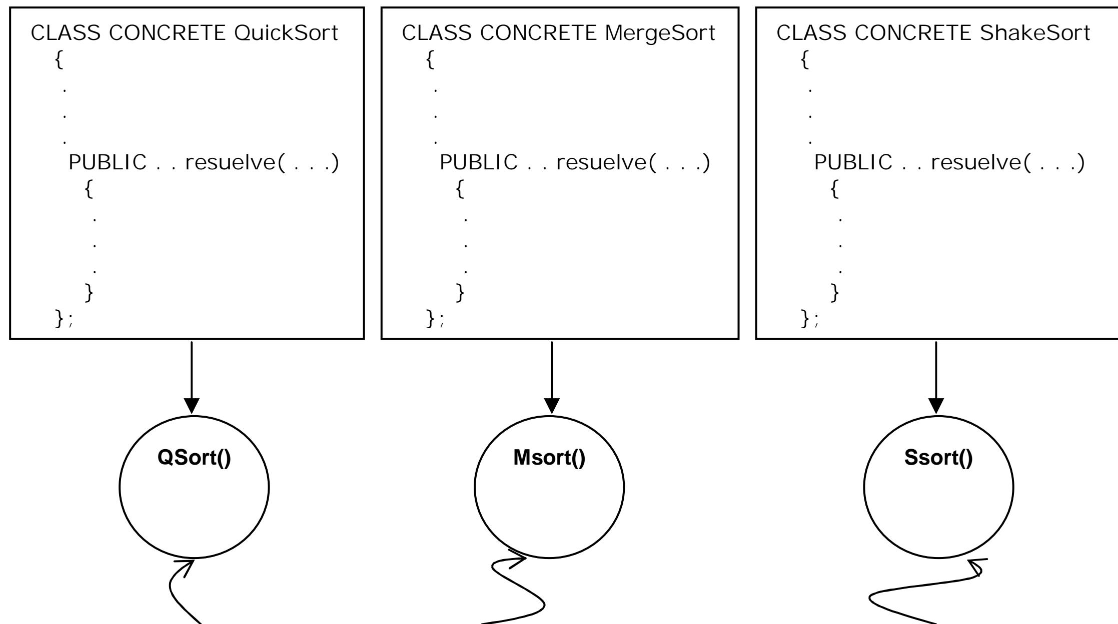


Fig. III.9. Objetos Esclavos y sus métodos de ejecución para el CPAN Farm

De forma que:

1. Se crean objetos de diferentes clases de ordenación que representan los objetos esclavos y se almacenan en un array de referencias a *object*:

```
Object obj[]= {
               Qsort CREATE(. . .),
               BubleSort CREATE( . . .),
               Isort CREATE(. . . )
              };
```

2. Se crean los objetos que representan los métodos asociados a los objetos esclavos y se almacenan en un array de referencias a *method*.

```
method meth[]={
               method CREATE (. . . resuelve),
               method CREATE (. . . resuelve),
               method CREATE (. . . resuelve)
              };
```

3. Se crea una lista de asociaciones (objeto_esclavo, método_asociado)

```
asociacion pareja=crea_asociacion(obj,meth,3);
```

4. Se crean las instancias de la clase *FarmManager* que se inicializarán con la lista de asociaciones del punto anterior.

```
FarmManager CpanFarm[] = {
                          FarmManager CREATE(pareja);
                          FarmManager CREATE(pareja);
                          }
```

5. Se especifican los datos iniciales a procesar mediante la creación específica del tipo y de los datos definidos como un objeto por el usuario, en este ejemplo dicho objeto se denomina *MyType*. Cada objeto *MyType* representa un problema a resolver

```
int nums1[t1]={-11,-14,-6,. . ., -1, -15};
int nums2[t2]={5,1,9, . . . ,11,4,8};

ANYTYPE data[]= {
                 MyType CREATE(nums1),
                 MyType CREATE(nums2)
                };
```

6. Se imprimen los datos iniciales en pantalla

```
FOR i =(0,num_problems)
     {
      imprime_datos(data[i]);
     }
```

7. Las instancias *FarmManager* se encuentran listas para trabajar, es decir, se pide la ejecución de la operación *execution( )* de cada instancia en paralelo para ordenar los arrays de datos que cada *CpanFarm* acepta en su operación *execution( )*.

```
FUTURETYPE resul[num_problems];
FOR i=(0,num_problems)
   {
    resul[i]= THREAD CpanFarm[i].execution(data[i]);
   }
```

8. Se imprimen los resultados finales en pantalla

```
ANYTYPE resultados[num_problems];
FOR i =(0,num_problems)
   {
    resultados[i]=resul[i];
    imprime_datos(resultados[i]);
   }
```

III.4. EL CPAN TREEDV

Finalmente, se presenta la técnica de programación *Divide y Vencerás* como un *CPAN*, aplicable a un amplio rango de problemas que pueden ser paralelizables bajo este esquema.

III.4.1. La técnica de Divide y Vencerás

La técnica de *Divide y Vencerás* se caracteriza por la división de un problema en subproblemas que tienen la misma forma que el problema completo [BRA98]. La división del problema en subproblemas más pequeños se lleva a cabo utilizando la recursión. El método recursivo continúa dividiendo el problema hasta que las partes divididas ya no puedan dividirse más, entonces se combinan los resultados parciales de cada subproblema para obtener al final la solución al problema inicial [BRA98].

En esta técnica la división del problema siempre se hace en dos partes, por tanto una formulación recursiva del método *Divide y Vencerás* forma un árbol binario, cuyos nodos serán procesadores, procesos o threads.

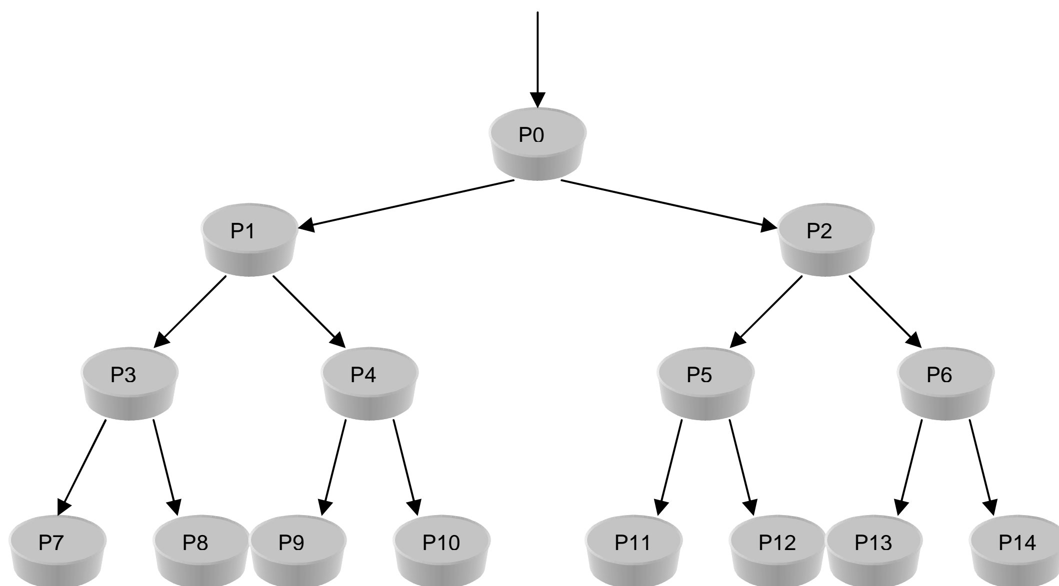


Fig. III.10. Árbol binario completo

El nodo raíz del árbol recibe como entrada un problema completo que se divide en dos partes. Una se envía al nodo hijo izquierdo, mientras que la otra se envía al nodo que representa al hijo derecho (fig. III.10). Este proceso de división es repetido

recursívamente hasta los niveles más bajos del árbol. Transcurrido un cierto tiempo, todos los nodos hoja reciben como entrada un problema dado por su nodo padre, lo resuelven y las soluciones (que son la salida del nodo hoja) son enviadas nuevamente a su progenitor. Cualquier nodo padre en el árbol obtendrá dos soluciones parciales de sus hijos y las combinará para proporcionar una única solución que será la salida del nodo padre. Finalmente el nodo raíz dará como salida la solución completa del problema, [BRI93].

En la fig. III.10 se muestra un árbol binario completo, esto es, un árbol perfectamente balanceado con los nodos hojas en el mismo nivel. Esto ocurre si el problema puede ser dividido en un número total de partes que sea potencia de dos. Pero si no es así, uno o más nodos hoja pueden estar en niveles distintos en el árbol.

Mientras que en una implementación secuencial un sólo nodo del árbol puede ser ejecutado o visitado a la vez, en una implementación paralela más de un nodo puede ser ejecutado al mismo tiempo por niveles, es decir, al dividir el problema en dos partes, ambas pueden ser procesadas de manera simultánea. Un ejemplo clásico del uso de esta técnica es el problema de la ordenación de un conjunto de datos. Existen varios algoritmos de ordenación que utilizan la técnica de *divide y vencerás* junto con la recursión, como lo es el algoritmo *QuickSort* u ordenación rápida o *MergeSort* por citar dos de ellos. Supongamos entonces que se tiene una lista de números enteros en desorden y queremos llevar a cabo la ordenación de dichos números por medio de la técnica de *divide y vencerás* en paralelo, utilizando el algoritmo de ordenación rápida o *Quicksort*.

Como primer paso el algoritmo selecciona como pivote uno de los elementos del conjunto de datos que haya que ordenar. A continuación el conjunto se parte a ambos lados del pivote. Se desplazan los elementos de tal manera que los que sean mayores que el pivote queden a su derecha, mientras que los que sean menores queden a su izquierda (ver fig. III.11(a). Posteriormente las partes del conjunto que quedan a ambos lados del pivote se ordenan independientemente, de forma paralela y recursiva. El resultado final es un conjunto completamente ordenado (ver fig. III.11(b))

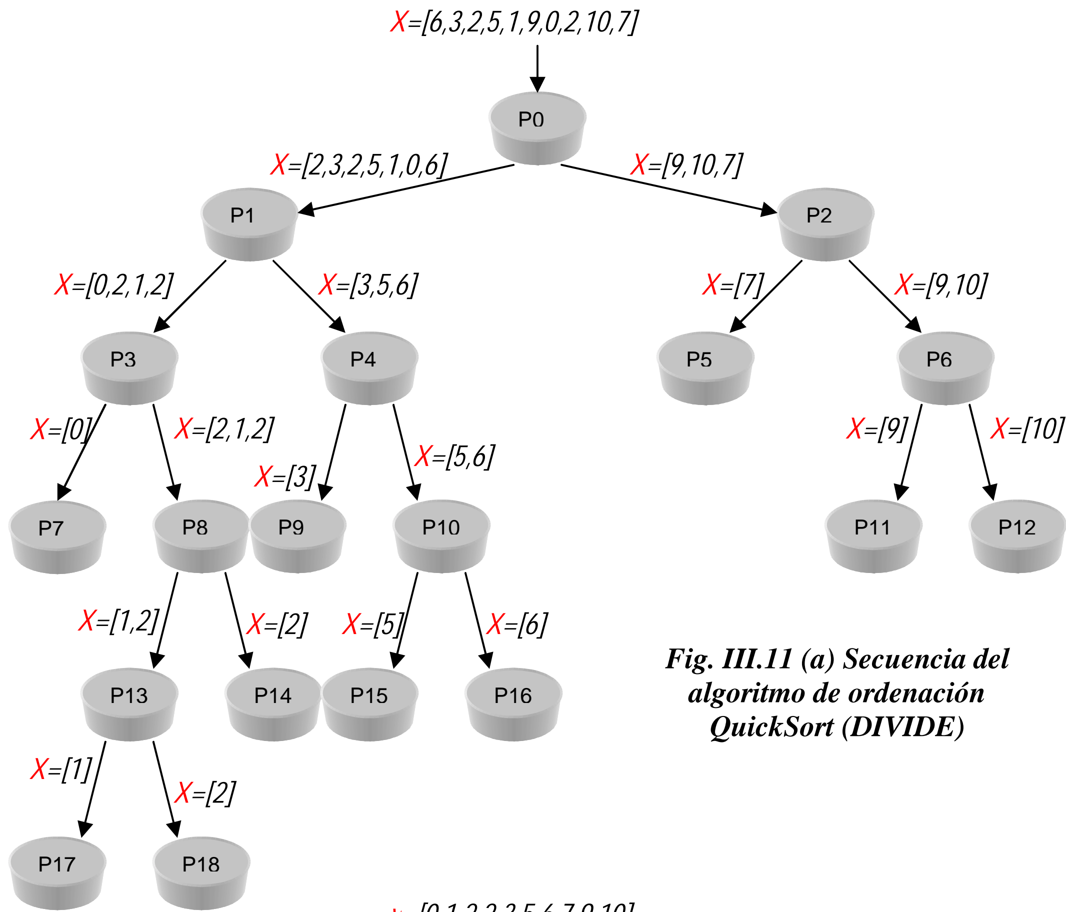


Fig. III.11 (a) Secuencia del algoritmo de ordenación QuickSort (DIVIDE)

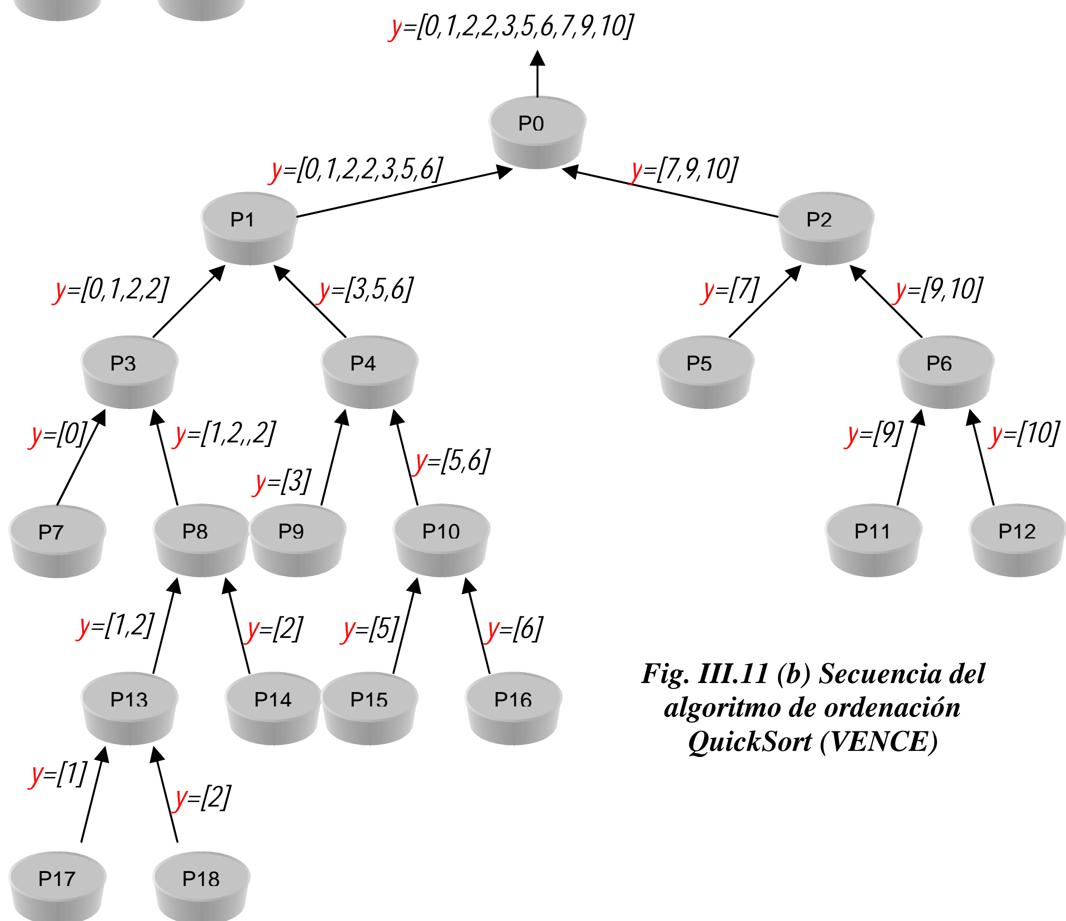


Fig. III.11 (b) Secuencia del algoritmo de ordenación QuickSort (VENCE)

Con la técnica de *divide y vencerás* muchos de los problemas computacionales que se llevan a cabo de manera secuencial pueden ser paralelizados como árboles binarios, donde cada nodo representa un proceso y cada proceso del árbol contiene una copia del mismo algoritmo de solución. Ejemplos de problemas que pueden ser solucionados con esta técnica son:

- Problemas de integración numérica
- Problemas de N-Body
- Problemas de ordenación de datos
- Problemas de búsqueda de datos

III.4.2. Representación de la técnica Divide y Vencerás como un CPAN

La representación del patrón *treeDV* que define la técnica de Divide y Vencerás como *CPAN* tiene su modelo representado en la fig. III.12.

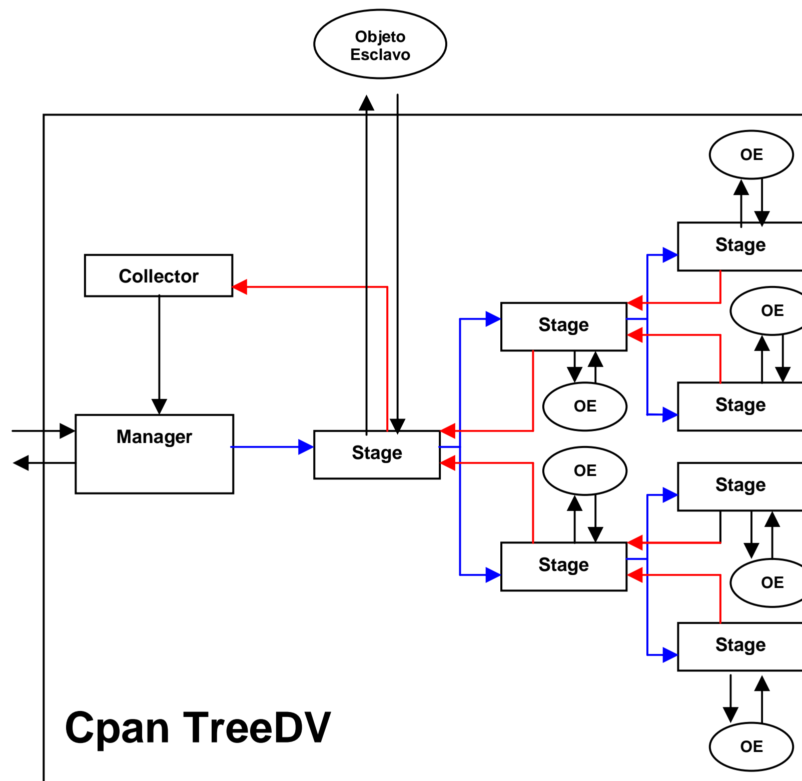


Fig. III.12. El CPAN de un TreeDV

A diferencia de los modelos anteriores, donde los objetos esclavos eran predeterminados fuera del modelo *CPAN*, en este modelo sólo un objeto esclavo es predefinido estáticamente y asociado al primer *stage* del árbol. Los siguientes objetos esclavos serán creados internamente por los propios *stages* de forma dinámica, pues los niveles del árbol dependen del problema a resolver y a priori no se sabe el número de nodos que pueda tener el árbol, ni su nivel de profundidad.

No obstante, al igual que en los modelos anteriores, los objetos *Manager y stage_i* de la fig. III.12, son instancias de clases concretas digamos *TreeDVManager y TreeDVStage*, que heredan de las clases base denominadas *ComponenManager* y *ComponentStage*, respectivamente. El objeto *Collector* es aquí también una instancia de la clase base *ComponentCollector*.

III.4.3. Definición Sintáctica y Semántica del CPAN TreeDV

Se crea la clase *TreeDVManager*, la cual hereda de *ComponentManager*, e implementa un patrón de comunicación de un árbol binario bajo la técnica de *Divide y Vencerás*. Los nodos del árbol binario están representados por los *stages* que son objetos de la clase *TreeDVStage* que hereda de *ComponentStage*. Cualquier instancia de la clase *TreeDVManager* se encarga sólo del primer *stage* o nodo raíz del árbol en su inicialización. Durante la ejecución de una petición de servicio, la raíz del árbol binario, es decir, el primer *stage* de la estructura, es comandada por el manager e internamente cada nodo padre creado en los niveles más bajos del árbol comandarán a sus respectivos nodos hijos en la solución del problema.

```
CLASS CONCRETE TreeDVManager EXTENDS OF ComponentManager
  {
   PUBLIC VOID init (ASOCIACION[ ] list)
     {
      stages[0] = TreeDVStage CREATE (list);
     }

   PRIVATE VOID  commandStages(ANYTYPE datain,
                               ComponentCollector res)
     {
      THREAD stages[0].request(datain,res);
      THREAD res.put(datain);
     }
   };
```

Cualquier objeto *TreeDVStage* se encargará de crear un nodo del árbol binario (izquierdo o derecho). Cuando el nodo raíz o *stage* inicial ejecuta la operación *request( )* en paralelo, se evalúa el problema con el método del objeto esclavo asociado, retornando la división del problema en dos partes. Posteriormente, se llamará al método *commandOtherStages( )*, quien tomará dichos subproblemas, creará dos nodos *stages* asociados a su nodo padre, asociándoles a éstos últimos sus respectivos objetos esclavos, que serán creados dinámicamente conforme se vayan creando los nodos del árbol binario, y les enviará a cada uno, una parte del problema a resolver. Recursivamente los nodos *stages* hijos recibirán el subproblema y ejecutarán su método *request( )* en paralelo, llevando a cabo el mismo proceso hasta que el subproblema ya no pueda dividirse más. Los últimos objetos *TreeDVStage* del árbol, es decir, las hojas, envían el resultado de su proceso de cálculo a su progenitor y éste combinará las soluciones para enviarlas, a su vez, a su progenitor y así sucesivamente, en el regreso de la recursión, hasta llegar al nodo raíz quien enviará el resultado final a un objeto *collector*, el cual a su vez, pasará el resultado al *manager* de la composición.

```
CLASS CONCRETE  TreeDVStage EXTENDS OF ComponentStage
 {
  VAR
   ASOCIACION list;

   PUBLIC VOID init(ASOCIACION[] list)
     {
      this.list=list;
      stage.init(list);
      am_i_last= FALSE;
     }

   PRIVATE VOID commandOtherStages (ANYTYPE datain, ComponentCollector res)
      VAR
       ANYTYPE data_izq, data_der;
      {
      IF(datain.inicio<datain.fin)
      {
       otherStages[0]= TreeDVStage CREATE(list);
       otherStages[1]=TreeStage CREATE(list);

       THREAD
otherStages[0].request(dv(datain,datain.inicio,datain.medio),res);
       THREAD otherStages[1].request(dv(datain,datain.medio+1,datain.tam-
1),res);
      }
     }

   PRIVATE ANYTYPE dv(ANYTYPE dataout, int inicio, int fin)
     {
      datain.inicio=inicio;
      datain.fin=fin;
      RETURN datain;
     }
 };
```

III.4.4. Uso del CPAN TreeDV

Uno de los problemas más comunes que se resuelven utilizando la técnica de divide y vencerás es el de la ordenación de un conjunto de datos puestos en desorden. Dependiendo de los datos que se quieran ordenar se pueden utilizar distintos algoritmos de ordenación, en particular, si lo que queremos ordenar son números enteros, el algoritmo de *QuickSort* resulta ser muy eficiente y rápido para resolver el problema.

