Universidad Nacional del Sur

Tesis de Magister en Ciencias de la Computación

Paradigmas de Programación en Paralelo: Paralelismo Explícito y Paralelismo Implícito

Ing. Natalia Luz Weinbach

Bahía Blanca Argentina

2008

UNIVERSIDAD NACIONAL DEL SUR

TESIS DE MAGISTER EN CIENCIAS DE LA COMPUTACIÓN

Paradigmas de Programación en Paralelo: Paralelismo Explícito y Paralelismo Implícito

Ing. Natalia Luz Weinbach

BAHÍA BLANCA ARGENTINA

2008

Prefacio

Esta Tesis se presenta como parte de los requisitos para optar al grado Académico de Magister en Ciencias de la Computación, de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones llevadas a cabo en el ámbito del Departamento de Ciencias e Ingeniería de la Computación, durante el período comprendido entre el 18 de mayo de 2004 y el 8 de octubre de 2008, bajo la dirección del Dr. Alejandro J. García.

Natalia Luz Weinbach
nweinbach@hotmail.com
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Bahía Blanca, 2 de Octubre de 2008.

UNIVERSIDAD NACIONAL DEL SUR
Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el .../.../..., mereciendo la calificación de ... (. . . . .).

Agradecimientos

Esta Tesis tiene los condimentos más variados. En paralelo ;) a lo largo de su desarrollo, he aprendido muchas cosas de la vida poco relacionadas con lo académico, pero que a su vez me han inculcado valores o han generado cambios que han hecho efecto aquí.

Agradezco a Alejandro, mi director, por haber imaginado esta Tesis cuando yo recién me convertía en una profesional. Por su apoyo al andar y sus reiterados empujones cuando más lo necesitaba. Gracias por ser mi guía en esta aventura tan particular.

A mi círculo familiar más cercano: ese *cuarteto de personas especiales* donde cada uno supo colaborar a su manera para que hoy este trabajo pueda ser Tesis. A mi madre en particular, por su fortaleza y su espíritu de lucha que siempre han sido un gran ejemplo.

A la montaña que ha sabido forjar la fuerza y la perseverancia en mí, que me ha dado el temple para afrontar las situaciones difíciles y que continúa presentándome nuevos desafíos, nuevas cumbres. Y por supuesto a los grandes personajes que en ella habitan: mi querido Leo, el Chegu y la gente linda de Abrazo a Ventania y del GAEMN.

No puede quedar afuera la gente de la Universidad Nacional del Sur: a mis compañeros becarios con los que dimos los primeros pasos en la investigación, y a mis amigos Barbie y JaviZ que siempre apostaron en mí.

A todos mis amigos y familia que siempre me acompañan y regalan momentos de felicidad. A todos ellos les debo lo que soy hoy y lo que puedo obtener en la vida.

Resumen

Esta Tesis se centra en el estudio, análisis y comparación de un conjunto de lenguajes y librerías que permiten programar sistemas utilizando paralelismo. Su investigación está motivada por el impetuoso ritmo de avance que posee actualmente el hardware y las herramientas de programación en paralelo. Tal situación es visible, por ejemplo, al observar que los fabricantes de chips ya no derivan la capacidad de cómputo escalando la velocidad de reloj de los procesadores, sino que han comenzado a incrementar el potencial agregando CPUs o núcleos adicionales. El impulso al estudio del paralelismo es clave, ya que el paralelismo deja de ser algo únicamente centrado en el ámbito académico o en grandes laboratorios de investigación de computación de alta performance.

A su vez, hoy en día, en la mayoría de los ámbitos científicos deben resolverse problemas computacionales muy complejos y con altos requerimientos en tiempo y recursos. Para resolver este tipo de problemas, el paralelismo se ha convertido en una de las herramientas computacionales de mayor relevancia. Para esto, no sólo es necesario contar con hardware que permita realizar cómputo paralelo, sino también con lenguajes de programación para el desarrollo de aplicaciones con paralelismo.

Como se muestra a lo largo de la Tesis, en la literatura se reconoce que la escritura de programas destinados a una ejecución paralela no es una tarea trivial. La elección del lenguaje y el formalismo apropiado son un problema central. Es por este motivo que se ha estudiado un conjunto heterogéneo de lenguajes y librerías a fin de mostrar las distintas herramientas que se disponen en la actualidad para programar en paralelo. Entre los lenguajes estudiados se encuentran de alto y de bajo nivel, con paralelismo implícito y explícito, compilados e interpretados. Se analizan tanto lenguajes imperativos, como declarativos y de desarrollo de sistemas multi-agentes. También se describen nuevas tendencias como la programación en Grid y los lenguajes de propósito general para la GPU (Graphic Processing Unit).

Abstract

The present Thesis is focused on the study, analysis and comparison of a set of languages and libraries that can be used to add parallelism to our programs. The research is motivated by the striking evolution of current hardware and parallel programming tools. That situation is evident, for example, when recalling that manufacturers are not increasing clock speed anymore: nowadays, they do rise up the computational power by adding extra CPUs or cores. The encouragement to parallel programming study is key as parallelism stops to be something uniquely centered in academic institutions or high performance computing laboratories.

Moreover, in most scientific environments, very complex computational problems with high resources and time requirements must be currently solved. Parallelism has evolved to be one of the most relevant computing tools to deal with these types of problems. For that purpose, not only some kind of specialized hardware for parallel computing is needed, but also a set of programming languages for the development of parallel applications.

As it is shown through the present Thesis, literature recognizes that the codification of programs targeting a parallel execution is not a trivial task. The central problem is constituted by both the language election and the appropriate formalism. In sync with these reasons, a set of heterogeneous languages and libraries are studied with the aim of showing different tools that are available today for parallel programming. Among the analyzed languages we have some of high and some of low abstraction level, with implicit and explicit parallelism, compiled and interpreted. A number of imperative, declarative and multi-agent system languages are studied. New tendencies like Grid Computing and General-purpose Programming in the GPU (Graphic Processing Unit) are also described.

Índice general

1. Introducción **1**

1.1. Aporte . . . 2

1.2. Motivación . . . 4

1.3. Razones para el estudio y uso de paralelismo . . . 6

1.4. Requerimientos básicos de la programación en paralelo . . . 13

1.4.1. Clasificación de los distintos tipos de paralelismo . . . 14

1.5. Organización de la tesis . . . 16

2. Arquitecturas de computadoras y modelos de paralelismo **17**

2.1. Paralelismo, Concurrencia y Sistemas Distribuidos . . . 18

2.2. Taxonomía de computadoras paralelas . . . 19

2.3. Ejemplos concretos de arquitecturas paralelas . . . 23

2.3.1. Multiprocesadores . . . 24

2.3.2. Multicomputadoras . . . 25

2.3.3. Comparación . . . 27

2.4. Redes de interconexión . . . 29

2.4.1. Redes de interconexión dinámicas . . . 30

2.4.2. Redes de interconexión estáticas . . . 30

2.5. Modelos de algoritmos paralelos . . . 32

2.5.1. Clases de algoritmos 33
2.5.2. Problemas típicos 35
2.5.3. Análisis de granularidad de los problemas 36
2.6. Resumen 37

3. Paralelismo en lenguajes imperativos **39**
3.1. Clasificación de lenguajes paralelos 40
3.2. Lenguajes y librerías de paralelismo analizadas 42
3.2.1. HPF 45
3.2.2. OpenMP 50
3.2.3. PVM 55
3.2.4. MPI 60
3.3. Ejemplo comparativo: RankSort 65
3.4. Comparación de lenguajes y primitivas 73
3.5. Resumen 75

4. Paralelismo en lenguajes de programación en lógica **77**
4.1. Oportunidades de paralelismo 78
4.2. Paralelismo OR 82
4.2.1. Modelos de ejecución con paralelismo OR 86
4.3. Paralelismo AND Independiente 91
4.3.1. Fase de ordenamiento 92
4.3.2. Fase de ejecución hacia adelante 94
4.3.3. Fase de ejecución hacia atrás 95
4.4. Paralelismo AND Dependiente 98
4.4.1. Detección de paralelismo 99
4.4.2. Manejo de variables 101

4.4.3. Backtracking . . . 102
4.5. Paralelismo en Prolog . . . 103
4.5.1. Andorra-I . . . 108
4.5.2. CIAO Prolog . . . 109
4.5.3. Parlog . . . 111
4.5.4. Quintus Prolog . . . 112
4.5.5. PVM-Prolog . . . 114
4.5.6. DeltaProlog . . . 117
4.6. Ejemplo comparativo: RankSort . . . 119
4.7. Comparación de lenguajes . . . 125
4.8. Resumen . . . 128

5. Programación en paralelo con un modelo de Sistemas Multi-Agente 131

5.1. Agentes y Sistemas Multi-agentes . . . 132
5.2. Programación en paralelo utilizando un modelo de SMA . . . 133
5.2.1. Soluciones clásicas . . . 135
5.2.2. Solución de SMA . . . 137
5.3. Características deseables de un lenguaje basado en SMA . . . 137
5.4. Caso de estudio . . . 139
5.4.1. Resumen de Orgel . . . 140
5.4.2. Detalles de Orgel . . . 141
5.4.3. Ejemplo comparativo: RankSort . . . 145
5.4.4. Evaluación . . . 148
5.5. Resumen . . . 151

6. Tendencias e Integraciones **153**

6.1. Tendencias actuales . . . 155

6.1.1. Grid Computing . . . 155

6.1.2. Computación de propósito general en la GPU . . . 159

6.2. Integraciones . . . 166

6.2.1. Primitivas básicas para paralelismo explícito . . . 167

6.2.2. Ideas para la integración de modelos con paralelismo implícito . . . 169

6.2.3. Ideas para la integración de modelos con paralelismo explícito . . . 172

6.3. Resumen . . . 174

7. Conclusiones **175**

Índice de lenguajes y librerías mencionadas **181**

Bibliografía **183**

Índice de figuras

1.1. Evolución de las arquitecturas de computadoras paralelas [CSG99] 11
1.2. Demandas computacionales de distintas aplicaciones [CSG99] 13
1.3. Obstáculos de la programación en paralelo 14
1.4. Clasificación de los distintos tipos de paralelismo 15

2.1. Una taxonomía de computadoras paralelas [Tan98] 19
2.2. Arquitecturas MIMD 22
2.3. Intel Pentium Pro Quad 24
2.4. Sun Enterprise 25
2.5. Topología *torus* de tres dimensiones 26
2.6. Intel Paragon y Sandia Option Red 27
2.7. Comparación en cantidad de operaciones de punto flotante (GFLOPS) . . 28
2.8. Redes de interconexión dinámicas 31
2.9. Una red omega conectando 8 entradas con 8 salidas [KGGK94] 31
2.10. Redes de interconexión estáticas 33

3.1. Modelo de mapeo de datos de HPF 47
3.2. Distribución `BLOCK` y `CYCLIC` de un arreglo en 3 procesadores 49
3.3. Modelo de máquina virtual de PVM 56
3.4. Comunicación entre tareas en PVM 57
3.5. Operaciones de reducción de datos en MPI 64

3.6. Primitivas de transferencia de datos en MPI 65

4.1. Or-parallel search tree [GPA+01] 83
4.2. Tipos de variables y ligaduras (rama derecha de la Figura 4.1) 85
4.3. Modelos or-paralelos 86
4.4. Árbol de directorio 88
4.5. Ventanas de hashing 89
4.6. Binding arrays 90
4.7. Árbol AND en la ejecución de Fibonacci 91
4.8. Pérdida de la correspondencia entre la disposición lógica y física 96

5.1. Obstáculos de la programación en paralelo 134
5.2. Nivel de abstracción en las soluciones clásicas 135

6.1. Temas incluidos en esta Tesis 154
6.2. Arquitectura de Grid [CER] 157
6.3. Pipeline tradicional de GPU 160
6.4. GPU NVIDIA Tesla con 112 procesadores paralelos [NBGS08] 163
6.5. Niveles de granularidad paralela y repartición de memoria [NBGS08] . . . 166
6.6. Un grafo de dependencias 171

Índice de cuadros

1.1. Lenguajes y librerías mencionadas a lo largo de la Tesis 2
1.2. Lenguajes y librerías destacados 3

2.1. Cuadro comparativo de computadoras 28

3.1. Los lenguajes que se analizan y su clasificación 44
3.2. Resumen y clasificación de primitivas en los distintos lenguajes analizados . 45
3.3. Comparación de los distintos lenguajes analizados 74

4.1. Clasificación en costos de los distintos modelos 87
4.2. Clasificación de los lenguajes lógicos 107
4.3. Comparación de los distintos lenguajes lógicos 127

Capítulo 1

Introducción

Esta investigación se centra en el estudio de los principales paradigmas de programación en paralelo existentes, con el objetivo de relacionar técnicas de paralelismo implícito y explícito. De esta manera, se estudia la posibilidad de que cada modelo pueda nutrirse de elementos presentes en el otro y así combinar las ventajas de ambos.

El ritmo de avance en el hardware y las herramientas para programar con paralelismo es vertiginoso. Desde hace algunos años ha sido notorio el desarrollo de computadoras con más de un procesador o de procesadores con varios núcleos. Esto hace que el paralelismo ya no esté alejado del público en general, sino que está esperando a ser aprovechado dentro de cualquiera de nuestras computadoras personales. El impulso al estudio del paralelismo es clave: el paralelismo deja de ser algo únicamente centrado en el ámbito académico o en grandes laboratorios de investigación de computación de alta performance.

Sin embargo, se conoce que la escritura de programas destinados a una ejecución paralela no es una tarea trivial. La elección del lenguaje y el formalismo apropiado son un problema central. Como se verá en esta Tesis, uno de los factores determinantes para la elección del lenguaje será la caracterización del problema a resolver.

A medida que se avance en los capítulos, se presentarán algunos sistemas o lenguajes que se utilizan para programar en paralelo, así como también las librerías o extensiones que se emplean. Se mostrará cuáles de estos lenguajes están siendo utilizados para el desarrollo de aplicaciones concretas, se describirá cuales están destinados a ámbitos académicos o de investigación, y también las tendencias que influirán en el desarrollo de nuevos lenguajes producto de tecnologías emergentes, como ser Grids y GPUs. Asimismo, se hará una reseña de las áreas de aplicación y las arquitecturas en las cuales son útiles estos sistemas.

1.1. Aporte

La contribución principal de esta Tesis es el estudio, análisis y comparación de un conjunto de lenguajes y librerías que permiten programar sistemas utilizando paralelismo. En el Cuadro 1.1 se muestra un listado alfabético de todos los lenguajes y librerías que son mencionados en la Tesis; y en el Cuadro 1.2 el subconjunto de estos lenguajes que serán analizados y estudiados en profundidad, junto con una breve reseña de sus principales características.

&-Ace	Emerald	Orgel
&-Prolog	Epilog	P3L-SKIE
ACE	FCP	ParAKL
Andorra-I	GHC	Parlog
Aurora	HPF	PLoSys
BinProlog	Java threads	Pthread
Brook/Brook+	Jinni	PVM
C⋆	Linda	PVM-Prolog
CIAO Prolog	MPI	Quintus Prolog
Cilk	MPI-2	ROPM
Concurrent Prolog	Muse	SCL
CS-Prolog	NESL	SkelML
CUDA	Occam	Skil
DAOS	OpenMP	SR
DASWAM	Orca	STRAND
DeltaProlog		

Cuadro 1.1: Lenguajes y librerías mencionadas a lo largo de la Tesis

A lo largo de los Capítulos 3, 4, 5 y 6, los lenguajes del Cuadro 1.2 son analizados y comparados entre sí, teniendo en cuenta los objetivos de su concepción, el área de aplicación y las arquitecturas para las que fueron desarrollados. Para facilitar la descripción y comparación, se introduce un ejemplo de ordenamiento en paralelo, el cual es desarrollado en cada uno de los lenguajes para mostrar sus características, sus similitudes y sus diferencias.

Como puede verse en el Cuadro 1.2, en el conjunto elegido hay lenguajes de alto y de bajo nivel; con paralelismo implícito y explícito; compilados e interpretados; y diseñados

Lenguaje	Paralelismo	Nivel	Año	Arquitectura[1]	Tipo	Cap.
Andorra-I	implícito	alto	1989	MP	interpretado	4
CIAO Prolog	explícito	alto	1997	MP	interpretado	4
CUDA	explícito	bajo	2007	GPU	compilado	6
DeltaProlog	explícito	alto	1984	MC	interpretado	4
HPF	explícito	alto	1993	MP	compilado	3
MPI	explícito	bajo	1993	MP / MC	compilado	3
OpenMP	explícito	alto	1997	MP	compilado	3
Orgel	explícito	alto	2000	MP	compilado	5
Parlog	explícito	alto	1983	MP / MC	interpretado	4
PVM	explícito	bajo	1989	MP / MC	compilado	3
PVM-Prolog	explícito	bajo	1996	MP / MC	interpretado	4
Quintus Prolog	explícito	bajo	1986	MC	interpretado	4

Cuadro 1.2: Lenguajes y librerías destacados

para diferentes arquitecturas. Entre los lenguajes analizados hay imperativos (e. g. HPF), declarativos (e. g. Prolog) y de desarrollo de agentes (e. g. Orgel). Los interpretados corresponden en general a lenguajes declarativos; y muchos de estos lenguajes utilizan un preprocesador que traduce parte de las directivas para el compilador o intérprete. En la pág. 181 se incluye el índice de todos los lenguajes incluidos en el Cuadro 1.1, indicando los lugares en que son mencionados.

A fin de investigar la posibilidad de obtener un modelo de programación que permita reunir las ventajas de los dos tipos de paralelismo mencionados (explícito e implícito), se plantean los elementos presentes en algunos tipos de lenguajes que sería bueno trasladar a otros de distinta clase. Esto es, trabajando con paralelismo implícito se verá como las herramientas de lenguajes declarativos pueden adaptarse a lenguajes imperativos. Análogamente, al tratar con paralelismo explícito se evaluarán qué elementos de los lenguajes imperativos pueden ser ajustados para el caso de lenguajes declarativos. El estudio de estos modelos de programación se realiza sin perder de vista los requerimientos o cualidades que se pretenden del hardware.

[1]MP = Multiprocesador; MC = Multicomputadora; GPU = Graphic Processing Unit.

1.2. Motivación

Actualmente, en la mayoría de los ámbitos científicos deben resolverse problemas computacionales muy complejos y con altos requerimientos en tiempo y recursos. Para resolver este tipo de problemas, el paralelismo se ha convertido en una de las herramientas computacionales de mayor relevancia, ya que permite dividir el problema en partes, de modo tal que cada parte pueda ejecutarse un procesador diferente, simultáneamente con las demás. Para esto, es necesario contar con hardware que permita realizar cómputo paralelo, y lenguajes de programación para el desarrollo de aplicaciones con paralelismo.

Es interesante notar que un problema que es resuelto por una computadora de un único procesador en un tiempo T, idealmente debería ser resuelto en tiempo T/N por una computadora con N procesadores. Sin embargo, existen factores que podrían impedir que esta situación ideal pueda cumplirse. Por ejemplo, la naturaleza misma del problema podría no permitir su descomposición en N partes; o bien el lenguaje de programación podría restringir el paralelismo potencial si no dispone de las primitivas adecuadas para programar en paralelo.

Existen actualmente dos grandes grupos de lenguajes de programación en paralelo:

- **Lenguajes con paralelismo implícito**: el código del programa define la especificación de la solución del problema. En ejecución *automáticamente* se realiza el paralelismo que sea posible.
- **Lenguajes con paralelismo explícito**: a través de primitivas de programación, el programador indica explícitamente en el programa qué sección de código se ejecutará en paralelo.

Los lenguajes con paralelismo implícito permiten a los programadores escribir el código de sus programas sin preocuparse acerca de los detalles de cómo se realizará el paralelismo. Todo el paralelismo que existe en el programa será detectado automáticamente y explotado en ejecución. De este modo, el programa se ejecutará en paralelo en forma transparente para el programador. La principal ventaja de este tipo de lenguajes es que el programador se concentra en resolver el problema y no en la forma en que deben particionarse y sincronizarse las partes del problema para que se ejecute en paralelo. En consecuencia, el programador no necesita tener conocimientos específicos sobre el hardware subyacente,

ni de programación en paralelo, quedando así al alcance de un grupo más amplio de personas. Como ventaja adicional se tiene que el mismo programa puede ejecutarse en arquitecturas diferentes con una performance aceptable. La principal desventaja del paralelismo implícito radica en que al hacerse una paralelización automática, no siempre es posible obtener la mejor performance. Además, puede ocurrir que partes del código inherentemente secuenciales, sean consideradas como paralelizables, generando una sobrecarga innecesaria en ejecución.

En contraste con esto, los lenguajes con paralelismo explícito necesitan de un programador experto que resuelva cómo se particionará el problema entre los procesadores disponibles, y cómo se realizará la sincronización de las tareas que resuelven el problema en paralelo. La ventaja de este modelo es que, mediante una buena programación, puede obtenerse la mayor performance posible con el hardware disponible. Sin embargo, para lograr esto, el programador debe tener un acabado conocimiento del hardware donde se ejecutará el programa. Es importante destacar que, cuando se programa para un hardware particular, si luego el programa es ejecutado en un hardware distinto para el que fue concebido, su performance puede bajar considerablemente. Por ejemplo, programas construidos para arquitecturas con memoria compartida, tienen muy baja performance cuando se ejecutan en una máquina con diferentes módulos de memoria distribuida [Les93].

Hoy en día existe en el mercado una amplia variedad de arquitecturas de hardware con múltiples procesadores. Un buen modelo de computación paralela debería ser independiente de la arquitectura subyacente. De este modo, los programas podrían ser migrados fácilmente de una computadora paralela a otra, sin tener que ser desarrollados nuevamente o sin modificar gran parte de su código. Sin embargo, todavía no existe un modelo de computación en paralelo con esas características, que mantenga la performance al cambiar de hardware [KGGK94]. Los lenguajes con paralelismo implícito permiten que el código se pueda migrar más fácilmente de una arquitectura a otra. En lenguajes compilados, el compilador es el que genera la paralelización automática del programa. Para lograr esto es necesario contar con los compiladores apropiados para cada tipo de arquitectura.

En la práctica, cuando se desea incorporar paralelismo a un programa, existen tres aproximaciones básicas:

1. utilizar lenguajes procedurales (imperativos) extendidos con constructores para paralelismo;
2. utilizar compiladores que automáticamente paralelicen a los programas secuenciales;

3. utilizar lenguajes declarativos (funcionales o lógicos), que pueden ser implícitamente paralelizados de manera más sencilla que los procedurales debido a que carecen de estructuras de control.

Extraer el paralelismo implícito en un programa no es una tarea fácil. En general, se torna aún más difícil si el programa está escrito en un lenguaje procedural o imperativo como Fortran, C, o Pascal. Esto se debe, en gran parte, a que en estos lenguajes el programador debe especificar con un alto grado de detalle cómo se resolverá el problema. Sin embargo, en lenguajes declarativos (como el paradigma funcional o lógico), el programa presenta un nivel más alto de abstracción en la especificación de la solución del problema. La paralelización automática ha tenido mayor éxito en este tipo de lenguajes, que bien pueden ser interpretados o compilados.

En resumen, tanto los lenguajes con paralelismo implícito como los de paralelismo explícito poseen características deseables, pero en algún sentido complementarias. En general las ventajas de cada uno no están presentes en el otro. En esta Tesis se estudiaran los dos enfoques con el objeto de investigar la posibilidad de obtener un modelo de programación que permita reunir las ventajas de los dos tipos de paralelismo mencionados.

1.3. Razones para el estudio y uso de paralelismo

En la literatura, se han esgrimido numerosas razones que sostienen el uso de paralelismo. Cada autor se enfoca en algún aspecto particular, exponiendo los factores que impulsaron al paralelismo, el contexto tecnológico, su aplicabilidad en diversas ciencias, o bien sus propias predicciones. A continuación destacamos algunas de ellas.

- El paralelismo es una manera potente y comprobada de correr programas rápidamente. El paralelismo es la norma, una resolución puramente secuencial es la restricción anómala. [CG90]
- Un programa secuencial puede ser trasladado a un procesador paralelo, compilado de manera especial y ejecutado varias veces más rápido que en un procesador secuencial. [Lew94]
- El procesamiento paralelo de datos se ha vuelto una realidad. Grandes números de procesadores (computadoras) cooperando para computar una tarea dada han

impulsado un speed-up esencial para las computaciones secuenciales clásicas. Muchos problemas de computación que requerían gran tiempo para ser resueltos en tiempo real por máquinas secuenciales, pueden ser calculados en paralelo muy velozmente. [Hro97]

- La tecnología de las computadoras secuenciales ha sido llevada hasta la cercanía de sus límites, y existe una creciente creencia acerca de que las computadoras paralelas son el camino hacia una computación de alta performance. [Gup94]
- Las demandas de las aplicaciones en cuanto a performance continúan sobrepasando lo que un procesador individual puede entregar, y los sistemas multiprocesador ocupan un lugar cada vez más importante en computación en general. [CSG99]
- Examinando las tendencias actuales desde una variedad de perspectivas –economía, tecnología, arquitectura y demandas de la aplicación– se ve que la arquitectura paralela es crecientemente atractiva y central. La búsqueda de performance es tan aguda que el paralelismo se explota a diferentes niveles y en varios puntos del espacio de diseño de computadoras. [CSG99]
- Computadoras paralelas que contienen miles de procesadores se encuentran disponibles comercialmente. Estas computadoras proveen un poder de cómputo varios órdenes de magnitud superior al de las supercomputadoras tradicionales, a un costo mucho más bajo. Ellas abren nuevas fronteras en las aplicaciones de las computadoras: muchos problemas que anteriormente no se podían resolver pueden ser abordados si el poder de estas máquinas es usado efectivamente. [KGGK94]
- En ciertas ocasiones, las computadoras paralelas permiten resolver problemas que son imposibles de abordar secuencialmente, independientemente del tiempo que consuman. Estos problemas ocurren, por ejemplo, en aplicaciones en las que el número de datos o sus valores son funciones del tiempo, los resultados intermedios afectan futuras entradas, y las entradas consisten de múltiples flujos, o el efecto de la computación no puede ser revertido. En éstas y muchas otras circunstancias, no es la velocidad de la computadora paralela lo que importa, sino su habilidad para manejar varias situaciones a la vez. [Akl97]
- Las fuerzas más significativas que impulsan el desarrollo de computadoras más rápidas hoy en día, son las aplicaciones comerciales emergentes que requieren de una

máquina que sea capaz de procesar grandes cantidades de datos de manera sofisticada. La integración de computaciones paralelas, networking de alta performance y tecnologías multimedia está llevando al desarrollo de servidores de video, i. e., computadoras diseñadas para servir cientos o miles de requerimientos simultáneos de video en tiempo real. [Fos95]

- Existe una demanda creciente en cuanto a computadoras de alta performance en áreas de análisis estructural, pronóstico del tiempo, exploración de petróleo, investigaciones en fusión de energía, diagnóstico médico, simulaciones de aerodinámica, inteligencia artificial, sistemas expertos, automatización industrial, sensado remoto, defensa militar, ingeniería genética y socio-economía, entre muchas otras aplicaciones científicas y de ingeniería. [HB84]

- Las actividades humanas y la ley natural no son sólo secuenciales, sino altamente paralelas. El paralelismo es tan importante y fundamental como lo secuencial. [Les93]

- El comportamiento paralelo o pseudo-paralelo es la norma y debe ser completamente entendido por aquellos que quieren convertirse en profesionales en la ingeniería de computación. [BD93]

- El paralelismo se convertirá, en un futuro no muy lejano, en una parte esencial del repertorio de todo programador. Cada programador, ingeniero de software y científico de computación necesitará entender paralelismo. [CG90]

Como dejan relucir algunas de las citas, el ritmo de avance en el hardware y las herramientas para programar con paralelismo es vertiginoso. Desde hace algunos años ha sido notorio el desarrollo de computadoras con más de un procesador o de procesadores con varios núcleos. Esto hace que el paralelismo ya no esté alejado del público en general, sino que está esperando a ser aprovechado dentro de cualquiera de nuestras computadoras personales. El impulso al estudio del paralelismo es clave: el paralelismo deja de ser algo únicamente centrado en el ámbito académico o en grandes laboratorios de investigación de computación de alta performance.

No obstante, es de amplio conocimiento en la comunidad informática que la escritura de programas destinados a una ejecución paralela no es una tarea trivial. La elección del lenguaje y el formalismo apropiado son un problema central. Estas elecciones no son absolutas, sino que el lenguaje o formalismo puede resultar adecuado o no dependiendo de

las características del problema. Además, a la hora de ejecutar un programa seguramente buscaremos que el proceso tenga una buena performance. Esto no sólo depende de lo bien que se hayan utilizado las primitivas del lenguaje, sino que también tiene dependencias con el hardware sobre el que se va a ejecutar. El hardware también presenta variantes de acuerdo con el tipo de problema que se pretende resolver. A continuación se resumen las características de cada uno de estos puntos centrales: *hardware*, *lenguajes* y *problemas*.

Hardware y Arquitecturas

Existen distintas posibilidades de paralelismo a nivel de arquitectura o hardware. En una máquina uniprocesador tenemos:

- Multiplicidad de unidades funcionales: muchas funciones aritméticas y lógicas pueden ser distribuidas en múltiples unidades funcionales especializadas que pueden operar con paralelismo.
- Paralelismo y pipelining dentro de la CPU: varias fases en la ejecución de instrucciones se realizan tipo pipeline (búsqueda de instrucción, decodificación, búsqueda de operandos, ejecución y almacenamiento de resultados). Además, se han desarrollado técnicas para pre-fetch de instrucciones y buffering de datos.
- Solapamiento de las operaciones de E/S y CPU: las operaciones de CPU se pueden realizar en simultáneo con las operaciones de entrada/salida utilizando controladores separados, canales o procesadores de E/S; por ejemplo DMA (direct-memory-access) que permite una transferencia directa entre los dispositivos y la memoria principal.
- Sistema de memoria jerárquico: se utiliza para achicar la diferencia entre las velocidades de la CPU y del acceso a memoria. Incluye memoria caché y organizaciones de memoria *interleaved* (en varios bancos).
- Multiprogramación y sistemas de tiempo compartido: en el mismo intervalo de tiempo, pueden haber muchos procesos activos en una computadora, compitiendo por recursos de memoria, E/S y CPU. Una buena mezcla de programas (limitados por CPU y limitados por E/S) promueve una mejor utilización de recursos.