El patrón paralelo óptimo para llevar a cabo el algoritmo es el que implementa la técnica de *Divide y Vencerás*. Como ya se tiene implementado este patrón en base al modelo del *CPAN* se procedería entonces a la creación del objeto esclavo inicial que será una instancia de la clase que implementa el algoritmo de *QuickSort* (ver Fig. III.13).

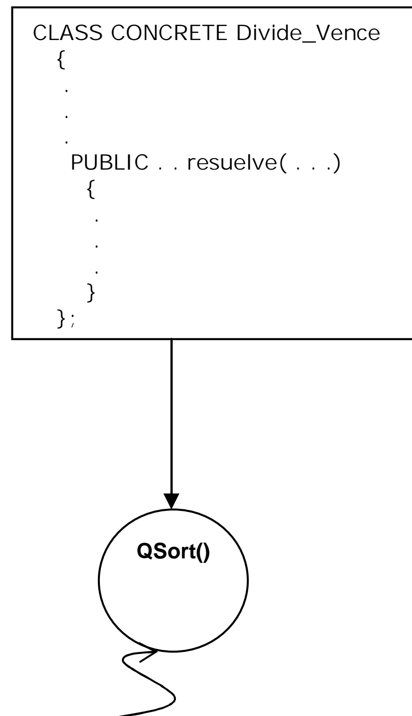


Fig. III.13. Objeto Esclavo y su método de ejecución para el CPAN TreeDV

De forma que:

1. Se crea el objeto esclavo que representa la clase de ordenamiento por *QuickSort* y se almacena en un array de referencias a *object*:

```
Object obj[]= {
               Qsort CREATE(. . .)
              };
```

2. Se crea el objeto que representa el método asociado al objeto esclavo y se almacena en un array de referencias a *method*.

```
method meth[]={
              method CREATE (. . . resuelve)
             };
```

3. Se crea una lista de asociaciones (objeto_esclavo, método_asociado)

```
asociacion pareja=crea_asociacion(obj,meth,1);
```

4. Se crean las instancias de la clase *TreeDVManager* que se inicializarán con la lista de asociaciones del punto anterior.

```
TreeDVManager CpanTreeDV[] = {
                              TreeDVManager CREATE(pareja);
                              TreeDVManager CREATE(pareja);
                             }
```

5. Se especifican los datos iniciales a procesar mediante la creación específica del tipo y de los datos definidos como un objeto por el usuario, en este ejemplo dicho objeto se denomina *MyTypeDV*. Cada objeto *MyTypeDV* representa un problema a resolver

```
int nums1[t1]={-11,-14,-6,. . ., -1, -15};
int nums2[t2]={5,1,9, . . . ,11,4,8};

ANYTYPE data[]= {
                MyTypeDV CREATE(nums1),
                MyTypeDV CREATE(nums2)
               };
```

6. Se imprimen los datos iniciales en pantalla

```
FOR i =(0,num_problems)
     {
      imprime_datos(data[i]);
     }
```

7. Las instancias *TreeDVManager* se encuentran listas para trabajar, es decir, se pide la ejecución de la operación *execution()* de cada instancia en paralelo para ordenar los arrays de datos que cada *CpanTreeDV* acepta en su operación *execution( )*.

```
FUTURETYPE resul[num_problems];
FOR i=(0,num_problems)
   {
    resul[i]= THREAD CpanTreeDV[i].execution(data[i]);
   }
```

8. Se imprimen los resultados finales en pantalla

```
ANYTYPE resultados[num_problems];
FOR i =(0,num_problems)
   {
    resultados[i]=resul[i];
    imprime_datos(resultados[i]);
   }
```

III.5. RESUMEN METODOLÓGICO DEL USO DE UN CPAN

Una vez implementados los *CPANS* de interés, la manera en que se utilizan en una aplicación de usuario es la siguiente:

1. Habrá que crear una instancia de la clase *manager* de interés, es decir, una que implemente el comportamiento paralelo requerido de acuerdo con los siguientes pasos:
   1.1. Inicializar la instancia con la referencia a los objetos esclavos que van a ser controlados por cada *stage* y el nombre del método requerido, como una asociación de pares *(obj_esclavo, método_asociado).*
   1.2. Se crean los *stages* internos (utilizando la operación *init( )*) y se les pasa a cada uno una asociación *(obj_esclavo, método_asociado)*, que utilizarán sobre su objeto esclavo .
2. El usuario pide al *manager* iniciar un cálculo a través de la ejecución dentro del *CPAN* del método *execution( )*. Dicha ejecución se lleva a cabo como sigue:
   2.1. Se crea el objeto *collector* referente a la petición.
   2.2. Se les pasa a los *stages* los datos de entrada (sin verificación de tipos) y la referencia al *collector*.
   2.3. Los resultados son obtenidos desde el objeto *collector*.
   2.4. El *collector* regresa los resultados al exterior nuevamente sin verificación de tipos.

3. Un objeto *manager* ha sido entonces creado e inicializado y algunas peticiones de ejecución pueden ser despachadas en paralelo.

Capítulo IV:

CASO DE ESTUDIO: Paralelización de la técnica de Ramificación y poda como un CPAN

IV.1. INTRODUCCIÓN

Las técnicas algorítmicas de optimización y búsqueda combinatoria se caracterizan por ofrecer a un determinado problema no sólo una solución única, sino un conjunto de soluciones potencialmente posibles. Para muchos problemas de búsqueda y optimización, una búsqueda exhaustiva no es factible. En lugar de ello se utiliza una búsqueda dirigida de forma que, más que encontrar la mejor solución al problema, esto es, la óptima, se busca a menudo una buena solución.

Problemas clásicos de la ciencia computacional han sido y siguen siendo abordados por técnicas de optimización y búsqueda. Estos incluyen el *problema del agente viajero*, el *problema de la mochila*, el *problema de las n-reinas* y el *problema de la asignación de tareas*, por citar algunos. En el *problema del agente viajero*, la meta es encontrar la ruta más corta que sigue un agente de viajes al iniciar en una determinada ciudad, visitando exactamente una vez cada una de las ciudades de un tour, antes de retornar a la ciudad de partida, esto es, la primera que visitó. En el *problema de la mochila*, existen objetos a los cuales se les asignan valores que representan ganancias y la meta es seleccionar algunos de estos objetos y colocarlos en la mochila de forma que los objetos seleccionados maximicen el valor de la ganancia. La meta en el *problema de las n-reinas* es colocar *n* reinas en un tablero de ajedrez de tamaño *nxn*, de tal forma que ninguna de las reinas se ataquen unas a otras. En el *problema de la asignación de tareas*, el objetivo es asignar *n-tareas* a *n-agentes*, de forma que cada agente realice exactamente una tarea. Para cada tarea un costo es asociado de forma que la meta es la minimización del costo total de las tareas que realizan los agentes. Todos estos problemas se caracterizan por tener un gran número de permutaciones y una búsqueda a través de todas ellas, daría como consecuencia un gran consumo de tiempo. En caso del *problema del Agente Viajero* no se conoce un algoritmo de tiempo polinomial que lo resuelva y el problema es clasificado como NP-Completo.

Existen muchas aplicaciones serias en el comercio, la banca o la industria que utilizan las técnicas de búsqueda y optimización para la solución de problemas, como son las siguientes :

- Ramificación y Poda (*Branch and Bound*)
- Programación Dinámica
- Algoritmos Genéticos

- Algoritmos Ávidos
- Vuelta a atrás

La técnica de *Ramificación y poda* es una de las técnicas de optimización y búsqueda fundamentales que esta siendo estudiada para su paralelización. Este capítulo se centra en la técnica de *Ramificación y Poda* como caso de estudio para su paralelización como un *CPAN* y demostrar así la utilidad del modelo de Objetos Paralelos propuesto, tomando como problema a resolver el del *Agente Viajero* denotado como *TSP*[17].

IV.2. LA TECNICA DE RAMIFICACIÓN Y PODA

Es una técnica de diseño algorítmica cuyo nombre en español proviene de ***Branch and Bound*** en el idioma Ingles y se aplica normalmente en la solución de problemas de optimización donde la complejidad computacional es grande. *Ramificación y Poda* es una variante de la técnica de *Vuelta Atrás*, siendo similar a ésta última en que se realiza una enumeración parcial del espacio de soluciones del problema basándose en la generación de un árbol de expansión, donde la raíz representa el punto inicial. Sin embargo la diferencia radica en que en *Ramificación y Poda* existe la posibilidad de generar nodos siguiendo distintas estrategias y en *Vuelta Atrás* no [GUE99]. A partir de la raíz, el siguiente nivel en el árbol de expansión representa las elecciones individuales que se pueden llevar a cabo para la solución del problema. El diseño de *Ramificación y Poda* puede seguir un recorrido de su árbol de expansión utilizando:

- El método *primero en profundidad* (estrategia LIFO) donde se comienza a recorrer el árbol de izquierda a derecha, de arriba hacia abajo a partir de un nodo, visitando los nodos que están más abajo en los niveles siguientes, continuando lo más lejos posible. Este método conduce rápidamente a encontrar soluciones subóptimas. Según esta estrategia, un nodo en un nivel más profundo será siempre examinado antes que un nodo en un nivel superior [CAP92].

[17] TSP son las siglas en ingles de Traveling Salesman Problem

- El método *primero en anchura* (estrategia FIFO), que realiza la expansión de cada nivel del árbol de izquierda a derecha, antes de ir al siguiente nivel más bajo. Esta heurística es la que puede producir los peores resultados, ya que se tarda mucho en encontrar soluciones, pues la acotación de nodos se hace al final del algoritmos [CAP92].
- El método de *primero el mejor* que utiliza el cálculo de funciones de costos para seleccionar el nodo que en principio parezca más prometedor a analizar (estrategia HEAP, usando el mínimo costo o LC). Esta es la estrategia que puede producir los mejores resultados. Conduce a encontrar soluciones óptimas rápidamente si la cota inferior que representa al LC es buena, ya que se exploran primero nodos que previsiblemente pueden conducir a buenas soluciones [CAP92].

Además de estas estrategias *Ramificación y Poda* utiliza cotas para hacer el podado de las ramas del árbol de expansión que conduzcan a la solución óptima. Para ello se calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése nodo. Si la cota muestra que cualquiera de estas soluciones tiene que ser necesariamente peor que la mejor solución hallada hasta ese momento, no se necesita seguir explorando por esa rama del árbol, lo que permite llevar a cabo el proceso de poda.

Dentro de esta técnica, para determinar en cada momento qué nodo será ramificado y dependiendo de la estrategia de búsqueda utilizada, se requerirá almacenar todos los nodos que no hayan sido podados, es decir, aquellos con posibilidad de ser ramificados (*nodos vivos)*, en alguna estructura de datos que podamos recorrer. Se utilizará una PILA de nodos generados que todavía no han sido examinados si la estrategia de búsqueda seleccionada es la de *primero en profundidad (LIFO)*. Pero si la estrategia seleccionada ha sido la de *primero en anchura (FIFO)*, entonces la estructura de datos a utilizar será una COLA, de tal forma que se van explorando nodos en el mismo orden en que son creados. Por el contrario, si la estrategia elegida es la de *primero el mejor* (LC), la estructura necesaria de uso será un *Montículo* (HEAP) para poder almacenar los nodos ordenados por su costo. La estrategia del mínimo costo utiliza una función de costos que decide en cada momento qué nodo debe explorarse, con la esperanza de alcanzar lo más rápidamente posible una solución más económica que la mejor encontrada hasta ese momento.

IV.2.1. El Algoritmo

En un algoritmo de ramificación y poda se llevan a cabo básicamente tres etapas:

- ***La etapa de Selección:*** Se encarga de extraer un nodo de entre el conjunto de los nodos que no han sido podados y que tienen la posibilidad de ser ramificados. La forma de elección depende directamente de la estrategia de búsqueda que se decida utilizar en el algoritmo.
- ***La etapa de Ramificación:*** Se construyen los posibles nodos hijos del nodo seleccionado en el paso anterior, formando el árbol de expansión.
- ***La etapa de Poda:*** Se eliminan algunos de los nodos creados en la etapa anterior, aquellos cuyo costo parcial sea mayor que la mejor cota mínima calculada en ese momento.

La contribución de éste algoritmo es la disminución en lo posible del espacio de búsqueda y por tanto la atenuación de la complejidad en la exploración del árbol de posibilidades de la solución óptima. Aquellos nodos no podados pasan a formar parte del conjunto de nodos con posibilidad de ser ramificados, y se comienza de nuevo el proceso de selección. El algoritmo finaliza una vez encontrada la solución del problema o bien cuando se agota el conjunto de nodos con posibilidad de ser ramificados.

Para cada nodo del árbol de expansión se dispondrá de una función de costo que estime el valor óptimo de la solución si se continuara por esa rama o camino. De esta forma, si la cota que se obtiene para un nodo, es peor que una solución ya obtenida por otra rama, se poda esa rama, pues no es interesante seguir por ella. Las funciones de costo son funciones crecientes respecto a la profundidad del árbol.

La utilidad de la técnica de *Ramificación y Poda* esta en la posibilidad de tener varias formas de exploración del árbol y al mismo tiempo el acotamiento de la búsqueda de solución, lo cual se traduce en *eficiencia* [GUE99]. Sin embargo la dificultad del algoritmo radica en encontrar una buena función de costo para el problema a resolver, donde "*buena*" es en el sentido de que garantice la poda y que su cálculo no sea muy costoso.

La estructura general de los algoritmos que implementan la técnica de *Ramificación y Poda* se basan en tres módulos principales:

1. El módulo que contiene el esquema de funcionamiento general de la técnica.
2. El módulo que maneja la estructura de datos en donde se almacenan los nodos que se van generando. La estructura de datos puede ser una *pila*, una *cola* o un *montículo*, según la estrategia adoptada[18]
3. El módulo que describe e implementa las estructuras de datos que conforman los nodos.

De los tres módulos citados, el primero es el único que permanece sin modificación alguna independientemente del problema que se resuelva con *Ramificación y Poda*, en otras palabras, es válido para todos los algoritmos que siguen esta técnica. El pseudo código[19] que lo implementa se presenta a continuación:

```
CLASS CONCRETE EsquemaBB
 {
  Estructura e;
  Nodo n;
  Nodo[] hijos;
  int numhijos,i,j;

  PUBLIC Nodo B&B()
   {
    e= Estructura CREATE();
    n= nodoInicial();
    e.inserta(n,n.h());
    WHILE (!E.esVacia())
      {
       n=e.extrae();
       numhijos=n.expandir(VAR hijos);
       eliminar(n);
       n.poner_cota_sup(numhijos,hijos);
       FOR i=(0,numhijos)
        {
         IF (n.aceptable(hijos[i]))
          {
           IF (n.esSolucion(hijos[i]))
             {
              FOR j=(0,numhijos)
               {
                IF (i!=j)
                  {
                   DELETE hijos[j];
                  }
               }
               e.clear();
               RETURN hijos[i];
             }
            ELSE
             {
              e.inserta(hijos[i], n.h(hijos[i]));
             }
          }
         ELSE
```

[18] estrategia primero en profundidad, estrategia primero en anchura o estrategia primero el mejor.
[19] basado en la gramática libre de contexto presentada en el capítulo II.

```
            {
             DELETE hijos[i];
            }
          }
        }
    }
 }
```

Este módulo encuentra una solución a un problema que pueda ser resuelto con la técnica de *Ramificación y Poda*. La definición del tipo abstracto de datos denominado *Estructura* dentro del pseudo código representa la *pila*, *cola* o *montículo*, según la estrategia adoptada donde se irán almacenando los nodos que se van generando. La clase que representa esta estructura de datos genérica es la siguiente:

```
CLASS ABSTRACT Estructura
  {
    PUBLIC Estructura crear( );
    PUBLIC VOID anadir(Nodo n, INT prioridad);
    PUBLIC Nodo extraer( );
    PUBLIC BOOL esVacia( );
    PUBLIC INT tamano( );
    PUBLIC VOID eliminar( );
    PUBLIC VOID limpiar( );
  }
```

En el caso que nos ocupa se utilizó un *Montículo o Heap*, pues la estrategia de solución adoptada fue la del mínimo costo (LC). No se presenta su implementación concreta pues además de que no aporta nada nuevo a la técnica de *Ramificación y Poda*, la mayoría de los lenguajes de programación orientados a objetos ya proporcionan esta clase de estructuras para su uso.

Por otro lado, la definición de la clase que representa el tipo *Nodo*, usado también en el pseudo código anterior se forma de los siguientes métodos:

```
CLASS CONCRETE Nodo
   {
    PUBLIC INT expandir(Nodo **n);
    PUBLIC INT h();
    PUBLIC INT h(Nodo *n);
    PUBLIC VOID imprimir();
   }
```

- *expandir( ):* Es el método que construye los nodos hijos de un nodo dado, retornando el número de hijos generados. Este es el método que realiza el proceso de ramificación del algoritmo.
- *h( ):* Es un método sobrecargado que implementa la función de costo de la estrategia LC y su valor es utilizado para priorizar el almacenamiento del nodo que usa este método, dentro de la estructura *HEAP* adoptada.
- *imprimir( ):* Es el método que se encarga de imprimir a pantalla el estado de un nodo en un determinado estado de tiempo, esto es, sus valores: su costo asociado, su matriz de adyacencia, el nivel del nodo en el árbol de expansión al que pertenece y su vector de soluciones que contiene la secuencia de recorrido del árbol de expansión desde la raíz hasta ese nodo.

La implementación de estos métodos en esta etapa genérica del algoritmo es abstracta. Es decir, habrá que adecuarlos al problema que se quiera resolver. Sin embargo, la ventaja añadida a estos algoritmos de ramificación y poda es la de posibilitar su implementación y ejecución de forma paralela. Puesto que se tiene un conjunto de nodos vivos, es decir, con posibilidad de ser ramificados, sobre los que se efectúan las tres etapas antes mencionadas, nada impide tener más de un proceso trabajando sobre este conjunto de nodos, extrayéndolos, expandiéndolos y llevando a cabo la poda.

Los *CPANS* proporcionan los algoritmos paralelizables necesarios para resolver problemas como el del *Agente Viajero* con la técnica de *Ramificación y Poda*. Sobre todo porque el abordar este tipo de problemas de forma paralela, se hace necesario si lo que se quiere además de obtener una buena solución es la de resolver el problema en tiempos razonables ya que la complejidad en ellos es intrínseca.

Esto último requiere de ciertos requerimientos que respecto a otras técnicas algorítmicas resultan caros, por ejemplo, aquellos que tienen que ver con la memoria. Se necesita por tanto, que cada objeto nodo sea autónomo, es decir, que contenga toda la información necesaria para que siendo objeto activo[20], realice los procesos de ramificación y poda para la reconstrucción de la solución encontrada hasta ese momento.

---

[20] Un objeto activo es aquel que tiene capacidad de ejecución en si mismo.

IV.3. EL PROBLEMA DEL AGENTE VIAJERO (TSP)

El Problema del Agente Viajero (TSP) suele ser representado mediante un grafo dirigido que consiste de un conjunto de vértices (ciudades) y arcos etiquetados (distancias entre las ciudades). Una solución optimizada del TSP es la de generar un camino o tour en el cual todos los vértices han sido visitados exactamente una vez con costo mínimo [CAP92]. En otras palabras, en el problema del TSP se conocen las distancias entre un cierto número de ciudades. Un agente de viajes debe, a partir de una de ellas, visitar cada ciudad exactamente una vez y regresar al punto de partida habiendo recorrido en total la menor distancia posible [BEL01].

IV.3.1. Solución del TSP usando la técnica de Ramificación y Poda

El problema es resuelto usando la técnica de Ramificación y Poda, la cual construye dinámicamente un árbol de búsqueda, cuya raíz representa el problema inicial y los nodos que se van generando en los niveles siguientes del árbol representan los caminos que deben seguirse para su solución. Existen numerosas estrategias de la técnica de *Ramificación y Poda* que resuelven el TSP. En el presente documento se adopta la primera de las tres estrategias propuestas en [HOR78], donde los siguientes elementos son definidos:

- ***LC(P) – Costo de un nodo:*** Es la distancia del camino completo en el grafo representado por ***P***, siempre y cuando ***P*** sea un nodo solución, de otra forma representará la distancia parcial de una solución de costo mínimo en el subárbol cuya raíz es ***P***.
- ***cota – Cota Superior:*** Es la longitud del camino más corto encontrado hasta este momento. Este valor deberá ser una variable global en el programa, que será utilizada para realizar la poda del árbol de búsqueda.
- ***h(P) – Cota Inferior:*** Es la cota inferior del costo de una solución para un problema o subproblema de ***P***. Si ***P*** es un nodo solución, entonces *LC(P)=h(P).*
- ***s[ ] – Vector Solución:*** Es un vector que indica el orden en el cual los vértices del grafo deben ser visitados para encontrar la solución óptima. Cada elemento del vector debe contener un número entre *1 y N*, siendo *N* el número de vértices del

grafo que define el problema. Obviamente, el vector no debe contener elementos repetidos.

- ***M[ ][ ] – Matriz de Adyacencia:*** Es la representación del grafo conexo ponderado, donde los vértices son representados por los índices *i, j* de la matriz, con $1 \leq i, j \leq N$ y las ponderaciones de los arcos del grafo serán los costos almacenados en dicha matriz. La matriz de adyacencia no necesariamente es simétrica aunque si de elementos no negativos.

El cálculo de la *cota inferior* de un nodo del árbol de búsqueda se lleva a cabo mediante la obtención de la *matriz de costos reducida* para cada nodo [CAP92]. Se dice que una fila (columna) de una matriz está reducida si contiene al menos un elemento cero, y el resto de los elementos son no negativos. Una matriz se dice reducida si y sólo si todas sus filas y columnas están reducidas [GUE99].