El paralelismo va a continuar siendo utilizado por los diseñadores de hardware para incrementar la performance de procesadores. Desde la tecnología Intel MMX (MultiMedia

eXtensions), que implementa una forma de paralelismo de datos de escala pequeña, hasta la ejecución de instrucciones fuera de orden en procesadores superescalares, como el microprocesador Digital Alpha [ERL97]. Ejemplos más actuales incluyen las tecnologías de HyperThreading o Dual-Core (que combina dos procesadores independientes en un mismo chip) presentes en las computadoras personales.

Para cada aplicación, existe una computadora paralela preferida que mejor se adapta a las necesidades [Akl97]. Las computadoras paralelas pueden dividirse en cuatro configuraciones básicas [HB84]:

1. *Computadoras pipeline*, i. e., utilizan un pipeline para la ejecución de instrucciones.
2. *Array processors*, i. e., donde los elementos de procesamiento operan sincrónicamente (ejecuta la misma instrucción sobre distintos datos, SIMD).
3. *Sistemas multiprocesador*, i. e., varios procesadores interconectados (bus común o una red conmutada), que pueden tener una memoria compartida.
4. *Sistemas multicomputadora*, i. e., varias computadoras conectadas en red con enlaces de alta velocidad, formando alguna topología en particular y que utilizan para comunicarse el mecanismo de pasaje de mensajes.

Tanto el *pipeline* como la ejecución *superescalar* (i. e., la habilidad de ejecutar más de una instrucción por ciclo de reloj, que se logra al tener múltiples unidades de ejecución o pipelines) incrementan el throughput de instrucciones. Los pipelines más largos (con más etapas) y los superescalares con más pipelines ponen más responsabilidad en el compilador para entregar una buena performance. Pero las dependencias de datos y de control en los programas, junto con las latencias de las instrucciones, ofrecen un límite superior a la performance que se entrega, ya que el procesador a veces debe esperar a que se resuelva una dependencia, como en el caso de un salto condicional mal predecido.

Programas y Lenguajes

Un programa que exhibe paralelismo está escrito de manera que el programador es consciente de que debe particionar las tareas y de que estas tareas potencialmente se ejecutarán en procesadores diferentes. Un programa de este estilo seguramente incluye llamadas a funciones que permitan la comunicación entre las subtareas y la sincronización de los procesos, dos de las principales actividades al trabajar con paralelismo.

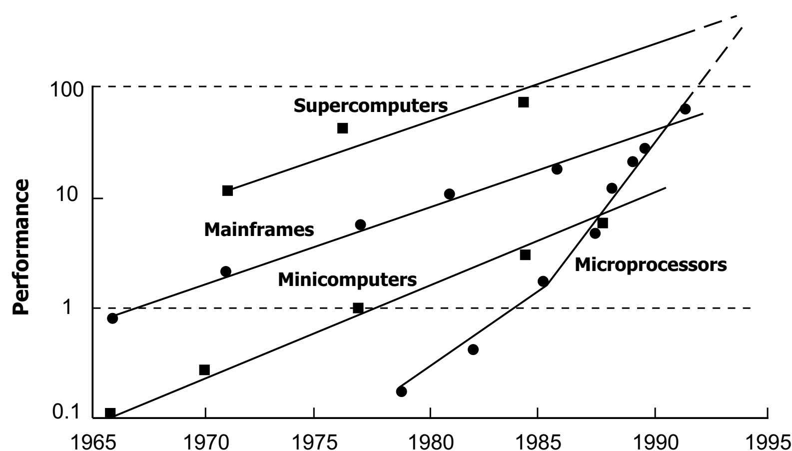

Figura 1.1: Evolución de las arquitecturas de computadoras paralelas [CSG99]

Los algoritmos paralelos son implementados en máquinas paralelas utilizando un lenguaje de programación. Este lenguaje debe ser lo suficientemente flexible como para permitir una implementación eficiente y debe ser fácil de utilizar. Nuevos lenguajes y paradigmas son desarrollados –continuamente– para alcanzar estos objetivos. Los modelos de paralelismo se implementan de varias formas: como librerías que se invocan desde lenguajes de programación secuenciales tradicionales, extensiones de lenguajes o modelos de ejecución completamente nuevos. Muchos de los lenguajes de programación para computadoras con memoria compartida o pasaje de mensajes son esencialmente lenguajes para programación secuencial aumentados con un conjunto de llamadas al sistema especiales. Estas llamadas al sistema incluyen primitivas de bajo nivel para pasaje de mensajes, sincronización de procesos, creación de procesos, exclusión mutua y otras funciones necesarias [KGGK94].

Los desarrolladores de software paralelo deben enfrentarse con problemas que no encontraban en la programación secuencial. El no-determinismo, la comunicación, sincronización, tolerancia a fallas, heterogeneidad, memoria compartida y/o distribuida, deadlock y condiciones de carrera presentan nuevos desafíos. Más aún, si el paralelismo en una aplicación no se amolda a la topología de una arquitectura paralela determinada, el diseñador debe modificar el programa para que se corresponda con la máquina. Debajo de todo esto está –siempre presente– la necesidad implícita de alta performance. [SS96]

Por otro lado, se realiza mucho trabajo en el diseño de compiladores que paralelizan el código, que extraen paralelismo implícito de programas que no han sido paralelizados

explícitamente. Tales compiladores permiten programar en una computadora paralela como si fuera secuencial. Se especula que esta aproximación tiene un potencial limitado para explotar el poder de las computadoras paralelas de gran escala [KGGK94], ya que el éxito de la paralelización automática depende de la estructura del código secuencial, pero fundamentalmente porque *el programa paralelo óptimo para resolver un problema puede diferir completamente de la mejor solución secuencial* [Zav99].

Además, para facilitar la programación de las computadoras paralelas, es importante el desarrollo de entornos y herramientas comprensibles. Éstas deben servir para desvincular a los usuarios de las características de bajo nivel de las máquinas y a la vez proveer herramientas de diseño y desarrollo como debuggers y simuladores.

Problemas y Algoritmos

Una computadora paralela brinda poca ayuda, a menos que se trate con algoritmos paralelos eficientes. El paralelismo puede incrementar la complejidad de una aplicación. Pero, cualquiera de sus ventajas puede transformarse en una desventaja si no se aplica adecuadamente.

Los problemas que presentan paralelismo son problemas que se plantean naturalmente en varias tareas simultáneas, que colaboran o interactúan para alcanzar la solución integral. Pueden ser problemas donde existen varias entradas que deben ser tratadas con ciertas restricciones de tiempo, señales de eventos que deben ser manejadas en el momento adecuado, o tareas con grandes tiempos de ejecución.

Para sacar provecho del enorme potencial ofrecido por el crecimiento de la tecnología de procesamiento paralelo, es necesario diseñar algoritmos paralelos que puedan mantener a un gran número de procesadores computando en paralelo para la concreción de la computación en general. En algunos casos, los algoritmos secuenciales estándar son fácilmente adaptados al dominio paralelo. Sin embargo, en la mayoría de los casos, el problema computacional debe ser re-analizado desde su base para desarrollar algoritmos paralelos enteramente nuevos.

La demanda creciente de performance para las aplicaciones es una característica común en cada aspecto de las ciencias de la computación. Los avances en la capacidad del hardware posibilitan nuevas funcionalidades en las aplicaciones, lo que crece en

significancia e impone aún más demandas en la arquitectura. Las aplicaciones más demandantes en la actualidad se pueden apreciar en la Figura 1.2.

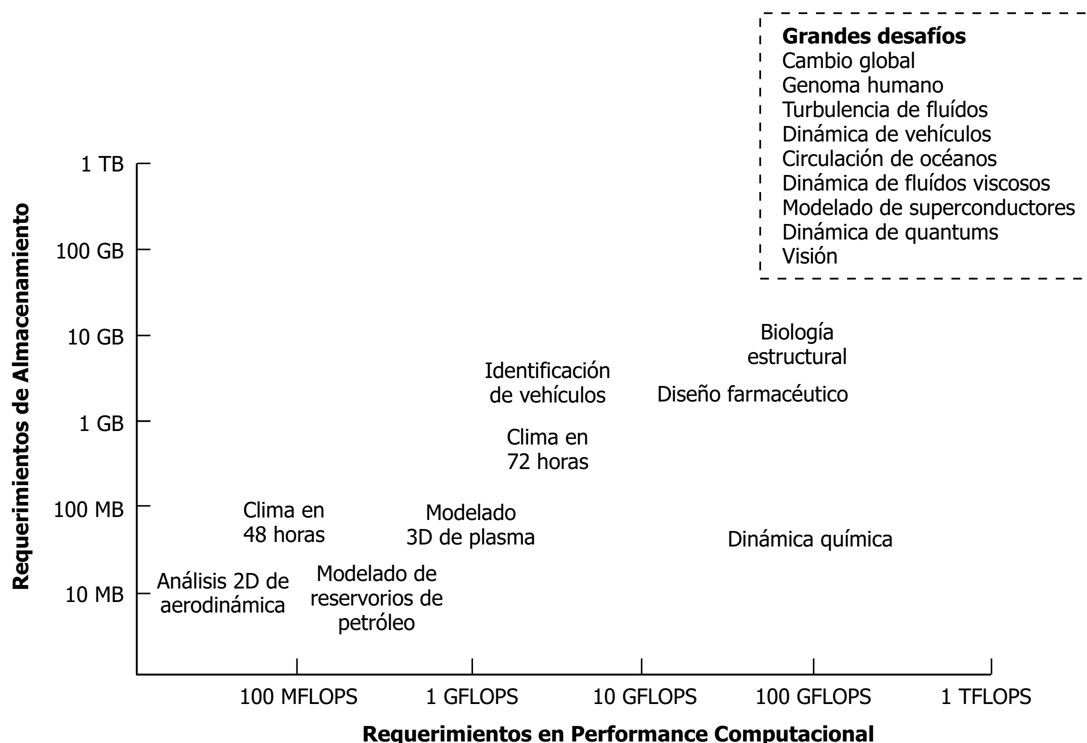

Figura 1.2: Demandas computacionales de distintas aplicaciones [CSG99]

1.4. Requerimientos básicos de la programación en paralelo

La programación en paralelo ofrece una herramienta computacional imprescindible para aprovechar el uso de múltiples procesadores y en la resolución de problemas que no pueden resolverse mediante técnicas clásicas. En el proceso de diseño de programas paralelos hay que tener en cuenta lo siguiente [HH03]:

1. *Descomposición*: involucra el proceso de dividir el problema y la solución en partes más pequeñas. Es decir, determinar qué parte del software realiza qué tarea.
2. *Comunicación*: se debe determinar cómo se lleva a cabo la comunicación entre los distintos procesos o computadoras, cómo sabe un componente de software cuando

otro terminó o falló, cómo se solicita un servicio a otro componente, qué componente debe iniciar la ejecución, etc.

3. *Sincronización*: se debe determinar el orden de ejecución de los componentes, si todos los componentes inician su ejecución simultáneamente, o alguno debe esperar mientras otros trabajan, etc.

De lo anterior es posible identificar los siguientes obstáculos que se encuentran comúnmente al programar en paralelo:

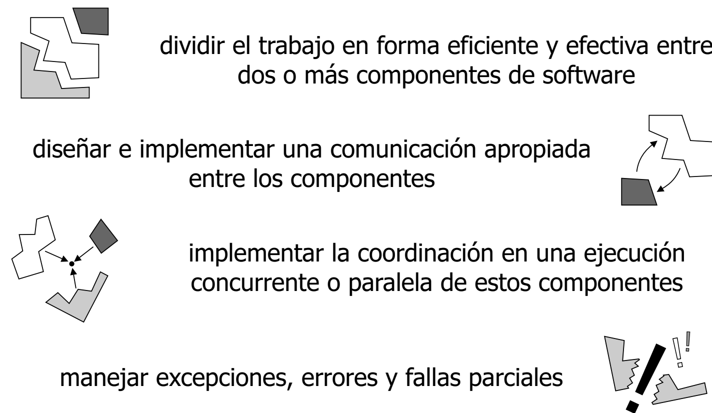

Figura 1.3: Obstáculos de la programación en paralelo

1.4.1. Clasificación de los distintos tipos de paralelismo

A partir del estudio de los sistemas de programación en paralelo, y debido a la gran variedad de lenguajes, se introduce una clasificación que considera tres grupos análogos de lenguajes de programación en paralelo [ST98, To95, Zav99]:

(1) *Lenguajes con paralelismo implícito completamente abstractos*: en este grupo se incluyen los enfoques en los cuales el programador está completamente aislado de la idea de ejecución paralela. Todos los pasos para crear un programa paralelo están a cargo de compiladores y sistemas de soporte en tiempo de ejecución. Este grupo incluye a los paradigmas lógico (ver Capítulo 4) y funcional y a compiladores que paralelizan código imperativo.

(2) *Lenguajes con paralelismo explícito de alto nivel*: incluye a los sistemas en los que el programador participa en algunas etapas del paralelismo (ver Capítulo 3).

(3) *Lenguajes con paralelismo explícito de bajo nivel*: incluye a los sistemas en los que el programador maneja explícitamente todos los problemas relacionados con el paralelismo (ver Capítulo 3).

A partir de lo estudiado, y para guiar la lectura de los siguientes capítulos, se puede ensayar la clasificación que se ve en la Figura 1.4. Los números en la figura se corresponden con las clases enumeradas anteriormente.

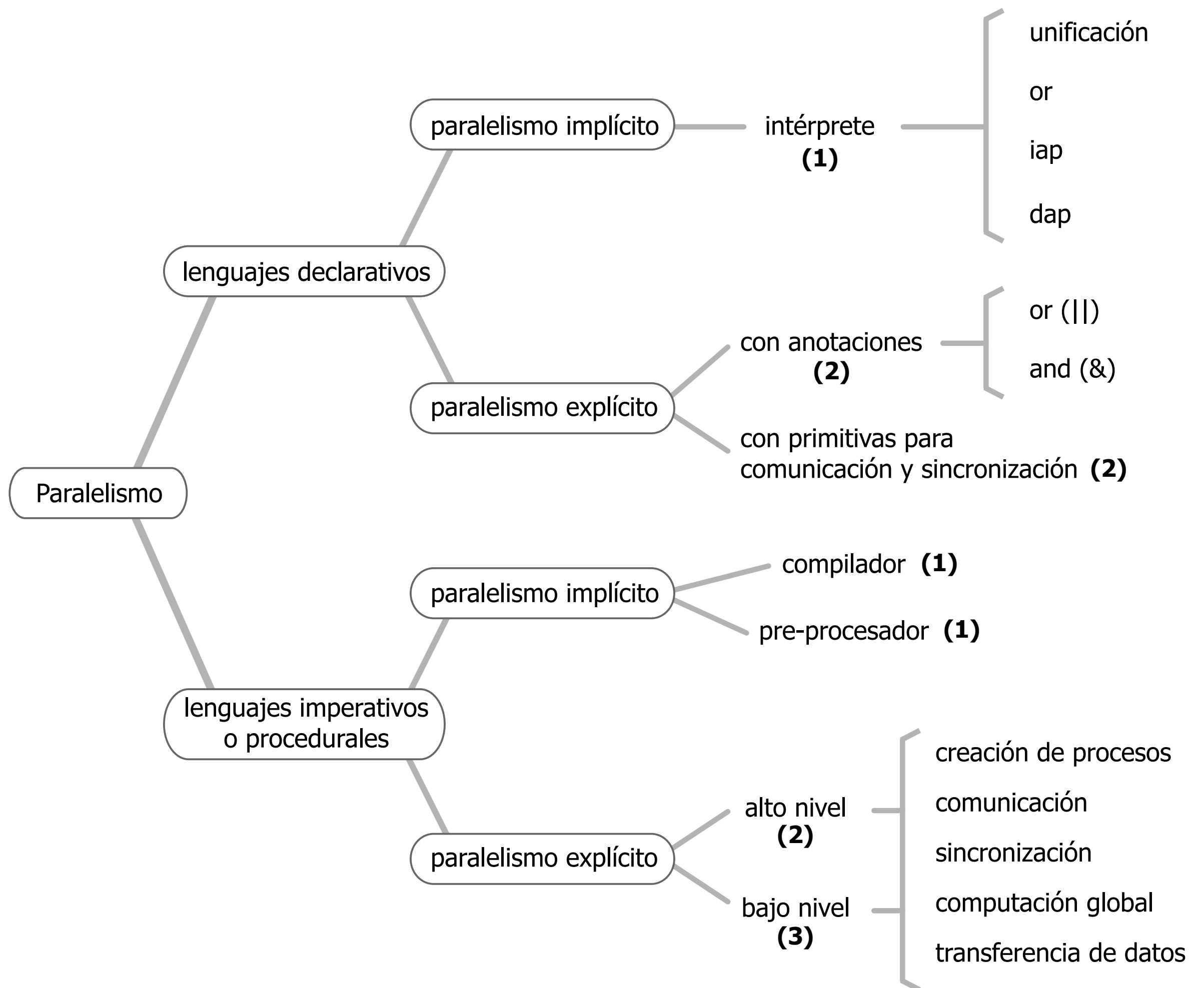

Figura 1.4: Clasificación de los distintos tipos de paralelismo

1.5. Organización de la tesis

Esta tesis está organizada de la siguiente manera:

Capítulo 2: Está dedicado a introducir los conceptos fundamentales de paralelismo, para lo que se incluye una taxonomía de computadoras paralelas. Se enfoca particularmente en arquitecturas MIMD, que trata a sistemas con múltiples procesadores y se subdivide en *multiprocesadores* y *multicomputadoras*.

Capítulo 3: Se analizan algunos de los lenguajes imperativos de programación en paralelo más populares, prestando especial atención a las primitivas que proveen para el manejo de paralelismo y a la arquitectura de hardware en que se inspiraron. Se presentarán a HPF y OpenMP como ejemplos de lenguajes de alto nivel, y a PVM y MPI representando a los lenguajes de bajo nivel.

Capítulo 4: Se estudian las oportunidades para ejecución paralela que ofrece la programación en lógica. Los tipos de paralelismo explotados en la práctica por soportes en tiempo de ejecución de programas lógicos son el paralelismo OR y el paralelismo AND. Se presentarán y evaluarán algunas implementaciones de Prolog que incorporan estas formas de paralelismo: Andorra-I, CIAO Prolog, Parlog, Quintus Prolog, PVM-Prolog y DeltaProlog.

Capítulo 5: Se describe cómo utilizar el modelo de Sistemas Multi-agente (SMA) para el diseño de programas con paralelismo. Uno de los objetivos es identificar las características que debería poseer un lenguaje de programación con paralelismo que utilice un modelo de Sistemas Multi-agente, cubriendo las componentes de descomposición, comunicación y sincronización. El lenguaje Orgel constituye una primera aproximación al conjunto de características deseables.

Capítulo 6: Se relacionará lo visto a lo largo de la tesis con las tendencias actuales en hardware (redes de alta velocidad, Grid Computing, computación en la GPU) y los problemas de software. Se presentan ideas para la integración de modelos con paralelismo implícito y paralelismo explícito, observando qué aspectos pueden ser trasladados de un paradigma al otro.

Capítulo 7: Finalmente se detallan los resultados y las conclusiones obtenidas. También se incluyen comparaciones entre los lenguajes vistos, realizando un contraste y destacando los aspectos de los lenguajes de programación en cada clase.

Capítulo 2

Arquitecturas de computadoras y modelos de paralelismo

A fin de lograr que esta Tesis sea lo más autocontenida posible, se explican en este capítulo los conceptos principales de paralelismo que servirán de base para comprender lo que se desarrolla en capítulos posteriores.

En este capítulo se incluye una taxonomía de computadoras paralelas: la clasificación extendida de Flynn [Tan98], basada en los conceptos de flujo de instrucciones y flujo de datos. A partir de ello, se describen los cuatro tipos de arquitecturas: SISD, SIMD, MISD y MIMD. Particularmente nos enfocamos en esta última categoría, que trata a sistemas con múltiples procesadores y se subdivide en *multiprocesadores* y *multicomputadoras*.

Luego de la descripción teórica, se exploran varios ejemplos concretos de supercomputadoras con diferentes arquitecturas que fueron o están siendo utilizadas en Laboratorios de Computación de Alta Performance, en distintos lugares del mundo. Con el objeto de dar una idea de la evolución de la ciencia en este sentido, se comparan las tecnologías actuales con las más antiguas.

Para utilizar el gran potencial ofrecido por esta tecnología computacional de procesamiento paralelo, se vuelve necesario el desarrollo de algoritmos que puedan mantener un gran número de procesadores trabajando en paralelo en beneficio de una misma computación. En consecuencia, se describirán algunos de los modelos de paralelismo que existen: paralelismo de datos, relajado o independiente, no relajado o sincrónico, tipo pipeline, trabajadores replicados y particionamiento de datos. Para concluir, se presentan los problemas más significativos que limitan la performance de los algoritmos paralelos.

2.1. Paralelismo, Concurrencia y Sistemas Distribuidos

No sería extraño que, a partir de la lectura de distintos libros sobre paralelismo o sistemas distribuidos, encontremos cierta ambigüedad entre las definiciones siguientes, o bien diferencias en el contexto donde se utilizan algunos términos. Para establecer un acuerdo y evitar así posibles confusiones, exponemos a continuación las definiciones que se adoptan a lo largo de la Tesis.

Paralelismo es la ejecución simultánea de *la misma tarea* (particionada y adaptada) en múltiples procesadores con el objetivo de obtener resultados *en menor tiempo*.

Concurrencia se relaciona con la acción de *compartir recursos* comunes entre computaciones que se ejecutan *solapadas en el tiempo* (incluyendo en paralelo). Esto frecuentemente implica encontrar técnicas confiables para coordinar su ejecución, intercambiar datos, reservar memoria y planificar el tiempo de procesamiento de manera tal que se minimice el tiempo de respuesta y se maximice el throughput[1].

Sistemas Distribuidos consiste de una colección de computadoras autónomas enlazadas por una red y equipadas con un sistema de software distribuido que *luce a los usuarios como si fuera en sistema único y centralizado*. Esto contrasta con una red, en donde el usuario es consciente de que hay varias máquinas, y su ubicación, replicación de datos, balance de carga y funcionalidad no son transparentes. Los sistemas distribuidos usualmente utilizan alguna clase de organización cliente-servidor.

Un sistema exhibe concurrencia cuando varios procesos o subtareas progresan al mismo tiempo. Cuando estas subtareas se pueden ejecutar en simultáneo en procesadores diferentes, se tiene *paralelismo*. Cuando un conjunto de subtareas tienen que compartir un procesador particular, intercalando sus ciclos de ejecución, se tiene *concurrencia*.

Cabe destacar que, la computación distribuida es más general y universal que la computación paralela. La distinción es sutil pero importante. El paralelismo se restringe a una forma de computación distribuida. La computación paralela es una computación

[1]Se define *throughput* como la cantidad de instrucciones por ciclo que el sistema es capaz de ejecutar.

distribuida, donde el sistema completo se utiliza para resolver *un solo problema en el menor tiempo posible.* Luego, las computadoras paralelas optimizan la performance. La computación distribuida es más general y abarca otras formas de optimización [ERL97].

2.2. Taxonomía de computadoras paralelas

Muchos tipos de computadoras paralelas han sido propuestas, por lo que resulta natural preguntarse si hay alguna manera de categorizarlas en una taxonomía. El esquema más utilizado es el que fue propuesto por Flynn en 1972, cuyo modelo extendido se puede apreciar en la figura 2.1.

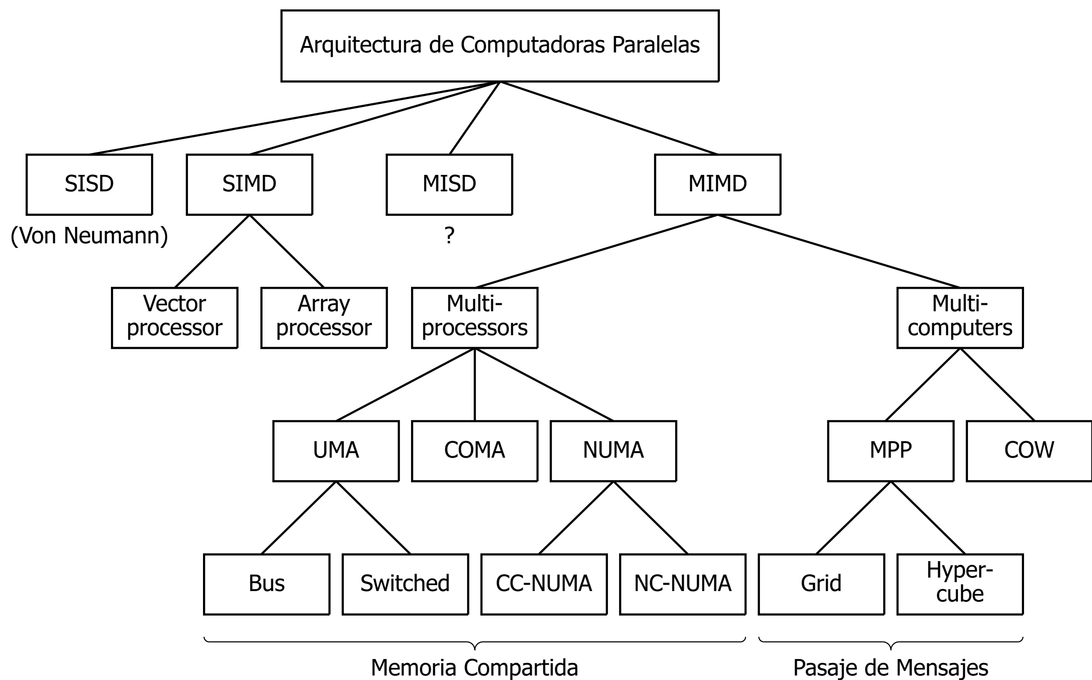

Figura 2.1: Una taxonomía de computadoras paralelas [Tan98]

La clasificación de Flynn está basada en dos conceptos: flujo de instrucciones y flujo de datos. Un flujo de instrucciones corresponde a un contador de programa. Un sistema con N CPUs tiene N contadores de programa, y por lo tanto N flujos de instrucciones. Un flujo de datos consiste de un conjunto de operandos.

El flujo de datos y de instrucciones son, hasta algún punto, independientes, entonces se pueden plantear cuatro combinaciones:

- **SISD** (simple instrucción, simple dato): consiste de un procesador y memoria (modelo clásico, secuencial de Von-Neumann). Toma una única secuencia de instrucciones y opera sobre una única secuencia de datos. Está limitada por la tasa de ejecución de instrucciones (se mejora agregando pipelining) y la velocidad con la que puede intercambiarse información entre memoria y CPU (se incrementa con entrelazado de memoria y caché).

- **SIMD** (simple instrucción, múltiple dato): única unidad de control que busca, decodifica y ejecuta una instrucción por sí misma o la deriva a un elemento de procesamiento (operan sincrónicamente). Tiene múltiples ALUs para llevar a cabo operaciones en múltiples conjuntos de datos simultáneamente. Se ajusta mejor a programas en los cuales el mismo grupo de instrucciones es ejecutado sobre un gran conjunto de datos.

- **MISD** (múltiple instrucción, simple dato): múltiples instrucciones operan sobre los mismos datos. No existen ejemplos claros de tales máquinas.

- **MIMD** (múltiple instrucción, múltiple dato): varios procesadores ejecutando de forma independiente, asincrónica, operando como parte de un gran sistema. Almacenan el programa y el sistema operativo en cada procesador. Se dividen en multicomputadoras y multiprocesadores.

Las cuatro arquitecturas descriptas difieren principalmente en la cantidad de procesadores que poseen, en el modo de operación de éstos (sincrónico/asincrónico), en cómo está organizada la memoria del sistema y dónde deben ubicarse el programa y datos (cuántas copias debe haber).

Extendiendo un poco el modelo, podemos considerar que la categoría de SIMD se divide en dos subgrupos. El primero es para supercomputadoras numéricas y otras máquinas que operan en vectores, realizando la misma operación en cada elemento del vector. El segundo subgrupo es para máquinas de tipo paralelo, como ILLIAC IV, en la cual una unidad de control maestra transmite masivamente (en broadcast) las instrucciones a muchas ALUs independientes.

La categoría MIMD se divide entre multiprocesadores y multicomputadoras. Los multiprocesadores poseen múltiples CPUs y un espacio de direcciones compartido, visible a todos los procesadores. Proveen soporte de hardware para accesos de lectura o escritura de

cualquier procesador al espacio de direcciones compartido. Algunas de éstas computadoras pueden simplemente compartir una memoria (shared-memory) que es igualmente accesible para todos los procesadores a través de una red de interconexión. Las multicomputadoras son arquitecturas MIMD que se comunican por pasaje de mensajes, cada procesador tiene su propia memoria –privada o local– que es accesible sólo por él. Presentan mejor escalabilidad: de 2.000 a 10.000 elementos de procesamiento.

Existen tres clases de multiprocesadores, distinguidas por la forma en que implementan la memoria compartida. Estas categorías existen porque en grandes multiprocesadores, la memoria usualmente se organiza en varios módulos. Basados en el tiempo que le toma a un procesador acceder a la memoria local y global, las computadoras de espacio de direcciones compartido se clasifican en UMA, NUMA y COMA. Si el tiempo de acceso para un procesador a cualquier palabra de memoria es idéntico, la computadora se clasifica como UMA (acceso a memoria uniforme). En cambio, si el tiempo para acceder a un banco de memoria remoto es mayor que el tiempo que toma acceder a uno local, la computadora es referida como NUMA (acceso a memoria no uniforme). Un mismo bloque de datos puede estar replicado en varios módulos (e. g., en cachés), lo que produce problemas de consistencia. Las arquitecturas NUMA se dividen en caché coherentes (CC-NUMA) y no cache coherentes (NC-NUMA). En el diseño de las computadoras COMA (acceso a memoria de sólo caché), se utiliza la memoria principal de cada CPU como una gran caché. Las páginas de memoria no tienen un lugar fijo, sino que migran dentro del sistema bajo demanda. En algunos casos sólo algunas CPUs pueden tener acceso a los dispositivos de E/S. Cuando no es así (todas acceden) recibe el nombre de SMP (multiprocesador simétrico).

Las multicomputadoras pueden dividirse a su vez en dos categorías. La primera contiene a las MPPs (procesadores masivamente paralelos), que son supercomputadoras costosas que consisten de muchas CPUs fuertemente conectadas mediante una red de interconexión propietaria de alta velocidad. El término Grid[2] referencia a una forma de interconexión entre los procesadores, que podría ser tanto una malla 2D o 3D, donde casi siempre se incluyen conexiones wrap-around (entre el último y el primer procesador de cada extremo). Esta red es más fácil de configurar pero tiene mayor diámetro[3] que una red hypercubo. Un Hypercubo es una red de interconexión que consiste en

[2]En la Sección 6.1.1 se utilizará el término Grid con otro significado para Grid Computing, que es el más utilizado en la actualidad.

[3]Se define *diámetro* como la distancia máxima que hay entre un par de procesadores en la red.

una malla multidimensional de procesadores con exactamente dos procesadores en cada dimensión (ver Sección 2.4.2). Luego el sistema contará con 2^N procesadores (siempre potencia de dos) donde cada uno está directamente conectado a otros N procesadores. Las supercomputadoras Cray T3E e IBM SP/2 son buenos ejemplos de esta categoría.

La otra categoría consiste de PCs regulares o estaciones de trabajo, posiblemente montadas en racks y conectadas por tecnologías de interconexión comerciales. Lógicamente no existe demasiada diferencia, pero las supercomputadoras gigantes de millones de dólares son usadas de forma diferente que las redes de PCs ensambladas por usuarios por una fracción del precio de una MPP. Las máquinas de esta categoría reciben el nombre de NOW (red de estaciones de trabajo) o COW (cluster de estaciones de trabajo).

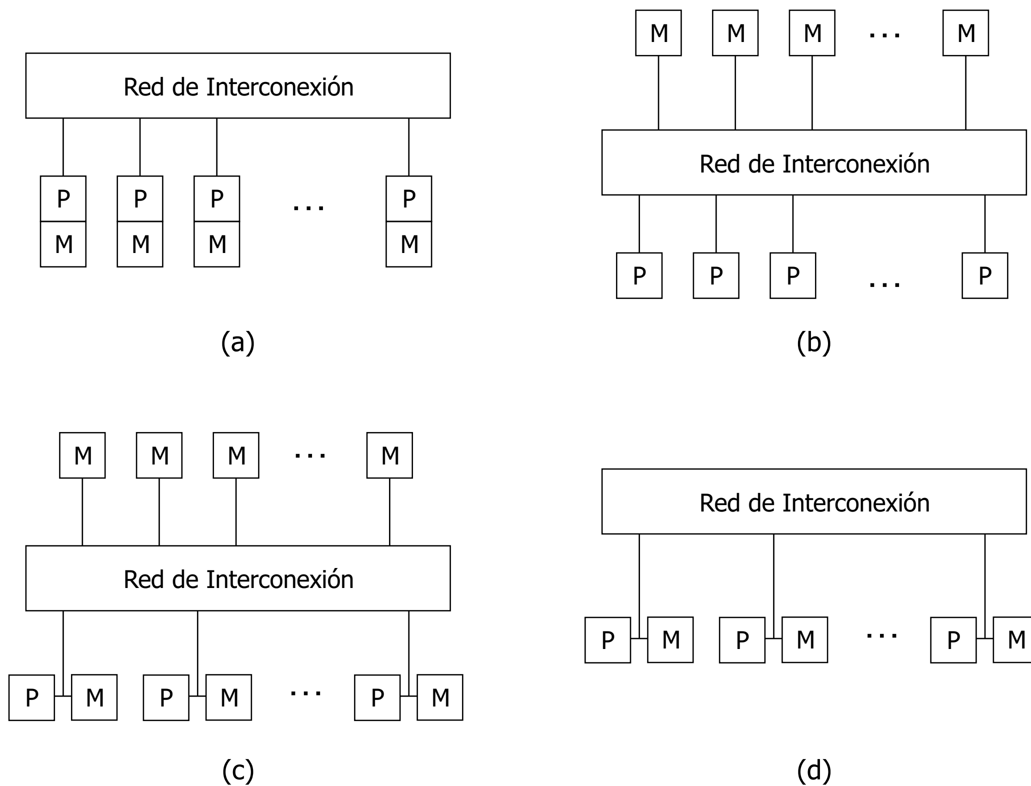

Figura 2.2: Arquitecturas MIMD

En la Figura 2.2 (a) se muestra la arquitectura típica de una multicomputadora, es decir, una arquitectura MIMD con pasaje de mensajes. Dentro de las arquitecturas MIMD con espacio de direcciones compartido tenemos tres configuraciones posibles. Las primeras computadoras de este tipo contenían una memoria compartida (posiblemente organizada en bancos) que era igualmente accesible para todos los procesadores a través

de una red de interconexión (Figura 2.2 (b)). En este caso el ancho de banda de la red de interconexión debía ser substancial para garantizar una buena performance. Una forma de relajar este requerimiento consiste en tener una memoria local a cada procesador, donde se almacena el programa que se ejecuta en ese procesador y las estructuras de datos no compartidas (ver 2.2 (c)). Las estructuras de datos globales se almacenan en la memoria compartida. Siguiendo con esta idea, se podría extender este concepto hasta el punto de eliminar completamente la memoria física compartida (Figura 2.2 (d)). Las referencias a celdas de memoria de otros procesadores son mapeadas por el hardware a los procesadores apropiados. El caso (b) es un ejemplo de arquitectura UMA, mientras que el (c) y (d) son ejemplos de arquitecturas NUMA.

Cabe notar que la arquitectura NUMA es muy similar a la de pasaje de mensajes, ya que la memoria está físicamente distribuida en ambos casos. La diferencia está en que la arquitectura NUMA provee soporte en hardware para las lecturas y escrituras en las memorias de procesadores remotos; mientras que, en las arquitecturas con pasaje de mensajes, los accesos deben ser emulados mediante el intercambio de mensajes explícito.

2.3. Ejemplos concretos de arquitecturas paralelas

Para ilustrar la taxonomía presentada en la sección anterior, vamos a ver algunos ejemplos concretos de computadoras paralelas con distintas arquitecturas. En primer lugar de describen dos multiprocesadores: el Intel Pentium Pro Quad que utiliza un canal compartido entre los procesadores para los accesos a memoria, y la Sun Enterprise que utiliza elementos de conmutación para conectar memorias y procesadores. En segundo lugar se presentan cuatro multicomputadoras (mayor número de CPUs): Cray T3E, Intel Paragon XP/S, Intel Sandia Option Red y Lemieux.

En tercer y último lugar presentaremos una comparación entre las supercomputadoras presentadas y los procesadores actuales que tendrá en cuenta el tiempo de vigencia de tales máquinas, la cantidad de procesadores, la cantidad de operaciones por segundo que pueden efectuar, la velocidad de transferencia de sus lazos de interconexión y su costo en dólares, que son los indicadores que se han considerado como más significativos.

2.3.1. Multiprocesadores

Intel Pentium Pro Quad

Esta computadora de Intel soporta hasta cuatro procesadores Pentium Pro de 166 MHz organizados en dos plaquetas, y hasta 2GB de memoria RAM. También tiene espacio en su gabinete para 12 módulos de disco *hot-swap* (se pueden quitar y poner aún cuando la máquina está encendida funcionando), fuentes de alimentación y ventiladores redundantes. Se puede apreciar un esquema de esta arquitectura en la Figura 2.3.

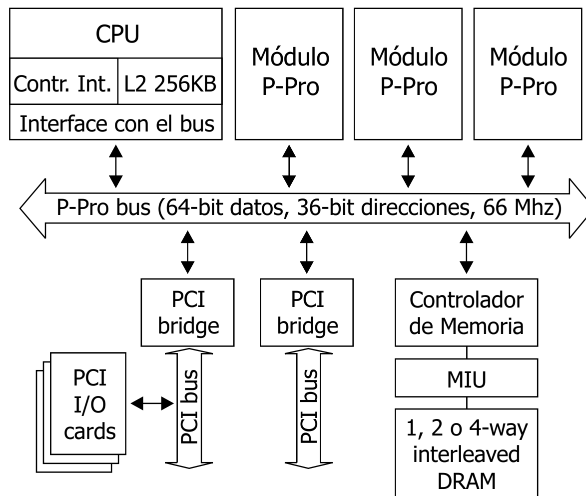

Figura 2.3: Intel Pentium Pro Quad

Arroja 65-75 MFLOPS[4] por procesador, dando un total estimado de 300 MFLOPS.

Sun Enterprise

Está compuesta por un multiprocesador UMA más un *crossbar switch*[5] en un gabinete de hasta 64 CPUs. Al crossbar switch se lo denomina Gigaplane-XB y está incluido en un circuito integrado con 8 slots a cada lado. Cada placa contiene 4 procesadores UltraSPARC de 333 MHz y 4GB de RAM.

[4]MFLOPS denota millones de operaciones de punto flotante por segundo, se adopta como medida de performance en varios benchmarks.

[5]Un *crossbar switch* es una conexión completa entre memorias y procesadores, i. e., cada procesador está conectado a cada una de las memorias.

Dados los rígidos requerimientos de temporizado y la baja latencia del crossbar, el acceso a una memoria externa no es más lento que el acceso a la memoria de la misma placa. Tiene un crossbar switch de 16 x 16 para mover datos entre memoria y caches, como se puede ver en la Figura 2.4. Además, hay 4 buses de direcciones que son utilizados para snooping (c/u se usa para mantener la coherencia de caché en 1/4 del espacio de direcciones). Llega a resolver hasta 51,2 GFLOPS.

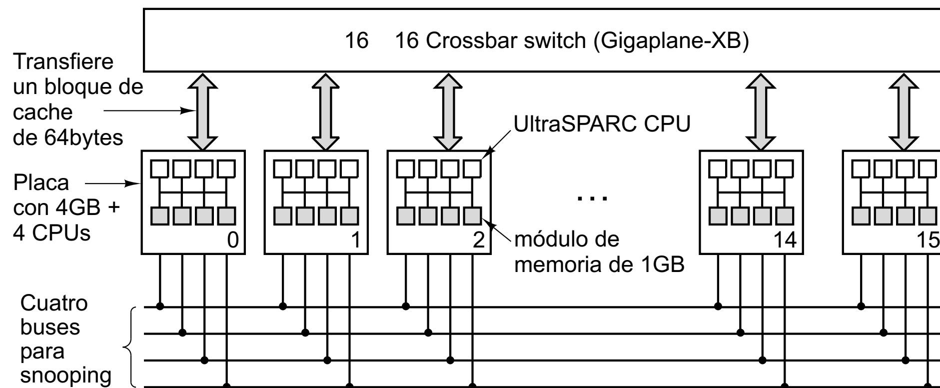

Figura 2.4: Sun Enterprise

2.3.2. Multicomputadoras

Cray T3E

Admite hasta 2048 CPUs por sistema, configuradas en un 3D Torus Full Duplex (ver Figura 2.5). Cada nodo está conectado a otros 6, la tasa de transferencia de estos links es de 480-600 MB/sec. Los nodos también están conectados a uno o más GigaRings, que componen el subsistema de E/S con un gran ancho de banda. Conecta a los nodos con otras redes, discos y otros periféricos.

Las CPUs utilizadas son DEC Alpha 21164:

- procesador superescalar RISC (4 instrucciones por ciclo),
- 64 bits (registros de 64 bits),
- 300 MHz (también hay de 450 y 600),
- 600 MFLOPS,
- L1 cache = 8KB data + 8KB instrucciones,

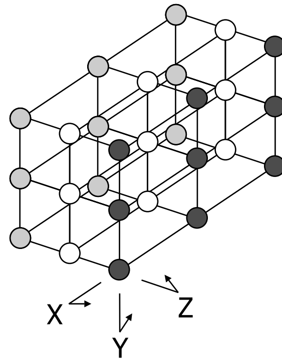

Figura 2.5: Topología *torus* de tres dimensiones

- L2 cache = 96KB unificada 3-way set-associative,
- 128Mb de RAM, expansibles a 2GB (con 2048 = 4TB).

Con 24 elementos de procesamiento (LimitingFactor supercomputer[6]) se obtienen 14,4 GFLOPS. Con la máxima cantidad posible de CPUs se tienen 1228 GFLOPS.

Intel Paragon XP/S

El sistema escala desde 64 a 4000 nodos y se estructura en 104 gabinetes. Posee una memoria RAM total de 1212 GB. Los procesadores son Intel Pentium Pro de 200 MHz; en el diseño se utiliza un total de 9152 procesadores.

Cada procesador arroja 75 MFLOPS. La performance ha variado por mejoras, desde 300 GFLOPS a 1 TFLOPS (con procesadores Xeon). Un ejemplo mejorado y adaptado de esta máquina es la Option Red de Sandia National Laboratories[7] que se detalla a continuación.

Intel Sandia Option Red

La máquina Option Red de Sandia consiste de 4608 nodos organizados en una malla 3D (torus). Los procesadores están incluidos en dos tipos de placas: *kestrel* (cómputo) y *eagle* (network, disk, boot, service). La distribución de los procesadores y las memorias puede observarse en la Figura 2.6.

[6] LimitingFactor supercomputer, Departamento de Ciencias de la Computación de la Universidad Tecnológica de Berlin: `http://swt.cs.tu-berlin.de/LimitingFactor/`

[7] Sandia National Laboratories: `http://www.cs.sandia.gov`

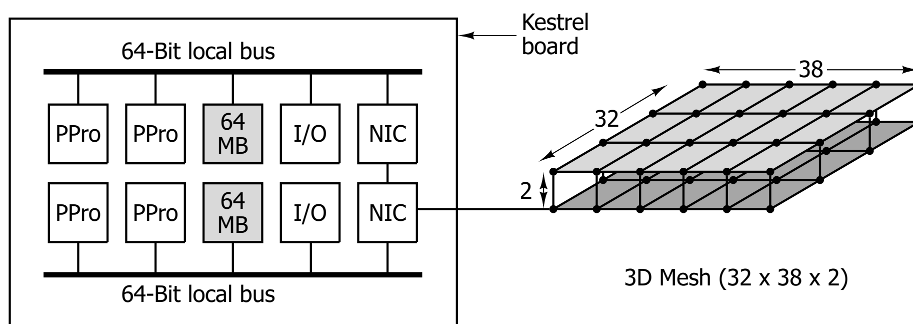

Figura 2.6: Intel Paragon y Sandia Option Red

Lemieux

Lemieux[8] consiste de más de 3000 procesadores y una velocidad máxima de 6 TFLOPS. Comprende 750 nodos Compaq Alphaserver ES45[9].

Cada nodo computacional contiene 4 procesadores Alpha EV6.8CB de 1 GHz y corre un sistema operativo Unix Tru64. Un nodo es un SMP de 4 procesadores, con 4GB de memoria.

Una red de interconexión Quadrics (QsNET) conecta a los nodos (topología fat tree, ver Sección 2.4.2).

2.3.3. Comparación

Teniendo en cuenta las arquitecturas recién descriptas y los procesadores actuales, se graficará su performance sobre una recta logarítmica a modo de comparación (ver Figura 2.7, entre paréntesis se nota el número de procesadores de cada computadora). Un procesador Pentium 4 Prescott de 3.20 GHz con soporte de HyperThreading, disponible para una PC de escritorio, tiene 6,624 GFLOPS hoy en día. Un procesador AMD Opteron de 1.8 GHz tiene 3,042 GFLOPS. Vemos que estos procesadores actuales para computadoras personales aparecen ya en la recta junto con las supercomputadoras de hace algunos años. Lejos se ubica la Cray XT3, uno de los modelos que ofrece actualmente Cray en computación de alta performance[10].

[8]Lemieux, Pittsburgh Supercomputing Center: `http://www.psc.edu/machines/tcs/lemieux.html`
[9]`http://www.psc.edu/publicinfo/news/2003/2003-06-13_interview.html`
[10]`http://www.cray.com/products/index.html`

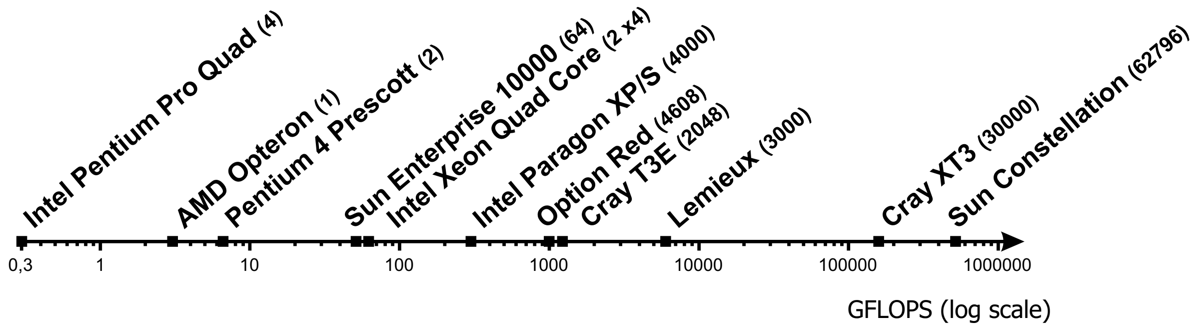

Figura 2.7: Comparación en cantidad de operaciones de punto flotante (GFLOPS)

En el Cuadro 2.1 se pueden apreciar otras características, como la vigencia de los sistemas, la cantidad de CPUs que pueden contener, la velocidad de transferencia en las conexiones y el costo en dólares (calculado para la cantidad de CPUs que se especifica).

Según los datos del cuadro, rápidamente podemos ver que, teniendo un cluster de 8 Pentium 4 tendríamos aproximadamente la misma performance que con la máquina Sun Enterprise 10000 (que posee 24 procesadores, i. e., una cantidad tres veces mayor), a un precio mucho menor. Con dos procesadores físicos AMD Opteron de estas características obtendríamos la misma performance que con un Pentium 4 de 3.20 GHz; aunque obviamente la comparación no es pareja por las velocidades de los procesadores.

PC	**vigencia**	**CPUs**	**GFLOPS**	**MB/s**	**CPUs x U$S**[11]
Intel Pentium Pro Quad	—	≤ 4	0,3	—	—
AMD Opteron Dual Core	2005/2006	1	3	—	U$S 450
Pentium 4 Prescott HT	2005/2006	2	6,6	—	1 x U$S 400
Sun Enterprise 10000	1997–2001	≤ 64	51,2	9930	24 x U$S 13K
Intel Xeon Quad Core[12]	hoy	2	63,5	—	1 x U$S 1200
Intel Paragon XP/S	1992–1996	~ 4000	300	400	—
Cray T3E	1996–1997	≤ 2048	1228	480	6 x U$S 630K
Lemieux	2003–	~ 3000	6000	1300	—
Cray XT3	hoy	~ 30000	159000	7600	2300 x U$S 9,7M
TACC Sun Constellation	hoy	62,796	504000	—	$\sim$ U$S 59M

Cuadro 2.1: Cuadro comparativo de computadoras

Periódicamente se realizan estudios sobre la performance de las supercomputadoras que existen en diferentes laboratorios y organizaciones del mundo. En el sitio **Top 500 Supercomputer Sites** (`http://www.top500.org/`) se publican los resultados y la información más reciente sobre supercomputadoras. La supercomputadora MareNostrum de Barcelona ocupaba el 5to puesto en Junio/2005 (puede encontrar una interesante galería de fotos en: `http://www.bsc.org.es/`). La supercomputadora BigBen[13] del Pittsburgh Supercomputing Center es un ejemplo de sistema Cray XT3, así como Jaguar[14] del Oak Ridge National Laboratory. Para la Sun Constellation System, lo que se reporta es la cantidad de núcleos, cada procesador cuenta con 4 núcleos. Es el último lanzamiento de Sun, y su primer implementación, llamada Ranger, se encuentra emplazada en el Texas Advanced Computing Center.

Lo que se quiere recalcar con esta comparación es que la computadora actual más potente podría haber sido considerada una supercomputadora hace 10 o 20 años, y que por la misma razón, una supercomputadora actual será considerada equipamiento estándar dentro de algunos años.

2.4. Redes de interconexión

Los procesadores y unidades de memoria en computadoras con espacio de direcciones compartido o pasaje de mensajes pueden conectarse usando distintas redes de interconexión. Las redes de interconexión pueden ser clasificadas como estáticas o dinámicas.

Las *redes estáticas* consisten en lazos de comunicación punto a punto entre procesadores, son típicamente usadas para construir computadoras con pasaje de mensajes. Las *redes dinámicas* son construidas utilizando switches y lazos de comunicación. Los enlaces son conectados entre sí dinámicamente por medio de los elementos de conmutación (switches)

[11] Precios en Septiembre/2007. La letra 'K' se utiliza para denotar miles y la 'M' para millones. Para el Pentium 4 y el AMD Opteron se considera principalmente el precio del motherboard y el o los procesadores, en una configuración estándar.

[12] Los resultados se obtienen con 2 procesadores X5355 de 4 núcleos cada uno.

[13] BigBen: `http://www.psc.edu/machines/cray/xt3/bigben.html`. Para mayor información sobre su lanzamiento y sus características, se puede visitar: `http://www.psc.edu/publicinfo/news/2005/2005-07-20-xt3.html`

[14] Jaguar: `http://info.nccs.gov/resources/jaguar`

para establecer caminos entre los procesadores y los bancos de memoria. Son normalmente utilizadas para construir computadoras con espacio de direcciones compartido.

2.4.1. Redes de interconexión dinámicas

Describiremos en esta sección algunas de las redes de interconexión dinámicas más importantes utilizadas en la práctica por arquitecturas con memoria compartida. Sus esquemas se pueden apreciar en la Figura 2.8.

Crossbar Switching: conecta p procesadores (filas) a b bancos de memoria (columnas) a través de una grilla de interconexión. Es una red no bloqueante ya que la conexión de un procesador a una memoria no bloquea la conexión de otro procesador a cualquier otro banco de memoria. La cantidad de conexiones necesarias puede ser costosa.

Bus: los procesadores están conectados a la memoria global a través de un camino de datos compartido, denominado bus. No es escalable a un gran número de procesadores (degrada la performance), aunque puede mejorarse con el uso de caché en cada procesador.

Multistage: es una clase intermedia, conecta los procesadores a los bancos de memoria a través de una serie de etapas. En estas etapas se encuentran los elementos de conmutación. Una red muy usada es la red *omega* (ver Figura 2.9). Consiste de $\log p$ etapas, donde p es el número de procesadores y de bancos de memoria. Cada etapa conecta p entradas a p salidas; el patrón de conexión utiliza la operación de rotación a izquierda en la representación binaria del procesador i. Cada etapa posee $p/2$ elementos de conmutación de 2×2, con dos modos de conexión (directa o cruzada que se establece dinámicamente; ver la línea punteada en la figura). Son redes bloqueantes en cuanto a que el acceso a un banco de memoria puede deshabilitar el acceso a otro banco por otro procesador.

2.4.2. Redes de interconexión estáticas

Las arquitecturas de pasaje de mensajes normalmente utilizan redes de interconexión estáticas para conectar sus procesadores. En esta sección discutiremos las redes de interconexión estáticas más conocidas (ver Figura 2.10) y sus propiedades.

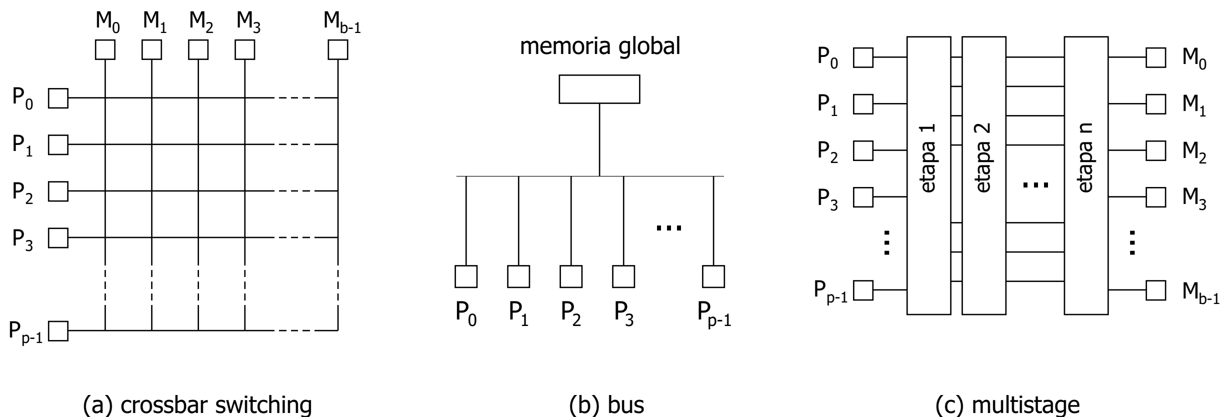

Figura 2.8: Redes de interconexión dinámicas

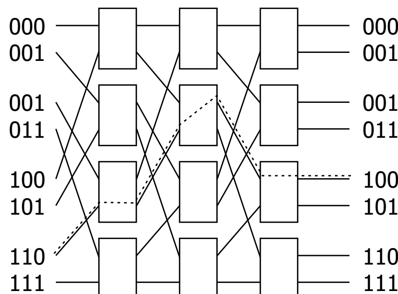

Figura 2.9: Una red omega conectando 8 entradas con 8 salidas [KGGK94]

Completamente conectadas: cada procesador tiene un enlace de comunicación directo a todo otro procesador en la red. Un procesador puede enviar un mensaje a otro procesador en un paso.

Estrella: existe un procesador distinguido que actúa como procesador central. Los demás procesadores tienen un enlace de comunicación que los conecta con el central. La comunicación es ruteada a través del nodo central, que naturalmente puede constituir un punto de congestionamiento.

Arreglo lineal: cada procesador en la red, excepto los de los extremos, tienen una conexión directa con otros dos procesadores, formando una lista. La conexión circular o *wraparound* (entre el último y el primero de la lista) forma un anillo. La comunicación se lleva a cabo pasando repetidamente un mensaje hacia el procesador de la derecha o la izquierda hasta que llega a destino.

Malla: es una extensión del arreglo lineal a dos dimensiones. Cada procesador tiene un lazo de comunicación directo que lo conecta a otros cuatro procesadores. Los procesadores en la periferia puede ser conectados por conexiones wraparound formando un toroide (también llamado *torus*). Los mensajes son ruteados en la malla enviándolos en una dimensión y luego en la otra hasta llegar a destino.

Árbol: existe un único camino entre cada par de procesadores. Redes árbol estáticas poseen un procesador en cada nodo. Redes árbol dinámicas pueden tener elementos de conmutación en los nodos internos y procesadores en las hojas. Para enviar un mensaje, el procesador fuente envía el mensaje hacia arriba en el árbol hasta llegar al procesador o switch en la raíz del menor subárbol que contiene a los procesadores fuente y destino, desde allí el mensaje comienza a bajar hasta su procesador destino. Pueden existir congestionamientos en los niveles más altos del árbol; para prevenirlos existe la posibilidad de definir un *fat tree*, reforzando los lazos de comunicación de los niveles superiores (no sólo existirá una conexión entre los nodos, sino que pueden existir varias para proveer redundancia).

Hipercubo: es una malla multidimensional con dos procesadores en cada dimensión. Un hipercubo d-dimensional consiste de $p = 2d$ procesadores. Un hipercubo puede construirse recursivamente: un hipercubo de dimensión cero es un único procesador; un hipercubo de dimensión $d+1$ se construye uniendo dos hipercubos de dimensión d. Los procesadores de un hipercubo de dimensión d pueden etiquetarse con d bits y cada procesador está directamente conectado a otros d procesadores. El número de lazos de comunicación que existen en el camino más corto entre dos procesadores s y t, coincide con el número de unos de la representación binaria de $s \oplus t$[15].

k-aria d-cubo: es la generalización del hipercubo, donde d es la dimensión de la red y k es el número de procesadores en cada dimensión o la raíz. El número de procesadores en la red es $p = k^d$.

2.5. Modelos de algoritmos paralelos

Para utilizar el gran potencial ofrecido por la tecnología computacional de procesamiento paralelo, se vuelve necesario el desarrollo de algoritmos que puedan mantener

[15] El símbolo $\oplus$ corresponde al operador binario *XOR*.