Con respecto a la interpretación del costo de un nodo, una de las formas para calcularlo es sumar las cantidades t_i que son restadas a las líneas (columnas) de la matriz de adyacencia que están siendo reducidas para obtener la matriz reducida. La suma final de dichas cantidades será una cota inferior del costo total de los caminos del grafo que se van generando. Este valor es el que se utiliza como la función de costos LC para la poda de nodos del árbol de expansión. Por tanto, a cada nodo se le asocia una *matriz reducida* y un *costo acumulado* de forma que si suponemos que *M[][]* es la matriz reducida asociada al nodo ***P*** y ***P'*** es el hijo de ***P*** que se obtiene incluyendo el arco *{i, j}* en el recorrido *s[]*, entonces:

1. Si ***P'*** es una hoja del árbol, esto es, una posible solución, su costo será el costo que llevaba ***P*** acumulado más *M[i][j]+M[j][1]*, que es lo que completa el recorrido. Esta cantidad coincide además, como ya se ha dicho, con el costo de tal recorrido.
2. Si ***P'*** no es una hoja, su matriz de costos reducida *M'[][]* se calcula a partir de los valores de *M[][]* como sigue:
   2.1. Primero, habrá que sustituir todos los elementos de la fila *i*, columna *j* por ∞. El objetivo es eliminar el uso posterior de aquellos caminos que parten del vértice *i* y de los que llegan al vértice *j*.
   2.2. Segundo, como ***P'*** no es una hoja, habrá que asignar M'[j][1]=∞ para eliminar la posibilidad de acabar el recorrido en el siguiente paso.

2.3. Tercero, reducir entonces la matriz *M'[][]* que será asignada al nodo ***P'***. El costo para ***P'*** será el costo de ***P*** más el costo de la reducción de *M'[][]* más el valor de *M[i][j]*.

IV.4. PARALELIZACIÓN DE LA TECNICA DE RAMIFICACIÓN Y PODA COMO UN CPAN

Los algoritmos de *ramificación y poda* tienen características que los hacen ser favorables a la paralelización. Por ejemplo, los procesadores por separado pueden realizar la búsqueda de soluciones en diferentes partes del árbol de expansión de forma independiente. El enfoque más natural es el de asignar una porción del árbol de expansión a partir de un nodo hacia abajo a un procesador. Este procesador puede entonces realizar una búsqueda en este subárbol del árbol de expansión.

Desafortunadamente existen problemas que hacen que su implementación paralela sea más compleja y menos eficiente que la que uno podría imaginar. *Ramificación y poda* requiere que la cota corriente calculada, (superior o inferior según sea el caso) sea conocida por todos los procesadores de forma que cada uno de ellos pueda trabajar de forma óptima en cualquier instante, con su porción del árbol de expansión asignada. Además, el tamaño del árbol de expansión en cualquier partición es no conocido en cada avance.

Existen varios esquemas para la paralelización de los algoritmos de Ramificación y Poda dependiendo de las características de las diferentes arquitecturas paralelas que se tengan [CAP92]. Pero en resumen los esquemas se clasifican en tres grandes grupos:

1. Esquemas Basados en Memoria Compartida
2. Esquemas Basados en Sistemas Distribuidos
3. Esquemas Basados en Memoria Compartida Distribuida

Estos esquemas pueden seleccionar diferentes tareas en paralelo por los procesos que permiten alcanzar una buena eficiencia dentro de las restricciones que impone cada implementación [CAP92].

Uno de los mayores problemas de la técnica de Ramificación y Poda es la implementación de los algoritmos que la diseñan. Sin embargo, el uso del paradigma de

la orientación a objetos hace menos difícil esta tarea gracias al uso de sus propiedades como son la herencia, la modularidad y la reutilización de código. En ese sentido, para demostrar la utilidad que tiene el reuso de los *CPANS* que hasta este momento se han propuesto[21], se ha adoptado el diseño y uso del *Cpan Farm* para diseñar la técnica de *Ramificación y Poda* como una ***C**omposición **P**aralela de **A**lto **N**ivel.* La implementación paralela que aquí se muestra se basa entonces en aquel tipo de programas que genera estructuras de datos globales compartidas por todos los procesadores.

La idea es que la *ramificación* (es decir, la división y generación de nuevos sub-problemas) esté separada de la *poda* (esto es, la comparación de los valores-cotas de la función LC que se pretende optimizar para un sub-problema dado con la mejor cota superior obtenida hasta ese momento). Estas dos estructuras son implementadas usando el *Cpan Farm* (cuya implementación y análisis de rendimiento se muestran en los capítulos III y V, respectivamente), de tal forma que la ramificación y la distribución del trabajo a los procesos se lleva a cabo usando el esquema del patrón paralelo de comunicación *farm*, formando con ello el correspondiente árbol de expansión.

La poda se lleva a cabo de forma implícita dentro de los *farms* que constituyen el modelo usando un esquema *totalmente conectado* entre todos los procesos para garantizar la comunicación inmediata de una cota sub-óptima encontrada en un proceso a todos los procesos *ramificadores*, evitando así ramificaciones de sub-problemas inútiles, es decir, evitar que se sigan ramificando sub-problemas que no pueden conducir a mejorar la mejor cota superior obtenida hasta ese momento en otro proceso.

IV.4.1. Representación de la técnica de Ramificación y Poda como un CPAN

El *CpanBB* (Cpan Branch&Bound) se compone de un conjunto de *Cpans Farm* que representan procesos trabajadores (stages) y un controlador (*manager),* formando un nuevo tipo de *patrón paralelo*: el *FarmBB* (Farm Branch&Bound) que ahora se incluye como un *CPAN* más a la librería propuesta en el apéndice A.

[21] El *Cpan Farm,* el *Cpan Pipe* y el *Cpan TreeDV*

Los procesos trabajadores o *stages* del *Cpan FarmBB* son ejecutados en paralelo formando el árbol de expansión de los nodos de la técnica de *Ramificación y Poda*. El proceso controlador inicial del *Cpan FarmBB* representa la raíz del árbol de expansión que es el encargado de distribuir el trabajo y de controlar el progreso del cálculo global que al final será enviado al objeto collector del *FarmBB* y éste a su vez enviará la solución al proceso controlador del *Cpan FarmBB* quien finalmente la mostrará al usuario que hizo la petición inicial.

La fig. IV.1. representa el patrón de comunicación *FarmBB* como un *CPAN*. En este modelo, los objetos *ManagerBB* y *CollectorBB* son instancias de las clases concretas *FarmBBManager* (que hereda de la clase *FarmManager* definida en el capítulo III) y *ComponentCollector* (que es una de las clases base que conforman el modelo CPAN definido en el capítulo II), respectivamente.

Los objetos *stageBB_i* representan instancias concretas de la clase *FarmBBStage* (que hereda de la clase *FarmStage* definida en el capítulo III). Cada objeto *stageBB_i* es en sí un *Cpan FarmBB* formado por objetos *Manager, Collector y stages* que conforman el *farm* de procesos que irán construyendo el árbol de expansión de la técnica de Ramificación y poda, como lo muestra la fig. IV.1. Es hasta este nivel en el que a cada objeto *stage_i* del modelo, se le asocia un objeto esclavo de forma dinámica, que es el que contendrá el algoritmo secuencial de solución del problema que se pretende resolver con el *CpanBB*.

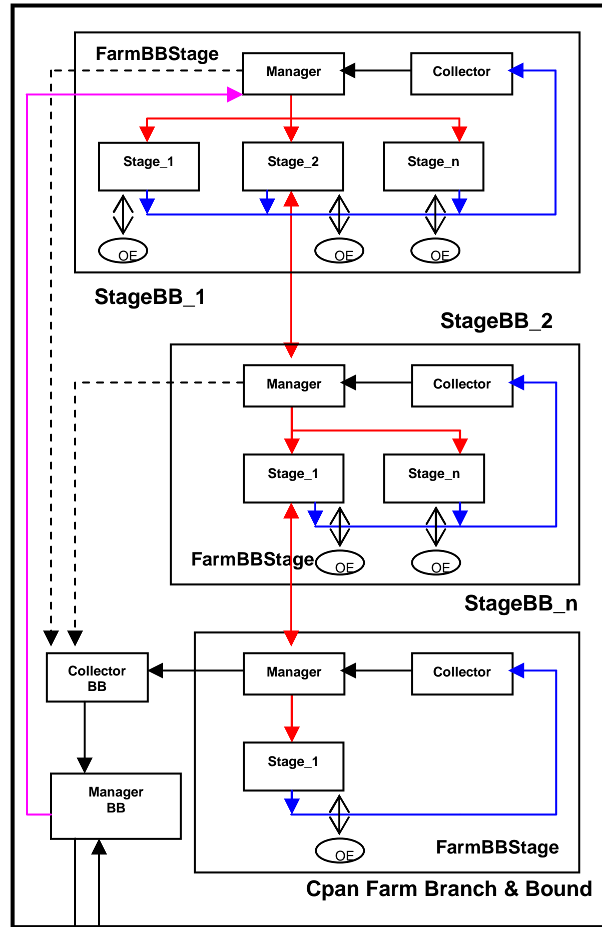


Fig. IV.1. El Cpan FarmBB de la técnica de Ramificación y Poda

IV.4.2. Definición Sintáctica y Semántica del Cpan FarmBB

Se crea la clase *FarmBBManager* que hereda de la clase *FarmManager* que inicialmente implementa el patrón *Farm* como *Cpan*. En esta nueva implementación se sigue preservando la política de composición del patrón, es decir, el *manager* recibirá resultados de sus *stages* a través de su *collector*[22], sin embargo a diferencia de la implementación del *Cpan farm* descrita en el capítulo III, donde el *manager* esperaba sólo al primer resultado dado por cualquiera de sus objetos *stage*, en el *Cpan FarmBB* el *manager* tendrá que esperar al último resultado dado por sus *stages*, ya que cada resultado no final, representa una solución parcial del árbol de expansión que se va creando por los propios *stages*, mediante un *esquema de conexión total* implícito. Al final, éste último resultado almacenado en el *collector* del *Cpan FarmBB* contendrá la

[22] Objetos *ManagerBB*, *StageBB_i* y *CollectorBB* en la Fig. IV.1

solución completa al problema inicial, solución que el *manager* mandará al usuario como resultado final del *CPAN*.

Una característica más del *Cpan FarmBB* es que cualquier instancia de la clase *FarmBBManager* se encarga sólo del primer *stageBB* el cual contendrá la raíz inicial del árbol de expansión de la técnica de *Ramificación y Poda*.

```
CLASS CONCRETE FarmBBManager EXTENDS OF public FarmManager
   {
     PRIVATE VOID  commandStages(ANYTYPE datain, ComponentCollector res)
      VAR
        INT i;
     {
      FOR i =(0,nWorker)
        {
         THREAD stages[i].request(datain, res);
        }
      FOR i =(0,nWorker)
        {
         JOIN stages[i];
        }
     }
   };
```

Recordar que la clase *FarmManager* hereda de la clase *ComponentManager* del modelo *CPAN*, por lo que la clase *FarmBBManager* incorpora la operación *init()* que crea todos los *stages* necesarios, la operación *execution( )* que es lanzada en paralelo de forma asíncrona distribuyendo datos a todos los *stages* y las restricciones de sincronización para la planificación de procesos.

Los objetos *stageBB_i* que constituyen los *stages* del *Cpan FarmBB*, son instancias de la clase *FarmBBStage* la cual hereda de la clase *FarmStage* que forma parte del modelo del *Cpan Farm* descrito en el capítulo III. En este caso, los objetos *stageBB* si se conectan unos con otros mediante un esquema de *conexión total* para formar el árbol de expansión a medida que se va resolviendo el problema, ramificando y podando ramas. Cada objeto *stageBB* será un *FarmBBStage* constituido por un *collector*, un *manager*, que será la raíz del subárbol de expansión que se irá generando y *k* objetos *stage* en el mismo nivel que serán los nodos hijos del nodo raíz. A cada *stage* se le asocia entonces un objeto esclavo que contiene el algoritmo secuencial de solución del problema a resolver y se lanzan en paralelo de forma asíncrona. Aquellos *stages* que representen nodos vivos, se ramificarán, esto es, serán capaces de crear el siguiente nivel del árbol de expansión a través de la generación de un nuevo objeto *StageBB*,

repitiéndose el proceso hasta llegar a la solución final. Aquellos nodos que, de acuerdo al problema no puedan ser ramificados, se podarán.

```
CLASS CONCRETE FarmBBStage EXTENDS OF FarmStage
   {
     VAR
       ASOCIACION[] lista;
       Nodo[] nh;
       STATIC INT cota=INFINITO;

     PUBLIC VOID init (ASOCIACION[ ] list)
       {
        FarmStage.init(list);
        am_i_last = false;
       }

     PRIVATE VOID commandOtherStages(ANYTYPE dataout, ComponentCollector res)
       {
        INT i,j,cont=0;
        ANYTYPE dataoutin;
        nh=dataout;
        estructura.eliminar(estructura.extraer());
        poner_cota_sup(nh.size);
        FOR i=(0,nh.size)
           {
            IF (aceptable(nh[i]))
              {
               IF (esSolucion(nh[i]))
                 {
                  FOR j=(0,nh.size)
                      IF (i!=j) DELETE nh[j];
                  estructura.limpiar();
                  otherstages[i]= FarmBBStage CREATE(nh[i],res);
                  otherstages[i].am_i_last=true;
                  THREAD otherstages[i].request(nh[i],res);
                 }
               ELSE {
                     estructura.anadir(nh[i], nh[i].coste);
                     otherstages[i]= FarmBBStage CREATE (dataout,res);
                     THREAD otherstages[i].request();
                    }
              }
            ELSE DELETE nh[i];
           }
       }
```

```
PRIVATE BOOL aceptable(Nodo n)
  {
  RETURN (n.coste<=cota);
 }

PRIVATE BOOL esSolucion(Nodo n)
 {
  RETURN (n.k==N-1);
 }

PRIVATE VOID poner_cota_sup(INT numhijos)
 {
  INT minimo=INFINITO,i;
  FOR i=(0,numhijos)
     {
      IF (minimo>nh[i].coste)
          minimo=nh[i].coste;
     }
  cota=minimo;
 }
} ;
```

Recordar que la clase *FarmStage* hereda de la clase *ComponentStage* del modelo *CPAN*, por lo que la clase *FarmBBStage* incorpora la operación *init()* que inicializará a un objeto *stage* con la lista de asociaciones *objeto_esclavo-método* y lo conectará o no con otro *stage* de la lista de asociaciones. Incorpora también la operación *request( )* de forma que cuando el manager comanda a los *stages* en paralelo, cada uno de ellos hace que el objeto esclavo ejecute su método asociado, capturando los resultados que serán mandados al siguiente *stage* si el algoritmo decide ramificar la rama del árbol en la que esta trabajando o al *collector*, si se opta por la poda. Como cada *stage* comanda a otros en la ejecución del cómputo relativo a la *ramificación*, se hace necesario ofrecer una implementación del método *commandOtherStages( )* heredado. Adicionalmente, para completar el algoritmo de la técnica de *ramificación y poda* se han añadido los métodos *aceptable( )*, *esSolución( )* y *ponerCotaSup( )* cuyo propósito para cada uno de ellos es el siguiente:

- *aceptable( ):* Método que realiza la poda. Dado un nodo vivo decide si es factible seguir analizándolo o bien rechazarlo.

- *esSolución( ):* Es el método que decide cuando un nodo es una hoja del árbol, es decir, una posible solución al problema original. Recordar que tal solución no necesariamente es la optima, pero si una "buena" solución.
- *poner_cota_sup( ):* Método que establece la cota superior del problema. La función que lleva a cabo el proceso de poda utiliza este método para podar aquellos nodos cuyo valor sea superior a la cota que ya se obtuvo de una solución.

IV.2.3. Uso del CPAN FarmBB en la solución del TSP

Supongamos que el TSP a resolver es el definido en el siguiente grafo conexo (Fig. IV.2.) que representa una red de 5 ciudades que el agente viajero tiene que visitar, cada una representada por un nodo del grafo etiquetado con un número entero. Los arcos dirigidos entre las ciudades representan los caminos que existen para ir de una ciudad a otra. Cada camino esta etiquetado con un costo no negativo.

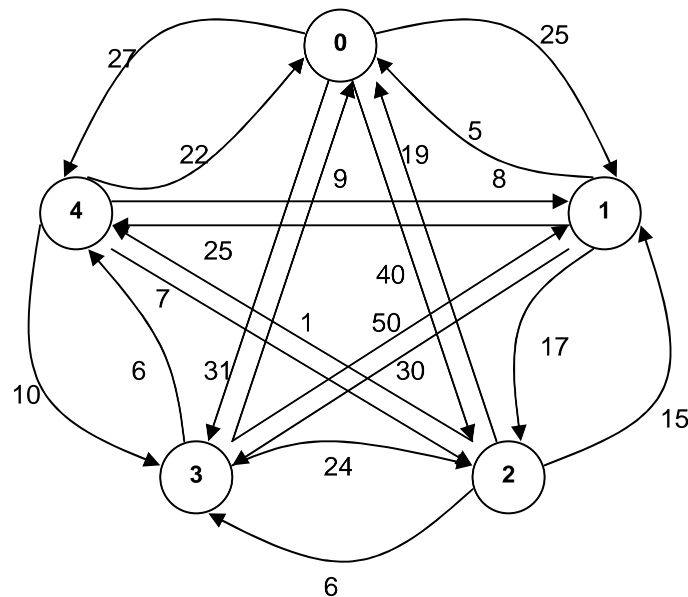


Fig. IV.2. Grafo conexo que representa una red de 5 ciudades

El objetivo es entonces que, a partir de una ciudad, digamos, la ciudad etiquetada con el número cero, el agente viajero visite cada una de las ciudades del grafo exactamente una vez, y regrese a la ciudad de donde partió, habiendo recorrido en total la menor distancia posible.

Si utilizamos el modelo del *CPAN* para su solución, el *Cpan FarmBB* será el encargado de solucionar el problema mediante la técnica de *Ramificación y Poda*. Se

procede entonces a crear el objeto esclavo inicial como una instancia de la clase que define el problema secuencial, en este caso el del TSP. En dicha clase se ha de definir un método de ejecución del algoritmo del TSP en cuestión.

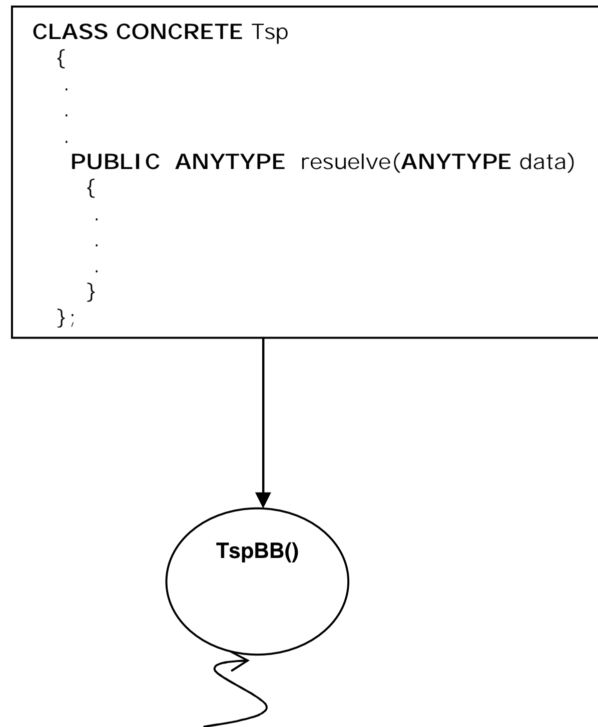


Fig. IV.3. Objeto Esclavo y su método de ejecución del TSP para el Cpan FarmBB

De forma que:

1. Se crea el objeto esclavo que representa el algoritmo del TSP y se almacena en un array de referencias a *object*:

```
Object obj[]= {
               Tsp CREATE(. . .)
              };
```

2. Se crea el objeto que representa el método asociado al objeto esclavo y se almacena en un array de referencias a *method*.

```
method meth[]={
               method CREATE (. . . resuelve),
              };
```

3. Se crea una lista de asociaciones (objeto_esclavo, método_asociado)

```
asociacion pareja=crea_asociacion(obj,meth,1);
```

4. Se crean las instancias de la clase *FarmBBManager* que se inicializarán con la lista de asociaciones del punto anterior. Cada instancia representa un *Cpan FarmBB* que solucionará un problema diferente.

```
FarmBBManager CpanFarmBB[] = {
                              FarmBBManager CREATE(pareja);
                              . . .
                              FarmBBManager CREATE(pareja);
                             }
```

5. Se especifican los datos iniciales a procesar mediante la creación específica del tipo y de los datos definidos como un objeto por el usuario. En este ejemplo el tipo definido se representa mediante la *clase Nodo*[23] y cualquier instancia de ésta clase representa en este nivel el nodo inicial que habrá de contener el nodo raíz del árbol de expansión de la técnica de *Ramificación y Poda* para un problema TSP a resolver.

```
int mat_ady[][] = { ∞ ,  25,  40,  31,  27
                     5,   ∞,  17,  30,  25
                    19,  15,   ∞,   6,   1
                     9,  50,  24,   ∞,   6
                    22,   8,   7,  10,   ∞ }

ANYTYPE data[]= { Nodo CREATE(mat_ady) };
```

6. Se imprimen los datos iniciales en pantalla

```
FOR i =(0,num_problems)
     {
      imprime_datos(data[i]);
     }
```

7. Las instancias *FarmBBManager* se encuentran listas para trabajar, es decir, se pide la ejecución de la operación *execution( )* de cada instancia en paralelo para resolver el problema del TSP que cada *Cpan FarmBB* acepta en su operación *execution( )*.

```
FUTURETYPE resul[num_problems];
```

[23] Ver sección IV.2.1.

```
FOR i=(0,num_problems)
   {
    resul[i]= THREAD CpanFarmBB[i].execution(data[i]);
   }
```

8. Se imprimen los resultados finales en pantalla

```
ANYTYPE resultados[num_problems];
FOR i =(0,num_problems)
   {
    resultados[i]=resul[i];
    imprime_datos(resultados[i]);
   }
```

La solución (ver Fig. IV.4.) que obtuvo el *Cpan FarmBB* para el problema enunciado fue el camino:

Nodo 0 → Nodo 3 → Nodo 4 → Nodo 2 → Nodo 1 → Nodo 0 con un costo mínimo de 64.

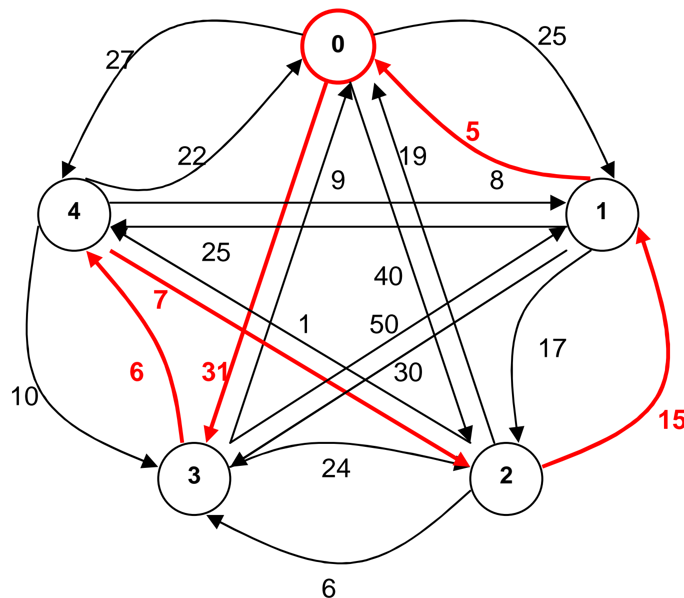


Fig. IV.4. Solución del Grafo conexo de la Fig. IV.2 que representa el camino de costo mínimo a seguir por el agente viajero

El problema mostrado aquí como ejemplo de ejecución de la aplicación, fue tomado de la referencia [GOO77], en donde se llegó a la misma solución.