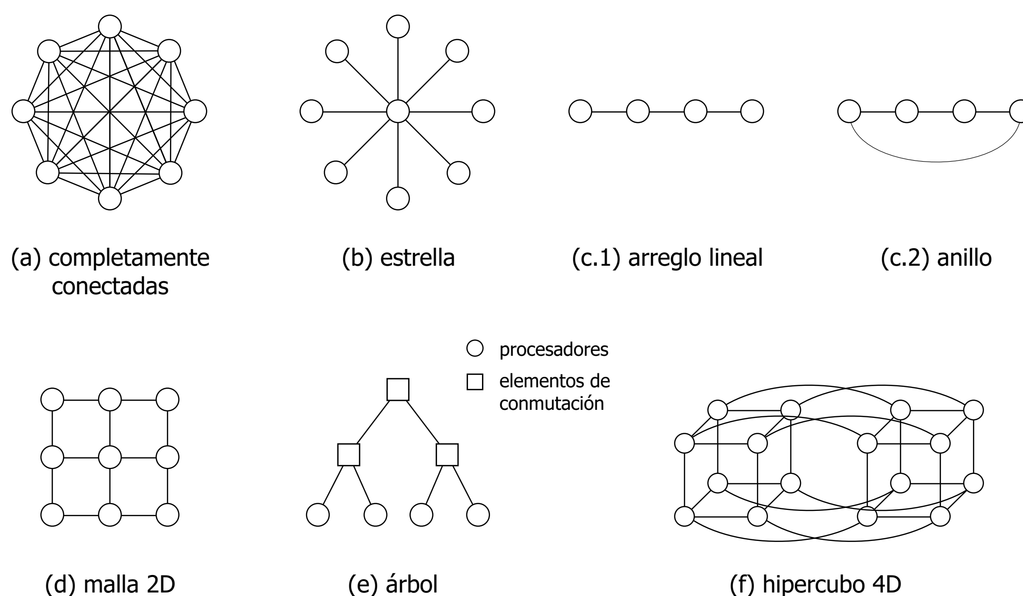

Figura 2.10: Redes de interconexión estáticas

un gran número de procesadores trabajando en paralelo para la completitud de una computación en general. En algunos casos, los algoritmos secuenciales estándar son fácilmente adaptados al caso paralelo. Sin embargo, en la mayoría de los casos, el problema computacional debe ser re-analizado desde su base para desarrollar algoritmos paralelos enteramente nuevos.

2.5.1. Clases de algoritmos

Dentro del campo del diseño de algoritmos paralelos se ha realizado una intensa investigación que abarca distintas áreas de aplicación, incluyendo ordenamiento de colecciones de datos, procesamiento de imágenes y gráficos, solución de ecuaciones lineales, solución de ecuaciones diferenciales y simulación de procesos. Al observar la variedad de algoritmos paralelos que han sido desarrollados hasta el momento, ciertos patrones claves comienzan a emerger, con algunas técnicas simples de organización recurrentes en distintos algoritmos. Se puede ensayar una lista de categorías de estas técnicas [Les93]:

Paralelismo de datos: Un gran número de items de datos distintos están sujetos a un procesamiento idéntico o similar, en paralelo. Este tipo de paralelismo es muy útil en algoritmos numéricos, donde se manejan arreglos o matrices extensas.

Particionamiento de datos: Es un tipo especial de paralelismo en donde el espacio de datos es naturalmente particionado en regiones adyacentes, cada una de las cuales es operada en paralelo por un procesador diferente. Puede existir un intercambio ocasional de datos en las fronteras de las regiones.

Algoritmos relajados: Cada proceso paralelo realiza sus cómputos de manera autosuficiente, sin sincronización o comunicación entre procesos. Estos algoritmos relajados o independientes funcionan bien tanto en multicomputadoras como en multiprocesadores, y pueden dar speed-ups[16] considerablemente altos. A pesar de que los procesadores pueden acceder a datos comunes, no necesitan conocer datos intermedios producidos por otros procesadores, y de allí viene su facilidad de programación.

Iteraciones sincrónicas: Cada procesador realiza la misma computación iterativa sobre una porción diferente de los datos. No obstante, los procesadores necesitan ser sincronizados al final de cada iteración. Esto asegura que ningún procesador comience la próxima iteración hasta que todos los demás hayan terminado de computar la iteración previa. Tal sincronización es necesaria porque los datos producidos por un procesador en la iteración i son utilizados por otros procesadores durante la iteración $i + 1$. Este tipo de algoritmo funciona mejor en multiprocesadores, ya que el tiempo que se requiere en esta arquitectura para la sincronización es pequeño y de esa manera no hay tanta penalización.

Trabajadores replicados: Existe un repositorio central (o *pool*) de tareas similares. A su vez, hay un gran número de procesos trabajadores que recuperan tareas del repositorio, realizan la computación esperada, y posiblemente agregan nuevas tareas al repositorio. La computación global finaliza cuando el repositorio se encuentra vacío y todos los trabajadores están ociosos. Esta técnica es útil en problemas combinacionales, como búsquedas en un árbol o un grafo. En dichos problemas, se genera dinámicamente un gran número de pequeñas tareas a lo largo de la computación. No se conoce de antemano el número de tareas que serán generadas.

Computación en pipeline: En este tipo de algoritmos, se organizan los procesos en una estructura regular, como un anillo o una malla de dos dimensiones. Los datos fluyen a

[16] El *speed-up* de un programa paralelo es el tiempo de ejecución secuencial dividido por el tiempo de ejecución en paralelo.

través de toda la estructura de proceso, asumiendo que cada proceso realiza una fase de la computación en general (análogamente al proceso en una línea de ensamblado). Estos algoritmos funcionan mejor en multicomputadoras, por el patrón ordenado de flujo de datos y por la carencia de datos compartidos que sean globalmente accedidos.

Este es un conjunto posible de clases de algoritmos. Distintos autores realizan diferentes clasificaciones. Por ejemplo, Hansen en [Han95] discute cinco clases de algoritmos paralelos: computación entre todos los pares (producto cruzado), multiplicación de tuplas, dividir y conquistar, ensayos de Monte Carlo y autómatas celulares; como ejemplos de casos generales que luego se aplican a la resolución de problemas con paralelismo.

2.5.2. Problemas típicos

En teoría, muchos algoritmos pueden exhibir una muy buena performance a primera vista, pero cuando son implementados en sistemas reales el speed-up es mucho menor debido a cierta sobrecarga[17] que se presenta en la práctica. Los siguientes problemas limitan la performance de los algoritmos paralelos:

1. *Contención de memoria*: la ejecución del procesador es demorada mientras se espera acceder a una celda de memoria que está siendo actualmente utilizada por otro procesador. Este problema puede existir cuando un número grande de procesadores paralelos comparten valores de datos globales. La contención se da típicamente cuando existe memoria compartida, como en las máquinas multiprocesadores.

2. *Código secuencial excesivo*: en un algoritmo paralelo, siempre habrá porciones de código secuencial donde se realizan ciertos tipos de operaciones centralizadas, como por ejemplo al inicializar las estructuras de datos. Como indica la **Ley de Amdahl**, las porciones de código secuencial pueden afectar severamente el speed-up alcanzable en algoritmos paralelos. Matemáticamente:

$$\textit{Speed-up}_{MAX} = \frac{1}{f + \frac{1-f}{p}}$$

donde f representa la fracción de operaciones que deben ejecutarse secuencialmente ($0 \leqslant f \leqslant 1$) y p es el número de procesadores físicos. Se puede observar que, en la expresión anterior, mientras p tiende a infinito, el speed-up se aproxima a $1/f$.

[17]En esta tesis se utilizarán las palabras *overhead* y *sobrecarga* con significados análogos.

3. *Tiempo de creación de procesos*: en un sistema real, la creación de procesos paralelos requiere de cierta cantidad de tiempo de ejecución. Si los procesos creados son relativamente cortos en duración, el overhead de la creación puede ser mayor que el tiempo ahorrado gracias al paralelismo.

4. *Retardo en las comunicaciones*: este overhead ocurre sólo en multicomputadoras porque los procesadores interactúan entre sí a través del pasaje de mensajes. En algunos casos, la comunicación entre procesos puede requerir que se direccione el mensaje a varios procesadores intermedios en la red de comunicaciones (los que estén en el camino).

5. *Retardo en la sincronización*: cuando los procesos paralelos se sincronizan, generalmente implica que un proceso es forzado a esperar a otro proceso. En algunos programas, las demoras resultantes pueden causar un cuello de botella en la performance y una reducción en el speed-up total.

6. *Desequilibrio en la carga*: en algunos programas, las tareas son producidas dinámicamente de una manera no predecible y deben ser asignadas a los procesadores a medida que son generadas. Esto resulta en la posibilidad de que algunos procesadores estén ociosos mientras otros tienen más tareas computacionales de las que pueden manejar. Este tipo de desequilibrio es común en programas de trabajadores replicados.

2.5.3. Análisis de granularidad de los problemas

Una de las tantas maneras de clasificar un problema en ciencias de la computación es por su grado de paralelismo. Como se ha visto, si un problema se puede dividir en pequeños sub-problemas que pueden ser resueltos por diferentes procesadores en paralelo, se pueden agilizar las computaciones utilizando muchas computadoras y por ende se obtienen grandes speed-ups.

El grado de paralelismo está afectado por la granularidad del problema, que determina cuán independientes son los sub-problemas (y los resultados) entre sí. Si son altamente dependientes se tienen cálculos paralelos con granularidad fina: hay mucha comunicación y sincronización. En la práctica, este tipo de cálculos requieren de una programación

inteligente para sacar el máximo paralelismo posible, y para que la información correcta esté disponible en el tiempo preciso para los procesadores que la necesiten.

En el otro extremo se encuentran los cálculos de granularidad gruesa, donde cada subproblema es independiente de los demás. Este es el caso, por ejemplo de las simulaciones de Monte Carlo, donde cada cálculo se puede hacer independientemente de los demás.

Como regla general, las computaciones de granularidad fina son más aptas para supercomputadoras, o para clusters de computadoras fuertemente acopladas (multiprocesadores o multicomputadoras con redes de alta velocidad), que tienen muchos procesadores idénticos conectados por medio de una red altamente confiable y veloz, para asegurar que no existan cuellos de botella en las comunicaciones. Por otro lado, los cálculos con granularidad gruesa son ideales para redes de computadoras ligeramente acopladas, ya que los retardos que resultan de comunicar los resultados de un procesador no afectarán al trabajo de los demás.

2.6. Resumen

En el capítulo actual hemos definido los conceptos centrales que construyen al paralelismo: desde una discusión de la definición de paralelismo y una descripción de la clasificación de Flynn con ejemplos reales de sistemas, hasta una presentación de modelos de algoritmos y de redes de interconexión.

Se presentaron una serie de ejemplos concretos de computadoras paralelas. Modelos algo antiguos de multiprocesadores, como son el Intel Pentium Pro Quad y la Sun Enterprise, comparados con tecnologías más actuales como el Intel Xeon Quad Core, en cuanto a capacidad y costo. En multicomputadoras se estudió el modelo T3E de Cray y la supercomputadora Lemieux del PSC, entre otros.

Es importante notar que, durante el desarrollo de esta tesis, el aumento de la capacidad de cómputo de estos modelos de supercomputadoras fue sorprendente, partiendo desde los 150 TFLOPS a los 500 TFLOPS, y llegando actualmente al Petaflop (IBM RoadRunner con 1,375 PFLOPS y configuraciones de Cray XT5 con 7 TFLOPS por gabinete). Hoy en día, entre las supercomputadoras más potentes se encuentran la Cray XT5 y la Sun Constellation.

Se vieron además una serie de redes de interconexión estáticas y dinámicas. Todas las supercomputadoras actuales emplean alguna de estas formas de interconexión: e. g., la Cray XT5 emplea un Torus 3D.

Además, como el hardware no funciona aislado, se analizaron varias clases de algoritmos que surgen a partir de la agrupación de técnicas simples de organización, recurrentes en distintos problemas. También se postularon las consideraciones que siempre están presentes al programar con paralelismo: contención de memoria, cantidad de código secuencial presente, tiempo de creación de procesos, retardos en la comunicación, retardos en la sincronización y balance de carga. Finalmente se realizó un análisis de granularidad de los problemas, recalcando cuáles resultan más adecuados para cada tipo de computadora.

En lo que sigue haremos continua referencia a términos que fueron introducidos aquí.

Capítulo 3

Paralelismo en lenguajes imperativos

En la actualidad, el mundo de la programación en paralelo es diverso y complejo. Un gran número de sistemas nuevos han sido desarrollados durante los últimos años, pero pocos se encuentran en uso en la práctica. Muchos de estos nuevos sistemas alimentaron una gran promesa, pero no escalaron bien en las aplicaciones del mundo real. Otros nunca salieron del estado de test. Los pocos sistemas que se utilizan en aplicaciones reales tienen en su mayoría cerca de diez años de antigüedad (como MPI o HPF) y están usualmente basados en lenguajes que son más viejos aún (C, C++, Fortran) [SL05].

En este capítulo, se analizarán algunos de los lenguajes de programación en paralelo más conocidos, prestando especial atención a las primitivas que proveen para el manejo de paralelismo y a la arquitectura de hardware en que se inspiraron. Veremos cuáles se trasladan a aplicaciones del mundo real, cuáles han sido desarrollados en el pasado y en cuáles se trabaja en este momento. Los compararemos y destacaremos los puntos fuertes y débiles de cada uno.

Se presentarán a HPF y OpenMP como ejemplos de lenguajes de alto nivel, y a PVM y MPI representando a los lenguajes de bajo nivel. Para cada uno de los lenguajes se implementará el algoritmo de Rank Sort como punto de partida para las comparaciones en cuanto a facilidad de programación, claridad y nivel de abstracción.

3.1. Clasificación de lenguajes paralelos

Como se vio en el Capítulo 1, en paralelismo se clasifica a los lenguajes de programación en tres grandes grupos [ST98, To95, Zav99]:

(1) *Lenguajes con paralelismo implícito completamente abstractos*: en este grupo se incluyen los enfoques en los cuales el programador está completamente aislado de la idea de ejecución paralela. Todos los pasos para crear un programa paralelo están a cargo de compiladores y sistemas de soporte en tiempo de ejecución. Este grupo incluye a los paradigmas lógico (ver Capítulo 4) y funcional, y a compiladores que paralelizan código imperativo.

(2) *Lenguajes con paralelismo explícito de alto nivel*: incluye a los sistemas en los que el programador participa en algunas etapas del paralelismo.

(3) *Lenguajes con paralelismo explícito de bajo nivel*: incluye a los sistemas en los que el programador maneja explícitamente todos los problemas relacionados con el paralelismo.

En este capítulo nos enfocaremos en los dos últimos tipos de lenguajes, aplicados a lenguajes imperativos o procedurales.

En los lenguajes con paralelismo explícito de alto nivel se adopta un estilo de programación donde el programador es consciente del paralelismo y puede contribuir en el desarrollo de una implementación eficiente. Los métodos que se incluyen en esta clase esconden la mayoría de los detalles de la ejecución paralela de bajo nivel. Ejemplos de sistemas paralelos de este tipo incluyen [Zav99]:

- *Lenguajes con paralelismo de datos*[1], que generalmente introducen operadores masivos para modificar los tipos de datos paralelos. NESL es un lenguaje funcional con operadores paralelos explícitos, y está ligeramente basado en ML. Un ejemplo de operador masivo que introduce el lenguaje podría ser `apply-to-each` que tiene la semántica de un mapeo funcional. Un ejemplo de lenguaje data-parallel imperativo es C⋆.

[1]Estos lenguajes en inglés se denominan "*data-parallel*".

- *Extensiones de lenguajes imperativos*, que usualmente extienden el lenguaje secuencial con un conjunto de directivas para paralelización de código. Comparado con la aproximación implícita, no se necesita un análisis de dependencias ni una distribución automática de los datos. La mayoría de los lenguajes en esta clase son extensiones de lenguajes conocidos, como Fortran, C y otros.

- *Lenguajes de esqueletos* [DFH+93], que requieren que el programador especifique la estructura del paralelismo usando un conjunto fijo de patrones de paralelismo predefinidos. La implementación con paralelismo del programa se genera automáticamente componiendo conjuntos de plantillas optimizadas (*templates*). Los lenguajes de esqueletos son usualmente conocidos como un método portable de programación, ya que no asumen ninguna característica particular sobre la máquina en la que se ejecutará el programa. La solución a la portabilidad es rediseñar las plantillas para cada arquitectura destino. Ejemplos de lenguajes de esqueletos son SkelML [Bra93], SCL [DGTY95], Skil [BK96] y P3L-SKIE [BDO+93, BCP+98, DPV95].

La clase de lenguajes con paralelismo explícito de alto nivel también incluye los lenguajes lógicos como Parlog [RR88] (ver Sección 4.5.3) y Concurrent Prolog. Estos lenguajes básicamente extienden el lenguaje lógico estándar (i. e., Prolog) para que los programadores puedan utilizan anotaciones a fin de especificar qué cláusulas se resolverán en paralelo.

Se tienen, por otro lado, los lenguajes con paralelismo explícito de bajo nivel. Ellos son modelos de programación en paralelo que requieren un manejo de problemas de bajo nivel por parte del programador, como mapeo, comunicación y sincronización de las tareas paralelas para asegurar performance y correctitud. Los dos grupos de sistemas que se mencionan en esta clase son:

- *Sistemas de pasaje de mensajes*: el pasaje de mensajes es una forma de comunicación entre procesos utilizada en sistemas concurrentes, paralelos y distribuidos y en programación orientada a objetos. La comunicación se lleva a cabo mediante el envío de mensajes a los destinatarios especificados. El envío y la recepción de mensajes se caracteriza por primitivas comunes de send/receive, y puede ser sincrónico (bloqueante) o asincrónico (no bloqueante). Existen formas especiales de mensajes para invocación de funciones, señales y paquetes de datos. Existen varios sistemas que utilizan pasaje de mensajes, como por ejemplo Occam, PVM y MPI.

- *Sistemas multithreaded*: en un multiprocesador, los *threads*[2] pueden ser utilizados para explotar el paralelismo e incrementar la performance, ya que cada thread podría ejecutarse en paralelo en un procesador diferente. Incluso múltiples threads pueden ser útiles en un uniprocesador, dado que el procesador podría evitar esperas intercalando la ejecución de ellos (algunos procesando entrada/salida mientras otros realizan cómputos en background). Usualmente utilizan un modelo de memoria compartida para la comunicación, i. e., simplemente pueden intercambiar información leyendo y escribiendo datos en éstas áreas de memoria. Ejemplos populares de sistemas multithreaded son Cilk y Java threads, que permiten al programador crear sus threads, asignarles tareas y destruirlos cuando finalizan de computar.

Estos sistemas de bajo nivel, aunque son más difíciles de usar, exhiben características interesantes en cuanto a efectividad. Los sistemas de pasaje de mensajes son generalmente más eficientes en sistemas con memoria distribuida, y los sistemas multithreaded en arquitecturas con memoria compartida.

3.2. Lenguajes y librerías de paralelismo analizadas

Muchos de los lenguajes de programación para computadoras con memoria compartida o pasaje de mensajes son esencialmente lenguajes para programación secuencial aumentados con un conjunto de llamadas al sistema especiales. Estas llamadas al sistema incluyen primitivas de bajo nivel para pasaje de mensajes, sincronización de procesos, creación de procesos, exclusión mutua y otras funciones necesarias [KGGK94].

Estas primitivas pueden clasificarse de la siguiente manera:

- *Creación de procesos*: las primitivas de creación de procesos incluyen a las funciones para creación dinámica de procesos, identificación de tareas y detección de terminación, entre otras.

[2]Un *thread*, también conocido como 'proceso liviano', es la unidad básica de utilización de CPU; comprende un identificador de thread, un contador de programa, un conjunto de registros y una pila. Comparte con otros threads que pertenezcan al mismo proceso la sección de código, datos y otros recursos del sistema operativo, como archivos abiertos y señales.

- *Comunicación*: las primitivas de comunicación incluyen a las funciones para enviar y recibir mensajes, o a funciones para el manejo de variables de alto nivel que se utilicen en las comunicaciones.
- *Sincronización*: las primitivas de sincronización incluyen a las funciones para controlar el acceso a los datos compartidos. También puede utilizarse operaciones de comunicación bloqueantes para sincronización.
- *Computación global*: las primitivas de computación global aplican operadores a datos provistos por distintos procesos. Son más útiles en multicomputadoras.
- *Transferencia de datos*: las primitivas de transferencia de datos permiten recopilar o distribuir datos entre los procesadores.

En este capítulo se analizarán HPF (High Performance Fortran [Hig97]), OpenMP [Ope02], MPI (Message Passing Interface [ERL97, GKP96]) y PVM (Parallel Virtual Machine [ERL97, GBD+94]), los cuales constituyen extensiones a lenguajes imperativos (Fortran o C). Cada uno de ellos aborda de diferente manera el problema de programar en paralelo. Se presentará, a modo de introducción, una pequeña ficha técnica de cada uno:

HPF: High Performance Fortran (1993)

- Definición: es una extensión de Fortran 90.
- Creador: HPF Forum (HPFF), un grupo formado por fabricantes de software y hardware e instituciones académicas de todo el mundo.
- Arquitecturas: SIMD y MIMD (multiprocesadores).
- Modelo: paralelismo de datos con espacio de direcciones compartido, distribuido convenientemente en las memorias de los distintos procesadores; el programador especifica un comportamiento paralelo abstracto y el compilador debe implementar estas directivas en la máquina destino.

OpenMP: Open specifications for Multi Processing (1997)

- Definición: es una API basada en directivas que soporta programación paralela en máquinas con memoria compartida, multiplataforma.

- Creador: OpenMP Architecture Review Board.
- Arquitecturas: MIMD (multiprocesadores).
- Modelo: utiliza el modelo fork-join para ejecución paralela (multithreaded).

PVM: Parallel Virtual Machine (1989)

- Definición: es un sistema de pasaje de mensajes que permite que una red de computadoras pueda ser utilizada como una única computadora paralela con memoria distribuida.
- Creadores: V. Sunderam y A. Geist; principalmente desarrollado en Oak Ridge National Labs y en la Universidad de Tennessee.
- Arquitecturas: MIMD.
- Modelo: el modelo de programación se basa en un conjunto de programas secuenciales que se comunican con otros procesos intercambiando mensajes; el conjunto de hosts componen la máquina virtual.

MPI: Message Passing Interface (1993/1994)

- Definición: es una especificación de una librería de pasaje de mensajes para computación paralela.
- Creador: MPI Forum.
- Arquitecturas: MIMD.
- Modelo: utiliza el modelo de pasaje de mensajes para proveer una forma portable y eficiente de comunicación en MPPs, clusters de SMP o estaciones de trabajo y redes heterogéneas.

Nivel de abstracción	Alto		Bajo	
Data-parallel	HPF			
Shared-Memory		OpenMP		MPI-2
Message-Passing			PVM	MPI

Cuadro 3.1: Los lenguajes que se analizan y su clasificación

En el Cuadro 3.1 se clasifica a los lenguajes o librerías de acuerdo a su nivel de abstracción y clase. Tanto HPF como OpenMP se consideran lenguajes de alto nivel, siendo el primero orientado a los datos y el segundo a memoria compartida. Se clasifican como de bajo nivel a PVM y MPI. Ambos utilizan formas de pasaje de mensajes, salvo que en las extensiones de MPI, i. e. MPI-2, se incorporan algunas primitivas para manejar memoria compartida (RMA). El Cuadro 3.2 resume las principales primitivas de cada uno según la clasificación vista con anterioridad. En las secciones siguientes se explicarán en detalle cada uno de ellos, para luego finalizar mostrando un ejemplo unificado que completa el entendimiento acerca del uso de las primitivas.

	Creación de procesos	*Comunicación*	*Sincronización*	*Computación global*	*Transf. de Datos*
HPF	FORALL INDEPENDENT (compilador)	—	DATE_AND_TIME SYSTEM_CLOCK ...	SCATTER PREFIX SUFFIX	SCATTER
OpenMP	PARALLEL PAR. FOR PAR. IF SECTIONS MASTER	—	CRITICAL BARRIER ATOMIC	—	—
PVM	SPAWN	SEND MCAST BCAST RECV [UN]PACK	BARRIER	REDUCE	—
MPI	(estática)	SEND RECV TEST WAIT BCAST	BARRIER	REDUCE ALLREDUCE SCAN	SCATTER GATHER

Cuadro 3.2: Resumen y clasificación de primitivas en los distintos lenguajes analizados

3.2.1. HPF

HPF (High Performance Fortran) es un conjunto de extensiones de Fortran 90 que permite a los programadores especificar cómo los datos serán distribuidos a través de

múltiples procesadores en un ambiente de programación paralelo. Fue definido por el High Performance Fortran Forum (HPFF), una coalición de representativos del ámbito industrial, académico y de laboratorios gubernamentales estadounidenses [Hig97]. Las extensiones de HPF proporcionan acceso a las características de arquitecturas de alto rendimiento manteniendo la portabilidad entre diferentes plataformas.

El modelo fundamental de paralelismo en HPF es el de paralelismo de datos con un espacio de direcciones compartido. HPF utiliza el paradigma SIMD (Single Instruction Multiple Data) que consiste en ejecutar el mismo conjunto de instrucciones en diferentes procesadores, cada uno de los cuales tiene memoria independiente en la que se almacena la fracción de los datos que va a ser utilizada con mayor frecuencia por el respectivo procesador.

El aprovechamiento de la localidad de datos es crítico para obtener un buen rendimiento en una computadora de alta performance, ya sea un uniprocesador, una red de computadoras o una computadora paralela. En una computadora paralela NUMA (Non Uniform Memory Access), una distribución efectiva de los datos entre las memorias de los procesadores es muy importante cuando se busca reducir el overhead por operaciones de movimiento o transferencia de datos. Una de las características claves de HPF es la facilidad que brinda al usuario para que especifique el mapeo de los datos (particionamiento). HPF provee una vista lógica de la computadora paralela como un arreglo rectilíneo de varios procesadores abstractos en una o más dimensiones. El programador puede especificar el alineamiento relativo de los elementos de los arreglos o vectores del programa, y la distribución del arreglo sobre la grilla lógica de procesadores.

La Figura 3.1 resume el modelo de mapeo de datos de HPF. El mapeo de datos se especifica a través de directivas que ayudan al compilador a optimizar la performance, pero que no tienen efecto en la semántica del programa. Este proceso se realiza en varios pasos, el primer paso relaciona un grupo de datos con respecto a otro dato, mediante el uso de las directivas `ALIGN` y `REALIGN`. El segundo paso consiste en distribuir los datos en los procesadores mediante las directivas `DISTRIBUTE` o `REDISTRIBUTE`. Esta distribución se realiza según la disposición lógica de los procesadores que se establece con la directiva `PROCESSORS`. Estas directivas y otras funciones importantes se describirán a continuación.

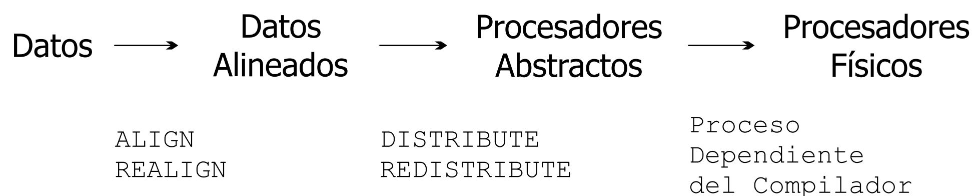

Figura 3.1: Modelo de mapeo de datos de HPF

Creación de procesos en HPF:

- **FORALL:**

  `FORALL <forall_header> <forall_assignment>`

  e. g., `FORALL (I=1:3,J=1:3) A(I,J)=A(J,I)`

  La sentencia y el constructor `FORALL` de HPF están diseñados para permitir la paralelización de asignaciones; de esta manera, se pueden paralelizar las iteraciones de un bucle.

  En las sentencias de asignación, se evalúa el lado derecho de la expresión para todos los valores del índice, y luego se asignan los resultados a lo referenciado en el lado izquierdo. Una asignación de arreglos no debe causar que al mismo elemento del arreglo se le asigne más de un valor. Sin embargo, en diferentes sentencias se puede realizar más de una asignación sobre un elemento de un arreglo, o las asignaciones de una sentencia pueden afectar a una sentencia posterior.

- **INDEPENDENT:**

  `!HPF$ INDEPENDENT`

  Esta directiva indica que las operaciones en un bucle `DO` o en una sentencia o constructor `FORALL`, pueden ser ejecutadas en cualquier orden. Precediendo a una sentencia `DO`, informa al compilador que las operaciones dentro del bucle pueden ser ejecutadas en cualquier orden y de forma independiente sin cambiar la semántica del programa. Precediendo a una sentencia `FORALL`, informa al compilador que los diferentes valores de los índices (todos activos al mismo tiempo) no interfieren entre sí.

  Como el compilador no está en capacidad de determinar cuáles lazos son independientes y cuáles no lo son, el programador dispone de la directiva `INDEPENDENT`

para realizar anotaciones y evitar la necesidad de que el compilador incorpore un analizador de dependencias.

- **PURE:**

```
PURE <return_type> FUNCTION <name_and_parameters>
PURE SUBROUTINE <name_and_parameters>
```

En HPF una función o una subrutina *pura* es un procedimiento que no produce efectos colaterales y no hace referencia a otras variables que no sean las mapeadas en sus argumentos. Esto implica que el único efecto de una *función* pura en el estado del programa es el retorno de un valor, y el único efecto de una *subrutina* pura en el estado del programa es la modificación de sus parámetros `INTENT(OUT)` e `INTENT(INOUT)`[3].

Las funciones definidas por el usuario pueden ser llamadas dentro del constructor `FORALL` sólo si son funciones puras. Las subrutinas que se llaman desde funciones puras deben ser puras. Este atributo se requiere solamente para funciones que se llaman desde un `FORALL`. Las funciones que se llaman desde sentencias de asignación de arreglos o bucles `INDEPENDENT` no necesitan ser puras.

Distribución de datos:

- **ALIGN:**

```
!HPF$ ALIGN <array_of_data> WITH <array_of_data>
```

Generalmente los arreglos de un programa están relacionados entre sí, por lo que al momento de distribuirlos en distintos procesadores será importante que las partes de los arreglos que estén relacionadas sean accesibles rápidamente por el procesador local. La directiva `ALIGN` se utiliza para informar al compilador sobre la existencia de estas relaciones. La alineación de los datos no especifica el procesador en el cual los datos serán distribuidos, sólo indica qué datos serán distribuidos en el mismo procesador. El propósito de esta alineación es producir código más eficiente, reduciendo las comunicaciones.

Además, la directiva `ALIGN` puede utilizarse para generar réplicas de los datos que son frecuentemente consultados en el programa.

[3]Es decir, sus parámetros de salida y de entrada/salida, respectivamente. Los parámetros de sólo entrada se marcan como `INTENT(IN)`.

- **DISTRIBUTE:**

  `!HPF$ DISTRIBUTE (BLOCK|CYCLIC) :: <array_of_data>`

  Esta directiva define la forma en que los arreglos son distribuidos sobre el arreglo abstracto de procesadores. La distribución se realiza por bloques. El tamaño de los bloques puede ser especificado directamente o calculado automáticamente en función del número de procesadores disponibles. Todos los mapeos de datos de HPF están construidos a partir de varias combinaciones de dos tipos básicos de distribución paralela: `BLOCK` y `CYCLIC`. En la distribución `BLOCK` de un arreglo unidimensional, los elementos del arreglo son distribuidos en cada procesador en grandes bloques. En la distribución `CYCLIC`, los elementos del arreglo de datos son asignados en forma cíclica (en orden *Round-Robin*) a cada uno de los procesadores.

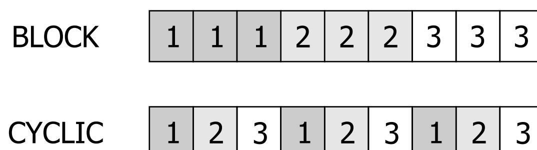

Figura 3.2: Distribución `BLOCK` y `CYCLIC` de un arreglo en 3 procesadores

  Además, HPF tiene dos directivas que permiten que los datos sean distribuidos en tiempo de ejecución: `REDISTRIBUTE` y `REALIGN`.

- **PROCESSORS:**

  `!HPF$ PROCESSORS <processor_name> <shape_and_number_of_processors>`
  e. g., `PROCESSORS P(100)`

  Declara una grilla abstracta de procesadores. Si el número de procesadores se omite, se asume un procesador escalar. Si el número de procesadores tiene una dimensión, se tendrá un arreglo de procesadores; en cambio, si el número de procesadores es un par ordenado, se tendrá una grilla de procesadores. Los procesadores no necesitan tener la misma forma que en el hardware subyacente, el mapeo lo hace el compilador.

Sincronización:

Las siguientes rutinas fuerzan automáticamente la sincronización de los procesadores cuando son llamadas desde rutinas globales de HPF: `DATE_AND_TIME`, `SYSTEM_CLOCK`, `DATE`, `IDATE`, `TIME` y la función `SECNDS`.

Computación global:

- **SCATTER, PREFIX, SUFFIX:**

  ```
  X_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
  ```

  Esencialmente, estas rutinas devuelven un resultado que es igual a la `BASE` operada con alguna combinación de los elementos del arreglo. Estos elementos se envían a los destinos que define el arreglo `INDX` en coordenadas. Los valores que se envían a un mismo elemento de un arreglo son combinados por una operación `X` que puede ser cualquiera de las funciones de reducción de Fortran 90 (`ALL`, `ANY`, `COPY`, `COUNT`, `MAXVAL`, `MINVAL`, `PRODUCT` y `SUM`) o las funciones de reducción de HPF (`IALL`, `IANY`, `IPARITY`, `PARITY`).

  Además, existen operaciones prefijo y sufijo de arreglos. Una función prefijo aplica una operación a lo largo del vector, tal que cada elemento es el resultado de las operaciones realizadas sobre los elementos que lo preceden. La función sufijo opera en dirección diferente a la operación prefijo, i. e., el resultado en una posición dada depende de los elementos que lo suceden. Las operaciones permitidas son las mismas que en el caso de la función `SCATTER`.

En la práctica, al utilizar las directivas de HPF el programador especifica un comportamiento paralelo abstracto y el compilador debe, de alguna manera, implementar estas directivas en la máquina destino. Existen compiladores comerciales de subconjuntos del estándar de HPF para muchas plataformas paralelas como la Cray T3E, SGI Origin, IBM SP2, Sun Ultra Enterprise, entre otras.

3.2.2. OpenMP

OpenMP [Ope02] es una API que soporta programación paralela en máquinas con memoria compartida, multiplataforma. Soporta todas las arquitecturas, incluyendo Unix y Windows NT. Fue conjuntamente definido por un grupo de fabricantes de hardware y software; es portable, escalable y flexible. Las directivas de compilación, funciones de librería y variables de entorno definidas en OpenMP son usadas para crear y manejar programas paralelos permitiendo portabilidad. Las directivas extienden los lenguajes base C, C++ y Fortran con constructores SPMD (Single Program Multiple Data), primitivas

para división de tareas y sincronización, y soporte para datos compartidos y privados. Las directivas siempre se aplican a la siguiente sentencia, por ejemplo:

```
#pragma omp ...
    <sentencia>

#pragma omp ...
    { sentencia1; sentencia2; sentencia3; }
```

OpenMP soporta programación explícita, ya que el usuario especifica las acciones que deben ser tomadas por el compilador y el sistema en tiempo de ejecución para ejecutar el programa en paralelo. La compilación de OpenMP no chequea dependencias, conflictos, condiciones de carrera u otros problemas. El compilador tampoco es responsable de generar una paralelización automática. El usuario es responsable de usar las funciones de librería y las directivas para generar el paralelismo y producir programas confiables.

En el modelo de ejecución `FORK-JOIN` que es utilizado por OpenMP los programas empiezan con un único thread (*master thread*). El master thread se ejecuta en forma secuencial hasta llegar a la primera región paralela. Luego, este thread crea nuevos threads para ejecutar las instrucciones del programa que se encuentran dentro de la región paralela. Al final de las regiones paralelas, existe una barrera implícita que fuerza la sincronización de los threads. Esto es, cuando un thread acaba con la ejecución de las instrucciones a su cargo, espera la sincronización con el resto y termina, y sólo el master thread continúa la ejecución. Este esquema de regiones paralelas puede darse varias veces dentro de un mismo programa.

Dentro del modelo de memoria compartida, OpenMP provee consistencia relajada. Todo thread tiene acceso a memoria para guardar o recuperar variables. Además de la memoria central compartida, cada thread tiene una porción temporal de la memoria (almacenamiento temporal privado, cacheable) y un espacio privado (thread-private) no accesible desde otros threads. A continuación se describirán algunas directivas (encabezadas con "`#pragma omp`") y funciones destacadas.

Creación de procesos en OpenMP:

- **PARALLEL:**

```
#pragma omp parallel
```

Se crea cierto número de threads que ejecutan el mismo código. Cada thread espera al final. Es bastante similar a la funcionalidad de un conjunto de primitivas create/join en Pthread (la librería POSIX clásica de C).

- **PARALLEL FOR:**

```
#pragma omp parallel
    #pragma omp for
    for(...) {...}
```

Se crea cierto número de threads que ejecutan un subconjunto de las iteraciones del bucle `FOR`. Todos los threads esperan al final del `parallel for` (fuerza una sincronización).

```
#pragma omp parallel
    #pragma omp for nowait
    for(...) {...}
```

Existe un modificador especial, `nowait`, que permite que los threads continúen sin esperar a que los demás terminen.

```
#pragma omp parallel for
    for(...) {...}
```

Es una directiva especial para abreviar las primeras dos.

- **PARALLEL/FOR IF:**

```
#pragma omp parallel if (expression)
#pragma omp for if (expression)
#pragma omp parallel for if (expression)
```

Se ejecutan en paralelo si la expresión es verdadera, sino se ejecutan secuencialmente.

- **PARALLEL SECTIONS:**

```
#pragma omp parallel
{
#pragma omp sections
    {
    #pragma omp section {...}
    #pragma omp section {...}
    ...
    }
}
```

Esta directiva identifica un constructor no iterativo para dividir tareas. Cada sección es ejecutada una vez por un thread en el grupo.

- **MASTER:**

```
#pragma omp master
```

Identifica un constructor que especifica un bloque de sentencias que será ejecutado por el master thread del grupo. Los otros threads en el grupo no ejecutan el bloque asociado.

Identificación de threads:

```
int omp_get_thread_num()
int omp_get_num_threads()
```

Devuelven el identificador del thread y el número total de threads, respectivamente. Se pueden utilizar para hacer que los threads realicen tareas diferentes. Por ejemplo, para el caso de un master y un worker:

```
#pragma omp parallel
{
    if (!omp_get_thread_num())
        master();
    else
        slave();
}
```

Número de Threads:

```
#pragma omp parallel num_threads(int)
void omp_set_num_threads(int)
```

Cualquiera de estas dos funciones permite indicar el número de threads que se crearán. La directiva que se transcribe en primer lugar define una sección paralela junto con el número de threads que se crearán en ella para ejecutar las sentencias que contenga. La segunda función setea un valor que será usado en las directivas `parallel` que sigan, si es que éstas no contienen la cláusula `num_threads`.

Sincronización:

- **CRITICAL:**

```
#pragma omp critical
```

Esta directiva restringe la ejecución del bloque asociado a un único thread en un momento dado. Un thread espera al inicio de la sección crítica hasta que ningún otro thread se encuentre ejecutando código de la misma.

- **BARRIER:**

```
#pragma omp barrier
```

Sincroniza todos los threads de un grupo. Cuando un thread llega a la barrera, espera a que todos los demás threads en el grupo alcancen este punto. Después de que todos los threads encontraron la barrera, cada thread comienza a ejecutar las sentencias que se ubican después de la directiva de barrera.

- **ATOMIC:**

```
#pragma omp atomic
    (expression)
```

Asegura que la ubicación de memoria específica es actualizada atómicamente. Las expresiones que se permiten incluyen a operadores binarios en asignaciones simples, como `a++` o `p[i] -= 5`.

Además, existen otras directivas como `single` (especifica una región que será ejecutada por un solo thread) o `flush` (que fuerza la consistencia de memoria) que pueden ser emuladas con otras directivas. OpenMP también dispone de cláusulas para manejar datos

compartidos o privados para los threads, como `private`, `shared`, `default`, `reduction` y `copyin`, entre otras.

Para sincronización también provee el tipo `omp_lock_t` que implementa un semáforo, con funciones para adquirir y liberar un lock.

Intel ofrece un compilador de alta performance de C++/Fortran con soporte para las directivas de OpenMP, desarrollado con el fin de aprovechar su tecnología de Hyper-Threading [TBG+02]. Esta tecnología permite que un único procesador maneje los datos como si fueran en realidad dos procesadores, ejecutando las instrucciones de distintos threads en paralelo. Los procesadores que cuentan con Hyper-Threading pueden aumentar la performance de las aplicaciones con un alto grado de paralelismo. Sin embargo, para que esta ganancia potencial sea posible, es necesario que las aplicaciones sean multithreaded, ya sea por medio de técnicas de paralelización manuales, automáticas o semiautomáticas.

3.2.3. PVM

PVM (Parallel Virtual Machine) [GBD+94] permite el desarrollo de aplicaciones en un conjunto de computadoras heterogéneas conectadas por una red, que aparenta ser una única computadora paralela para los usuarios. Ofrece un conjunto poderoso de funciones para control de procesos y manejo dinámico de recursos. El sistema PVM utiliza el modelo de pasaje de mensajes para permitir que los programadores puedan realizar una computación distribuida a lo largo de una gran variedad de tipos de computadoras, incluyendo MPPs. El modelo de programación se basa en un conjunto de programas secuenciales, que se comunican con otros procesos intercambiando mensajes. PVM maneja de forma transparente el ruteo de mensajes, conversión de datos y tareas de planificación en una red compuesta por computadoras cuyas arquitecturas son incompatibles.

El usuario escribe sus programas en PVM como un conjunto de tareas cooperativas. Cada tarea progresa alternando entre etapas de cómputo y etapas de comunicación. Las tareas acceden a los recursos de PVM a través de una librería estándar de rutinas. Estas rutinas permiten la iniciación y terminación de tareas sobre la red, así como la comunicación y sincronización entre tareas. Las tareas se ejecutan en un conjunto de máquinas seleccionadas por el usuario para una corrida específica del programa. El pool de hosts, i. e., los hosts que conforman la máquina virtual, puede ser alterado agregando y

eliminando máquinas durante la operación (una característica importante para tolerancia a fallas).

PVM tiene dos componentes: un demonio[4], que debe correr en cada host que forme parte de la máquina virtual, y una librería de rutinas. Esta última contiene las rutinas que pueden ser llamadas por el usuario para pasaje de mensajes, creación de procesos, coordinación de tareas y modificaciones de la máquina virtual. Antes de correr una aplicación se debe ejecutar y configurar la máquina virtual, que para tres procesadores podría lucir como en la Figura 3.3. Cada programa es compilado individualmente para cada host (o al menos para cada arquitectura), y ubicado en un lugar accesible desde otros hosts.

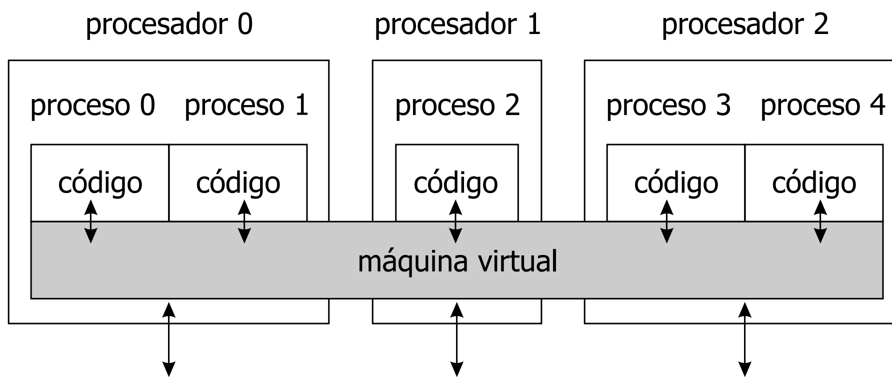

Figura 3.3: Modelo de máquina virtual de PVM

Como ya se mencionó, la comunicación en PVM se realiza a través del pasaje de mensajes. Esto se realiza utilizando la librería de rutinas y el demonio. Los mensajes que tienen como origen y destino a un proceso local se rutean contactando sólo al demonio local, que transmite el mensaje al otro proceso. Cuando el proceso receptor del mensaje se encuentra en otro procesador, el demonio local contacta al demonio que corre en el host remoto y realiza la transferencia, para que luego el demonio remoto entregue el mensaje al proceso correspondiente. La comunicación por defecto es asincrónica y buffered.

En los párrafos que siguen se hará un repaso de las rutinas más utilizadas al programar con PVM.

[4] Término común en sistemas UNIX donde en general se utiliza la palabra inglesa *daemon*.

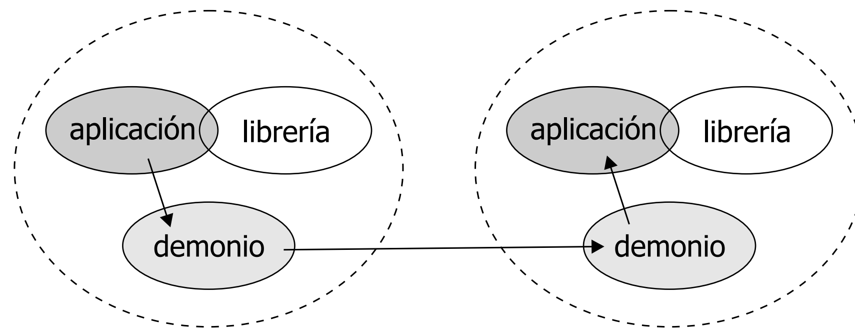

Figura 3.4: Comunicación entre tareas en PVM

Creación de procesos en PVM:

- **SPAWN:**

```
int pvm_spawn(hijo, argumentos, flag, lugar, cantidad, &tids)
```

Se utiliza en la creación dinámica de tareas para crear uno o más procesos. La función especifica el nombre del programa que se iniciará en el/los hijo(s) mediante el argumento `hijo`. Así también puede enviarle los argumentos en el segundo parámetro. Los identificadores de los procesos creados se devuelven en `tids`.

Los programas de aplicación pueden ver el entorno de hardware como una colección de elementos de procesamiento virtuales sin atributos, o pueden elegir explotar las capacidades de máquinas específicas del pool de hosts posicionando ciertas tareas en las computadoras más apropiadas. La ubicación de una tarea en una máquina específica o tipo de arquitectura se indica con el parámetro `lugar`.

Identificadores:

```
int pvm_mytid()
int pvm_parent()
```

Estas funciones obtienen el identificador del proceso actual y del proceso padre, respectivamente. El manejo de los identificadores es fundamental en el envío y recepción de mensajes.

Buffers:

```
int pvm_initsend(codificacion)
```

Crea un buffer para las comunicaciones. El argumento de `codificacion` es útil

cuando el mensaje se envía a una máquina que no es capaz que leer el mensaje en el formato nativo, por tener distinta arquitectura.

Comunicación:

- **Pack & Unpack:**

```
pvm_pkstr(char*)
pvm_pkint(int, n, 1)
pvm_upkint(int, n, 1)
pvm_upkfloat(float, m, 1)
```

  Pack y Unpack son las operaciones para escribir y leer datos del buffer de comunicaciones. Existe una función de acuerdo al tipo de datos que se maneja. Por ejemplo, la función `pvm_pkint` empaqueta una cantidad `n` de enteros (el tercer argumento indica la codificación). Al recibir un mensaje se debe proceder con las operaciones de desempaquetado en el mismo orden.

- **SEND:**

```
pvm_send(tid, tag)
```

  Es la función que se utiliza en las comunicaciones para enviar un mensaje a otro proceso (`tid`). El mensaje se identifica con un `tag` para distinguir los mensajes de distintas conversaciones. Es asincrónico. El mensaje que se envía es el que se encuentra en el buffer que fue creado anteriormente.

- **MCAST:**

```
pvm_mcast(tids, n, tag)
```

  Realiza un multicast del mensaje a `n` procesos cuyos identificadores se encuentren en el arreglo `tids`.

- **BCAST:**

```
pvm_bcast(grupo, tag)
```

  Realiza un broadcast del mensaje a todos los procesos del grupo.

- **RECV:**

  `pvm_recv(tid, tag)`

  Es la función que se utiliza en las comunicaciones para recibir un mensaje desde otro proceso (`tid`). Esta operación por defecto es sincrónica, i. e., el proceso se bloquea esperando a que arribe un mensaje con el `tid` y el `tag` deseados. No obstante, existen las versiones no bloqueantes (`nrecv`) y con timeout (`trecv`). Además, se pueden utilizar comodines en la recepción para cuando no se conoce la fuente del mensaje o su tag.

  Para clarificar el uso de estas primitivas, se presenta un ejemplo de comunicación en PVM. Supongamos que se envía un mensaje a partir de una cadena de caracteres y un entero, de la siguiente forma:

  ```
  pvm_initsend(PvmDataDefault);
  pvm_pkstr("se envia un numero");
  pvm_pkint(4, 1, 1);
  pvm_send(tid, 1);
  ```

  Luego, para recibir este mensaje habrá que ejecutar las operaciones simétricas de desempaquetado, a saber:

  ```
  pvm_recv(ptid, 1);
  pvm_upkstr(&cadena);
  pvm_upkint(&nro, 1, 1);
  ```

Sincronización:

- **BARRIER:**

  `pvm_barrier(grupo, n)`

  Crea una barrera para sincronización de procesos. Recibe como parámetros el nombre del grupo de procesos al que se aplica la barrera y la cantidad de procesos que deben llamar a la barrera antes de que cualquiera de ellos pueda continuar con la ejecución de las sentencias que siguen.

Computación global:

- **REDUCE:**

  `pvm_reduce(func, data, b, datatype, tag, grupo, raiz)`

  Realiza una operación de reducción sobre los datos, donde múltiples valores son reducidos a uno solo mediante la aplicación de un operador asociativo y conmutativo (`func`). La operación se realiza en los procesos que forman parte de `grupo` y el resultado es devuelto al proceso `raiz`.

3.2.4. MPI

MPI (Message Passing Interface) [Mes] provee una librería estándar de rutinas para escribir programas eficientes y portables que utilizan como forma de comunicación el envío y recepción de mensajes. MPI provee una completa colección de rutinas de comunicación punto a punto y de operaciones colectivas para transferencia de datos, computación global y sincronización. Existen implementaciones comerciales y gratuitas para MPPs, clusters y redes heterogéneas (ver por ejemplo la implementación MPICH [GLDS96]).

Uno de los objetivos principales de MPI fue producir un sistema que fuese de utilidad para los fabricantes de computadoras MPPs de alta performance para proveer implementaciones altamente optimizadas y eficientes. En contraste, PVM fue diseñado en primer orden para redes de computadoras, buscando portabilidad y sacrificando optimalidad en performance.

En MPI las operaciones de comunicación se realizan en el contexto de comunicadores. Un comunicador es una clase de grupo de procesos. En el pasaje de mensajes, tanto el emisor como el receptor deben ser parte del mismo comunicador. Existen diversas operaciones para crear comunicadores, duplicarlos, excluir/incluir procesos de un grupo, y realizar operaciones de conjuntos sobre grupos (unión, intersección, diferencia).

La experiencia ganada con el uso del estándar original ayudó al MPI Forum a expandir el estándar en MPI-2. En ese momento, se incluyeron extensiones para el manejo dinámico de tareas, soporte cliente/servidor, acceso a memoria remota y un número de funciones de alto nivel para E/S paralela, como por ejemplo, manipulación de archivos. Sin embargo, concentraremos nuestra descripción sobre el estándar original.

El modelo de control de procesos de MPI es estático. Los nodos quedan fijos en tiempo de compilación, de acuerdo al contenido de un archivo de configuración (el tamaño del sistema no cambia en ejecución), y los procesos son lanzados manualmente por línea de comandos. La falla de uno de los procesos condiciona a la aplicación completa, i. e., no hay manera de revertir esta situación [FD96]. No hay mecanismos para detectar fallas, para reportar que un proceso ha fallado, y no hay manera de recuperarse de una falla [Bec05]. En MPI-2 el modelo de tareas es dinámico, de acuerdo a las necesidades actuales de los sistemas distribuidos y los avances en la tecnología de redes. El modelo dinámico permite la creación y destrucción de tareas luego de que una aplicación de MPI ha comenzado su ejecución. Asimismo, MPI-2 incluye un mecanismo para que las nuevas tareas puedan comunicarse con la aplicación existente, e incluso para que aplicaciones independientes de MPI se comuniquen entre sí (*inter-communicators*).

El conjunto básico de funciones de MPI se describe a continuación.

Identificadores:

```
MPI_Comm_rank(MPI_Comm comunicador, int *mi_rango)
MPI_Comm_size(MPI_Comm comunicador, int *cantidad)
```

En MPI los identificadores de procesos reciben el nombre de *rangos*. La primera función nos permite conocer el rango del proceso que llama a la misma. La segunda función nos devuelve el tamaño del comunicador (grupo). Los rangos comienzan a numerarse desde 0 hasta `cantidad`−1.

Comunicación:

- **SEND:**

```
MPI_Send(void *buf, int n, MPI_Datatype tipo, int dest, int tag,
          MPI_Comm comm)
```

Al enviar un mensaje, el emisor se bloquea hasta que el mensaje ha sido copiado en el buffer receptor (o en un buffer temporal). Cuando la llamada retorna, el buffer emisor puede ser sobreescrito. El buffer consiste de `n` datos de tipo `tipo`. El parámetro `dest` corresponde al rango del proceso receptor.

También existen versiones buffered (no espera al `receive`, se copia a un buffer de sistema y retorna), sincrónica (no retorna hasta que se ejecute `receive`) y ready (sólo si ya se hizo un `receive`).

```
MPI_Isend(void *buf, int n, MPI_Datatype tipo, int dest, int tag,
          MPI_Comm comm, MPI_Request *request)
```

Es la versión no bloqueante del `send`. Retornará antes de que el mensaje sea copiado del buffer. El argumento `request` es utilizado para identificar las operaciones de comunicación y consultar sobre el estado de la misma.

- **RECV:**

```
MPI_Recv(void *buf, int n, MPI_Datatype tipo, int origen, int tag,
          MPI_Comm comm, MPI_Status *status)
```

`MPI_Recv` es la función de recepción estándar, no retornará hasta que sea recibido un mensaje en el buffer indicado (es bloqueante por defecto).

Si la fuente y/o el tag no son conocidos, se pueden utilizar comodines (`MPI_ANY_SOURCE`, `MPI_ANY_TAG`). En la información de estado, `status`, se pueden consultar el origen del mensaje, el tag y el tamaño del mensaje recibido, entre otros.

```
MPI_Irecv(void *buf, int n, MPI_Datatype tipo, int source, int tag,
          MPI_Comm comm, MPI_Request *request)
```

Iniciará la recepción de un mensaje, retorna inmediatamente sin esperar que el mensaje termine de ser copiado al buffer de recepción.

- **Testear la terminación:**

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *stat)
```

Para completar la comunicación, MPI provee funciones para testear la terminación de una operación no bloqueante. En el test, `flag` será 1 si la operación de comunicación se completó.

```
MPI_Wait(MPI_Request *request, MPI_Status *stat)
```

Espera a que la operación de comunicación asociada se complete.

- **BCAST:**

```
MPI_Bcast(void *buf, int n, MPI_Datatype tipo, int raiz, MPI_Comm comm)
```

Broadcast envía un mensaje desde el proceso `raiz` a todos los demás procesos en el comunicador.

Sincronización:

- **BARRIER:**

```
MPI_Barrier(MPI_Comm comm)
```

Los procesos en un comunicador pueden sincronizarse en un punto determinado. La función retorna luego de que todos los miembros del comunicador hayan ejecutado sus llamadas a la misma.

Computación global:

- **REDUCE:**

```
MPI_Reduce(void *sbuf, void *rbuf, int n, MPI_Datatype tipo,
          MPI_Op op, int raiz, MPI_Comm comm)
```

`MPI_Reduce` es un ejemplo de función para computación global, son operaciones para combinar grupos de datos. En este caso, un operador asociativo y conmutativo es aplicado a los datos (e. g., `MPI_SUM`). El resultado se envía al proceso `raiz`. Existe una variante, `Allreduce`, que realiza un broadcast del resultado a todos los procesos intervinientes (miembros del comunicador). Las dos operaciones se ilustran en la Figura 3.5.

- **SCAN:**

```
MPI_Scan(void *sbuf, void *rbuf, int n, MPI_Datatype tipo,
          MPI_Op op, int raiz, MPI_Comm comm)
```

Esta función realiza una reducción prefija en los datos asociados. Después de la reducción, el buffer de recepción del proceso con rango i tendrá la reducción de los valores de los buffers emisores de los procesos con rangos 0, 1, $\ldots$, i.

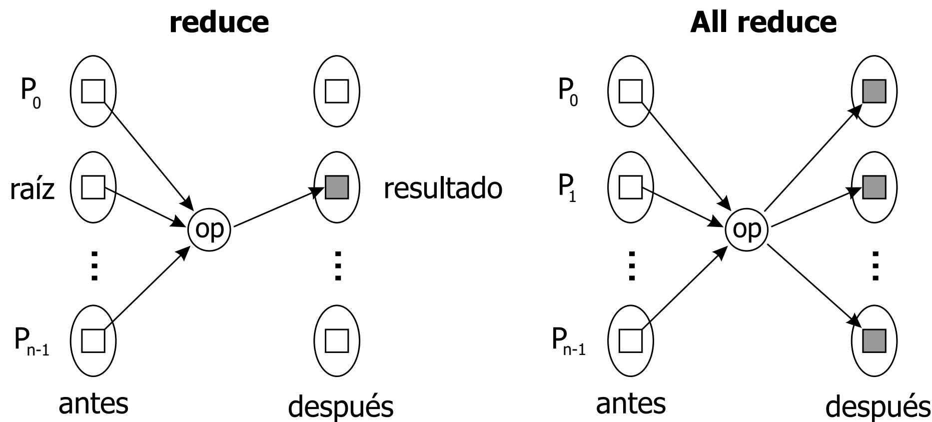

Figura 3.5: Operaciones de reducción de datos en MPI

Transferencia de datos:

- **SCATTER:**

```
MPI_Scatter(void *sbuf, int n, MPI_Datatype stipo, void *rbuf,
               int m, MPI_Datatype rtipo, int raiz, MPI_Comm comm)
```

La función `Scatter` permite que un proceso distribuya su buffer entre cada miembro del grupo. El buffer emisor del proceso `raiz` se divide en un número de segmentos, cada uno de tamaño `n`. Los primeros `n` elementos en el buffer son copiados en el buffer receptor del primer miembro del grupo, y así sucesivamente.

- **GATHER:**

```
MPI_Gather(void *sbuf, int n, MPI_Datatype stipo, void *rbuf,
               int m, MPI_Datatype rtipo, int raiz, MPI_Comm comm)
```

La función `Gather` es el complemento de `Scatter`: permite que un proceso construya su buffer a partir de piezas de datos recolectadas de los otros miembros del grupo. Cada tarea (incluyendo la raíz) envía el contenido de su buffer emisor al proceso `raiz`. Este proceso recibe los mensajes y los almacena según el orden de los rangos. El buffer emisor del primer miembro del grupo se copia en las primeras `m` ubicaciones del buffer receptor, y así sucesivamente.

En la Figura 3.6 se puede ver gráficamente la operación de estas dos primitivas específicas de MPI para transferencia de datos.

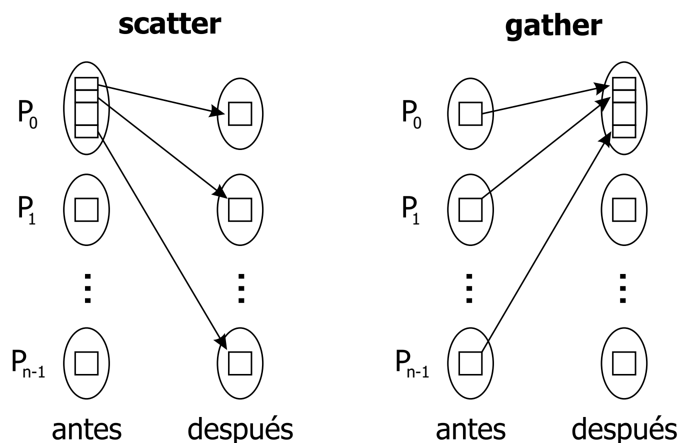

Figura 3.6: Primitivas de transferencia de datos en MPI

3.3. Ejemplo comparativo: RankSort

El algoritmo de RankSort, también conocido como *enumeration sort*, implementa una técnica de ordenamiento simple, donde se cuenta la cantidad de elementos que son menores que un elemento seleccionado. Esta cuenta provee la posición del elemento seleccionado en la lista, o bien su *rango* en la lista ordenada.

Supongamos que existen n elementos almacenados en un arreglo, $a[0] \dots a[n-1]$, y una relación de orden entre los mismos. Primero, el $a[0]$ es comparado con el resto de los elementos $a[1] \dots a[n-1]$, contando la cantidad de elementos menores que $a[0]$. Llamemos x a esa cantidad, que a su vez será la ubicación de $a[0]$ en la lista ordenada. Luego, $a[0]$ es copiado en la lista ordenada $b[0] \dots b[n-1]$ en la ubicación $b[x]$. Comparando cada elemento contra los $n-1$ restantes requiere $n-1$ pasos si es ejecutado secuencialmente. Ejecutar este procedimiento para los n elementos requiere $n(n-1)$ pasos, obteniendo un tiempo secuencial del orden de $O(n^2)$. Este orden de complejidad no representa lo que se considera un buen tiempo para un algoritmo de ordenamiento secuencial. Se conoce que algoritmos como Merge Sort o Heap Sort obtienen tiempos del orden de $O(n \log n)$ en el peor caso.

Para un arreglo sin elementos repetidos[5] el código secuencial sería el siguiente:

[5]La extensión es trivial para elementos repetidos si consideramos además la posición inicial de los elementos en el arreglo, donde a pesar de la igualdad de los elementos, se incremente el rank si el elemento a comparar se encuentra antes que el elemento al que se le está calculando el ranking. De esta manera se obtiene un ranking consecutivo para los elementos repetidos.