Cabe señalar que, además de encontrar un camino de costo mínimo, el *Cpan FarmBB* calcula 3 datos importantes:

- *El número de nodos generados:* Este dato proporciona información sobre el trabajo que ha tenido que realizar el *CPAN* hasta encontrar la solución. Mientras más pequeño sea este valor, menos parte del árbol de expansión habrá tenido que construir, y por tanto más rápida será la ejecución.
- *El número de nodos analizados:* En este dato se indica el número de nodos que el *CPAN* ha tenido que analizar para lo cual es necesario extraerlos de la estructura (*montículo)* utilizada para su almacenamiento y comprobar si han de ser podados o ramificados. Este es el valor que indica el número de nodos del árbol de expansión que se recorren de forma efectiva. Por tanto, sería deseable que dicho valor sea pequeño.
- *El número de nodos podados:* Es un valor que indica la efectividad de la poda y de las restricciones que se imponen al problema. Mientras más grande sea este valor, menos trabajo tendrá que realizar el *CPAN*.

Estos tres datos pueden ser de interés si lo que se quiere es por ejemplo, llevar a cabo un análisis comparativo de la solución del mismo problema usando distintas estrategias de la técnica de *Ramificación y Poda* (LC, LIFO y FIFO). De igual manera, aunque la implementación del *Cpan FarmBB* que aquí se propone encuentra la mejor solución de entre todas las soluciones de un problema, también puede resultar interesante tener la versión que encuentra sólo una solución, así como aquella que encuentra todas las soluciones posibles.

Capítulo V:

ANÁLISIS DEL RENDIMIENTO DE LOS CPANS: Evaluación de los CPANS Farm, Pipe,TreeDV y FarmB&B

V.1. INTRODUCCIÓN

El análisis del rendimiento de los *CPANS* que aquí se presenta se llevó a cabo en un sistema paralelo *Origin 2000 Silicon Graphics* cuyo nombre es *karnak.cepba.upc.es*, disponible físicamente en el *Centro Europeo de Paralelismo de Barcelona CEPBA* y cuenta con:

- 64 procesadores R1000 a 250 Mhz
- 8 Gigabytes de memoria principal
- Bus de alta velocidad (1.2 Gb/s) que conecta los procesadores a la memoria principal y los dispositivos de I/O
- Sistema Operativo IRIX 6.5

Esta máquina paralela cuenta además con un sistema de colas *batch NQE* que optimiza su uso, y cuyos comandos muestran información relevante tales como el listado de las colas y los límites de cada una de ellas en todo momento.

V.1.1. Configuración de las colas batch NQE

Karnak dispone de 7 colas batch de propósito general que siempre se encuentran activas para su uso:

COLA	UTILIZACION	TIEMPO	MEMORIA	DISCO	NUM. PROCS.
Short	SEQ/PAR	20 min	1 Gb	1 Gb	8
Médium	SEQUENCIAL	8 hrs	512 Mb	1 Gb	1
Xlarge	SEQUENCIAL	10 días	256 Mb	500 Mb	1
Ldisk	SEQUENCIAL	12 hrs	3 Gb	2 Gb	1
Xdisk	SEQUENCIAL	10 dias	1 Gb	2 Gb	1
Parallel	PARALELO	10 dias	1Gb	2 Gb	8
Research	PARALELO	6 hrs	1 Gb	500 Mb	8
Gui	GRAFICO	10 min	1Gb	500 Mb	?

La cola *gui* se utiliza únicamente para aplicaciones gráficas e interactivas que no necesiten de mucho CPU. Por otro lado, *Karnak* también cuenta con las siguientes colas batch de uso especial:

COLA	UTILIZACION	TIEMPO	MEMORIA	DISCO	NUM. PROCS.
Research16	PARALELO	2 hrs	1 Gb	500 Mb	16
Services	SEQ/PAR	10 días	2 Gb	2 Gb	8
cpuset2	PARALELO	??	250 Mb	Sin límite	2
cpuset4	PARALELO	??	1 Gb	Sin límite	4
cpuset8	PARALELO	??	1 Gb	Sin límite	8
cpuset16	PARALELO	??	4Gb	Sin límite	16
cpuset32	PARALELO	??	4 Gb	Sin límite	32
Mbench	PARALELO	15 min	1 Gb	1 Gb	?
Bench	PARALELO	8 hrs	2 Gb	2 Gb	?

Son colas que siempre están desactivadas de modo que para poderlas utilizar es necesario ponerse en contacto con el súper usuario del sistema.

V.2. Ejecución de los CPANS en el sistema de colas de Karnak

Las colas *batch* que se utilizaron en la ejecución de los *CPANS Farm*, *Pipe*, *TreeDV* y *FarmBB*, fueron las colas especiales *cpusetX* (donde *X* denota el número de procesadores de cada *cpuset* que pueden ser de *2,4,8,16 y 32*). Las colas *cpuset* se ejecutan en el sistema de *Karnak*, todos los días laborables en los siguientes horarios: 8:00, 10:00, 13:00 y 18:00 hrs. Para ello es necesario pertenecer al grupo de trabajo *cpuset*, y siempre que se quiera enviar algún *job* a cualquiera de las *cpuset*, habrá que ponerse en contacto con el administrador del sistema de colas de *Karnak*, solicitando por un lado la ejecución del *job* y por otro informando de los requerimientos de éste en cuanto a tiempo de ejecución y memoria se refiere.

El envío de un job a cualquiera de las *cpuset* se hace de dos formas:

- Ejecutando el *escript* que el sistema de Karnak proporciona, en lugar de utilizar el comando NQE para el envío de *jobs* a colas batch (*qsub*):

```
--------------------
karnak(554)% envia_cpuset
envia_cpuset <num_cpus> <script>
<num_cpus> => Numero de cpus del cpuset (2, 4, 8, 16, 24, 32)
<script> => El script normal pero con permisos de ejecución
--------------------
```

- Haciendo dos *scripts*, uno normal que es el que realmente se ejecutará y uno especial que se tendrá que enviar con el comando *qsub* a la cola *cpusetX* indicada. El *script* especial sólo servirá para ejecutar el *script* normal con el comando *miser_cpuset*:

```
--------------------
#!/bin/tcsh
cd path_script
/usr/sbin/miser_cpuset -q nom_cpuset -A ./script
---------------------
```

  donde *nom_cpuset* será cualquiera de las *cpuset* disponibles y *./script* será el script que normalmente se envía al sistema de colas con *qsub* y que ha de tener permisos de ejecución.

V.2.1. Metodología de trabajo para la evaluación de los CPANS

La evaluación de la ejecución de los *CPANS Farm, Pipe, TreeDV y FarmBB* en el sistema de colas NQE de *Karnak* sigue las siguientes acciones:

1. Creación de un *MakeFile* para la compilación de los *CPANS* en sus versiones secuenciales y paralelas
2. Creación y uso de un *script* por cada *CPAN* para enviar a las colas *cpuset* sus ejecuciones
3. Medición de parámetros de rendimiento que muestran el comportamiento de los *CPANS*, realizando medidas de:
   - 3.1. Tiempo de ejecución de cada *CPAN* (incluyendo los de las versiones secuenciales) y medición de los fallos de página provocados por el sistema en ellos, con el uso del comando *ssusage*.
   - 3.2. Ciclos por instrucción ejecutados (CPI) por cada *CPAN* (incluyendo los de las versiones secuenciales) con el uso del comando *perfex*
   - 3.3. Fallos a la caché de datos secundaria provocados en los *CPANS*, a través del uso del comando *ssrun*.
   - 3.4. *SpeedUp* para cada ejecución de los *cpans* en las colas *cpusetX*, respecto de sus ejecuciones secuenciales
   - 3.5. Cota superior del *SpeedUp* para cada *CPAN* que se ejecute, utilizando para ello la *ley de Amdahl*

Un ejemplo del fichero *makefile* que se creo para la compilación de un *CPAN*, digamos el *CPAN farm* es:

```
cpanfarm: cpanfarm.o
        CC -o cpanfarm cpanfarm.cpp -lpthread
cpanfarm.o: cpanfarm.cpp
        CC -D_POSIX_C_SOURCE=199506 -D_REENTRANT -g -c cpanfarm.cpp
```

Un ejemplo del *script* que se utilizó en la compilación del mismo *CPAN farm*, es:

```
#!/bin/tcsh
cd /user1/uni/ugr/ugr-sc/rossainz/CPANS
echo APLICACION DEL CPANFARM
echo ----------------------------------------
echo
echo EJECUCION DE SSUSAGE CPANFARM
echo ==============================
ssusage cpanfarm
echo EJECUCION DE PERFEX CPANFARM
echo ==============================
perfex -e 0 -e 17 cpanfarm
echo EJECUCION DE SSRUN CPANFARM
echo ==============================
ssrun -dsc_hwc cpanfarm
```

V.3. EL RENDIMIENTO DE LOS CPANS

V.3.1. Resultados de rendimiento del Cpan Farm

De la ejecución de la versión secuencial del *Cpan Farm* se obtuvieron los siguientes resultados.

- La ejecución dura 342.83 segundos y consume 332.31+0.39 = 332.70 segundos de CPU.
- Se consigue un CPI= 1.075.
- Los fallos de página provocados son 0 major y 27 minor.
- Realiza 51352*131= 6727112 fallos a la cache de datos secundaria

Las medidas obtenidas de la ejecución del *Cpan Farm* paralelo sobre las colas exclusivas cpuset2, cpuset4, cpuset8, cpuset16 y cpuset32 para 2, 4, 8, 16 y 32 procesadores respectivamente, fueron las siguientes

1. ***Cpan Farm paralelo sobre la cola exclusiva cpuset2***
    1.1. La ejecución dura 189.41 segundos, y el consumo de CPU es de 273.86 (273.70+0.16) segundos.
    1.2. El speedup obtenido sobre 2 procesadores es de: 342.83/189.41= 1.81
    1.3. No se reportaron fallos en la caché secundaria.
    1.4. El CPI conseguido es de 5612/6247=0.898, que es 1.19 veces inferior al de la versión secuencial.
    1.5. Los fallos de página provocados son 0 mayor y 39 minor ligeramente superior a la versión secuencial.
2. ***Cpan Farm paralelo sobre la cola exclusiva cpuset4***
    2.1. La ejecución dura 147.49 segundos, y el consumo de CPU es de 294.42 (294.19+0.23) segundos.
    2.2. El speedup obtenido sobre 4 procesadores es de: 342.83/147.49= 2.32
    2.3. La ejecución no reportó fallos en la caché secundaria.
    2.4. El CPI conseguido es de 5542/6247=0.887, que es 1.21 veces inferior al de la versión secuencial.
    2.5. Los fallos de página provocados son 0 mayor y 39 minor ligeramente superior a la versión secuencial.
3. ***Cpan Farm paralelo sobre la cola exclusiva cpuset8***
    3.1. La ejecución dura 145.13 segundos, y el consumo de CPU es de 287.42 (287.22+0.20) segundos.
    3.2. El speedup obtenido sobre 8 procesadores es de: 342.83/145.13= 2.36
    3.3. La ejecución no reportó fallos en la caché secundaria.
    3.4. El CPI conseguido es de 5462/6247=0.874, que es 1.23 veces inferior al de la versión secuencial.
    3.5. Los fallos de página provocados son 1 mayor y 39 minor ligeramente superior a la versión secuencial.
4. ***Cpan Farm paralelo sobre la cola exclusiva cpuset16***
    4.1. La ejecución dura 143.27 segundos, y el consumo de CPU es de 292.30 (292.17+0.13) segundos.
    4.2. El speedup obtenido sobre 16 procesadores es de: 342.83/143.27= 2.39
    4.3. La ejecución no reportó fallos en la caché secundaria.

4.4. El CPI conseguido es de 5420/6247=0.868, que es 1.24 veces inferior al de la versión secuencial.

4.5. Los fallos de página provocados son 0 mayor y 39 minor ligeramente superior a la versión secuencial.

5. ***Cpan Farm paralelo sobre la cola exclusiva cpuset32***

5.1. La ejecución dura 141.21 segundos, y el consumo de CPU es de 292.53 (292.38+0.15) segundos.

5.2. El speedup obtenido sobre 32 procesadores es de: 342.83/141.21= 2.43

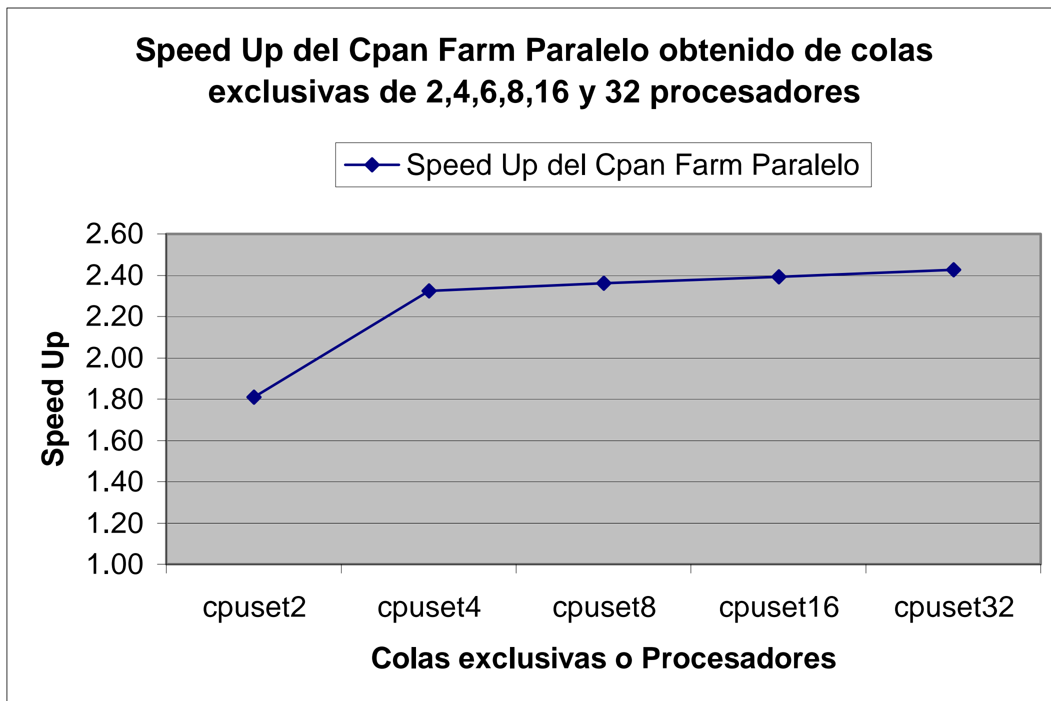
5.3. La ejecución no reportó fallos en la caché secundaria.

5.4. El CPI conseguido es de 5293/6247=0.847, que es 1.27 veces inferior al de la versión secuencial.

5.5. Los fallos de página provocados son 0 mayor y 39 minor ligeramente superior a la versión secuencial.

V.3.1.1. Representación gráfica del rendimiento del Cpan Farm

		Ejecución del Cpan Farm Paralelo en colas exclusivas de 2, 4, 8, 16 y 32 procesadores				
	Cpan Farm Secuencial	**cpuset2**	**cpuset4**	**cpuset8**	**cpuset16**	**cpuset32**
Tiempo de ejecución en seg.	***342.83***	**189.41**	**147.49**	**145.13**	**143.27**	**141.21**
Consumo de tiempo en seg. en modo usuario	*332.31*	273.70	294.19	287.22	292.17	292.38
Consumo de tiempo en seg. en modo Sistema	*0.39*	0.16	0.23	0.20	0.13	0.15
Consumo de tiempo en seg. de CPU	***332.70***	***273.86***	***294.42***	***287.42***	***292.30***	***292.53***
Número de Ciclos	*79783881216*	56126416176	55420223332	54624908189	54209885979	52936449087
Número de Instrucciónes	*74235179493*	62474668759	62471359081	62472731100	62478338966	62470515179
CPI	***1.075***	***0.898***	***0.887***	***0.874***	***0.868***	***0.847***
Speed Up	***1.00***	***1.81***	***2.32***	***2.36***	***2.39***	***2.43***
Amdalh	***1.00***	***1.82***	***3.08***	***4.71***	***6.40***	***7.80***



V.3.2. Resultados de rendimiento del Cpan Pipe

De la ejecución de la versión secuencial del *Cpan Pipe* se obtuvieron los siguientes resultados:

- La ejecución dura 238.50 segundos y consume 229.78+2.04 = 331.82 segundos de CPU.
- Se consigue un CPI= 1.862.
- Los fallos de página provocados son 0 major y 33 minor.
- Realiza 4175756*131= 547024036 fallos a la cache de datos secundaria

Las medidas obtenidas de la ejecución del *Cpan Pipe* paralelo sobre las colas exclusivas cpuset2, cpuset4, cpuset8, cpuset16 y cpuset32 para 2, 4, 8, 16 y 32 procesadores respectivamente, fueron las siguientes

1. ***Cpan Pipe sobre la cola exclusiva cpuset2***
   1.1. La ejecución dura 127.27 segundos, y el consumo de CPU es de 224.79 (221.23+3.56) segundos.
   1.2. El speedup obtenido sobre 2 procesadores es de: 238.50/127.27= 1.87
   1.3. La ejecución no reportó fallos en la caché secundaria.

1.4. El CPI conseguido es de 4443/4886=0.909, que es 2.04 veces inferior al de la versión secuencial.
1.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

2. ***Cpan Pipe sobre la cola exclusiva cpuset4***
    2.1. La ejecución dura 123.83 segundos, y el consumo de CPU es de 217.80 (214.27+3.53) segundos.
    2.2. El speedup obtenido sobre 4 procesadores es de: 238.50/123.83= 1.93
    2.3. La ejecución no reportó fallos en la caché secundaria.
    2.4. El CPI conseguido es de 4379/4844=0.904, que es 2.06 veces inferior al de la versión secuencial.
    2.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.
3. ***Cpan Pipe sobre la cola exclusiva cpuset8***
    3.1. La ejecución dura 116.97 segundos, y el consumo de CPU es de 203.98 (200.12+3.86) segundos.
    3.2. El speedup obtenido sobre 8 procesadores es de: 238.50/116.97= 2.04
    3.3. La ejecución no reportó fallos en la caché secundaria.
    3.4. El CPI conseguido es de 4352/4829=0.901, que es 2.07 veces inferior al de la versión secuencial.
    3.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.
4. ***Cpan Pipe sobre la cola exclusiva cpuset16***
    4.1. La ejecución dura 115.63 segundos, y el consumo de CPU es de 198.30 (194.90+3.40) segundos.
    4.2. El speedup obtenido sobre 16 procesadores es de: 238.50/115.63= 2.06
    4.3. La ejecución no reportó fallos en la caché secundaria.
    4.4. El CPI conseguido es de 4256/4802=0.886, que es 2.10 veces inferior al de la versión secuencial.
    4.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.
5. ***Cpan Pipe sobre la cola exclusiva cpuset32***
    5.1. La ejecución dura 111.03 segundos, y el consumo de CPU es de 191.89 (188.48+3.41) segundos.

5.2. El speedup obtenido sobre 32 procesadores es de: 238.50/111.03= 2.15

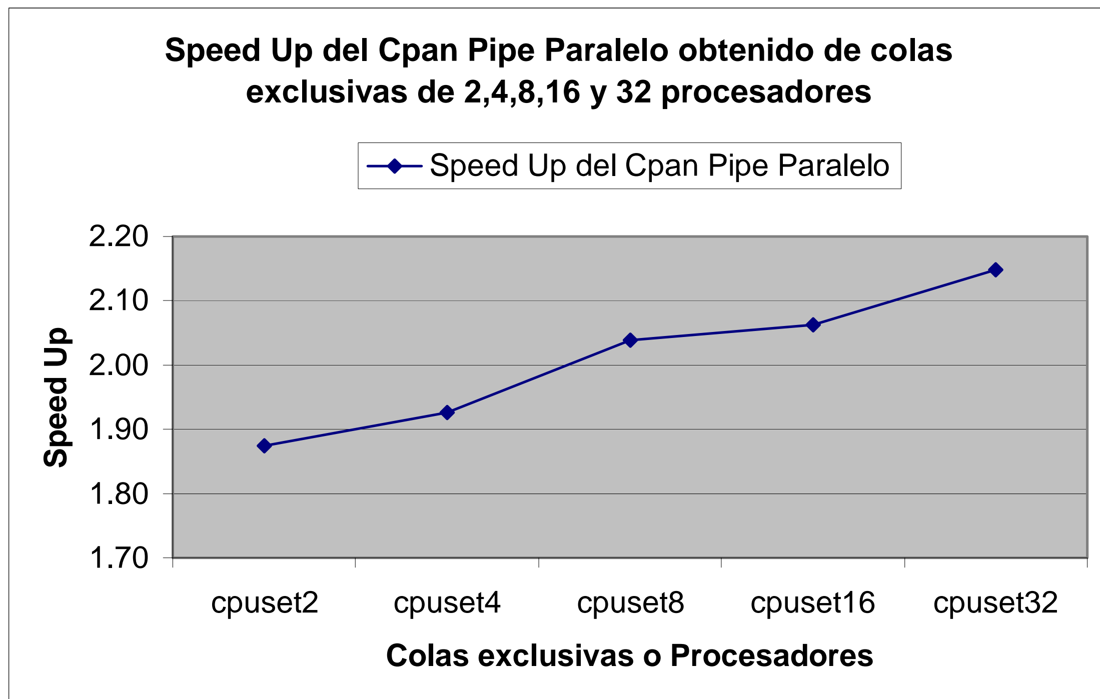
5.3. La ejecución no reportó fallos en la caché secundaria.

5.4. El CPI conseguido es de 4197/4820=0.871, que es 2.14 veces inferior al de la versión secuencial.

5.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

V.3.2.1. Representación gráfica del rendimiento del Cpan Pipe

		Ejecución del Cpan Pipe Paralelo en colas exclusivas de 2, 4, 8, 16 y 32 procesadores				
	Cpan Pipe Secuencial	**cpuset2**	**cpuset4**	**cpuset8**	**cpuset16**	**cpuset32**
Tiempo de ejecución en seg.	***238.50***	**127.27**	**123.83**	**116.97**	**115.63**	**111.03**
Consumo de tiempo en seg. en modo usuario	*229.78*	221.23	214.27	200.12	194.90	188.48
Consumo de tiempo en seg. en modo Sistema	*2.04*	3.56	3.53	3.86	3.40	3.41
Consumo de tiempo en seg. de CPU	***231.82***	***224.79***	***217.80***	***203.98***	***198.30***	***191.89***
Número de Ciclos	*561990287660*	44434829110	43799794339	43521535545	42569949308	41979138703
Número de Instrucciónes	*301773457276*	48867065068	48449064443	48298118230	48022837040	48207328639
CPI	***1.862***	***0.909***	***0.904***	***0.901***	***0.886***	***0.871***
Speed Up	***1.00***	***1.87***	***1.93***	***2.04***	***2.06***	***2.15***
Amdalh	***1.00***	***1.89***	***3.39***	***5.63***	***8.42***	***11.19***



V.3.3. Resultados de rendimiento del Cpan TreeDV

De la ejecución de la versión secuencial del *Cpan TreeDV* se obtuvieron los siguientes resultados:

- La ejecución dura 4.77 segundos y consume 0.25+1.40 = 1.65 segundos de CPU.
- Se consigue un CPI= 1.163.
- Los fallos de página provocados son 0 major y 25 minor.
- Realiza 13624*131= 1784744 fallos a la cache de datos secundaria.