```
for (i = 0; i < n; i++) {
   x = 0;
   for (j = 0; j < n; j++)
      if (a[i] > a[j]) x++;
   b[x] = a[i];
}
```

Pero, si tenemos forma de ejecutar este procedimiento en paralelo, con un procesador por componente del arreglo, podemos obtener un tiempo del orden de $O(n)$. Incluso se pueden obtener tiempos mejores si se cuenta con más procesadores. Con n procesadores, todos los procesos tienen acceso de lectura al arreglo de elementos desordenados, pero no lo modifican, por lo que son independientes. En notación *forall* tendrá la forma:

```
forall (i = 0; i < n; i++) {
   x = 0;
   for (j = 0; j < n; j++)
      if (a[i] > a[j]) x++;
   b[x] = a[i];
}
```

Es claro que, dependiendo del tamaño del arreglo, en la práctica no siempre podemos contar con n procesadores. Por lo tanto es muy probable que tengamos que asignar a cada procesador una porción del arreglo, que conste de p elementos (*grouping*). Cuando se trata con implementaciones para multicomputer, el arreglo de elementos desordenados no se encuentra en una memoria común accesible por todos los procesadores, sino que tiene que comunicarse de alguna manera. Una opción es enviar una copia del arreglo completo a cada procesador, así, con n procesadores y un arreglo de n componentes, se utilizan $n * n$ celdas de memoria (ver la versión que se presenta más adelante con PVM). Otra alternativa es dejar que los procesadores se comuniquen los valores individuales a medida que los van utilizando, esto es, el procesador inicial envía los datos del arreglo a su procesador vecino, que lo compara y lo vuelve a enviar al procesador que está al lado. De esta forma tenemos un canal para los elementos que se van pasando, y evitamos la necesidad de acumular los elementos en cada memoria local (ver versión con MPI).

A continuación se presentará una versión de RankSort para cada uno de los lenguajes vistos en este capítulo, no para comparar la eficiencia de las distintas implementaciones,

sino para poder ver la complejidad de cada uno de los lenguajes a través de un ejemplo común y sencillo de comprender. Estos programas ordenan un arreglo de enteros, aunque los elementos podrían ser de cualquier tipo que defina una relación de orden.

RankSort en HPF:

```
1  PROGRAM RankSort
2
3  INTEGER MAX, NUM_PROCS
4  PARAMETER (MAX=...)
5  PARAMETER (NUM_PROCS=...)
6  INTEGER I, fuente(MAX), destino(MAX), rangos(MAX)
7
8  !HPF$ PROCESSORS procs(NUM_PROCS)
9  !HPF$ ALIGN fuente(:) WITH destino(:)
10 !HPF$ DISTRIBUTE fuente (CYCLIC) ONTO procs
11 !HPF$ DISTRIBUTE destino (CYCLIC) ONTO procs
12 !HPF$ DISTRIBUTE rangos (CYCLIC) ONTO procs
13
14 PURE INTEGER FUNCTION Rank(nro, fuente)
15     INTEGER, INTENT(IN) :: nro
16     INTEGER, DIMENSION(MAX), INTENT(IN) :: fuente
17     INTEGER I, cant
18
19     cant = 0
20     FOR (I=1:MAX)
21         IF (fuente(I) < nro) THEN
22             cant = cant + 1
23     Rank = cant
24 END FUNCTION Rank
25
26 FORALL (I=1:MAX)
27     rangos(I) = Rank(fuente(I))
28     destino(rangos(I)) = fuente(I)
29 END FORALL
30
31 END
```

RankSort en OpenMP:

```
1  #include <omp.h>
2
3  #define MAX ...
4  int fuente[MAX]
5  int destino[MAX]
6
7  int Rank(int nro) {
8      int i, rank=0;
9
10     for (i=0; i<MAX; i++)
11         if (fuente[i] < fuente[nro]) then
12             rank++;
13     return rank;
14 }
15
16 void main() {
17     #pragma omp parallel num_threads(MAX)
18     {
19         int i;
20
21         #pragma omp parallel for
22         {
23             for (i=0; i<MAX; i++)
24                 destino[Rank(i)] = fuente[i]
25         }
26     }
27 }
```

RankSort en PVM:

```
1  #include "pvm3.h"
2  #define MAX ...
3  #define SPAWN_TYPE PvmTaskArch
```

```
4  #define SPAWN_WHERE "LINUX"
5
6  int Rank(int elemento, int fuente[]) {
7     int cant, i;
8     cant = 0;
9     for (i=0; i<MAX; i++) {
10       if (fuente[i] < elemento)
11          cant++;
12    }
13    return cant;
14 }
15
16 int main (int argc, char *argv) {
17    int fuente[MAX], destino[MAX], hijos[MAX];
18    int ranking, elem, mytid, pid;
19    char **argvc = NULL;
20
21    mytid = pvm_mytid();
22    pid = pvm_parent();
23    if (pid = PvmNoParent) {
24       /* proceso padre */
25       pvm_spawn(argv[0], argvc, SPAWN_TYPE, SPAWN_WHERE, MAX-1, hijos+1);
26       for (i=1; i<MAX; i++) {
27          pvm_initsend(PvmDataDefault);
28          pvm_pkint(fuente[i], 1, 1);
29          pvm_send(hijos[i], 1);
30       }
31       pvm_initsend(PvmDataDefault);
32       pvm_pkint(fuente, MAX, 1);
33       pvm_mcast(hijos+1, MAX-1, 2);
34
35       ranking = Rank(fuente[0], fuente);
36       destino[ranking] = fuente[0];
37
38       for(i=1; i<MAX; i++) {
39          pvm_recv(hijos[i], 3);
```

```
40          pvm_upkint(&ranking, 1, 1);
41          destino[ranking] = fuente[i];
42       }
43    }
44    else {
45       /* procesos hijos */
46       pvm_recv(pid, 1);
47       pvm_upkint(&fuente, MAX, 1);
48       pvm_recv(pid, 2);
49       pvm_upkint(&elem, 1, 1);
50
51       ranking = Rank(elem, fuente);
52
53       pvm_initsend(PvmDataDefault);
54       pvm_pkint(ranking, 1, 1);
55       pvm_send(pid, 3);
56    }
57    pvm_exit();
58    exit(0);
59 }
```

RankSort en MPI:

```
1  #include "mpi.h"
2  #define MAX ...
3
4  int main (int argc, char *argv) {
5     int elemento, actual, id, i, rank;
6     MPI_Status s;
7     int fuente[MAX], destino[MAX];
8
9     MPI_Init(&argc,&argv);
10    MPI_Comm_rank(MPI_COMM_WORLD, &id);
11
12    if (id == 0) {
13       /* proceso principal */
```

```
14         elemento = fuente[0];
15         for (i=1; i<MAX; i++) {
16            MPI_Send(fuente[i], 1, MPI_INT, i, 1, MPI_COMM_WORLD);
17         }
18
19         rank = 0;
20         for (i=0; i<MAX; i++) {
21            if (fuente[i] < elemento)
22                rank++;
23            MPI_Send(fuente[i], 1, MPI_INT, 1, 2, MPI_COMM_WORLD);
24         }
25         destino[rank] = elemento;
26
27         for(i=1; i<MAX; i++) {
28            MPI_Recv(&rank, 1, MPI_INT, i, 3, MPI_COMM_WORLD, &s);
29            destino[rank] = fuente[i];
30         }
31      }
32      else {
33         /* procesos esclavos */
34         MPI_Recv(&elemento, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &s);
35         rank = 0;
36         for (i=0; i<MAX; i++) {
37            MPI_Recv(&actual, 1, MPI_INT, id-1, 2, MPI_COMM_WORLD, &s);
38            if (actual < elemento)
39                rank++;
40            if (id+1 != MAX)
41                MPI_Send(actual, 1, MPI_INT, id+1, 2, MPI_COMM_WORLD, &s);
42         }
43         MPI_Send(rank, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
44      }
45
46      MPI_Finalize();
47      exit(0);
48   }
```

A primera vista podemos ver que en las versiones con HPF y OpenMP el programa presenta menor cantidad de líneas de código. Esto es una consecuencia inherente de que son lenguajes de más alto nivel que lo que representan MPI y PVM. Las estrategias de resolución de los cuatro casos tienen algunas diferencias. Existe, sin embargo, cierto grado de similitud entre las implementaciones de HPF y OpenMP (por ser de alto nivel y para multiprocesadores) y las de MPI y PVM (orientadas al pasaje de mensajes).

Las versiones de HPF y OpenMP reflejan casi directamente la idea inicial del algoritmo de Rank donde se utiliza la semántica del *forall*, ya que el lenguaje dispone de esta primitiva. Fuera de ello, como sentencias especiales, en HPF se tiene que declarar el número de procesadores y la distribución de los arreglos de entrada en los mismos (líneas 8-12). Para OpenMP directamente utilizamos las directivas del compilador que indican una ejecución en cierta cantidad de threads (líneas 17 y 21).

Las otras dos implementaciones son ligeramente diferentes. MPI y PVM están orientados a multicomputadoras, entonces debemos utilizar primitivas de pasaje de mensajes. Para esto pensamos otra estrategia que involucra enviar el arreglo a los hijos y esperar por un mensaje con el resultado. En PVM el proceso padre crea varios procesos hijos y luego les envía una copia del arreglo a cada uno, así como una de las componentes de ese arreglo (la que le toca computar). El proceso padre, de rango 0, se encarga del cómputo del rank del elemento en la primer componente del arreglo (línea 35); los hijos hacen una tarea análoga con el elemento que le fue enviado (línea 51). Luego el proceso padre se queda esperando por los mensajes de sus hijos que le van a indicar dónde ubicar ese elemento en el arreglo de resultado (línea 39).

En cambio, en MPI, como variante se eligió otra estrategia posible donde se evita la replicación del arreglo completo en cada uno de los hijos. En este caso, los procesadores se ubican linealmente y los valores del arreglo se van pasando de procesador en procesador (líneas 37 y 41). Cuando un proceso recibe un valor del arreglo, lo compara con el elemento que le toca computar (que es el único dato que se le pasó de antemano por mensaje) y así va obteniendo el rank. Una vez que por él pasaron todos los elementos del arreglo, forma un mensaje con el resultado y lo envía al proceso padre (línea 43). En esta alternativa tenemos menos gasto de memoria al no replicar el arreglo, pero tenemos más gasto de comunicación entre los procesadores, ya que los hijos se comunican entre sí. Sin embargo, la topología de línea se adapta perfectamente; como punto a favor, se conoce que todas

las otras topologías pueden emular fácilmente a una línea, si mapeamos los procesadores a procesos de manera inteligente.

3.4. Comparación de lenguajes y primitivas

En esta sección se ha elaborado un cuadro comparativo, el Cuadro 3.3, donde se destacan las características salientes de los lenguajes descritos. En cuanto a la abstracción que proveen, ya hemos visto que HPF y OpenMP son lenguajes de alto nivel, mientras que PVM y MPI son de bajo nivel. Todos los lenguajes de programación tienen un buen grado de portabilidad, ya que implementan sus compiladores para las distintas arquitecturas.

HPF es apto para multiprocesadores, i. e., computadoras con un espacio de direcciones compartido, ya sea un uniprocesador, una red de computadoras (NUMA) o una computadora paralela. OpenMP es una API que soporta programación paralela en máquinas con memoria compartida, multiplataforma. PVM es muy portable, su código ha sido compilado tanto en laptops como en CRAYs[6], es apto para multiprocesadores y multicomputadoras. También existen implementaciones de MPI para redes de estaciones de trabajo o para máquinas multiprocesador con memoria compartida. En todas las implementaciones (contando a MPI-2) la descomposición de procesos puede hacerse de manera dinámica, i. e., se pueden crear y terminar tareas en tiempo de ejecución. El mecanismo es distinto en MPI ya que para lanzar procesos se utiliza la línea de comandos y los nodos del sistema quedan fijos en tiempo de compilación.

El mecanismo de comunicación entre procesos concuerda con el tipo de arquitectura. En las computadoras multiprocesador se utiliza memoria compartida y en las multicomputadora se realiza pasaje de mensajes. En general las implementaciones para multicomputadoras pueden utilizarse también en multiprocesadores (es fácil emular el pasaje de mensajes en memoria compartida). Para la sincronización de procesos hay varias alternativas. HPF utiliza rutinas especiales (de fecha y hora) que tienen el efecto de una barrera, en el sentido que al ser invocadas por los distintos procesos fuerzan una sincronización de los mismos. En OpenMP se pueden definir secciones críticas, adquirir

[6]Cray Inc. - The Supercomputer Company: `http://www.cray.com/`

[7]MPI-2 incluye un esquema de notificación de fallas, pero no presenta mecanismos para recuperarse de una espontánea pérdida de un proceso.

	HPF	**OpenMP**	**PVM**	**MPI**
Nivel de Abstracción	alto	alto	bajo	bajo
Portabilidad	si	si	si	si
Multiprocesador(MP) vs. Multicomputadora(MC)	MP	MP	MP/MC	MP/MC
Descomposición	dinámica	dinámica	dinámica	estática y dinámica (MPI-2)
Comunicación	mem. compartida distribuida	mem. compartida	mensajes	mensajes
Sincronización	rutinas que fuerzan sincronización	secciones críticas y barreras	mensajes y barreras	mensajes y barreras
Tolerancia a Fallas	–no especifica–	–no especifica–	si	no[7]

Cuadro 3.3: Comparación de los distintos lenguajes analizados

y liberar locks y utilizar barreras. Para sincronizar procesos en PVM y MPI se tienen primitivas de barreras y se puede utilizar el intercambio de mensajes en modo bloqueante.

Por último, con respecto a la tolerancia a fallas, algunos lenguajes no reportan nada relativo. Sin embargo, generalmente poseen los mecanismos básicos de detección de errores en las operaciones (se informa el éxito o fracaso mediante algún código de error devuelto en forma de entero). Lo importante en el caso de una computadora paralela es detectar la caída de alguno de los hosts, y poder reestructurar o reconfigurar el sistema ante ese suceso (que sea dinámico). Esta facilidad permitiría continuar con la computación, quizás con una pequeña degradación en performance, asignando las tareas del host que falló a otro host.

3.5. Resumen

En este capítulo se presentaron, clasificaron y compararon cuatro de los lenguajes imperativos más utilizados para programación en paralelo: HPF, OpenMP, PVM y MPI.

La escritura de programas eficientes en HPF no es necesariamente una tarea trivial. Ciertamente, la naturaleza de alto nivel del lenguaje lo hace ineficiente. El programador necesita entender muy bien el programa y el modelo de ejecución de HPF para mapear los datos en forma efectiva. No obstante, HPF representa un paso importante hacia la simplificación de la programación paralela. Se basa en un lenguaje programación simple y ofrece gran portabilidad. Posee obvias ventajas sobre la programación SPMD explícita con pasaje de mensajes, partiendo de que los programas son más claros, cortos y fáciles de desarrollar. Una desventaja de HPF es que es un lenguaje muy difícil de compilar y que no ofrece gran performance a pesar de su nombre (ver por ejemplo la comparación de performance que presenta Shires en [Shi98]). Frecuentemente, invertir en código MPI resulta en una aplicación más performante con pasaje de mensajes explícito.

OpenMP cubre sólo el uso directo de paralelismo, donde el usuario especifica en forma explícita las acciones que deben ser tomadas por el compilador y el sistema en tiempo de ejecución para ejecutar el programa en paralelo. La compilación de OpenMP no chequea por dependencias, conflictos, condiciones de carrera u otros problemas. El compilador de OpenMP tampoco es responsable de generar paralelización automática. El usuario es responsable de usar OpenMP en estas condiciones y producir programas confiables. Muchos fabricantes de multiprocesadores ofrecen compiladores para este lenguaje: Sun, SGI, Intel, Fujitsu, etc.

MPI es usualmente mencionado como el '*lenguaje ensamblador*' de la programación en paralelo. Esta gran fortaleza es también su mayor debilidad: el programador tiene control final sobre todo. Toda tarea de comunicación se ejecuta a una tasa mucho más baja que la computación real, luego el programador debe asegurarse de que se realicen suficientes cómputos antes o incluso durante los eventos de comunicación.

Si una aplicación va a ser desarrollada y ejecutada en una única MPP, entonces se espera que MPI tenga mejor performance en las comunicaciones. MPI tiene un conjunto de funciones de comunicación más completo y permite que las aplicaciones aprovechen los modos de comunicación especiales que no están presentes en PVM, citando por ejemplo

al *send* no bloqueante. La intención de MPI era ser un estándar de una especificación de pasaje de mensajes que cada fabricante de MPP implementaría en su sistema (la librería utiliza el hardware nativo para ser lo más performante posible). La solución que presenta PVM deja de lado parcialmente el tema de performance, en vistas de obtener mayor flexibilidad. Cuando la comunicación se realiza localmente o a un host con la misma arquitectura, PVM utiliza las funciones de comunicación nativa, tal como lo hace MPI. En cambio, cuando la comunicación se realiza a una arquitectura diferente, PVM utiliza las funciones de comunicación estándar [GKP96]. Dado que la librería tiene que determinar el destino de cada mensaje para determinar qué funciones va a utilizar, se incurre en un pequeño overhead que impacta sobre la velocidad de ejecución.

Se han tenido que hacer algunos sacrificios en MPI para obtener esta alta performance en comunicación. Dos de los más notables son la falta de interoperabilidad y la intolerancia a fallas. Tanto en MPI como en PVM, los programas escritos para una arquitectura pueden ser copiados a una segunda arquitectura, compilados y ejecutados sin modificación. No obstante, PVM presenta algo extra: los ejecutables resultantes también se pueden comunicar el uno con el otro y cooperar (incluso provee interoperabilidad de lenguajes con programas escritos en C y Fortran). Como PVM nace sobre el concepto de máquina virtual, presenta estas ventajas cuando la aplicación se va a ejecutar sobre un conjunto de hosts heterogéneos. Además, PVM contiene funciones para control de procesos y manejo de recursos que son importantes a la hora de crear aplicaciones portables que corran en clusters de estaciones de trabajo y MPPs.

Mientras más grande sea el cluster de hosts, más importantes se vuelven las facilidades de tolerancia a fallas que presenta PVM. En las aplicaciones de computación distribuida, que generalmente se ejecutan durante períodos largos de tiempo, es muy importante contar con la posibilidad de escribir aplicaciones que continúen en pie ante el caso de fallas o cambios dinámicos en la carga. Para salvar este punto débil de MPI, Fagg y Dongarra [FD00] han trabajado sobre una versión del estándar que incluye mecanismos de tolerancia a fallas.

Capítulo 4

Paralelismo en lenguajes de programación en lógica

La programación en lógica ofrece muchas oportunidades para ejecución paralela [TF86], ya que existe una separación natural entre la lógica y el control de los programas. La estrategia típica en el desarrollo de sistemas paralelos de programación en lógica ha estado basada en trasladar alguna de las posibilidades de no determinismo presentes en la semántica operacional a computaciones paralelas. Usualmente, los tipos de paralelismo explotados en la práctica por soportes en tiempo de ejecución de programas lógicos son:

- el paralelismo OR, que es explotado cuando se unifica una submeta con la cabeza de más de una cláusula en paralelo. Por ejemplo, asumamos que tenemos la meta: `?- a(X)` y que tenemos dos cláusulas que coinciden: `a(X):- b(X)` y `a(X):- c(X)`, luego podemos usar paralelismo OR para que estas cláusulas se evalúen en paralelo.

- el paralelismo AND, que surge al aprovechar la evaluación concurrente de las submetas en las que se descompone la meta del programa.

Los problemas se presentan cuando distintos hilos de ejecución intentan ligar una misma variable compartida, cuando existe una dependencia entre los argumentos de las distintas submetas y al intentar mantener bajos los niveles de sobrecarga en el sistema asociados con el paralelismo, entre otros.

En este capítulo también se presentarán y evaluarán algunas implementaciones de Prolog que incorporan estas formas de paralelismo. Para cerrar se contrastarán los lenguajes elegidos y se compararán a través del mismo ejemplo que se utilizó en el capítulo anterior.

4.1. Oportunidades de paralelismo

La paralelización de los programas lógicos puede verse como una consecuencia directa del principio de Kowalski [Kow79]:

$$\textit{Programas} = \textit{Lógica} + \textit{Control}$$

Este principio separa la componente de control de la especificación lógica del problema, permitiendo que el control de la ejecución sea una característica ortogonal, independiente de la especificación del problema. Como consecuencia, los sistemas encargados de la ejecución de programas lógicos pueden adoptar diferentes estrategias (i. e., las operaciones pueden ejecutarse en distinto orden o incluso en paralelo), sin afectar el significado declarativo del programa.

Además de la separación entre lógica y control, desde la perspectiva de los lenguajes de programación, la programación en lógica exhibe tres características claves que hacen que la explotación del paralelismo sea más práctica que en los lenguajes imperativos tradicionales:

1. Los lenguajes de programación en lógica son lenguajes de asignación única: las variables son entidades matemáticas a las que se puede asignar a lo sumo un valor durante cada derivación. Esto releva al sistema paralelo de la tarea de llevar rastro de ciertos tipos de dependencias de flujo.

2. Los lenguajes de programación en lógica, al igual que los lenguajes funcionales, permiten que el programador refleje la estructura del problema de manera más natural y directa en su algoritmo. Esto hace que el paralelismo posible pueda ser accedido más fácilmente por un compilador.

3. La semántica de operación de la programación en lógica incluye cierto grado de no determinismo, el cual puede ser fácilmente transformado en paralelismo

sin modificaciones radicales. Esto lleva a la posibilidad de extraer paralelismo directamente desde el modelo de ejecución, sin modificaciones en el código fuente del programa (paralelismo implícito).

Los lenguajes con paralelismo implícito permiten a los programadores escribir el código de sus programas sin preocuparse acerca de los detalles de cómo se realizará el paralelismo. Todo el paralelismo que existe en el programa será detectado automáticamente por un compilador y explotado en ejecución. La estrategia típica en el desarrollo de sistemas paralelos de programación en lógica ha estado basada en la traslación de una (o más) de las posibilidades de no determinismo presentes en la semántica operacional en computaciones paralelas. Esto se traduce en tres formas clásicas de paralelismo:

- *Unification Parallelism*, que surge de la paralelización del proceso de unificación.
- *OR-Parallelism*, que se origina de paralelizar la selección de la cláusula a ser utilizada en la computación del literal, permitiendo que varias cláusulas se prueben en paralelo; y
- *AND-Parallelism*, que se origina paralelizando la selección del próximo literal a ser resuelto, permitiendo que múltiples literales se resuelvan concurrentemente;

Paralelismo de Unificación

Este tipo de paralelismo surge durante la unificación de los argumentos de una meta con los de la cabeza de la cláusula que tiene el mismo nombre y aridad. Los diferentes términos de argumentos pueden ser unificados en paralelo. Esto aprovecha la secuencia de unificaciones entre los argumentos de estructuras complejas. Por ejemplo, en:

```
persona(fecha_nac(dia(17),mes(3),año(95)),
        direccion(calle(roca),numero(1832),ciudad(cipo)))
persona(fecha_nac(dia(X),mes(1),Y),direccion(Z,W,ciudad(cipo)))
```

deben unificar:

```
fecha_nac(dia(17),mes(3),año(95)) = fecha_nac(dia(X),mes(1),Y)
direccion(calle(roca),numero(1832),ciudad(cipo)) = direccion(Z,W,ciudad(cipo))
```

y así recursivamente. El paralelismo de unificación también tiene que manejar problemas de dependencias entre los argumentos, similares a los que se presentan en el paralelismo AND (de hecho, el paralelismo de unificación puede verse como una forma de paralelismo AND).

Paralelismo OR

El paralelismo OR se presenta cuando más de una regla define una relación y la submeta unifica con más de una cabeza de regla. Se tienen entonces varias opciones posibles para buscar una solución a la meta; los cuerpos de las reglas correspondientes pueden ser ejecutados en paralelo. Este tipo de paralelismo es entonces una manera de buscar soluciones para una consulta más velozmente, explorando en paralelo el espacio de soluciones generado por la presencia de múltiples cláusulas aplicables en cada paso de resolución. Se debe observar que cada computación paralela está calculando potencialmente una solución alternativa a la meta original.

Por ejemplo, teniendo las reglas:

```
abuelo(X, Y):- padre(X, Z), padre(Z, Y).
abuelo(X, Y):- padre(X, Z), madre(Z, Y).
```

Al realizar la siguiente consulta tenemos dos cláusulas que unifican y pueden ejecutarse en paralelo:

```
?- abuelo(juan, marcos).

abuelo(juan, marcos):- padre(juan, Z), padre(Z, marcos).
abuelo(juan, marcos):- padre(juan, Z), madre(Z, marcos).
```

Paralelismo AND

El paralelismo AND surge cuando existe más de una submeta en el resolvente (en el cuerpo de una regla), y algunas de estas metas pueden ser ejecutadas en paralelo. Luego, este paralelismo permite la explotación del paralelismo dentro de la computación de una misma solución para la meta original. El paralelismo AND está presente en la mayoría de las aplicaciones, pero es particularmente relevante en programas que se organizan según la aproximación de 'dividir y conquistar', aplicaciones de procesamiento de listas, resolución de problemas con restricciones, y aplicaciones de sistema.

A continuación se plantea un ejemplo que exhibe paralelismo AND, donde las metas entre corchetes pueden ejecutarse en paralelo. Este ejemplo plantea una solución para el problema de buscar el M-ésimo término de la sucesión de Fibonacci, que se obtiene a partir de la suma de los términos $M - 1$ y $M - 2$. Sin embargo, no siempre la conversión a programas con paralelismo es tan directa, a veces hay que reescribir completamente la solución a partir de su equivalente secuencial.

```
fib(0, 1).
fib(1, 1).
fib(M, N):- [M1 is M - 1, fib(M1, N1)],
            [M2 is M - 2, fib(M2, N2)],
            N is N1 + N2.
```

Generalmente, en la literatura, el paralelismo AND tiene dos formas:

- *Independent AND-Parallelism (IAP)*: dadas dos o más submetas, las ligaduras en tiempo de ejecución para las variables en estas metas antes de su ejecución son tales que cada meta no tiene influencia en el resultado de sus pares. El ejemplo clásico de metas independientes está representado por metas que, en tiempo de ejecución, no comparten ninguna variable sin ligar; esto es, la intersección de los conjuntos de variables accesibles por cada meta es vacía.

- *Dependent AND-Parallelism (DAP)*: en tiempo de ejecución, dos o más metas en el cuerpo de una cláusula tienen una variable común y son ejecutadas en paralelo, *compitiendo* en la creación de ligaduras para esa variable (o *cooperando* si las metas comparten la tarea de crear las ligaduras para la variable común). El paralelismo AND dependiente puede ser explotado en varios grados, desde modelos que reproducen fehacientemente las semánticas observables de Prolog hasta modelos que utilizan formas especializadas de este tipo de paralelismo (e. g., paralelismo stream) para soportar corutinas y otras semánticas alternativas.

Comparación

El paralelismo OR y el paralelismo AND identifican oportunidades de transformar ciertas componentes secuenciales de la semántica operacional de la programación en lógica en operaciones concurrentes. En el caso de paralelismo OR, en un punto de elección

se paraleliza la exploración de las diferentes alternativas; mientras que en el caso del paralelismo AND, se paraleliza la resolución de las distintas submetas. En ambos casos, se espera que el sistema sea capaz de proveer un número de recursos de computación que hagan posible la ejecución de las diferentes instancias de trabajo paralelo.

Intuitivamente, se puede decir que el paralelismo OR y el paralelismo AND son ortogonales entre sí, siendo que paralelizan puntos independientes del no determinismo presente en la semántica operacional del lenguaje. Luego, se podría esperar que la explotación de una de las formas de paralelismo no afecte al aprovechamiento de la otra, e incluso podría ser factible su explotación simultánea. Sin embargo, en la práctica esta ortogonalidad no se traslada fácilmente al nivel de implementación, y en efecto, aún no se ha construido un sistema paralelo eficiente que logre este objetivo ideal.

Es importante destacar que el objetivo global de la investigación en programación en lógica paralela es lograr una alta performance a través del paralelismo. La obtención de buenos speed-ups puede no traducirse en una mejora real de la performance con respecto a los sistemas secuenciales actuales; por ejemplo, el costo de manejar la explotación del paralelismo podría hacer que la performance del sistema en un único procesador sea considerablemente inferior a la de un sistema secuencial estándar. La mayor parte de las investigaciones en el campo de programación en lógica paralela ha sido desarrollado en sistemas con arquitectura de memoria compartida. Actualmente, se ha renovado un poco el interés en arquitecturas de memoria distribuida gracias a su creciente disponibilidad a precios económicos y su escalabilidad [AR97, CMCP92, SW00, Tal94].

4.2. Paralelismo OR

El paralelismo OR surge cuando una submeta puede unificar con las cabezas de más de una cláusula. En tal caso, los cuerpos de estas cláusulas pueden ser ejecutados en paralelo. Una forma conveniente para visualizar este tipo de paralelismo es por medio de un árbol or-paralelo de búsqueda o *or-parallel search tree*. El árbol respeta la semántica operacional de Prolog: en cada nodo se consideran las cláusulas aplicables a la primer submeta, y los hijos del nodo se consideran ordenados de izquierda a derecha, de acuerdo al orden de las cláusulas en el programa. Esto es, durante una ejecución secuencial, el árbol or-paralelo es recorrido en profundidad. No obstante, si se cuenta con varias unidades de cómputo, múltiples ramas del árbol podrían recorrerse simultáneamente.

Por ejemplo, para el siguiente fragmento de código se obtiene el árbol de la Figura 4.1.

```
f:- t(X,three),p(Y),q(Y).
p(L):- s(L,M),t(M,L).
p(K):- r(K).
q(one).
q(two).
s(two,three).
s(four,five).
t(three,three).
t(three,two).
r(one).
r(three).
?-f.
```

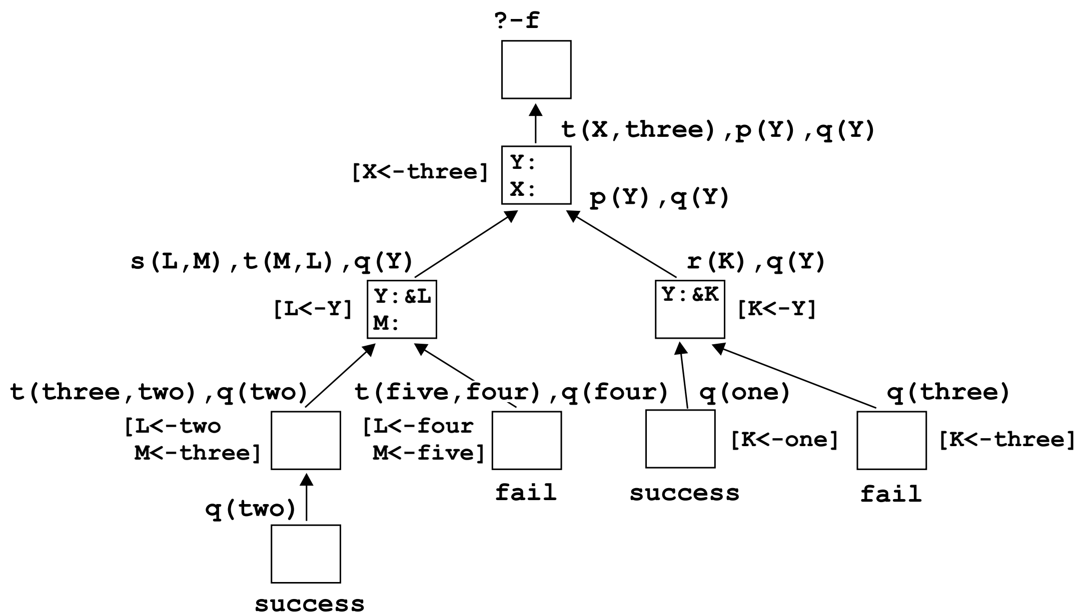

Figura 4.1: Or-parallel search tree [GPA+01]

Desde un punto de vista teórico, el paralelismo OR impone pocos problemas ya que las distintas ramas del árbol son independientes entre sí, y entonces se requiere poca comunicación entre los agentes que las exploran. A partir de ello, es más probable que la ejecución or-paralela de un programa lógico cumpla con la condición de *no-slowdown*, esto es, la ejecución paralela no resultará más lenta que su contraparte secuencial.

Desafortunadamente, la implementación del paralelismo OR no es tan fácil, ya que no es trivial mantener pequeños los overheads asociados con el paralelismo en tiempo de ejecución. Esto se debe a las complicaciones prácticas que emergen del hecho de que algunos de los nodos del árbol son comunes: dados dos nodos en ramas diferentes, todos los nodos hacia arriba y en el mismo nivel que el primer ancestro común, se comparten. Una variable que se crea en uno de los nodos ancestros puede ser ligada con distintos valores en las dos ramas. Por ejemplo, se puede ver esta situación en la Figura 4.1, con la variable `K` que se instancia en dos ramas con los valores `one` y `three`, o con `M` que recibe los valores `three` y `five` en distintas ramas. Los entornos de las ramas deben estar organizados de forma que, sin importar los ancestros que se comparten, las ligaduras correctas para cada rama sean fácilmente discernibles.

En una ejecución secuencial, la ligadura de una variable es almacenada en la ubicación de memoria asignada a esa variable. Como las ramas de árbol se exploran de a una, las ligaduras se deshacen durante el backtracking y no surgen problemas. En contraste, en una ejecución paralela, múltiples ligaduras existen simultáneamente, por lo que no pueden ser almacenadas en una única locación de memoria asignada a la variable. Este problema, conocido como el problema de las múltiples representaciones de entorno, es el problema principal en la implementación del paralelismo OR.

Se distinguen dos tipos de variables y ligaduras:

- *Incondicionales*: son variables que se crearon en un nodo de nivel inferior, y cuya ligadura ya fue asignada con anterioridad. Por ejemplo, en la Figura 4.2 la variable `X` es incondicional para $N2$ o $N3$, la ligadura incondicional fue creada en $N1$ por lo tanto $N2$ y $N3$ ven el mismo valor (`X=three`).
- *Condicionales*: son variables que se crearon en un nodo que está más arriba en el árbol, pero que aún no fueron ligadas cuando ocurre la ramificación, entonces cada rama puede ligarla con un valor distinto. Esto ocurre con la variable `K` en la Figura 4.2, que recibe distintos valores en $N2$ y $N3$. La ligadura `one` para la variable `K` es aplicable a sólo aquellos nodos que están debajo de $N2$.

Es importante notar que el problema del manejo de múltiples entornos es justamente el de la representación y acceso eficiente de las ligaduras condicionales. Las ligaduras incondicionales pueden ser tratadas como en una ejecución secuencial normal. El problema

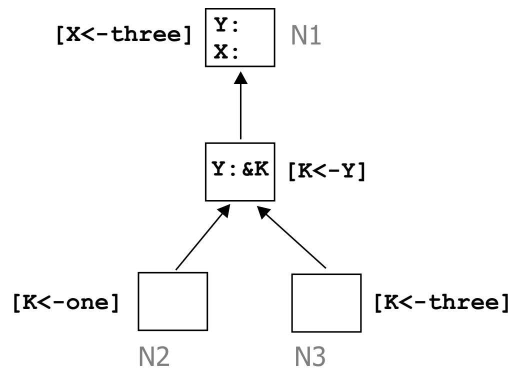

Figura 4.2: Tipos de variables y ligaduras (rama derecha de la Figura 4.1)

del manejo de múltiples entornos o ambientes debe ser resuelto diseñando un mecanismo donde cada rama tenga ciertos datos en áreas privadas, donde puede almacenar las ligaduras condicionales aplicables a sí mismo. Hay varias formas para lograr esto:

a) almacenar la ligadura condicional creada por una rama en un arreglo o tabla hash privada a esa rama;

b) mantener una copia separada del entorno para cada rama del árbol;

c) guardar las ligaduras condicionales en una estructura de datos global, junto con un identificador único para cada ligadura que identifica a la rama a la que pertenece.

Cada aproximación tiene sus costos. Estos costos no son constantes y son incurridos en el momento del acceso a variables, en el momento de la creación de un nodo, o en el momento en que un proceso comienza con la ejecución de una nueva rama (conmutación). Idealmente se buscaría un modelo en que estos costos fueran constantes, i. e., que el tiempo para realizar las operaciones sea independiente de la cantidad y el tamaño de los nodos en el árbol. Sin embargo, es imposible satisfacer los tres criterios simultáneamente, es decir, mantener los tres costos constantes a la vez en un sistema con un número finito de procesadores (demostrado en [GJ93]). En el manejo de múltiples entornos or-paralelos siempre habrá un tiempo no constante. A pesar de que estos tiempos no constantes no pueden evitarse, se pueden reducir mediante un diseño cuidadoso del planificador, cuya función es asignar las tareas a los workers (procesos). Estas tareas involucran explorar una rama del árbol, representando una alternativa sin probar en un punto de elección. El

diseño del planificador es importante para evitar una excesiva cantidad de conmutaciones y para manejar cálculos especulativos apropiadamente.

4.2.1. Modelos de ejecución con paralelismo OR

Según el valor –constante o no constante– de los costos de acceso, creación y conmutación, se obtiene la clasificación de la Figura 4.3 [GPA+01], donde se muestra para cada combinación uno de los modelo propuesto en la literatura (ver Cuadro 4.1 para una clasificación más completa). Como se puede observar, hay varios de estos modelos que logran dos costos constantes; se tomará para análisis un representante de cada uno de estos grupos: árbol de directorio, ventanas de hashing y arreglos de ligaduras.

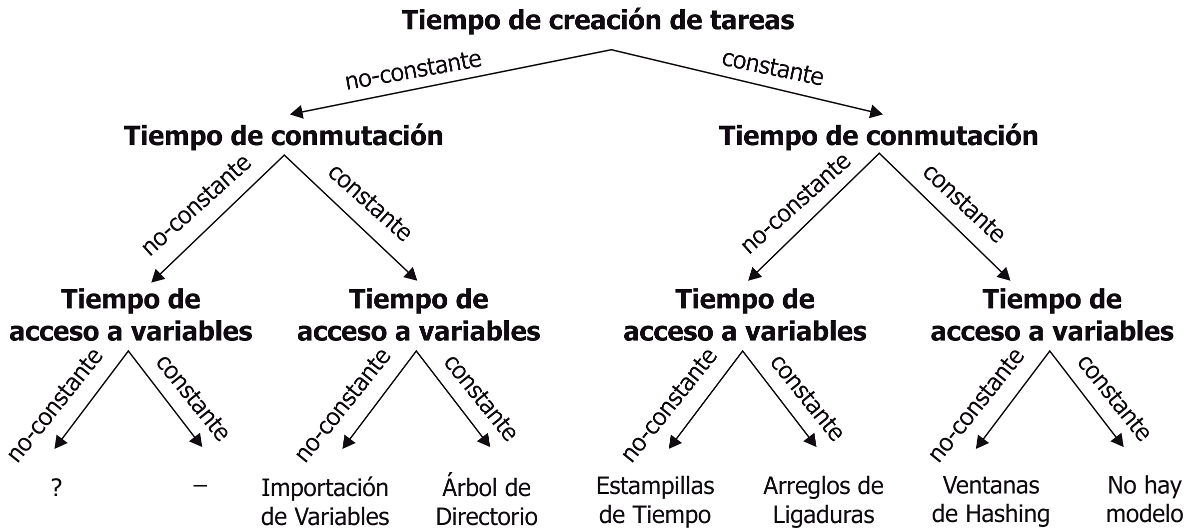

Figura 4.3: Modelos or-paralelos

A continuación se describirán aquellos modelos que están en negrita en el Cuadro 4.1, y que tienen tiempos no constantes sólo en uno de los tres criterios (los más eficientes). Cada uno de los modelos analizados, al haber sido desarrollados por distintos autores, presentan sus propios ejemplos con una notación particular lo cual dificulta el análisis y comparación de los mismos. Aquí ilustraremos cada uno de los métodos presentados respetando siempre el ejemplo de árbol or-paralelo de búsqueda de la Figura 4.1.

Método	Tiempo constante en creación de tareas	Tiempo constante en acceso a variables	Tiempo constante en conmutación
Vectores de versión (Version vectors)	✓	✓	
Arreglos de ligaduras (Binding arrays)	✓	✓	
Método Argonne-SRI	✓	✓	
Manchester-Argonne	✓	✓	
Estampillas de tiempo (Time stamping)	✓		
Ventanas de hashing (Hashing windows)	✓		✓
Modelo Simple (Naive model)	✓		✓
Modelo Argonne	✓		✓
Árbol de directorio (Directory tree)		✓	✓
Cierre de entorno (Environment closing)		✓	✓
Importación de variables (Variable import)			✓
Kabu-wake	✓	✓	
Máquina BC	✓	✓	
Muse	✓	✓	
Modelo abstracto (Abstract method)		✓	✓
Delphi	✓	✓	
Modelo aleatorio (Randomized method)	✓	✓	

Cuadro 4.1: Clasificación en costos de los distintos modelos

Árbol de directorio

En el modelo de árbol de directorio cada rama del árbol or-paralelo tiene un proceso asociado [CH86]. Un proceso es creado cada vez que se crea un nuevo nodo en el árbol, y el proceso expira una vez que se completa la creación de los procesos hijos.

El directorio de un proceso es un arreglo de referencias a contextos. El entorno del proceso consiste de contextos apuntados por su directorio. La ubicación i en el directorio contiene una referencia al i-ésimo contexto para ese proceso. Se crea un nuevo contexto

para cada cláusula invocada. Cada proceso tiene un entorno de ligaduras separado, pero puede compartir algunos de los contextos en su entorno con procesos de otras ramas. El entorno completo de ligaduras está descrito por el directorio.

Cuando ocurre una ramificación, se crea un nuevo directorio por cada proceso hijo. Por cada contexto en el proceso padre que tiene al menos una variable sin ligar, se crea una nueva copia, y se ubica un puntero a ella en el directorio del hijo con el mismo desplazamiento que tenía en el directorio del padre. Los contextos que no tienen variables sin ligar pueden ser compartidos ubicando un puntero a ellos en el directorio del hijo.

Una variable condicional se representa por una terna ⟨*dirección del directorio, offset del contexto, offset de la variable*⟩. Con este modelo, todas las variables son accedidas en tiempo constante, y la conmutación de procesos no involucra ningún cambio de estado. Sin embargo, el costo de la creación del directorio es potencialmente muy alto y el método conduce a un gran consumo de memoria y escasa localidad.

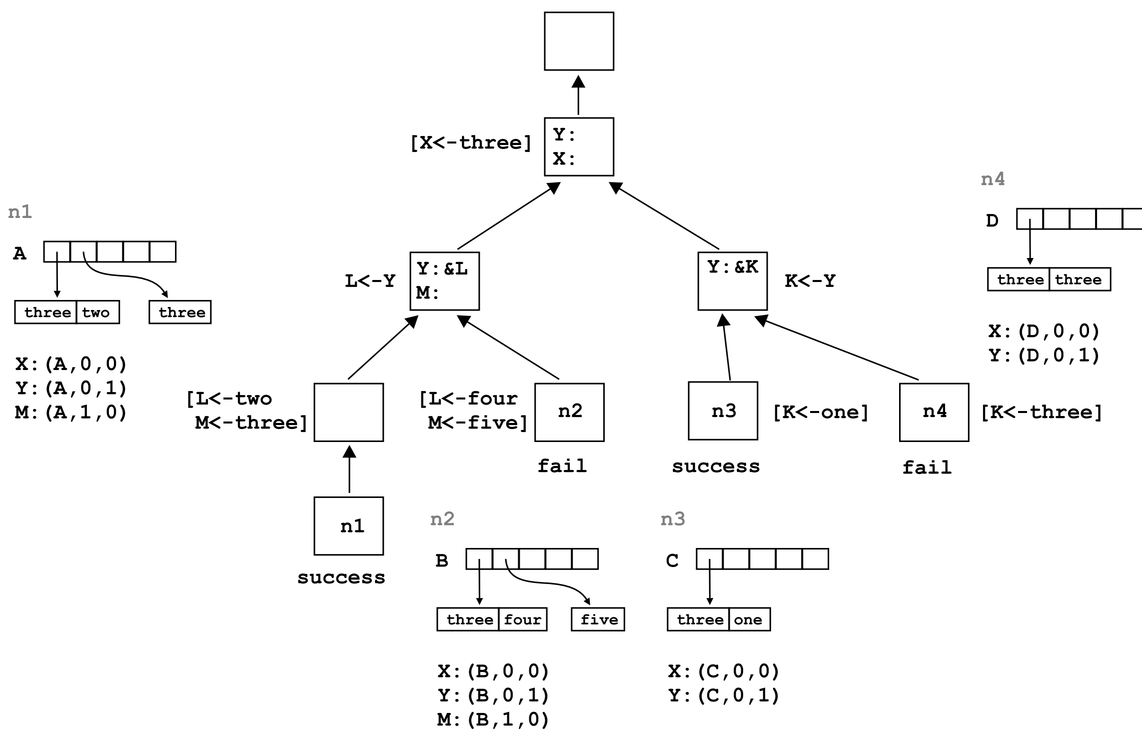

Figura 4.4: Árbol de directorio

Ventanas de hashing

En este esquema se mantienen los distintos entornos mediante ventanas de hashing (*hashing windows*). La ventana de hashing es esencialmente una tabla hash. Cada nodo en el árbol or-paralelo tiene su propia ventana de hashing donde se almacenan las ligaduras condicionales de un nodo particular. La función hash se aplica a la dirección de la variable para computar la dirección de la cubeta donde la ligadura condicional será almacenada. Las ligaduras incondicionales no se ubican en la ventana de hash, sino que se guardan en el mismo nodo que creó la ligadura (éste será alguno de los ancestros del nodo actual).

Durante el acceso a las variables, la función hash se aplica a la dirección de la variable cuya ligadura se necesita y se chequea la cubeta resultante en la ventana hash del nodo actual. Si no se encuentra ningún valor, se busca recursivamente en la cubeta del padre hasta que se encuentra la ligadura o se llega al nodo en donde se creó la variable. Si se llega a este último nodo es porque la variable permanece sin ligar. Las ventanas de hash no necesitan ser duplicadas cuando ocurren ramificaciones porque son compartidas.

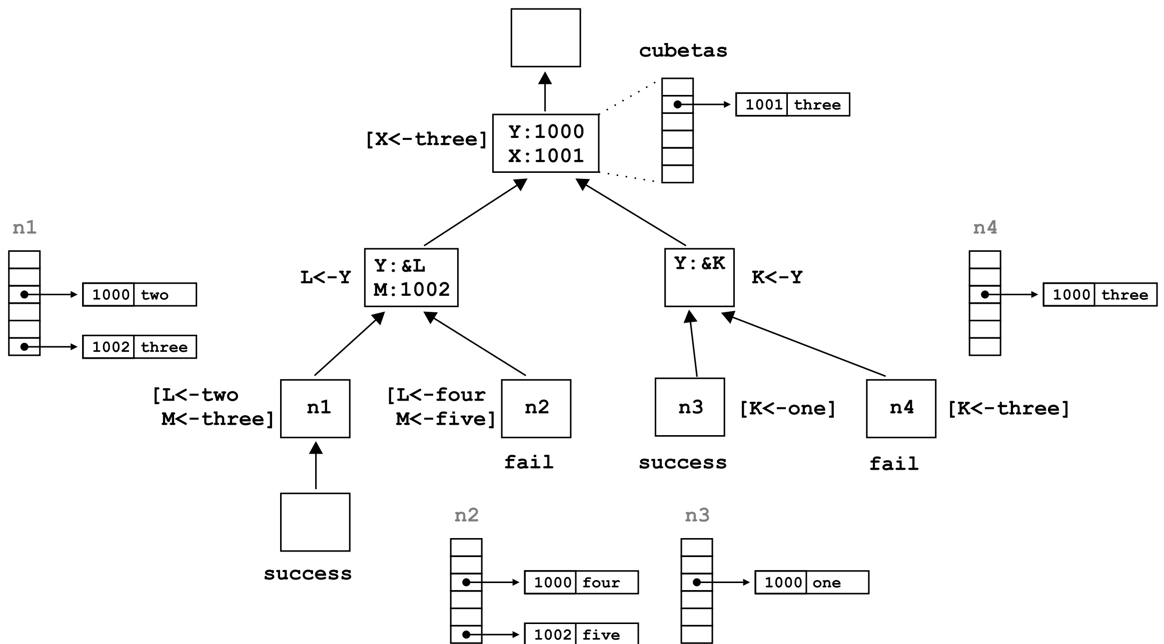

Figura 4.5: Ventanas de hashing

Arreglos de ligaduras

En el modelo de *binding arrays* cada procesador –y no un proceso– posee una estructura auxiliar llamada arreglo de ligadura. Cada variable condicional a lo largo de una rama es numerada secuencialmente desde la raíz. Para realizar esta numeración, cada rama mantiene un contador (cuando ocurre una ramificación cada rama obtiene una copia del contador), como se muestra en la Figura 4.6. Cuando se crea una variable condicional se marca como uno –seteando un bit–, y el valor del contador se graba en ella (este valor se conoce como el counter-offset de una variable). Luego, el contador se incrementa. Cuando una variable condicional recibe un valor, la ligadura se almacena en el arreglo de ligaduras del procesador en la ubicación dada por el counter-offset de esa variable. Asimismo, la ligadura condicional junto con la dirección de la variable condicional se almacena en el trail de la WAM [AK91, War83] (se extiende el trail para incluir también ligaduras). Las entradas almacenadas en el trail se muestran entre corchetes en la figura. Si después se necesita la ligadura de la variable, se utiliza el valor del counter-offset de la variable para indexar el arreglo de ligaduras y obtener la ligadura. Se debe notar que las ligaduras de todas las variables, condicionales o incondicionales, son accesibles en tiempo constante.

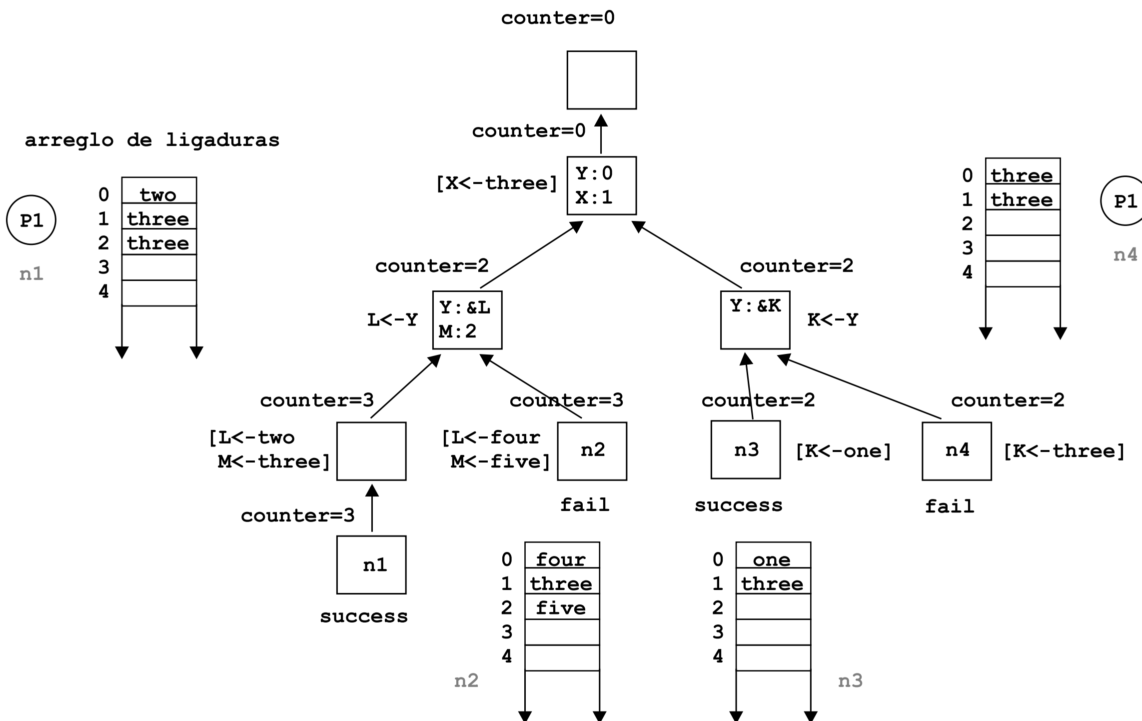

Figura 4.6: Binding arrays

Para asegurar la consistencia, cuando un procesador conmuta de una rama del árbol or-paralelo a otra, debe actualizar su arreglo de ligaduras desinstalando los valores del trail del nodo origen e instalando las ligaduras correctas del trail del nodo destino. Por ejemplo, en la Figura 4.6, hay dos ramas que se ejecutan en el procesador $P1$, luego, al comenzar con la ejecución de la segunda rama debe desinstalar los valores correspondientes a $n1$ para establecer los correctos para $n4$.

4.3. Paralelismo AND Independiente

El paralelismo AND independiente (IAP) se refiere a la ejecución paralela de metas que no tienen dependencias de datos, y entonces no se afectan entre sí.

Análogamente al caso del paralelismo OR, el desarrollo de una computación and-paralela puede ser descrita a través de una estructura de árbol (*árbol and*). En este caso, cada nodo del árbol se etiqueta con la conjunción de las submetas que representa y tiene tantos hijos como submetas en la conjunción. El paralelismo AND se presenta, por ejemplo, en algoritmos de 'dividir y conquistar', donde muchas llamadas recursivas independientes pueden ser ejecutadas en paralelo (e. g., multiplicación de matrices, quicksort). Un ejemplo de este tipo de árboles se presenta en la Figura 4.7.

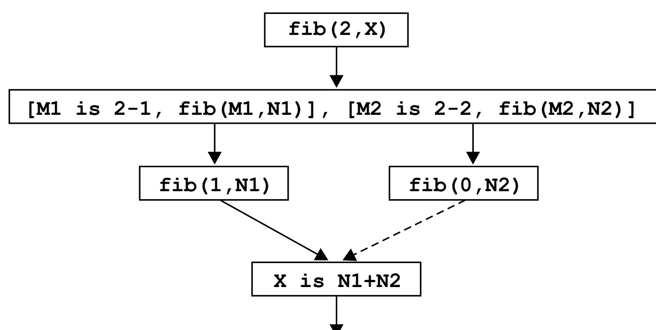

Figura 4.7: Árbol AND en la ejecución de Fibonacci

La ejecución and-paralela se divide en tres fases:

- *Fase de ordenamiento*: trata con la detección de dependencias entre metas.
- *Fase de ejecución hacia adelante*: realiza los pasos necesarios para seleccionar la próxima submeta y comenzar su ejecución.

- *Fase de ejecución hacia atrás*: se encarga de los pasos a seguir cuando una meta falla, esto es, la operación de backtracking.

4.3.1. Fase de ordenamiento

El objetivo de este proceso es descubrir todo el paralelismo que sea posible, mientras se garantiza la computación de los resultados correctos y se mejoran o preservan (no-slowdown[1]) otras características observables del programa, como el tiempo de ejecución (que influirá en el speed-up).

Se puede analizar la correctitud y eficiencia del paralelismo utilizando la técnica instrumental de reordenamiento (*interleaving*) de sentencias. Por ejemplo, en:

```
1  Y:= W+2;
2  X:= Y+Z;
```

Las sentencias 1 y 2 en lenguaje imperativo son consideradas dependientes porque si se altera su orden se obtendrá un resultado incorrecto; esto es, se viola la condición de correctitud (es una dependencia de flujo). En cambio, en un programa lógico como el siguiente, revertir el orden de las sentencias 2 y 3 arroja el resultado correcto (X=a).