Las medidas obtenidas de la ejecución del *Cpan TreeDV* paralelo sobre las colas exclusivas cpuset2, cpuset4, cpuset8, cpuset16 y cpuset32 para 2, 4, 8, 16 y 32 procesadores respectivamente, fueron las siguientes

1. ***Cpan TreeDV sobre la cola exclusiva cpuset2***

   1.1. La ejecución dura 3.24 segundos, y el consumo de CPU es de 1.42 (0.20+1.22) segundos.

   1.2. El speedup obtenido sobre 2 procesadores es de: 4.77/3.24= 1.47

   1.3. La ejecución no reportó fallos en la caché secundaria.

1.4. El CPI conseguido es de 1165/2033=0.573, que es 2.02 veces inferior al de la versión secuencial.

1.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

2. *Cpan TreeDV sobre la cola exclusiva cpuset4*

2.1. La ejecución dura 3.02 segundos, y el consumo de CPU es de 1.26 (0.18+1.08) segundos.

2.2. El speedup obtenido sobre 4 procesadores es de: 4.77/3.02= 1.58

2.3. La ejecución no reportó fallos en la caché secundaria.

2.4. El CPI conseguido es de 1155/2025=0.570, que es 2.04 veces inferior al de la versión secuencial.

2.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

3. *Cpan TreeDV sobre la cola exclusiva cpuset8*

3.1. La ejecución dura 2.94 segundos, y el consumo de CPU es de 1.31 (0.18+1.13) segundos.

3.2. El speedup obtenido sobre 8 procesadores es de: 4.77/2.94= 1.62

3.3. La ejecución no reportó fallos en la caché secundaria.

3.4. El CPI conseguido es de 1124/2011=0.559, que es 2.08 veces inferior al de la versión secuencial.

3.5. Los fallos de página provocados son 2 mayor y 38 minor ligeramente superior a la versión secuencial.

4. *Cpan TreeDV sobre la cola exclusiva cpuset16*

4.1. La ejecución dura 2.89 segundos, y el consumo de CPU es de 1.30 (0.18+1.12) segundos.

4.2. El speedup obtenido sobre 16 procesadores es de: 4.77/2.89= 1.65

4.3. La ejecución no reportó fallos en la caché secundaria.

4.4. El CPI conseguido es de 1077/2014=0.535, que es 2.17 veces inferior al de la versión secuencial.

4.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

5. *Cpan TreeDV sobre la cola exclusiva cpuset32*

5.1. La ejecución dura 1.99 segundos, y el consumo de CPU es de 1.32 (0.18+1.14) segundos.

5.2. El speedup obtenido sobre 32 procesadores es de: 4.77/1.99= 2.40

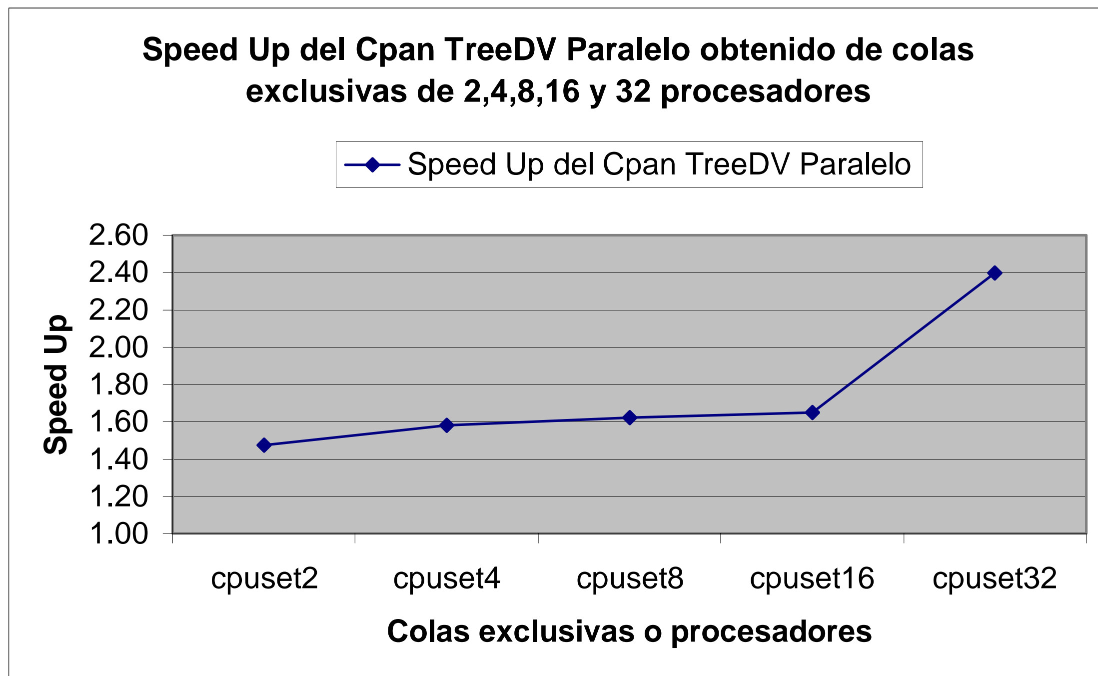
5.3. La ejecución no reportó fallos en la caché secundaria.

5.4. El CPI conseguido es de 1076/2052=0.524, que es 2.22 veces inferior al de la versión secuencial.

5.5. Los fallos de página provocados son 0 mayor y 38 minor ligeramente superior a la versión secuencial.

V.3.3.1. Representación gráfica del rendimiento del Cpan TreeDV

		Ejecución del Cpan TreeDV Paralelo en colas exclusivas de 2, 4, 8, 16 y 32 procesadores				
	Cpan Tree Secuencial	**cpuset2**	**cpuset4**	**cpuset8**	**cpuset16**	**cpuset32**
Tiempo de ejecución en seg.	***4.77***	**3.24**	**3.02**	**2.94**	**2.89**	**1.99**
Consumo de tiempo en seg. en modo usuario	*0.25*	0.20	0.18	0.18	0.18	0.18
Consumo de tiempo en seg. en modo Sistema	*1.40*	1.22	1.08	1.13	1.12	1.14
Consumo de tiempo en seg. de CPU	***1.65***	***1.42***	***1.26***	***1.31***	***1.30***	***1.32***
Número de Ciclos	*56168656*	11656651	11550967	11247449	10779621	10763400
Número de Instrucciónes	*48307686*	20337798	20257566	20112183	20144420	20527527
CPI	***1.163***	***0.573***	***0.570***	***0.559***	***0.535***	***0.524***
Speed Up	***1.00***	***1.47***	***1.58***	***1.62***	***1.65***	***2.40***
Amdalh	***1.00***	***1.90***	***3.48***	***5.93***	***9.14***	***12.55***



V.3.4. Resultados de rendimiento del Cpan FarmBB

De la ejecución de la versión secuencial del *Cpan FarmBB* se obtuvieron los siguientes resultados:

- La ejecución dura 35.42 segundos y consume 25.80+1.30 = 27.10 segundos de CPU.
- Se consigue un CPI= 1.321.
- Los fallos de página provocados son 0 major y 33 minor.
- Realiza 58438*131= 7655378 fallos a la cache de datos secundaria

Las medidas obtenidas de la ejecución del *Cpan FarmBB* paralelo sobre las colas exclusivas cpuset2, cpuset4, cpuset8, cpuset16 y cpuset32 para 2, 4, 8, 16 y 32 procesadores respectivamente, con una *N=50 ciudades* y costos aleatorios (entre ciudades) con valores entre 1 y100 , fueron las siguientes

1. ***Cpan FarmBB sobre la cola exclusiva cpuset2***
   1.1. La ejecución dura 21.88 segundos, y el consumo de CPU es de 23.25 (22.14+1.11) segundos.
   1.2. El speedup obtenido sobre 2 procesadores es de: 35.42/21.88=1.62
   1.3. La ejecución no reportó fallos en la caché secundaria.

1.4. El CPI conseguido es de 6987/7338=0.952, que es 1.38 veces inferior al de la versión secuencial.

1.5. Los fallos de página provocados son 0 mayor y 37 minor ligeramente superior a la versión secuencial.

2. *Cpan FarmBB sobre la cola exclusiva cpuset4*

2.1. La ejecución dura 14.21 segundos, y el consumo de CPU es de 21.17 (20.16+1.01) segundos.

2.2. El speedup obtenido sobre 4 procesadores es de: 35.42/14.21=2.49

2.3. La ejecución no reportó fallos en la caché secundaria.

2.4. El CPI conseguido es de 6907/7321=0.943, que es 1.40 veces inferior al de la versión secuencial.

2.5. Los fallos de página provocados son 0 mayor y 37 minor ligeramente superior a la versión secuencial.

3. *Cpan FarmBB sobre la cola exclusiva cpuset8*

3.1. La ejecución dura 11.34 segundos, y el consumo de CPU es de 19.19 (18.16+1.03) segundos.

3.2. El speedup obtenido sobre 8 procesadores es de: 35.42/11.34=3.12

3.3. La ejecución no reportó fallos en la caché secundaria.

3.4. El CPI conseguido es de 6766/7293=0.928, que es 1.42 veces inferior al de la versión secuencial.

3.5. Los fallos de página provocados son 0 mayor y 37 minor ligeramente superior a la versión secuencial.

4. *Cpan FarmBB sobre la cola exclusiva cpuset16*

4.1. La ejecución dura 10.27 segundos, y el consumo de CPU es de 22.69 (21.67+1.02) segundos.

4.2. El speedup obtenido sobre 16 procesadores es de: 35.42/10.27=3.45

4.3. La ejecución no reportó fallos en la caché secundaria.

4.4. El CPI conseguido es de 6694/7245=0.924, que es 1.43 veces inferior al de la versión secuencial.

4.5. Los fallos de página provocados son 0 mayor y 37 minor ligeramente superior a la versión secuencial.

5. *Cpan FarmBB sobre la cola exclusiva cpuset32*

5.1. La ejecución dura 9.10 segundos, y el consumo de CPU es de 22.18 (21.14+1.04) segundos.

5.2. El speedup obtenido sobre 32 procesadores es de: 35.42/9.10=3.89

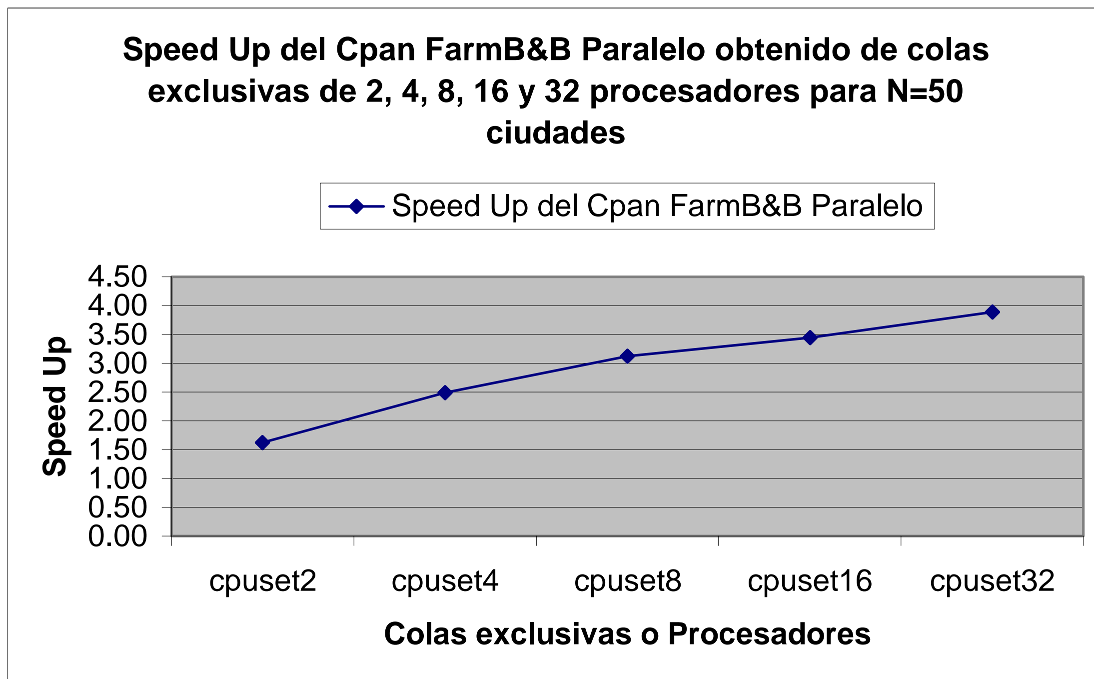
5.3. La ejecución no reportó fallos en la caché secundaria.

5.4. El CPI conseguido es de 6599/7223=0.914, que es 1.44 veces inferior al de la versión secuencial.

5.5. Los fallos de página provocados son 0 mayor y 37 minor ligeramente superior a la versión secuencial.

V.3.4.1. Representación gráfica del rendimiento del Cpan FarmBB

		Ejecución del Cpan FarmBB Paralelo en colas exclusivas de 2, 4, 8, 16 y 32 procesadores con N=50 ciudades				
	Cpan FarmBB Secuencial	**cpuset2**	**cpuset4**	**cpuset8**	**cpuset16**	**Cpuset32**
Tiempo de ejecución en seg.	***35.42***	**21.88**	**14.21**	**11.34**	**10.27**	**9.10**
Consumo de tiempo en seg. en modo usuario	*25.80*	22.14	20.16	18.16	21.67	21.14
Consumo de tiempo en seg. en modo Sistema	*1.30*	1.11	1.01	1.03	1.02	1.04
Consumo de tiempo en seg. de CPU	***27.10***	***23.25***	***21.17***	***19.19***	***22.69***	***22.18***
Número de Ciclos	266948719	69877161	69073500	67663433	66942422	65992570
Número de Instrucciónes	202106112	73380571	73219450	72932731	72454022	72235919
CPI	***1.321***	***0.952***	***0.943***	***0.928***	***0.924***	***0.914***
Speed Up	***1.00***	***1.62***	***2.49***	***3.12***	***3.45***	***3.89***
Amdalh	***1.00***	***1.68***	***2.55***	***3.43***	***4.16***	***4.64***



V.4. Porcentajes de código secuencial y paralelo de los CPANS para la ley de Amdahl

Cpanfarm:

tiempo de ejecución de la parte secuencial:	30.02 seg.	(10%)
tiempo de ejecución de la parte paralela:	312.81seg.	(90%)
tiempo de ejecución secuencial total:	342.83seg.	

Cpanpipe:

tiempo de ejecución de la parte secuencial:	15.32 seg.	(6%)
tiempo de ejecución de la parte paralela:	223.18seg.	(94%)
tiempo de ejecución secuencial total:	238.50seg.	

CpanTreeDV:

tiempo de ejecución de la parte secuencial:	0.10 seg.	(2%)
tiempo de ejecución de la parte paralela:	4.67 seg.	(98%)
tiempo de ejecución secuencial total:	4.77 seg.	

CpanFarmBB:

tiempo de ejecución de la parte secuencial: 6.73 (19%)

tiempo de ejecución de la parte paralela: 28.69 (81%)

---

tiempo de ejecución secuencial total: 35.42 seg.

V.5. CONCLUSIONES DEL ANÁLISIS DE RENDIMIENTO DE LOS CPANS

1. La ejecución paralela de los *cpans* tienen un tiempo de ejecución inferior al tiempo utilizado por sus correspondientes versiones secuenciales como era de esperarse.
2. Los tiempos de ejecución de los *CPANS* en sus versiones paralelas mejoran a medida que se aumentan el número de procesadores en sus ejecuciones, es decir, conforme va aumentando el número de procesadores con los que se ejecutan los *CPANS*, sus tiempos de ejecución van disminuyendo.
3. Se aprecia un *SpeedUp* siempre hacia arriba en la mejora de los tiempos de ejecución de los *CPANS* paralelos respecto de su contraparte secuencial, como lo muestran las gráficas y por debajo de las cotas de la Ley de Amdahl calculadas.
4. En cuanto a los CPI se observa una mejora en ellos a medida que se aumentan el número de procesadores, es decir, a mayor número de procesadores utilizados en la ejecución de los *CPANS* menor es el porcentaje de los ciclos por instrucción. Esto indica que mientras que el número de instrucciones en las ejecuciones de la aplicación dentro de las *cpusetX* se mantiene más o menos constante, el número de ciclos por instrucción va disminuyendo, lo cual reditúa en una ganancia en el valor final del CPI.
5. Los fallos de memoria en la caché secundaria desaparecen en las versiones paralelas de los *CPANS*, no así en sus correspondientes versiones secuenciales.

Capítulo VI:

CONCLUSIONES

VI. 1. CONCLUSIONES

1. Se ha desarrollado una metodología de programación original basada en Composiciones Paralelas de Alto Nivel o *CPANS*.
2. Se han implementado bajo el modelo del *CPAN* los patrones de comunicación/interacción más comúnmente utilizados en la programación paralela: *El Pipeline, el Farm y el Tree*.
3. Se ha creado una librería de clases de Objetos Paralelos que proporciona al usuario los patrones citados como *CPANS*: *El Cpan Pipe, el Cpan TreeDV* (que representa a la técnica de Divide y Vencerás)*, el Cpan Farm y el Cpan FarmBB* (que representa la técnica de Ramificación y Poda).
4. Los *CPANS* implementados pueden ser explotados (con conocimientos mínimos de concurrencia y paralelismo) gracias a la adopción del enfoque Orientado a Objetos, utilizando los diferentes mecanismos de reusabilidad del paradigma, para definir nuevos patrones en base a los ya construidos. Un ejemplo de ello es el *Cpan FarmBB* que implementa la técnica de Ramificación y Poda y que ha sido definido y construido en base al *Cpan Farm* mediante el mecanismo de herencia.
5. Se han Transformado algoritmos conocidos que resuelven problemas secuenciales (algoritmos de ordenación, búsqueda y optimización) en algoritmos paralelizables, y con ellos se ha probado la utilidad de la metodología y de los componentes software desarrollados en la presente tesis. En particular, se ha resuelto el problema del Agente Viajero (TSP), que es un problema NP-Completo usando para ello el *Cpan FarmBB*.
6. Se han programado de forma original las restricciones de sincronización sugeridas por el modelo del *CPAN* para su funcionamiento paralelo y concurrente: el paralelismo máximo (*MaxPar*), la exclusión mutua (*Mutex*) y la sincronización de comunicación de procesos en base al modelo lectores/escritores (*Sync*).
7. De igual forma, la programación del *modo de comunicación futuro asíncrono* para resultados *"futuros"* dentro de los *CPANS* se ha llevado a cabo de manera original mediante objetos y clases. Además, se han programado los otros dos modos de comunicación clásicos entre procesos: *el modo de comunicación síncrono y asíncrono*.
8. Se ha probado el rendimiento de cada uno de los *CPANS* propuestos mediante métricas de SpeedUp, la ley de Amdahl y eficiencia, demostrando que el

comportamiento paralelo de los *CPANS* es mejor que la contraparte secuencial en todos los casos.

9. El problema denominado *PPP (Programmability, Portability, Performance*) a quedado resuelto en la implementación de los *CPANS*. La *programabilidad* esta garantizada al adoptar el enfoque basado en patrones como clases y objetos paralelos en la creación de los *CPANS*. La *Portabilidad* se da por la transparencia en la distribución de aplicaciones paralelas en el lenguaje C++, lo que permite que los *CPANS* puedan ser portables desde algún sistema centralizado hacia un sistema distribuido sin necesidad de afectar el código fuente. Y finalmente el *Rendimiento o Performance* se ha logrado en los *CPANS* al garantizar que su comportamiento paralelo, para diferente número de procesadores, es siempre mejor que su contraparte secuencial.

VI.2. APORTE DE LA TESIS A LA CIENCIA COMPUTACIONAL

1. Los resultados obtenidos en la presente investigación acortan la brecha existente en cuanto a la aceptación de entornos de programación paralela estructurada para el desarrollo de aplicaciones paralelas.
2. Los usuarios cuentan con una librería de clases de objetos paralelos, que instancian composiciones paralelas de alto nivel y que pueden definirse y utilizarse dentro de su infraestructura (lenguaje de programación) orientada a objetos.
3. La librería de patrones propuesta es una librería completa pues cuenta con una sintaxis y una semántica bien definida con la suficiente generalidad para que pueda ser aceptada por parte de la comunidad científica.
4. El uso del enfoque orientado a objetos en la construcción de los *CPANS* soluciona el problema de la falta de existencia de un conjunto completo de patrones a los que se les permite añadir nuevos patrones mediante la herencia.
5. El programador o usuario obtiene ventajas útiles al utilizar la librería propuesta bajo el paradigma de la orientación a objetos como: genericidad, uniformidad, y facilidad de uso.

VI.3. TRABAJOS FUTUROS

1. Robustecer la librería de *CPANS* propuesta con nuevos patrones paralelos de comunicación que puedan facilitar aun más el desarrollo de aplicaciones paralelas en usuarios y programadores novel. Algunas propuestas son los patrones geométricos, los de pares totales, los cubos, los hipercubos, etc.
2. Crear todo un ambiente integrado de desarrollo, una herramienta CASE o ambiente Visual donde por medio de una interfaz gráfica de usuario amigable, el programador pueda visualmente hacer una composición de sus CPANS para la solución de sus problemas. El ambiente tendrá que tener la posibilidad no solo de crear gráficamente la solución del problema sino además, generar código ejecutable en algún lenguaje de programación orientado a objetos como puede ser el mismo C++, así como la posibilidad de incluir en la librería los nuevos patrones creados por el propio programador.

Apéndice A

LA LIBRERÍA DE CLASES DE LOS CPANS: Descripción de la librería de clases que constituyen los CPANs propuestos

A.1. INTRODUCCIÓN

La implementación paralela del modelo *CPAN* se concretiza con la librería de clases de objetos paralelos que se propone y que define los patrones paralelos de comunicación de procesos con los que se han venido trabajando, vistos como Composiciones Paralelas de Alto Nivel[24]: El patrón *Farm* representado por el *Cpan Farm*, el patrón *Pipeline* representado por el *Cpan Pipe*, el patrón *Binary-tree* junto con la técnica de *Divide y Vencerás,* representado por el *Cpan TreeDV* y el patrón *FarmB&B* que implementa la técnica de *Ramificación y poda* representado por el *Cpan FarmBB*.

La librería ha sido implementada en el lenguaje *C++* bajo el paradigma de la orientación a objetos en su parte secuencial y para la contraparte paralela se ha utilizado el estándar *PThread*, para el manejo de hilos y compartición de recursos.

La adopción del enfoque de la orientación a objetos en la implementación de la librería ha hecho posible solucionar el problema de encontrar un conjunto completo de patrones paralelos de comunicación, ya que el paradigma permite añadir nuevos patrones a un conjunto inicial incompleto mediante la definición de subclases. La línea de investigación que se siguió fue el encontrar representaciones de patrones paralelos como *clases*, a partir de las cuales se pueden instanciar objetos paralelos o *CPANS* que son, a su vez, ejecutados como consecuencia de una petición de servicio externa a dichos objetos y procedentes de la aplicación de usuario.

La conveniencia de utilizar los *PThreads* en la implementación de los *CPANS* radica en el hecho de que los *PThreads* como paquete de hilos, permiten escribir programas con varios puntos simultáneos de ejecución, sincronizados a través de memoria compartida. La librería de clases de los *CPANS* se forma principalmente de tres grupos de clases:

1. El grupo de las clases base necesarias para construir un *CPAN*, o dicho de otra forma, las clases que implementan los Objetos Paralelos de éste.
2. El grupo de las clases que definen los *CPANS* concretos que constituyen la librería: *Cpan Farm*, *Cpan Pipe*, *Cpan TreeDV* y *Cpan FarmBB*.

[24] Es decir, construyendo el objeto manager, el objeto collector, los objetos Stage y los objetos esclavo, así como sus interconexiones de acuerdo a la estructura requerida: granja, cauce o árbol.

3. El grupo de clases definidas por el usuario para la creación de los objetos esclavos a ser trabajados por los *CPANS*. En este caso se han incorporado a la librería a manera de ejemplo, clases que constituyen los objetos esclavos para los *CPANS farm, pipe y treeDV*, que implementan la solución de problemas de ordenamiento de un conjunto de valores y clases que constituyen los objetos esclavos del *CPAN FarmBB* que resuelven el problema del agente viajero o TSP.

De los tres grupos de clases citados, el tercero es el único cuya programación se ha llevado a cabo de forma secuencial. El resto ha sido programado en forma paralela utilizando las primitivas del manejo de hilos con el uso de los *PThreads*, implementando las políticas de planificación de procesos[25] y los modelos de lectores-escritores ya descritos en capítulos anteriores.

A.2. EL GRUPO DE LAS CLASES BASE DE LA LIBRERÍA DE LOS CPANS

CLASES BASE PARA LA CONSTRUCCION DE UN CPAN	
Object	Utilizada para definir objetos esclavos genéricos. Debe ser la superclase de las clases que definen objetos esclavos específicos.
CHilo	Proporciona métodos para iniciar un hilo de ejecución. Es una clase abstracta con un método virtual puro *fnHilo( )* que se corresponderá con el hilo de ejecución. Será la clase padre de las clases *ComponentCollector, ComponentStage y ComponentManager* de un *CPAN*, las cuales tendrán que redefinir el método *fnHilo()*.
ComponentCollector	Es la clase Concreta que define el componente *"collector"* de un *CPAN* y genera objetos colectores que recopilan las soluciones de los objetos *"stage"* conectados a él.
ComponentStage	Es la clase Abstracta que define el componente *"stage"* de un *CPAN* que debe ser especializado y definido en aquellas clases que hereden de ésta.
ComponentManager	Es la clase Abstracta que define el componente *"manager"* de un *CPAN* que debe ser especializado y definido en aquellas clases que hereden de ésta.
Util	Fichero de encabezado que contiene la definición de diversas *"funciones primitivas"* y de varios tipos de datos abstractos necesarios en la implementación de los CPANS

A.2.1. La clase base Object

Es una clase abstracta que debe ser utilizada en un *CPAN* cuando se quieran definir o instanciar objetos esclavos. Esta formada por un constructor, un destructor y un método virtual *"resuelve( )"* que deberá ser implementado por cualquier clase que

[25] Políticas de mutex (exclusión mutua), maxpar (paralelismo máximo) y sync (sincronización usando el modelo lectores-escritores).

herede las características de *Object* y que represente un objeto esclavo que será trabajado en el *CPAN* a implementar.

FUNCIONES MIEMBRO	DESCRIPCION
Object( )	Constructor de la clase *Object*
virtual ~Object( )	Destructor virtual de la clase *Object*
virtual void * resuelve(void *)=0	Función virtual pura que será redefinida en las clases hijas de *Object*. Contendrá el algoritmo de solución del problema. Recibe como entrada los datos a procesar como un puntero a *void* y regresa como resultado la solución del problema como una referencia a *void*.

A.2.2. El archivo de encabezado Util

Contiene primitivas y definición de tipos abstractos útiles en la implementación de los *CPANS*. Esta formado por:

1. El tipo *asociación* que es un *map* que asocia el nombre del método a ejecutar por un objeto esclavo con una referencia a éste. El nombre del método esta definido dentro de una estructura llamada *"method"*.
2. El tipo *vector_sol* es un *vector* que almacena referencias a *void* y que representa el tipo de datos utilizado en la clase *collector* para definir la variable de almacenamiento de soluciones que se colectan en esta clase.
3. La función *head( )* es una función sobrecargada utilizada para regresar la cabeza de una lista de elementos. En una de sus versiones la función *head( )* recibe una lista de objetos de tipo *asociación* y regresa la cabeza de esa lista, es decir, el objeto asociación que esta al principio de la lista. En su segunda versión *head( )* recibe una lista de objetos *vector_sol* y regresa el objeto *vector_sol* que esta al principio de la lista como una referencia a *void*.
4. La función *tail( )* es también una función sobrecargada que recibe una lista de elementos y regresa la cola o el resto de dicha lista. En su primera versión, *tail( )* trabaja con objetos de tipo asociación, y en su segunda versión trabaja con referencias a objetos de tipo *vector_sol*. Ambas funciones, *head( )* y *tail( )*[26] son

[26] Las funciones head y tail han sido implementadas en C++ con la misma semántica de las funciones "car" y "cdr" de LISP, [COR95].

utilizadas dentro de las clases *ComponentStage* y *ComponentCollector* y pueden ser utilizadas en las clases específicas de los *CPANS* según se requiera.

5. La función *cuenta_par( )* es una función de ayuda para las clases específicas, por ejemplo para la clase *ComponentManager*, donde en ella, la función se utiliza para saber si una lista de datos de tipo *asociación* tiene más de un elemento. Recibe como argumento la lista y regresa como resultado un entero que define el número de elementos en la lista.
6. La función *eval( )* es una de las primitivas más importantes y útiles de este archivo de encabezado ya que se encarga de que un objeto esclavo pueda ejecutar su método de solución del problema secuencial con los datos del problema. Recibe como parámetros la referencia al objeto esclavo de interés, la referencia al método relacionado con el objeto esclavo y la referencia a los datos, esta última como un puntero a *void*, retornando como salida, el resultado de la ejecución de dicho método por parte del objeto esclavo con los datos. Esta función es utilizada por la clase *ComponentStage*.
7. La función *cons( )* se encarga de ir construyendo la lista de resultados de tipo *vector_sol* y es utilizada por la clase *ComponentCollector*. Esta función recibe como parámetros la referencia al objeto que representa el vector de soluciones y el dato como una referencia a *void*, que será incluido dentro de dicho vector.

TIPOS ABSTRACTOS DEFINIDOS	DESCRIPCION
Method	Tipo de datos abstracto que define una estructura con un único campo: *void * (Object::*fun.)(void *)*. Sintaxis: *method *var_meth= new method(&Object::resuelve)* donde *Object* es la referencia a la clase *Object* y *resuelve* es el nombre de la función miembro virtual de dicha clase. Se utiliza para definir el método asociado a un objeto esclavo en el tipo asociación
Asociación	Es un *map<method *, Object *>* que define una asociación metodo-objeto_esclavo
vector_sol	Es un *vector<void *>* utilizado como un vector de soluciones que almacena objetos referenciado como punteros a *void*.

PRIMITIVAS	DESCRIPCION
asociación * head(asociación *lista)	Recibe como entrada una lista de asociaciones (metodo, objeto_esclavo) y da como resultado la cabeza de esa lista, esto es, una asociación.
void * head(vector_sol *content)	Primitiva sobrecargada que recibe como entrada un vector de soluciones y da como resultado la cabeza de esa lista como una referencia a *void*.
	Recibe como entrada una lista de asociaciones (metodo,

asociacion * tail(asociacion *lista)	objeto_esclavo) y da como resultado el resto de esa lista, esto es, la misma lista de asociaciones pero sin la cabeza.
void tail(vector_sol *content)	Primitiva sobrecargada que recibe como entrada un vector de soluciones para borrar la cabeza de esa lista y dejar el resto en la misma lista **content*.
int cuenta_par(asociación *lista)	Recibe como dato de entrada una lista de asociaciones (metodo, objeto_esclavo) y da como resultado el tamaño de dicha lista.
void * eval(Object *obj, method *meth, void *data)	Hace que el objeto esclavo referenciado por **obj* evalúe su método asociado referenciado en **meth* con los datos de entrada **item* y da como salida el resultado de dicho cálculo como una referencia a *void*
void cons(vector_sol *content, void *item)	Primitiva que va construyendo el vector de soluciones **content* con el dato dado en **item*.
asociacion * crea_asociacion(Object *obj[ ], method *meth[ ], int tam)	Crea un objeto asociación de forma dinámica. Recibe como entrada un *array* de objetos esclavos en **obj[ ]*, un *array* de los métodos asociados a cada objeto esclavo en **meth[ ]* y el tamaño de los arreglos en *tam*.
void elimina_obj(Object *obj[ ], int tam)	Elimina de memoria objetos de tipo *Object* que hayan sido creados dinámicamente. Recibe como entrada en **obj[ ]* el conjunto de objetos *Object* a eliminar, así como el tamaño del arreglo en *tam*..

A.2.3. La clase base CHilo

La clase *CHilo* proporciona métodos para iniciar un hilo y para que otro hilo pueda esperar a que éste finalice. Tiene un constructor que permite asignar un nombre a un hilo y establecer los atributos con los que éste se creará, un método para acceder al nombre del hilo, uno más para obtener el identificador de éste, un método para asignación de atributos y un método para modificar el nombre del hilo.

FUNCIONES MIEMBRO	DESCRIPCION
CHilo(string nom, pthread_attr_t *attr)	Es el constructor de la clase. Inicia los datos *nombre y atributos* del hilo. Si los atributos no se especifican se utilizan los definidos por omisión.
virtual ~CHilo( )	Es el destructor de la clase. Finaliza el hilo sólo en caso de que no lo haya hecho por sí mismo.
static void * ejec_hilo(void *arg)	Método estático y privado. Corresponde a la función que tiene que ejecutar el hilo. Esta función recibe como parámetro la dirección (this) del objeto que encapsula al hilo con el fin de invocar a la función virtual *fnHilo( )*.
virtual void * fnHilo( ) =0	Es un método virtual puro que hace que la clase *CHilo* sea abstracta. Esto hace necesario derivar una clase concreta de ésta que redefina el método, donde se escribirá el código que ejecutará el hilo. En el caso que nos ocupa, las clases concretas serán *ComponentCollector, ComponentStage y ComponentManager*.
void iniciar( )	Inicia la ejecución del hilo representado por el objeto *CHilo* que reciba este mensaje. Esto es, lanza el hilo.
void * esperar_finalizacion( )	Este método permite, al hilo que lo invoca, esperar a

	que el hilo que recibe el mensaje finalice su ejecución. El valor devuelto hace referencia al valor devuelto por el hilo, o en su defecto al estado del mismo.
string obtener_nombre( )	Es el método que devuelve el nombre del hilo
void asignar_nombre(string)	Es el método que asigna un nombre al hilo
pthread_t obtener_id( )	Es el método que devuelve el identificador del hilo
void asignar_atributos(pthread_attr_t *attr)	Es el método utilizado para asignar los atributos al hilo

Puesto que todo Objeto Paralelo de un *CPAN* (objetos *collector*, *manager* y *stages*) es un proceso o hilo que se ejecuta en paralelo con otros procesos o hilos, éstos deberán heredar de *CHilo* y redefinir la función virtual *fnHilo( ).*

A.2.4. La clase base ComponentCollector

Es una clase concreta que hereda de la clase *CHilo* las características necesarias para definir la estructura del objeto paralelo *collector*, cuya finalidad es almacenar los resultados finales de los cálculos derivados de los objetos esclavos comandados por los *stages* de un *CPAN*.

VARIABLES DE INSTANCIA	DESCRIPCION
vector_sol content	*content* es una variable de tipo *vector_sol* utilizada como un *"vector"* de soluciones. Esta variable es utilizada en exclusión mutua por las funciones *put( )* y *get( )* de un objeto de la clase *ComponentCollector*.

FUNCIONES MIEMBRO	DESCRIPCION
ComponentCollector( )	Es el constructor de la clase que sirve para crear un objeto *collector* como un hilo de ejecución. Inicializa los *mutex* y *variables de condición* de la clase para definir los semáforos que utilizarán los métodos *put()* y *get()* en su sincronización
void put(void *item)	Coloca dentro del vector *content* el conjunto de soluciones finales dado en **item* como una referencia a *void*, utilizando para ello una sincronización con el método *get( )* del tipo productor-consumidor.
void * get( )	Obtiene de *content* el primer elemento del vector retornándolo como una referencia a *void*, utilizando para ello una sincronización con el método *put( )* del tipo productor-consumidor.
void *fnHilo()	Es la función virtual heredada de su clase padre *CHilo* que es redefinida y que será ejecutada al lanzar el hilo *collector* a ejecución. Esta función ejecuta el método *put( )* en paralelo. Un objeto *collector* puede ser accedido a la vez por más de un objeto *stage* de forma que en *fnHilo( )* se establecen las restricciones de sincronización oportunas para el acceso al *collector* como recurso compartido de más de un *stage*, cuando éstos quieren almacenar un dato en él.

A.2.5. La clase base ComponentStage

Es una clase abstracta que hereda de la clase *CHilo* las características necesarias para definir la estructura del objeto paralelo *stage* de un *CPAN*, de donde se derivarán todos los *stages* concretos dependiendo del comportamiento paralelo que se contemple.

VARIABLES DE INSTANCIA	DESCRIPCION
ComponentStage *otherstages[ ]	Variable de instancia definida como un *array* de referencias a objetos *ComponentStage* que contendrá a *stages* asociados a otro *stage* dependiendo del patrón paralelo que se implemente.
bool am_i_last	*am_i_last* será verdadera si la cola de la lista de asociaciones (objeto-esclavo, metodo) esta vacía y será falsa en caso contrario. (Esta variable se utiliza dentro del método *init( )*.
Object *obj	Variable utilizada para almacenar la referencia de un objeto esclavo de una asociación de la lista de asociaciones pasada como argumento en *init( ).*
method *meth	Variable utilizada para almacenar la referencia del método de ejecución del algoritmo de solución del objeto esclavo *obj,* en una asociación de la lista de asociaciones pasada en *init( ).*
void *datas	Variable de instancia que será inicializada en el constructor de la clase y que almacena el conjunto de datos con los que un objeto *stage* trabajará para la solución de un problema.
ComponentCollector *col	Es un puntero que hace referencia al objeto *collector* pasado como argumento en el constructor de la clase y que estará asociado al objeto *stage* que en ese momento se crea.

FUNCIONES MIEMBRO	DESCRIPCION
ComponentStage(void *data, ComponentCollector *colecta)	Es el Constructor de la clase que sirve para crear un objeto *stage* como un hilo de ejecución. Inicializa los *mutex* y variables de condición de la clase para definir los semáforos que utilizará el método *fnHilo( )* en la planificación de procesos mediante el uso del paralelismo máximo. Además instancia sus variables *datas* y *col* con el conjunto de datos a procesar y la referencia al objeto *collector* asociado, que entran como parámetros respectivamente.
void init(asociación *lista)	Es el método de inicialización del objeto *stage* que se crea. Recibe en **lista* la referencia a una lista de asociaciones (objeto-esclavo, método). Obtiene la cabeza de la lista para inicializar las variables *obj y meth*, y utiliza el resto de la lista para inicializar *am_i_last*.
void request(void* datain, ComponentCollector *res, bool band)	El método *request( )* ejecuta en paralelo una petición de servicio comandada por el *manager*. Recibe como entradas los datos a procesar en **datain* y la referencia al objeto *collector* asociado en **res*, además de una bandera *band*, la cual si es verdadera indicará al *stage* corriente que deberá, antes de finalizar su ejecución, esperar la finalización de los *stages* asociados a él. Ejecuta la primitiva *EVAL* para evaluar los datos con el

	algoritmo asociado en **meth* a través del objeto esclavo **obj* y almacena el resultado de *EVAL* en el objeto *collector* utilizando exclusión mutua.
virtual void commandOtherStages(void *dataout)=0	Función virtual a ser definida por el usuario dentro de la clase que herede de *ComponentStage*. Recibe como entradas los datos a procesar, como referencia a *void*. Deberá contener el código que comande a los objetos *stages* almacenados en la variable **otherstages[ ]*, que serán los *stages* asociados a un *stage corriente*.
void * fnHilo( )	Es la función virtual heredada de su clase padre *CHilo* que es redefinida y que será ejecutada al lanzar el hilo *stage* a ejecución. Esta función ejecuta el método *request( )* en paralelo, de forma que la política de planificación empleada dentro de *fnHilo( )*, es la que proporciona la restricción de sincronización de paralelismo máximo aplicado a *request( )*.

A.2.6. La clase base ComponentManager

Es una clase abstracta que hereda de la clase *CHilo* las características necesarias para definir la estructura del objeto paralelo *manager* de un *CPAN*, que habrá que especializar a un manager concreto dependiendo del comportamiento paralelo que se quiera implementar.

VARIABLES DE INSTANCIA	DESCRIPCION
ComponentCollector *res	Puntero que hace referencia al objeto *collector* que se asocia al *manager* que se define.
ComponentStage *stages[ ]	Variable de instancia que contendrá las referencias a los objetos *stages* asociados al *manager* como un arreglo de apuntadores a objetos de tipo *ComponentStage*.
void *datos	Variable de instancia que será inicializada en el constructor de la clase y que almacena el conjunto de datos con los que un objeto *manager* trabajará para la solución de un problema ayudándose de los *stages* asociados a él.

FUNCIONES MIEMBRO	DESCRIPCION
ComponentManager(void *data)	Es el Constructor de la clase que sirve para crear un objeto *manager* como un hilo de ejecución. Inicializa los *mutex* y variables de condición de la clase para definir los semáforos que utilizará el método *fnHilo( )* en la planificación de procesos mediante el uso del paralelismo máximo. Además instancia su variable *datos* con el conjunto de datos a procesar que entra como parámetro e internamente crea el objeto *collector* asociado, almacenando la referencia en su puntero **res*..
virtual void init(asociación *lista)=0	Función abstracta que inicializa un objeto *manager*. Deberá ser implementada por el usuario dentro de la clase que herede de *ComponentManager*. Recibe como entrada una referencia a una lista de asociaciones (objeto-esclavo, método) y deberá crear y asociar los *stages* correspondientes a dicha lista de asociaciones.

void * execution( )	Método que lleva a cabo la ejecución en paralelo de una petición de servicio por parte del usuario. Utiliza su objeto *collector* y comanda a los *stages* definidos en *init( )* mediante el llamado a *commandStages( )* para satisfacer la petición de servicio. Recoge los resultados finales del *collector* y los retorna al usuario como una referencia a *void*.
virtual void commandStages( )=0	Función miembro o método virtual protegido implementado en las clases hijas de *ComponentManager*. Deberá contener el código necesario para comandar en paralelo a los *stages* asociados al manager.
void * fnHilo( )	Es la función virtual heredada de su clase padre *CHilo* que es redefinida y ejecutada al lanzar el hilo *manager* a ejecución. Esta función ejecuta el método *execution( )* en paralelo, de forma que la política de planificación empleada dentro de *fnHilo( )*, es la que proporciona la restricción de sincronización de paralelismo máximo aplicado a *execution( )*.

A.3. EL GRUPO DE LAS CLASES QUE DEFINEN LOS CPANS FARM, PIPE, TREEDV Y FARMBB

CLASES QUE DEFINEN LOS CPANS FARM, PIPE,TREEDV Y FARMBB	
FarmManager	Definen la construcción de un objeto *"manager"* concreto para los patrones *FARM, PIPE, Tree-Divide y Venceras y Farm-Ramificación y poda*, respectivamente. Heredan de la clase base *"ComponentManager"*
PipeManager	
TreeDVManager	
FarmBBManager	
FarmStage,	Definen la construcción de un objeto *"stage"* concreto para los patrones *FARM, PIPE, Tree-Divide y Venceras y Farm-Ramificación y poda*, respectivamente. Heredan de la clase base *"ComponentStage"*
PipeStage	
TreeDVStage	
FarmBBStage	

A.3.1. La clase concreta FarmManager

La clase *FarmManager*, es una subclase de *ComponentManager* que implementa el patrón paralelo de comunicación *FARM*. Cualquier instancia de la clase *FarmManager* será un objeto que representa en sí mismo el *CPAN FARM* ya que en su inicialización se generan los objetos *stages* correspondientes, los comanda y lleva a cabo la ejecución de los objetos esclavos a través de dichos *stages*[27].

VARIABLES DE INSTANCIA	**DESCRIPCION**
int nWorker	Variable que contabiliza el número de *stages* o procesos trabajadores con los que cuenta el *FARM*

[27] La implementación se puede ver en el capítulo III de la presente tesis.

FUNCIONES MIEMBRO	DESCRIPCION
FarmManager(asociación *lista, void *data)	Constructor de la clase *FarmManager* que instancia objetos manager específicos para un *FARM* y que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo) y el conjunto de datos a trabajar como una referencia a *void* que será pasada a su constructor padre, esto es, al constructor de *ComponentManager*.
void init(asociación *lista)	Función que contiene la instrumentación que define la inicialización de un objeto *farmManager*
void commandStages( )	Función que contiene la instrumentación en paralelo que define como un objeto *FarmManager* comandará a sus objetos *stages* en la ejecución de una petición de servicio

A.3.2. La clase concreta FarmStage

La subclase *FarmStage* es derivada de la clase *ComponentStage*. Los *stages* del *FARM* no se conectan unos con otros, por tanto, la operación *CommandOtherStages* carece de cuerpo en su implementación (ver capítulo III).

FUNCIONES MIEMBRO	DESCRIPCION
FarmStage(asociación *lista, void *data, ComponentCollector *colecta)	Constructor de la clase *FarmStage* que instancia objetos *stage* específicos para un *FARM* y que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo), el conjunto de datos con los que va a trabajar como una referencia a *void* y la referencia al objeto *collector* asociado al *stage* que se crea. Los dos últimos parámetros son enviados como argumentos al constructor padre, esto es, al constructor de *ComponentStage*.
void commandOtherStages(void *dataout)	Función que contiene la instrumentación en paralelo que define como un objeto *FarmStage* comandará a los objetos *stages* que estén relacionados con él. En este caso en particular, la función carece de cuerpo (ver capítulo III para más detalles).

A.3.3. La clase concreta PipeManager

La clase *PipeManager*, es subclase de la clase *ComponentManager* e implementa el patrón de comunicación *PIPELINE* de forma paralela. Cualquier instancia de la clase *PipeManager* será un objeto que representa en sí mismo el *CPAN PIPELINE* ya que en su inicialización se generan los objetos *stages* correspondientes, los comanda y lleva a cabo la ejecución de los objetos esclavos a través de dichos objetos.

FUNCIONES MIEMBRO	DESCRIPCION
PipeManager(asociación *lista, void *data)	Constructor de la clase *PipeManager* que instancia objetos *manager* específicos para un *PIPELINE* que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo) y el conjunto de datos con los que trabajará, como una referencia a *void* que será pasada a su constructor padre, esto es, al constructor de *ComponentManager*.
virtual void init(asociación *lista)	Función que contiene la instrumentación que define la inicialización de un objeto *PipeManager*
void commandStages( )	Función que contiene la instrumentación en paralelo que define como un objeto *PipeManager* comandará a sus objetos *stages* en la ejecución de una petición de servicio

A.3.4. La clase concreta PipeStage

La clase *PipeStage*, deriva de la clase *ComponentStage*, es decir, hereda de ésta última. Cualquier instancia de *PipeStage* es encargada de crear el siguiente *stage* del *pipeline* durante la fase de inicialización. Pero en su fase de ejecución cualquier objeto *stage* comanda al siguiente, el último objeto *PipeStage* envía el resultado a una instancia de la clase *ComponentCollector*[28].

FUNCIONES MIEMBRO	DESCRIPCION
PipeStage(asociación *lista, void *data, ComponentCollecto *colecta)	Constructor de la clase *PipeStage* que instancia objetos *stage* específicos para un *PIPELINE* que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo), los datos a procesar como una referencia a *void,* y una referencia al objeto *collector* asociado al *stage* que se crea. Los dos últimos parámetros son enviados como argumentos al constructor padre, esto es, al constructor de *ComponentStage*.
void init(asociación *lista)	Función que redefine la función *init( )* de la clase padre *ComponentStage* para inicializar un objeto *PipeStage*
void commandOtherStages(void *dataout)	Función que contiene la instrumentación en paralelo que define como un objeto *PipeStage* comandará a los objetos *stages* que están relacionados con él de acuerdo con el modelo del *PIPELINE*. Recibe como entrada el conjunto de datos con los que trabajará esta función, a través de una referencia a *void*.

A.3.5. La clase concreta TreeDVManager

La clase *TreeDVManager*, es subclase de la clase *ComponentManager* e implementa el patrón de comunicación *Tree-Divide_y_Vencerás* de forma paralela.

[28] Para más detalles remitirse al capítulo III.

Cualquier instancia de la clase *TreeDVManager* es un objeto que representa en sí mismo el *Cpan TreeDV* ya que en su inicialización se genera el objeto *stage* correspondiente al nodo raíz del árbol que se utilizará para la solución del problema, comanda el nodo *stage* raíz y lleva a cabo la ejecución del objeto esclavo asociado a dicho nodo (su implementación puede observarse en el capítulo III).

FUNCIONES MIEMBRO	DESCRIPCION
TreeDVManager(asociación *lista, void *data)	Constructor de la clase *TreeDVManager* que instancia objetos manager específicos para un *TREE-Divide_y_Vencerás* que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo) y el conjunto de datos a procesar como una referencia a *void* que será pasada a su constructor padre, esto es, al constructor de *ComponentManager*.
void init(asociación *lista)	Función que contiene la instrumentación que define la inicialización de un objeto *TreeDVManager*
void commandStages( )	Función que contiene la instrumentación en paralelo que define como un objeto *TreeDVManager* comandará a sus objetos *stages* en la ejecución de una petición de servicio. En éste caso particular se comandará al *stage* o nodo raíz del árbol binario que se genere.

A.3.6. La clase concreta TreeDVStage

La clase *TreeDVStage*, hereda de la clase *ComponentStage*. Cualquier instancia de *TreeDVStage* esta encargada de crear dos *stages* hijos o dependientes del *stage* corriente y comandarlos en su fase de ejecución para que resuelvan cada uno de ellos una parte del problema. El árbol binario se va generando por niveles, ejecutándose en paralelo de manera recursiva para que en el *backtraking* se retornen las soluciones parciales al *stage* raíz quien enviará la solución final a una instancia de la clase *ComponentCollector* (los detalles de la implementación se encuentran en el capítulo III).

VARIABLES DE INSTANCIA	DESCRIPCION
asociación *lista	Variable de instancia utilizada para almacenar la lista de asociaciones que es pasada como argumento en el constructor de la clase

FUNCIONES MIEMBRO	DESCRIPCION
TreeDVStage(asociación *lista, void *data, ComponentCollector *colecta)	Constructor de la clase *TreeDVStage* que instancia objetos *stage* específicos para un *Tree-Divide_y_vencerás* y que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo), el conjunto de datos a procesar como una referencia a *void*, y la referencia al objeto *collector* asociado al *stage* que se

	crea. Los dos últimos parámetros son enviados como argumentos al constructor padre, esto es, al constructor de *ComponentStage*.
void init(asociación *lista)	Función que redefine la función *init( )* de la clase padre *ComponentStage* para inicializar un objeto *TreeDVStage*
void commandOtherStages(void *dataout)=0	Función que contiene la instrumentación en paralelo que define como un objeto *TreeDVStage* comandará a los objetos *stages* que están relacionados con él de acuerdo con el modelo del *Tree-Divide_y_vencerás*, en este caso, comandará al *stage* hijo izquierdo y derecho del subárbol binario que se esté creando.
void *dv(void *dataout, int inicio, int fin)	Función utilizada para hacer la división de un problema o conjunto de datos en dos sub-problemas o sub-conjunto de datos. Recibe como entrada una referencia a los datos que se procesan como una referencia a *void*, y dos índices de inicio y fin de la lista de datos para poder hacer la división.

A.3.7. La clase concreta FarmBBManager

La clase *FarmBBManager*, es una subclase de *FarmManager*, la cual implementa el patrón paralelo de comunicación *FARM* para la técnica de *Ramificación y Acotación*. Cualquier instancia de la clase *FarmBBManager* será un objeto que representa la técnica.

FUNCIONES MIEMBRO	DESCRIPCION
FarmBBManager(asociación *lista, void *data)	Constructor de la clase *FarmBBManager* que instancia objetos manager específicos para un *FARM Branch&Bound* y que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo) y un objeto *Nodo* como una referencia a *void* que será pasado a su constructor padre, esto es, al constructor de *FarmManager*.
void commandStages( )	Función que contiene la instrumentación en paralelo que define como un objeto *FarmBBManager* comandará a sus objetos *stages* en la ejecución de una petición de servicio. En este caso, el objeto *FarmBBManager* esperará al último resultado de sus *stages* para obtener un conjunto de soluciones posibles y no sólo al primero como sucede en el *FarmManager*.

A.3.8. La clase concreta FarmBBStage

La subclase *FarmBBStage* es derivada de la clase *FarmStage*. En este caso, los objetos *stageBB* que se generen si se conectan unos con otros para formar el árbol de expansión de la técnica de Ramificación y Poda a medida que se va resolviendo el problema, ramificando ramas por un lado y podando otras por otro. Como cada *stageBB*

comanda a otros en la ejecución del cómputo relativo a la *ramificación*, se hace necesario ofrecer una implementación del método *commandOtherStages( )* heredado[29].

VARIABLES DE INSTANCIA	DESCRIPCION
asociación *lista	Variable de instancia utilizada para almacenar la lista de asociaciones que es pasada como argumento en el constructor de la clase dentro de un objeto *stage*.
Nodo **nh	Es un doble apuntador de tipo Nodo que representa el conjunto de nodos hijos que un objeto Nodo actual (representado por un stage de tipo *FarmBBStage*) tiene a su cargo, y que juntos: nodos hijos y nodo (padre) actual, representan el subárbol de expansión que en ese momento el *stage actual* esta procesando.
static int cota	Es una variable compartida por todos los objetos *stage* de tipo *FarmBBStage* que representa la cota superior calculada y que sirve para la poda de nodos del árbol de expansión que se genera.
Multimap estructura	Es un multimap que representa la estructura donde se almacenan los nodos del árbol de expansión próximos a ser analizados. Dicha estructura trabaja como un *montículo o heap*.

FUNCIONES MIEMBRO	DESCRIPCION
FarmBBStage(asociación *lista, void *data, ComponentCollector *colecta)	Constructor de la clase *FarmBBStage* que instancia objetos *stage* específicos para implementar la técnica de *Ramificación y Poda* y que recibe como entrada una lista de asociaciones (metodo,objeto_esclavo), el *Nodo* activo a procesar como una referencia a *void*, y la referencia al objeto *collector* asociado al *stage* que se crea. Estos parámetros son enviados como argumentos al constructor padre, esto es, al constructor de *FarmStage*.
void init(asociación *lista)	Función que redefine la función *init( )* de la clase padre *FarmStage* para inicializar un objeto *FarmBBStage*. Llama a la función *init()* de *FarmStage* y pone la variable *am_i_last* (heredada) a false
void commandOtherStages(void *dataout)	Función que contiene la instrumentación en paralelo que define como un objeto *FarmBBStage* comandará a los objetos *stages* que están relacionados con él de acuerdo con la técnica de *Ramificación y Poda*, en este caso, comandará a los *stages* hijos del subárbol de expansión que se esté generando. Esta función hace uso de las funciones *aceptable( ), esSolucion( ) y poner_cota_sup( )* para la implementación de la técnica en cuestión.
bool aceptable(Nodo *n)	Es la función que realiza la poda. Dado un nodo vivo en su parámetro ***n***, decide si es factible seguir analizándolo o bien rechazarlo.
bool esSolucion(Nodo *n)	Es la función que decide cuando el nodo que entra en su parámetro ***n***, es una hoja del árbol, es decir, una posible solución al problema original.
void poner_cota_sup(int numhijos)	Función que establece la cota superior del problema en base al número de hijos nodo pasado como argumento y que tiene asignados un *stage*. El proceso de poda utiliza esta función para podar aquellos nodos cuyo valor sea superior a la cota que ya se obtuvo de una solución.
void imprime_cota_sup()	Función que imprime a pantalla el valor de la cota

[29] Para más detalles sobre la implementación del Cpan FarmBB remítase al capítulo IV.

	superior compartida por todos los objetos *stage*.

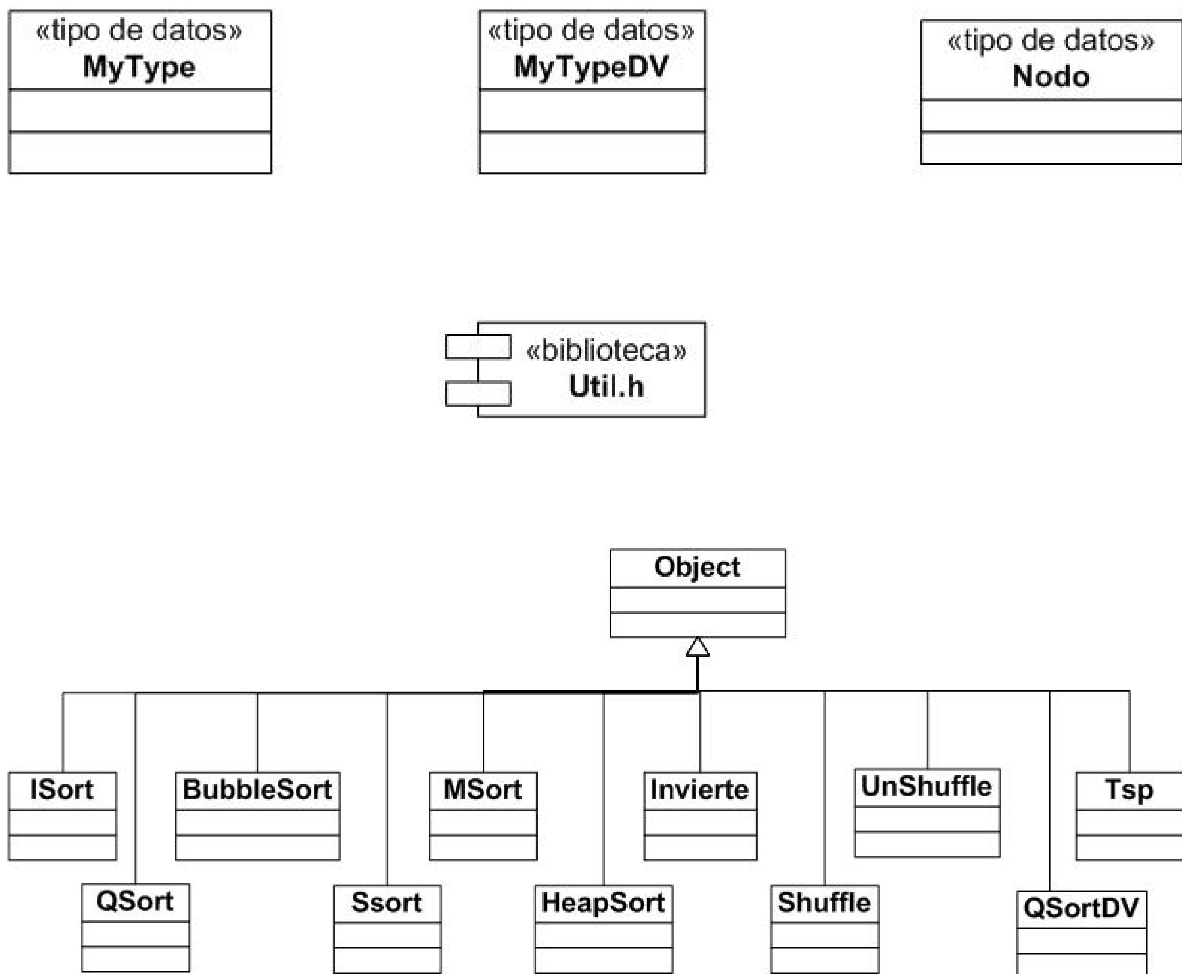
A.4. EL GRUPO DE LAS CLASES DE LOS OBJETOS ESCLAVOS

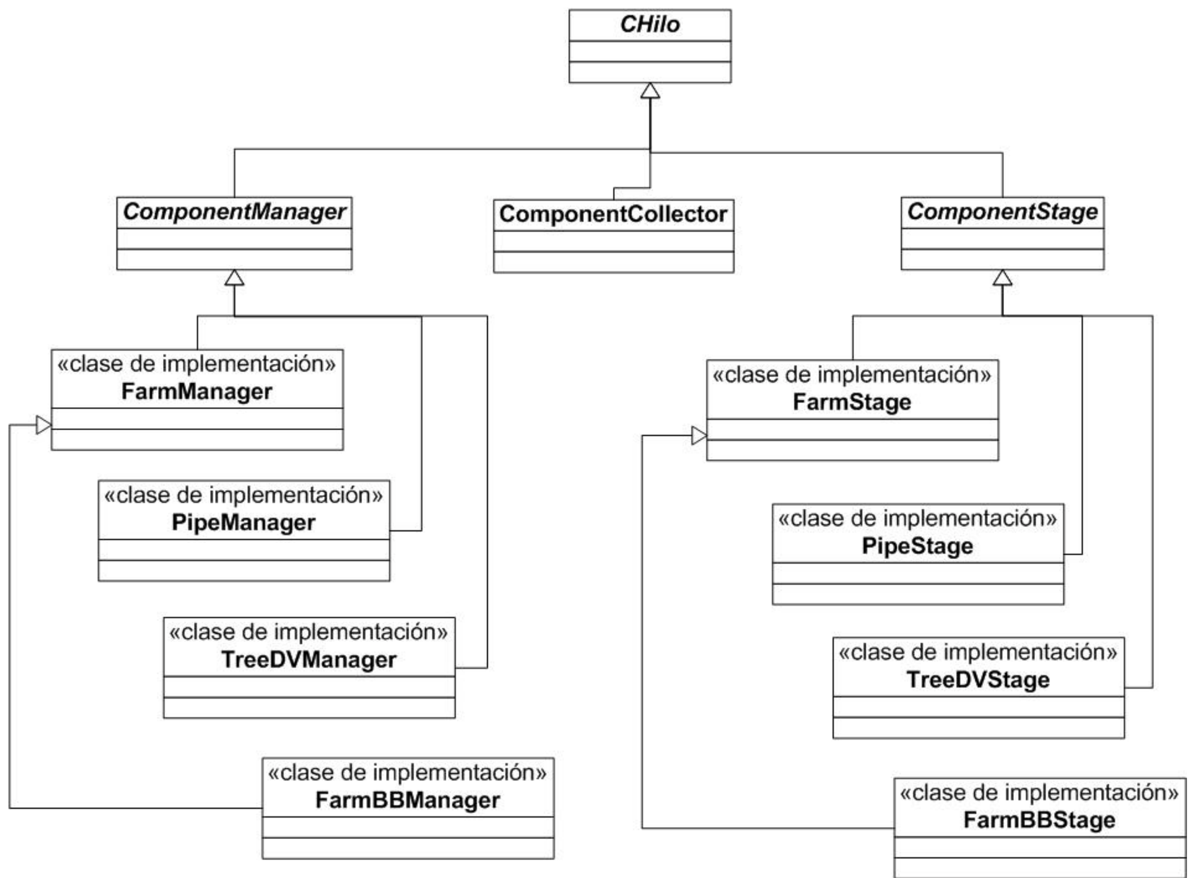
CLASES "EJEMPLO" DE LOS TIPOS ABSTRACTOS DE DATOS UTILIZADOS EN LOS CPANS	
MyType	Clase definida por el usuario donde se establece el tipo de datos enteros iniciales a procesar así como la estructura de almacenamiento de estos. Esta clase es utilizada para proporcionar el tipo abstracto de datos así como la su estructura de almacenamiento. Elementos que serán utilizados por los *CPAN Farm y Pipe* en su prueba de ejecución .
MyTypeDV	Clase definida por el usuario donde se establece el tipo de datos enteros iniciales a procesar así como el almacenamiento de estos dentro del *CPAN TreeDV* en su prueba de ejecución .
Nodo	Clase definida por el usuario donde se establece el tipo de datos a procesar por un *Cpan FarmBB*: En este caso el tipo *Nodo*, formado de variables que almacenan datos como por ejemplo, el costo acumulado de un nodo, su matriz de costos reducida, el nivel donde se encuentra en el árbol de expansión y el vector de solución que muestra el camino de costo mínimo calculado en ese nodo. Además cuenta con funciones miembro que calculan: la reducción de una matriz de costos, y funciones que imprimen los datos anteriormente citados.
CLASES "EJEMPLO" DE LOS OBJETOS ESCLAVOS UTILIZADOS EN LOS CPANS	
Isort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación por inserción,* implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos a los *CPAN Farm y Pipe* en su prueba de ejecución .
QSort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación rápida,* implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos a los *CPAN Farm y Pipe* en su prueba de ejecución .
Bubblesort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación por el método de la burbuja,* implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos a los *CPAN Farm y Pipe* en su prueba de ejecución .
Ssort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación ShellSort* implementado en el método *resuelve( ),* que es redefinido en este nivel. La clase se utilizó para proporcionar objetos esclavos a los CPAN Farm y Pipe en su prueba de ejecución .
Msort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación por Mezcla* implementado en el método *resuelve( ),* que es redefinido en este nivel. La clase se utilizó para proporcionar objetos esclavos a los CPAN Farm y Pipe en su prueba de ejecución .
HeapSort	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un conjunto de datos en desorden usando el algoritmo de *ordenación por Montículo* implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos a los *CPAN Farm y Pipe* en su prueba de ejecución .
Invierte	Clase definida por el usuario, en este caso para crear objetos esclavo que tienen la función de invertir el orden un conjunto de datos. Esta clase hereda de Object e implementa el método "resuelve( )" que es donde se soluciona el problema de invertir los datos. La clase fue utilizada para proporcionar objetos esclavos al *CPAN Pipe* en su prueba de ejecución .
Shuffle	Clase definida por el usuario, en este caso para crear objetos esclavo que tienen la función de intercalar el orden un conjunto de datos. Esta clase hereda de Object e implementa el método "resuelve( )" que es donde se implementa el algoritmo de intercalación de datos. La clase fue utilizada para proporcionar objetos esclavos al *CPAN Pipe* en su prueba de ejecución .
UnShuffle	Clase definida por el usuario, en este caso para crear objetos esclavo que tienen la función de des-intercalar el orden un conjunto de datos. Esta clase hereda de Object e implementa el método "resuelve( )" que es donde se implementa el algoritmo de des-intercalación de datos. La clase fue utilizada para proporcionar objetos esclavos al *CPAN Pipe* en su prueba de ejecución .
QsortDV	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que ordenan un

	conjunto de datos en desorden usando el algoritmo de *ordenación rápida,* implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos al *CPAN TreeDV* en su prueba de ejecución .
Tsp	Clase que hereda de *Object* y que es utilizada para crear objetos esclavos que resuelven el problema del agente viajero implementado en el método *resuelve( ).* La clase se utilizó para proporcionar objetos esclavos al *CPAN FarmBB* en su prueba de ejecución .

A.5. JERARQUÍA DE HERENCIA DE LA LIBRERÍA DE LOS CPANS

La jerarquía de clases que a continuación se presenta, constituye la librería de los *CPANS* propuestos, proporcionando con ello una gran ventaja al programador ya que dado que se esta trabajando con un lenguaje orientado a objetos, el programador tiene la posibilidad de, mediante la herencia, simplificar la definición de nuevas Composiciones Paralelas o *CPANS* haciendo uso de la jerarquía inicial definida en la librería de clases.




CHilo
ComponentManager
ComponentCollector
ComponentStage
«clase de implementación»
FarmManager
«clase de implementación»
PipeManager
«clase de implementación»
TreeDVManager
«clase de implementación»
FarmBBManager
«clase de implementación»
FarmStage
«clase de implementación»
PipeStage
«clase de implementación»
TreeDVStage
«clase de implementación»
FarmBBStage

Apéndice B

SISTEMA SGI ORIGIN 2000: El Sistema Paralelo del Centro Europeo de Paralelismo de Barcelona

B.1. INTRODUCCIÓN

El Centro Europeo de Paralelismo de Barcelona, *CEPBA*, ofrece a sus usuarios el sistema paralelo *Origin 2000 Silicon Graphics*. Éste sistema es el que se utilizó en la ejecución y análisis del rendimiento de los *CPANS*.

El *Sistema Origin 2000 Silicon Graphics* se define como un multiprocesador de alto rendimiento, basado en servidores multiproceso con *memoria compartida distribuida*. Este tipo de servidores tiene una arquitectura S2MP que es altamente flexible, modular y de muy bajo costo, que soporta un gran ancho de banda, Fig. B.1.

Fig. B.1. El Sistema Paralelo SGI Origin 2000

El *Origin 2000* es un multiprocesador de alta modularidad con una latencia muy baja, con escalabilidad de 1 a 128 procesadores distribuidos en nodos de 2 procesadores con un ruteador y una red de interconexión.

El CEPBA cuenta actualmente con 64 procesadores *MIPS R10000*. Estos procesadores son de propósito general y de muy alto rendimiento. La disposición de los procesadores es simétrica en base al modelo S2MP, es decir, un nodo se compone de 2 procesadores y dentro del nodo se encuentra una parte de la memoria distribuida del sistema que puede ser accedida por cualquier procesador a través de una red de interconexión modular *crossbar*. Fig. B.2.

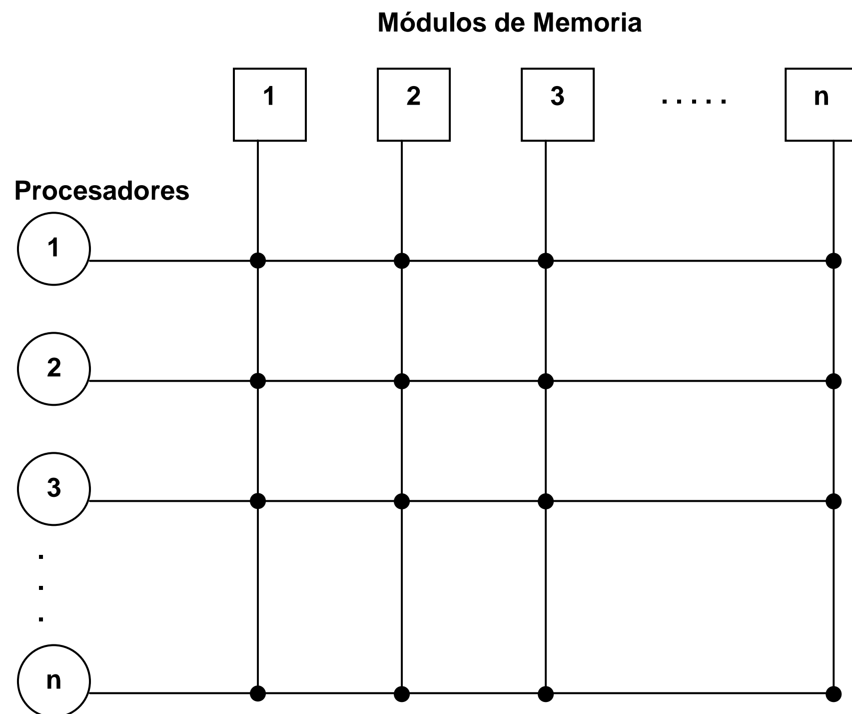


Fig. B.2. Red de Interconexión CrossBar del Sistema Origin 2000

Los nodos del sistema están interconectados a través de una red de interconexión formada de ruteadores, tal como lo muestra la fig. B.3.

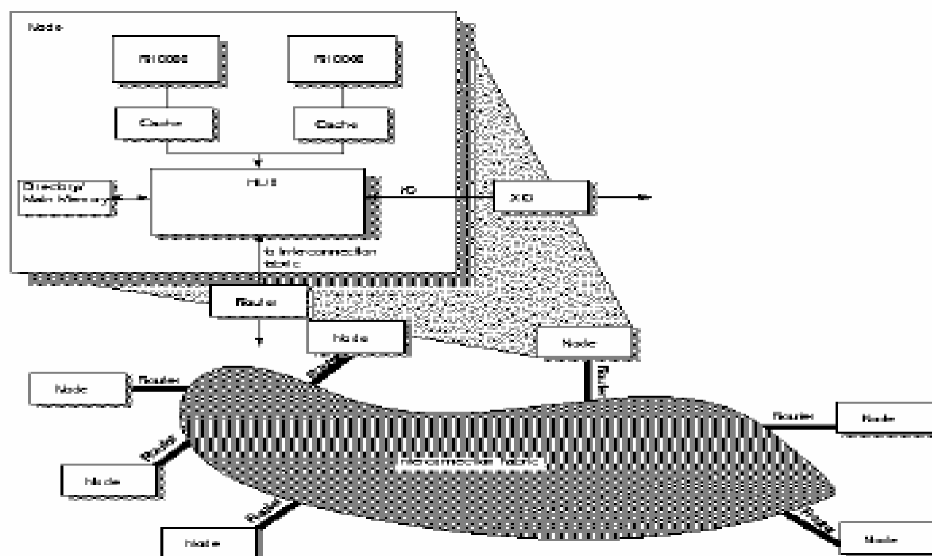


Fig. B.3. Esquema del Sistema Origin 2000

La topología de interconexión de los nodos del *Origin 2000* es un hipercubo, tal como se muestra en la fig. B.4.

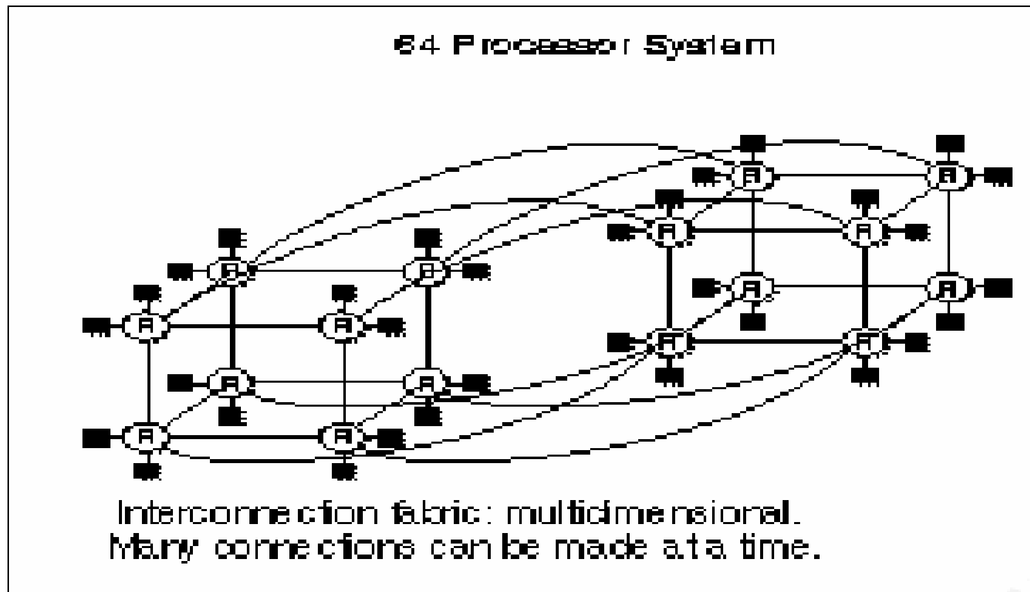


Fig. B.4. Topología de Interconexión de los procesadores del Origin 2000

Esta topología tiene una serie de ventajas:

- Modularidad
- Más de un camino para ir a memoria.
- Tolerancia a fallos.
- Latencia proporcional al logaritmo del número de nodos.
- Ancho de banda que crece en forma proporcional al número de nodos.

B.2. LOS PROCESADORES MIPS R10000

Cada nodo del *Origin 2000* soporta uno o dos procesadores *MIPS R10000*, ver Fig. B.5., de forma que cada procesador esta montado en un *HIMM* (*H*orizontal *I*n-line *M*emory *M*odule) con *4 MB* de memoria caché de segundo nivel para cada procesador. Las características principales de los procesadores *MIPS R10000* son las siguientes:

- Procesador superescalar de 64 bits
- 5 unidades de ejecución separadas

- caches de mapeo 2-asociativas no bloqueantes
- frecuencia de reloj de 250 MHz

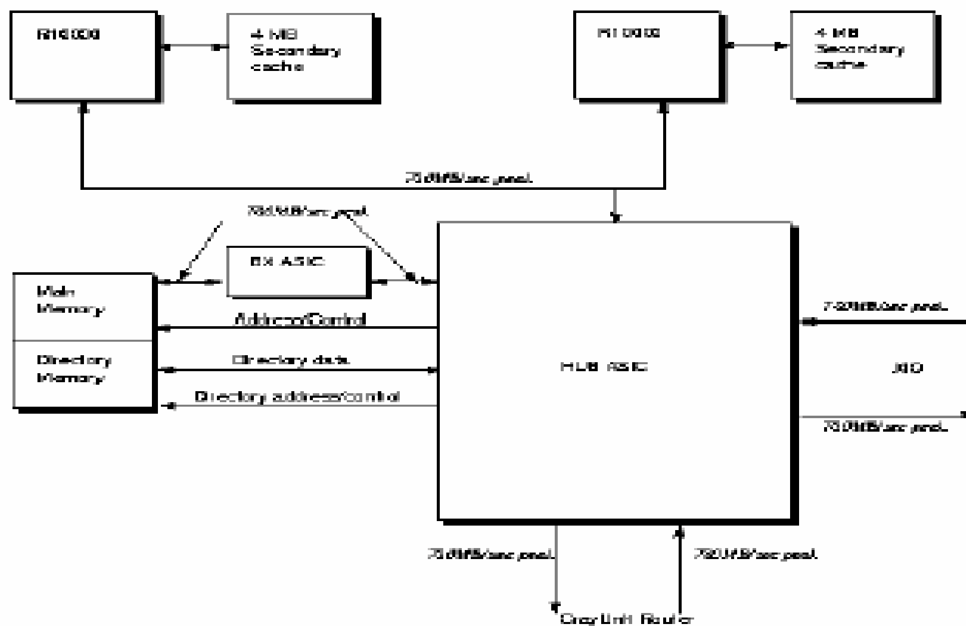


Fig. B.5. Esquema del Procesador MIPS R10000

B.3. LA MEMORIA CACHE DEL ORIGIN 2000

La memoria caché es una memoria pequeña y rápida que se asocia al procesador. Se utiliza para aprovechar la localidad referencial de los datos para acceder a la información de manera rápida.

Dentro del *Origin 2000* se disponen de 2 niveles de memoria cache: la cache primaria, dentro del procesador, y la cache secundaria fuera de éste pero asociada a él, es decir, sólo utilizable desde el procesador. Siempre que se referencia la cache del procesador se está hablando de la cache secundaria o externa.

Como ya se mencionó, el *Origin 2000* utiliza un sistema de memoria compartida distribuida (*DSM, Distributed Shared Memory*) para la memoria principal, que se particiona entre los nodos del procesador (un nodo por cada dos procesadores), pero que puede ser accedida por todos los procesadores. Las características de la cache secundaria son:

- 4 MB de cache secundaria en cada procesador, dispuesta en dos bancos de 2 MB
- Mapeo 2-asociativo

- Capaz de leer 8 palabras (32 bits) y transferirlas durante 2 ciclos consecutivos
- Tamaño de línea de 128 bytes (16 elementos en doble precisión, o 32 de precisión simple)

Las características de la cache primaria son:

- Mapeo 2-asociativo
- Subdividida en 2 memorias cache dentro del *R10000*
- Tamaño de línea de cache de 64 bytes (8 elementos de doble precisión)
- Cache primaria de instrucciones
- Cache primaria de datos

B.4. EL SISTEMA OPERATIVO DEL ORIGIN 2000

El sistema operativo del *Origin 2000* es un *UNIX IRIX 6.5* de 64 bits que soporta la mayoría de las plataformas estándares de la industria como son:

- Motif™ 1.2.4
- X11R6
- POSIX 1003.1/1003.2
- FIPS 151.2
- Display PostScript®
- Tooltalk
- Triteal CDE (through Triteal)
- AppleTalk® (through Xinet)
- XFS®
- NFS™
- QuickTime™
- Cinepak®
- JPEG
- MPEG 1
- AVI file formats

En particular el sistema *Origin 2000* del CEPBA ofrece a sus usuarios y programadores diversas utilidades como son:

- Compiladores de fortran
  - GNU proyect Fortran Compiler V0.5.24
  - MIPSpro F77 compiler V7.30
  - MIPSpro Fortran 90 compiler V7.30
- Compiladores de C y C++
  - GNU proyect C and C++ Compiler V2.95
  - MIPSpro (MIPS C, MIPSpro C, MIPSpro C++) compilers V7.30
- Utilidades varias
  - *consum*: muestra un reporte del consumo
  - *gdb:* GNU debugger
  - *ipcclean:* Limpia la memoria compartida y los semáforos
  - *nedit:* Editor multi propósitos para X Windows

Bibliografía y Referencias

BIBLIOGRAFIA Y REFERENCIAS

[AHO74] Aho Ullman, Hopcroft. "The Design and Analysis of Computer Algorithms", Addison-Wesley, 61-65. 1974.

[AND96] Andreoli Jean-Marc, Pareschi Remo. "Integrated Computational Paradigms for flexible Client-Server Communication". ACM Computing Surveys, Vol. 28, No.2, pp.297-299. 1996.

[AND83] Andrews G. R., Schneider F. B. "Concepts and Notations for Concurrent Programming". Computing Surveys, Vol. 15, No. 1, pp. 3-43. 1983.

[AND91] Andrews, G. R. "Paradigms for Process interaction in Distributed Programs". ACM Computing Surveys, Vol. 15, No. 1, 49-90. 1991.

[AND91] Andrews, G. R. "Concurrent Programming. Principles and Practice". The Cummings Publishing. 1991.

[ARA96] Araque F., Capel M., Palma A. "Paradigms for Parallel Distributed Programming". International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'96), Sunnyvale (CA). 1996.

[ARJ96] Arjomandi E., O´Farrell W.G., Wilson G. V. "An Object-Oriented Communication Mechanism for Parallel Systems". Conference on Object-Oriente Technologies. Toronto, Ontario, Canada, 1996. USENIX http://www.usenix.org

[BAC99] Bacci, Danelutto, Pelagatti, Vaneschi. "SklE: A Heterogeneous Environment for HPC Applications". Parallel Computing 25, pp. 1827-52. 1999.

[BEL01] Bello L. P., DeIta L.G., "Diseño de Heurísticas para resolver el problema del Agente Viajero". Tesis de Licenciatura. Proyecto Financiado por CONACYT, con número de registro 28025-A. Puebla, México 2001.

[BIR89] Birrell Andrew. "An Introduction to programming with threads". Digital Equipment Corporation. Systems Research Center. 1989.

[BLE96] Blelloch G. E. "Programming Parallel Algorithms". Comunications of the ACM. Volume 39, Number 3, 85-97, 1996.

[BRA98] Brassard G., Bratley P. "Fundamentos de Algoritmia", Prentice-Hall. Madrid. 1998.

[BRI93] Brinch Hansen. "Model Programs for Computational Science: A programming methodology for multicomputers", Concurrency: Practice and Experience, Vol. 5, No. 5, Pp. 407-423. 1993.

[BRI94] Brinch Hansen. "SuperPascal- a publication language for parallel scientific computing", Concurrency: Practice and Experience, Vol. 6, No. 5, Pp: 461-483. 1994.

[BUT97] Butenhof, D. R. "Programming with POSIX® Threads". Addison Wesley. 1997.

[CAP92] Capel M., Palma A., "A Programming tool for Distributed Implementation of Branch-and-Bound Algorithms". Parallel Computing and Transputer Applications. IOS Press/CIMNE. Barcelona 1992.

[CAP94] Capel M., Troya J. M. "An Object-Based Tool and Methodological Approach for Distributed Programming". Software Concepts and Tools, 15, pp. 177-195. 1994.

[CEB03] Ceballos F.J. "Programación Orientada a Objetos con C++". Editorial RA-MA. Madrid, 2003.

[CEPBA] CEPBA. Grupo de Soporte a la Paralelización. "Ejemplos de Paralelización con Origin 2000 Silicon Graphics y DEC Alpha 8400". http://www.cepba.upc.es/geninf_i.htm

[CHE95] Chen S, Eshaghian M. M., et. al. "Evaluation of two programming paradigms for heterogeneous computing". Journal of Parallel and Distributed Computing 31, pp: 41-55. 1995.

[CHI91] Chin R. S., Chanson S. T. "Distributed Object-Based Programming Systems". ACM Computing Surveys. Vol. 23, No. 1. 1991.

[COF] Coffin Stephen. "UNIX Sistema V. Versión 4". Manual de Referencia. Mc-Graw Hill.

[COL89] Cole M. "Algorithmic Skeletons: Structured Managment of Parallel Computation". The MIT Press. 1989.

[COR91] Corradi A., Leonardi L. "PO Constraints as tools to synchronize active objects". Journal Object Oriented Programming 10, pp: 42-53. 1991.

[COR95] Corradi A, Leonardo L, Zambonelli F. "Experiences toward an Object-Oriented Approach to Structured Parallel Programming". DEIS technical report no. DEIS-LIA-95-007. 1995

[DAN] Danelutto M., Orlando S, et al. "Parallel Programming Models Based on Restricted Computation Structure Approach". Technical Report-Dpt. Informatica. Universitá de Pisa.

[DAR93] Darlington et al. "Parallel Programming Using Skeleton Functions". Proceedings PARLE'93, Munich (D). 1993.

[DEI97] Deitel & Deitel. “C++ How to program” fourth Edition. Prentice Hall. 1999.

[DES97] De Simone, et al. “Design Patterns for Parallel Programming”. PDPTA’96 International Conference, pp: 231-240. 1997.

[ECK99] Eckel Bruce. “Thinking in C++” 2nd edition. VERSION TICA18, Prentice Hall Upper Saddle River, New Jersey 07458. http://www.phptr.com. 1999

[GAM02] Gamma, Erich, et al. “Designs Patterns”. Addison Wesley. 2002.

[GEH88] Gehani N.H., Roome W. D. “Concurrent C++: Concurrent Programming with Class”, Software-Practice and Experience, Vol. 18, pp: 1157-1177. 1988.

[GOO77] GoodMan S.E., Hedetniemi S.T. “Introduction to the Design and Analysis of Algorithms. Mc Graw Hill Book Company. United States of America. 1977.

[GUE99] Guerrequeta G.R., Vallecillo M.A. “Técnicas de Diseño de Algoritmos”. Manuales. Universidad de Málaga.1999.

[HAR98] Hartley Stephen J. “Concurrent Programming. The JAVA Programming Language”, New York, Oxford University Press. 1998.

[HOR78] Horowitz Sahni. “Fundamentals of Computer Algorithms”. Ed. Computer Sc. Press. 1978.

[HOR03] Horstmann Cay. “Computing Concepts with C++ Essentials”. Third Edition. John Wiley. 2003.

[KER] Kernigan B. W. “El entorno de programación UNIX”. Prentice Hall.

[KIN88] Kindervater G.A.P., Trienekens H.W.J.M. “Experiments with Parallel Algorithms for Combinatorial Problems”. European Journal of Operational Research, Vol. 33, pp: 65-81. 1988.

[KLE96] Kleiman S., Shah D., Smaalders B. “Programming with Threads”. Prentice Hall. 1996.

[KUM94] Kumar V., et al. “Introduction to parallel computing. Design and Analysis of Algorithms”, USA, Benjamin-Cummings. 1994.

[KUN89] Kung H.T. “Computational Models for Parallel Computers”. Scientific Applications of Multiprocessors. Prentice-Hall. 1989.

[LAV] Lavander G. R., Kafura D. G. “A Polymorphic Future and First-class Function Type for Concurrent Object-Oriented Programming”. Journal of Object-Oriented Systems. http://www.cs.utewxas.edu-users/lavender/papers

[LEA96] Lea Doug. "Concurrent Programming in JAVA. Design, Principles and Patterns", Addison-Wesley. 1996.

[LIE87] Lieberman, Henry. "Concurrent Object-Oriented Programming in Act 1". Object-Oriented Concurrent Programming. MIT Press, 1987. http://lieber.www.media.mit.edu/people/lieber/Lieberary/OOP/Act-1/Concurrent-OOP-in-Act-1

[MAR99] Martínez F., Capel M., Sánchez P.J. "Distributed Objects for Programming Multicomputers". LSI technical report. 1999.

[PAR] Pratt T. W., Zelkowitz M. V. "Lenguajes de Programación. Diseño e Implementación". Prentice Hall.

[PUL01] Puliafito A., Riccobene S., Scarpa M. "Which paradigm should I use? An analytical comparison of the client-server; remote evaluation and mobile agent paradigms". Concurrency and Computation: Practice and Experience. 13, pp: 71-94. 2001.

[RAB95] Rabhi, Fethi. "A Parallel Programming Methodology based on Paradigms"; In Transputer and Occam Development (18th WoTUG Technical Meeting), P. Nixon IOS Press, 239-249. 1995.

[ROB] Robbins, K. A., Robbins S. "UNIX Programación Práctica. Guía para la concurrencia, la comunicación y los multihilos". Prentice Hall.

[ROS99] Rossainz M. "Una Metodología de Programación Paralela en Java". Technical Report-Dpt. Lenguajes y Sistemas Informáticos. Universidad de Granada. 1999.

[ROS04] Rossainz M., Capel M. "A Parallel Programming Methodology based on High Level Parallel Compositions (CPANs)". Procedings of XIV International Conference on Electronics, Communications, and Computers, CONIELECOMP 2004. ISBN 0-7695-2074-X.

[SAN98] Sanz A. J. "Utilización de Threads", Sólo Programadores. Año 5, No. 48, pp: 56-62. 1998.

[SAR02] Sarwar S. M., Koretsky R., Sarwar S. A. "El libro de LINUX". Addison Wesley. 2002.

[SCH95] Schildt H. "C++ Manual de Referencia". Osborne-Mc Graw Hill. 1995.

[SED98] Sedgewick R. "Algorithms", Addison-Wesley Publishing Company, Second Edition. 1998.

[SEL99] Seller Roosta. "Parallel Processing and Parallel Algorithms. Theory and Computation". Springer. 1999.

[STA02] Stallman R. “Using the GNU Compiler Collection”. Free Software Foundation. 2002.

[STR99] Stroustrup B. “A brief look at C++”. AT&T Bell Laboratories. New Jersey. 1999.

[STR00] Stroustrup B. “The C++ Programming Languaje”. Special Edition. Addison Wesley. 2000.

[SUN95] Sun Microsystems. “POSIX.1c/D10 Summary”. SunSoft Inc. California. 1995.

[TAN92] Tanenbaum A. S., Bal H. E., Kaashoek F., “ORCA: An Language for Parallel Programming of Distributed Systems”. IEE Transactions on Software Engineering, Vol. 18, No. 3. 1992.

[TAU99] Taura K., Tabata K., Yonezawa A. “Stack Threads/MP: Integrating Futures into calling standars”. Technical Report. Department of Information Science Faculty of Science, University of Tokio. Japan 1999.

[WAR95] Warner T., Towsley D. “Getting Started with POSIX Threads”. Department of Computing Science. MIT, Press. 1995

[WIL99] Wilkinson B., Allen M. “Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers”. Prentice-Hall. U.S.A. 1999.