```
1  main :-
2      p(X),
3      q(X),
4      ...
5  p(X) :- X=a.
6  q(X) :- X=b, <computación intensa>
7  q(X) :- X=a.
```

De hecho, este caso es instancia de una regla general: *"si no hay efectos secundarios, reordenar las sentencias en un programa lógico no afecta a su correctitud"*.

Este hecho derivó en los primeros modelos en una estrategia de ejecución donde el paralelismo era explotado "completamente" (i. e., todas las sentencias eran elegibles para su paralelización). Sin embargo, el problema fue que tal paralelización usualmente violaba

[1]Esta condición afirma que la ejecución paralela de programas lógicos no debe ser más lenta que su contraparte secuencial

el principio de eficiencia: para un número finito de procesadores, el programa paralelizado podía ser más lento que su versión secuencial, aún bajo suposiciones ideales con respecto a la sobrecarga en tiempo de ejecución. Por ejemplo, en el fragmento de código anterior, revertir el orden de las llamadas a `p` y `q` en el cuerpo de `main` implica que la llamada a `q(X)` (`X` no tiene ningún valor) entrará en la primer alternativa, ejecutando la computación intensa. En el retorno de `q` (con `X` apuntando a la constante `b`) la llamada a `p` fallará y el sistema hará backtracking a la segunda alternativa de `q`, en la que tendrá éxito con `X=a`. Lo que se debe observar es que, en la ejecución secuencial, `p` afecta a `q`, en el sentido de que limita sus elecciones.

Este tipo de comportamiento motivó a que se definieran nuevas nociones de independencia que capturan los conceptos de correctitud y eficiencia al mismo tiempo. Luego, las metas independientes serán aquellas cuyo comportamiento en tiempo de ejecución, si se paralelizan, produce los mismos resultados que en su ejecución secuencial y un incremento (o al menos, no un decremento) en la performance. Una meta o llamada a procedimiento, `q`, no puede ser afectada por otra, `p`, si no comparten variables lógicas en el momento de su ejecución. En estos casos de *independencia estricta*, se respetan la correctitud y eficiencia y se garantiza la condición de no-slowdown.

En el ejemplo siguiente se puede ver que en `main1`, `p` y `q` son estrictamente independientes, porque a pesar de que comparten a `Y`, en ejecución `Y` referencia a un valor constante. Sin embargo, `r` y `q` no son estrictamente independientes porque comparten `L`.

```
main1 :-                     main2 :-
    X=f(K, g(K)),                t(X, Y),
    Y=a,                         p(X),
    Z=g(L),                      q(Y).
    W=h(b, L),
    p(X, Y),
    q(Y, Z),
    r(W).
```

Desafortunadamente, no siempre es fácil determinar si dos metas son independientes con sólo observar un predicado. En el caso de `main2`, es posible determinar que `p` y `q` no son (estrictamente) independientes de `t`, ya que al entrar al cuerpo del procedimiento, `X` e `Y` son variables libres que están compartidas con `t`. Sin embargo, luego de la ejecución

de `t` las estructuras creadas (y referenciadas por `X` e `Y`) pueden no compartir variables. Para determinar esto debe realizarse un análisis global –interprocedural– del programa (en este caso, para determinar el comportamiento de `t`). Como alternativa, se puede realizar un test en tiempo de ejecución justo después de la computación de `t`, para detectar la independencia de `p` y `q`. Esto tiene un efecto indeseable debido a los overheads que implica el test, que hace que la propiedad de no-slowdown no se cumpla automáticamente, pero sigue siendo potencialmente útil.

Varias aproximaciones han sido propuestas para abordar el tema de la detección de dependencias de datos discutido anteriormente. Éstas van desde técnicas puramente en tiempo de compilación a otras completamente en tiempo de ejecución. Existe obviamente un trade-off entre la cantidad de paralelismo explotado y la sobrecarga del análisis de las dependencias de datos incurrido en tiempo de ejecución. Las dependencias de datos no siempre pueden ser completamente detectadas en tiempo de compilación, aunque las herramientas de análisis pueden cubrir una porción significativa de tales dependencias. Algunos de los modelos propuestos se enumeran a continuación:

1. *Input output modes*: el programador indica explícitamente el modo de las variables (si son de entrada o de salida).

2. *Static data dependency analysis*: el análisis es hecho en tiempo de compilación, asumiendo el peor caso.

3. *Run-time dependency graphs*: genera un grafo de dependencias en tiempo de ejecución. Para esto, examina las ligaduras de las variables relevantes cada vez que una submeta termina de ejecutarse.

4. Un híbrido entre las dos últimas aproximaciones, *RAP (restricted and-parallelism)*: encapsula la información de dependencias en el código generado por el compilador, agregando información sobre las variables. De esta manera, se pueden hacer chequeos simples en ejecución para analizar las dependencias.

4.3.2. Fase de ejecución hacia adelante

Una vez cumplida la fase de ordenamiento, la fase de ejecución hacia adelante selecciona las metas independientes que pueden ser corridas con paralelismo AND independiente,

e inicia su ejecución. La ejecución continúa como una ejecución normal secuencial de Prolog hasta que se encuentra una solución u ocurre una falla, en cuyo caso se entra en la fase de ejecución hacia atrás.

La implementación de esta fase es bastante directa, donde el problema más importante es la determinación eficiente de las metas que están listas para su ejecución paralela. Esto pone énfasis en la importancia de una buena estrategia de planificación. Se han desarrollado técnicas sofisticadas de planificación para garantizar una mejor correspondencia entre la organización lógica de la computación y su distribución física en memoria (pilas), con el objetivo de simplificar el backtracking y la performance de la memoria.

Algunos investigadores proponen una metodología que adapta los mecanismos de planificación desarrollados para sistemas or-paralelos al caso de paralelismo AND independiente (e. g., [Dut94], [PG95], [Her87]). Análogamente a la forma en que un sistema or-paralelo intenta planificar primero el trabajo que tiene mayor posibilidad de éxito, los sistemas and-paralelos buscarán provecho al planificar primero las tareas que tienen mayor probabilidad de fallar. La ventaja de realizar esto deriva del hecho que la mayoría de los sistemas con IAP soportan formas inteligentes de backtracking sobre las llamadas and-paralelas, lo que permite propagar rápidamente la falla de una submeta a toda la llamada. Luego, si la llamada no tiene solución, se encontrará más pronto una submeta que falle y entonces se iniciará el backtracking.

4.3.3. Fase de ejecución hacia atrás

La necesidad de una fase de ejecución hacia atrás se justifica a partir de la naturaleza no determinística de la programación en lógica: la ejecución de un programa involucra la elección de una cláusula candidata en cada paso de la resolución, y esta elección puede conducir potencialmente a distintas soluciones. La fase de ejecución hacia atrás aparece cuando ocurre una falla, o cuando se buscan más soluciones para una consulta de alto nivel después de que se reportó una. Se determina la submeta hacia la que se debe hacer el backtracking, se restaura el estado de la máquina abstracta (WAM [AK91, War83]) y se inicia la fase de ejecución hacia adelante de la submeta seleccionada.

En presencia de IAP, el backtracking se vuelve considerablemente más complejo, en especial si el sistema intenta explorar el espacio de búsqueda en el mismo orden que en una ejecución secuencial de Prolog. Distintas características justifican esta complejidad:

- El IAP lleva a la pérdida de la correspondencia entre la organización lógica de la computación y su disposición física; esto significa que submetas lógicamente contiguas (i. e., submetas que están una después de la otra en el cuerpo de la regla) pueden estar físicamente ubicadas en partes no contiguas de la pila, o en pilas de diferentes workers (en distintos procesadores). Para ejemplificar esta situación se puede observar la Figura 4.8. El worker 1 comienza con la ejecución paralela, dejando a `b` y `c` disponibles para ejecución remota y empezando localmente con la ejecución de `a`. El worker 2 inmediatamente comienza y completa la ejecución de `b`. Mientras tanto, el worker 1 abre una nueva llamada paralela, localmente ejecutando `d` y dejando `e` para otros workers. En este punto, el worker 2 puede elegir ejecutar `e`, y luego `c`. En la figura se muestra la disposición final de las submetas en las pilas de los dos workers.
- El proceso de backtracking puede necesitar continuar con la submeta (lógicamente) precedente, la cual puede estar aún ejecutándose cuando el backtracking toma lugar.
- Estos problemas se acentúan más si consideramos que las submetas and-paralelas independientes pueden tener otras submetas IAP anidadas en ejecución, las cuales deben ser finalizadas o procesadas en backtracking.

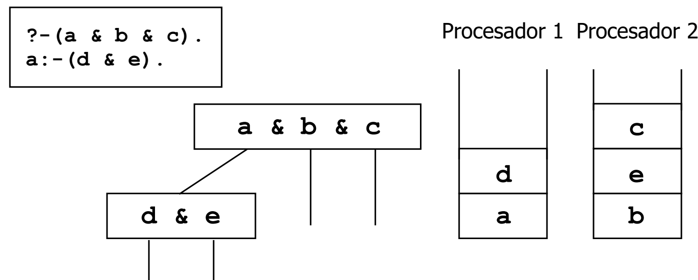

Figura 4.8: Pérdida de la correspondencia entre la disposición lógica y física

Varias aproximaciones han sido adoptadas en la literatura para manejar la fase de ejecución hacia atrás. La más simple se basa en remover la necesidad del backtracking sobre las metas and-paralelas por medio de la utilización de paralelismo y reuso de soluciones. Por ejemplo, diferentes threads son asignados a distintas submetas y se utilizan para generar (con técnicas locales de backtracking estándar) todas las soluciones, y luego se realizan los productos cruzados. Esto tiene algunas desventajas, como la complejidad

que existe para recrear la semántica de Prolog, es decir, para establecer el orden correcto de ejecución de los predicados sensibles al orden. Además, tiene un alto costo ya que en una meta como `?- p, q`, la submeta `q` se recomputa completamente para cada solución de `p`.

En una aproximación más reciente y ampliamente utilizada se plantean distintas formas de backtracking. Este algoritmo también ha sido extendido para manejar paralelismo AND dependiente. Para presentarlo consideremos la siguiente consulta:

```
?- b1, b2, (q1 & q2 & q3), a1, a2
```

donde "&" indica conjunción paralela, i. e., las submetas pueden resolverse concurrentemente (mientras "," se mantiene para representar conjunción secuencial). Se pueden dar distintos casos si alguna submeta falla:

- Si `a2` o `b2` fallan, se utiliza un backtracking estándar a partir de `a1` o `a2` respectivamente.
- Si `a1` falla (*outside backtracking*) entonces el backtracking debe proceder dentro de la llamada paralela, en la submeta `q3`. El hecho de que `a1` se estaba ejecutando implica que toda la llamada paralela estaba completa. En este caso, lo más preocupante es identificar la ubicación de la computación `q3`, que puede residir en una parte distinta de la pila (no inmediatamente debajo de `a1`) o en la pila de otro worker. Si `q3` no ofrece soluciones alternativas, entonces, como en Prolog estándar, se hará backtracking hacia `q2` y eventualmente a `q1`.
- Si algún $\mathtt{q}_i$ falla (*inside backtracking*) entonces:
  - las submetas $\mathtt{q}_j$ con $j > i$ deben removerse de la pila (interrumpir o cancelar su ejecución).
  - cuando la computación de $\mathtt{q}_{i-1}$ se completa, debe iniciarse el backtracking para buscar nuevas alternativas.

  Lo más importante es que, en la práctica, todos estos pasos pueden evadirse confiando en el hecho de que las submetas paralelas son independientes. Luego, la falla de una de las submetas no puede solucionarse haciendo backtracking en cualquiera de las otras submetas paralelas. Hermenegildo [HN86] sugirió una forma

de backtracking semi-inteligente donde la falla de cualquiera de los $\mathtt{q}_i$ causa la falla de toda la conjunción paralela y el backtracking se hace en `b2`.

4.4. Paralelismo AND Dependiente

El paralelismo AND dependiente (DAP) generaliza al independiente permitiendo la ejecución concurrente de submetas que acceden a conjuntos de variables no disjuntos. Un ejemplo podría ser la consulta `?- p(X) & q(X)` donde las dos submetas pueden competir (o cooperar) en la creación de una ligadura para `X`. Una ejecución sin restricciones de este tipo puede producir un comportamiento no determinístico: el resultado dependerá del orden en que las dos submetas acceden a `X`. Luego, uno de los objetivos de cualquier sistema que explota el paralelismo AND dependiente es asegurar que el comportamiento operacional de la ejecución and-paralela es consistente con la semántica pretendida (las semánticas observables de Prolog, en este caso). Ello requiere:

- asegurar que todas las submetas paralelas coinciden en los valores dados a las variables compartidas; y
- garantizar que el orden en que se realizan las ligaduras no lleva a ninguna violación en el comportamiento observable del programa.

Típicamente, en una consulta como la anterior, `p` producirá una ligadura para `X` mientras `q` la procesará (o la consumirá). Para que se preserve este orden, `q` será suspendido hasta que la ejecución de `p` termine. Sin embargo, no siempre es así, y la ejecución de `p` y `q` puede ser solapada:

1. `q` puede realizar una cantidad significativa de cálculos antes de necesitar el valor de `X`; esta computación puede ser solapada con la computación de `p`, porque no depende de `X`;
2. `p` puede instanciar parcialmente a `X`. En tal caso, `q` puede empezar a trabajar con ese valor, mientras `p` continúa ocupado computando el resto de la ligadura para `X`.

El alcance de la explotación del DAP depende fuertemente de la semántica del lenguaje de programación en lógica elegido. Por ejemplo, la ejecución DAP de programas en Prolog

puro (donde no hay predicados sensibles al orden), conduce a una implementación simple y podría producir speed-ups potencialmente altos.

Soportar DAP incluye:

- *detección de paralelismo*: involucra determinar qué submetas serán consideradas para su ejecución en paralelo;
- *manejo de metas DAP*: activación y manipulación de las submetas paralelas;
- *manejo de variables compartidas*: validación y control de las variables compartidas para garantizar las semánticas de Prolog; y
- *backtracking*: manejo del no determinismo.

El manejo de submetas (segundo item) no presenta nuevos desafíos con respecto al manejo de submetas paralelas en el contexto de IAP. Por lo tanto, se considerarán en las siguientes secciones los items restantes.

4.4.1. Detección de paralelismo

Realizar anotaciones en un programa para una ejecución fructífera en DAP se asemeja en algunos aspectos a una paralelización automática para IAP. Esto se justifica pensando que DAP no es más que una instancia con granularidad más fina del principio de independencia, aplicado al nivel de ligaduras de variables.

Una detección automática y semiautomática de fuentes potenciales de paralelismo DAP identifica y hace explícitas las variables que se comparten entre metas en la conjunción paralela. Para las variables compartidas se realiza un renombramiento, que reemplaza cada variable por una nueva. Esto hace que cada submeta en la conjunción tenga acceso puro e independiente a cada variable compartida. Por ejemplo, en el siguiente programa que invierte una lista, la variable Z es compartida:

```
nrev([X|Xs], Y):-
        nrev(Xs, Z),
        append(Z, [X], Y).
```

entonces procede a renombrarla con `Z1` y `Z2`:

```
nrev([X|Xs], Y) :-
        $mark([Z]),
        ( $and_goal([[Z,Z1]], nrev(Xs, Z1)) &
        $and_goal([[Z,Z2]], append(Z2, [X], Y)) ).
```

La directiva al compilador `$mark/1` identifica variables compartidas. Las variables compartidas obtienen nombres distintos en cada una de las metas paralelas (marcadas con `$and_goal`).

El proceso de anotaciones en un programa para la explotación de paralelismo AND dependiente opera a través de sucesivos refinamientos:

1. La identificación de las cláusulas que tienen una estructura compatible con la explotación de DAP, esto es, contienen al menos un grupo de predicados consecutivos que no son predefinidos. Cada grupo maximal de metas contiguas forma una *partición*.

2. Se identifican las variables compartidas en cada partición.

3. Se refina la partición para mejorar el comportamiento DAP:

   – se colapsan submetas consecutivas,

   – se separan particiones en subparticiones, y

   – se remueven submetas que se encuentran al comienzo o al final de una partición.

Las transformaciones anteriores surgen de los siguientes principios:

- las submetas paralelas deben presentarse en una granularidad suficientemente grande para superar los overheads de paralelización; y

- las submetas dependientes dentro de una partición deben demostrar un buen grado de solapamiento en sus ejecuciones.

4.4.2. Manejo de variables

La primer necesidad que surge al manejar variables compartidas es la de garantizar la exclusión mutua durante la creación de una ligadura para tal variable. Otro item, y quizás más importante, concierne al proceso de validación de la ligadura, i. e., garantizar que el resultado de la computación respete las semánticas observables de una ejecución secuencial de Prolog.

Exclusión Mutua

La mayoría de los esquemas DAP utilizan una única representación de cada variable compartida: todos los threads de la computación acceden al mismo área de memoria que representa a la variable compartida. En cualquier momento, a lo sumo uno de esos threads tendrá permitido ligar la variable. Sin embargo, la construcción de una ligadura para la variable no es una operación atómica, a menos que el valor asignado sea atómico. En general, la variable sin ligar primero recibe una plantilla del término que será construido, y luego sucesivamente se construyen los subtérminos de la ligadura.

Si se permite a los consumidores un acceso continuo a las ligaduras creadas por el productor, se necesitan introducir mecanismos capaces de garantizar la atomicidad de una ligadura en variables compartidas.

Para lograr esto, Parlog [RR88] hace que el compilador genere un orden distinto de instrucciones durante la construcción de los términos complejos, pero el puntero a la estructura no se escribe hasta que la estructura completa ha sido construida. En el modelo de Andorra-I [SCWY91a], los términos que necesitan corresponder con un término compuesto se bloquean, y se agrega una instrucción especial (`last`) para marcar el final de la construcción de términos y liberar el lock. En el sistema DASWAM [She92] el enfoque consiste en modificar las instrucciones WAM `unify` y `get` de manera que sobrescriban ubicaciones sucesivas de memoria en el heap con un valor especial. Cada acceso al término inspeccionará tal ubicación sucesiva para verificar si la ligadura está completa.

Validación de ligaduras

Los distintos enfoques pueden clasificarse de acuerdo a ciertos criterios ortogonales:

Tiempo de validación:

- remover las inconsistencias en las ligaduras de las variables compartidas sólo si aparece un conflicto y amenaza las semánticas de Prolog (esquemas curativos).
- prevenir inconsistencias retardando y ordenando apropiadamente las ligaduras en variables compartidas (esquemas preventivos).

Resolución de la validación:

- realizar la actividad de validación al nivel de submetas paralelas (goal-level validation).
- realizar la actividad de validación a nivel de las variables compartidas individuales (binding-level validation).

Los enfoques curativos a nivel de submeta implican usualmente que cada submeta and-paralela desarrolla su computación en copias locales del entorno. Al final de la llamada paralela se introduce un paso adicional para verificar la consistencia de los valores producidos por las distintas computaciones para las variables compartidas (Epilog [Wis86], ROPM [Kal85]). A nivel de ligaduras, la validación permite que las ligaduras se realicen, pero se necesitan acciones especiales de rollback cuando se detecta una violación a la semántica del programa.

En los enfoques preventivos las ligaduras de variables compartidas se previenen a menos que esté garantizado que no amenaza a las semánticas de Prolog. A nivel de metas, estos esquemas retardan la ejecución de la submeta completa hasta que su ejecución no afecte la semántica. A nivel de ligaduras, los esquemas preventivos permiten explotar un mayor grado de paralelismo. La mayoría de tales esquemas cuentan con reforzar una noción de semántica más fuerte (strong Prolog semantics): las ligaduras a variables compartidas se realizan en el mismo orden que en la ejecución secuencial de Prolog.

4.4.3. Backtracking

Mantener las semánticas de Prolog durante una ejecución paralela también significa soportar computaciones no determinísticas, i. e., computaciones que pueden potencialmente producir múltiples soluciones. En muchos casos, DAP ha sido restricto a computaciones

donde `p` y `q` son determinísticos, dada la complejidad del backtracking distribuido. Pero, esto limita demasiado la cantidad de paralelismo que puede explotarse.

El backtracking en DAP es más complejo que en el caso de IAP (ver Sección 4.3.3). Mientras que el *outside backtracking* permanece sin cambios, el *inside backtracking* pierde su naturaleza independiente, que garantizaba el backtracking semi-inteligente descripto anteriormente. La falla de una submeta dentro de una conjunción paralela no lleva a la falla de toda la conjunción, pero requiere 'matar' a las submetas de la derecha y hacer backtracking sobre la submeta inmediatamente a la izquierda, una actividad asincrónica, ya que la submeta puede estar ejecutándose.

Asimismo, el backtracking dentro de una submeta paralela también puede afectar la ejecución de otras submetas paralelas. En una conjunción paralela como `p(X) & q(X)`, el backtracking sobre `p(X)` puede llevar a una modificación del valor de `X` que requiere deshacer la ejecución de `q(X)`, porque `q(X)` puede haber consumido el valor de `X` que acaba de ser retrocedido.

4.5. Paralelismo en Prolog

El interés en la ejecución paralela de programas lógicos deriva de los siguientes hechos:

- El paralelismo puede ser explotado implícitamente (i. e., sin intervención del usuario).
- Los lenguajes lógicos permiten que el programador pueda expresar su algoritmo de una forma que refleje la estructura del problema más directamente.
- Existe la creencia de que los lenguajes de programación en lógica tienen una eficiencia en ejecución muy baja. Esto ha motivado a los investigadores a utilizar el paralelismo como una alternativa para alcanzar velocidad.

Existen muchos sistemas que explotan sólo paralelismo AND como &-Prolog y &-Ace, sólo paralelismo OR como Muse y PLoSys, o una combinación del paralelismo OR y AND. Sistemas que combinan ambos tipos continúan en desarrollo e investigación, entre ellos se destacan ANDORRA-I, ACE, ParAKL y DAOS.

Claramente, la paralelización también puede ocurrir de manera explícita. Esto puede lograrse extendiendo al lenguaje de programación en lógica con constructores explícitos para concurrencia o modificando la semántica del lenguaje lógico de programación de manera adecuada. Los lenguajes que requieren una especificación explícita se pueden clasificar en tres categorías:

1. Aquellos que agregan primitivas de *pasaje de mensajes* a Prolog, por ejemplo DeltaProlog y CS-Prolog. Múltiples procesos de Prolog se corren en paralelo y ellos se comunican entre sí a través del intercambio de mensajes u otro mecanismo de *rendez-vous* (e. g., eventos).

2. Aquellos que agregan primitivas de *blackboard* a Prolog. El modelo Linda [CG89] puede utilizarse para la comunicación de procesos en programas lógicos y también para sincronización, utilizando el blackboard compartido. CIAO Prolog soporta multithreading y primitivas de concurrencia que permiten al programador utilizar la base de datos como un blackboard (también implementa la librería de Linda).

3. Aquellos basados en *guardas*, committed choice, por ejemplo Parlog, GHC y Concurrent Prolog. Esta categoría también incluye a algunas extensiones de sistemas de programación en lógica más tradicionales para ejecución distribuida (e. g., &-Prolog/Ciao y ACE).

A continuación se evaluarán las formas de paralelismo presentes en algunas implementaciones de Prolog. Se eligieron los lenguajes intentando cubrir un espectro lo más amplio posible, tomando un ejemplo significativo de cada clase (paralelismo OR, IAP y DAP). Se describirá Andorra-I como un ejemplo de lenguaje con paralelismo implícito. En cuanto a paralelismo explícito, veremos los predicados que brindan CIAO Prolog, Parlog, Quintus Prolog, PVM-Prolog y DeltaProlog.

Gupta *et al.* en [GPA+01] discute y compara sistemas como DASWAM, Muse, Aurora, &-Ace y &-Prolog. Bal [Bal91] analiza a Emerald, SR, Orca, Parlog y Linda.

Andorra-I: (1989)

- Definición: es un sistema experimental de Prolog paralelo que implementa paralelismo implícito.

- Creadores: D. H. D. Warren, V. Santos Costa y R. Yang. University of Bristol, UK.
- Implementa: paralelismo OR (binding arrays) y AND dependiente.
- Arquitecturas: MIMD (multiprocesadores).
- Características: las metas determinantes se ejecutan en primer orden; provee una forma de corutinas implícitas; los programas son ejecutados por equipos de agentes abstractos (workers).

CIAO Prolog: (1997)

- Definición: el sistema Ciao es un ambiente de programación completo para desarrollar programas en lenguaje Prolog y en otros lenguajes que son extensiones o modificaciones a Prolog.
- Creadores: M. Carro, M. Hermenegildo y otros. Universidad Politécnica de Madrid, España.
- Implementa: paralelismo AND independiente (se encuentra en versión experimental).
- Arquitecturas: MIMD (multiprocesadores).
- Características: provee predicados para chequear la independencia de variables; también soporta multithreading e implementa primitivas de concurrencia que permiten al programador utilizar la base de datos como un blackboard. Se encuentra aún en desarrollo.

Parlog: (1983)

- Definición: es una versión de Prolog paralelo que permite la evaluación concurrente de cláusulas de Horn.
- Creadores: K. L. Clark y S. Gregory. Imperial College.
- Implementa: paralelismo OR y AND explícito.
- Arquitecturas: MIMD.
- Características: utiliza la sintaxis tradicional de Prolog para denotar paralelismo e incluye una sintaxis alternativa para especificar lo que es secuencial;

los argumentos de los predicados tienen asociada una declaración de modos (entrada/salida).

Quintus Prolog: (1986)

- Definición: es la implementación industrial estándar de Prolog.
- Creador: Quintus Computer Systems Inc. Swedish Institute of Computer Science (SICS), Suecia.
- Implementa: una forma de RPC (remote procedure calling) con estructura cliente/servidor.
- Arquitecturas: MIMD (multicomputadoras).
- Características: el paralelismo que puede explotarse surge a partir de la comunicación entre procesos y es algo limitado; muy buena performance en general.

PVM-Prolog: (1996)

- Definición: es una interface entre el sistema PVM y el lenguaje Prolog.
- Creadores: R. F. P. Marques y J. C. Cunha. Universidade Nova de Lisboa, Portugal.
- Implementa: paralelismo a través del pasaje de mensajes.
- Arquitecturas: MIMD.
- Características: los procesos de PVM-Prolog se corresponden con tareas en PVM; mantiene todas las funcionalidades de PVM accesibles al nivel de Prolog. Soporta aplicaciones multi-lenguaje.

DeltaProlog: (1984)

- Definición: es un lenguaje de programación en lógica que extiende Prolog con constructores para composición de metas secuenciales y paralelas, comunicación entre procesos y sincronización, y no-determinismo externo.
- Creadores: L. Pereira y R. Nasr. Universidade Nova de Lisboa, Portugal.

- Implementa: paralelismo OR y DAP.
- Arquitecturas: MIMD (multicomputadoras).
- Características: define metas de eventos sincrónicas y asincrónicas; incluye un mecanismo de backtracking distribuido y ambas formas de no-determinismo (*don't care* y *don't know*). Provee paralelismo OR a partir de las metas para elección no determinística. Provee paralelismo DAP realizando renombramiento de variables para resolver las dependencias.

Para resumir las características recién presentadas, se clasificarán en el siguiente cuadro los distintos lenguajes lógicos vistos de acuerdo al tipo de paralelismo que implementan (OR, AND) y si se codifica implícita o explícitamente. Los lenguajes que no implementan paralelismo OR o paralelismo AND, utilizan algún otro mecanismo para expresar las computaciones paralelas, quizás rescatado de algún lenguaje imperativo (ver en el Cuadro 4.2 las notas al pie para esos casos).

	Andorra-I	**CIAO Prolog**	**Parlog**	**Quintus Prolog**	**PVM-Prolog**	**DeltaProlog**
Paralelismo OR	si	no	si	no	no	si
Paralelismo AND	DAP	IAP	DAP	no	no	DAP
Implícito	si	no	no	no	no	no
Explícito	no	si	si	si[2]	si[3]	si
Hardware	MP	MP	MP/MC	MC	MP/MC	MC

Cuadro 4.2: Clasificación de los lenguajes lógicos

[2] QUINTUS PROLOG utiliza un mecanismo cliente/servidor, similar a RPC, no se puede clasificar como ninguna de las formas de paralelismo descritas.

[3] PVM-PROLOG implementa el paralelismo con un sistema de pasaje de mensajes como se realiza en PVM [GBD+94].

4.5.1. Andorra-I

Andorra-I [SCWY91a] es un sistema experimental de Prolog paralelo que aprovecha de manera transparente el paralelismo OR y AND dependiente (ver Secciones 4.2 y 4.4). Constituye la primera implementación del modelo Básico de Andorra [War88, Har90], un modelo de ejecución paralela de programas lógicos en el cual las metas determinantes son ejecutadas antes que otras metas. Este modelo, además de combinar las dos formas más importantes de paralelismo implícito, también provee una forma de corutinas implícitas.

La idea principal es que, en vez de ejecutar la meta de más a la izquierda como en el Prolog tradicional, primero se ejecutan todas las metas para las cuales exista a lo sumo una cláusula que unifica (i. e., metas determinantes). Estas metas se ejecutan con paralelismo AND. Cuando no hay más metas con esa forma, se crea un *punto de elección* como en Prolog. Las diferentes ramas desde ese punto de elección se exploran con paralelismo OR. El modelo da origen naturalmente a una forma de corutinas implícitas, donde las ligaduras de las variables durante la ejecución de dos metas pueden ser pasadas tanto de derecha a izquierda, como de izquierda a derecha en el Prolog estándar.

La conjunción es tratada con paralelismo por defecto. No obstante, se introduce un nuevo operador para representar una conjunción que está restringida a una ejecución secuencial.

Los programas en Andorra-I son ejecutados por equipos de agentes abstractos llamados *workers*. Cada worker normalmente se corresponde con un procesador físico. Cada equipo, cuando está activo, se asocia con una rama separada del árbol or-paralelo, y se encuentra en alguna de las dos fases de computación:

- **Determinante**: mientras existan metas determinantes dentro de la rama, todas esas metas serán candidatas para su ejecución inmediata. Esta fase termina cuando no hay más metas determinantes disponibles o cuando alguna de ellas falla. Si se da el primer caso, el equipo se mueve hacia la fase no determinante. Si una meta falla, la rama correspondiente debe ser abandonada. El equipo iniciará el backtracking y eventualmente intentará encontrar una nueva rama para explorar.
- **No determinante**: Si no hay metas determinantes, se reduce la meta de más a la izquierda. Se crea un punto de elección para representar el hecho de que la rama actual se ha subdividido en varias ramas del árbol or, mientras que el equipo en

sí mismo va a explorar una de ellas. Si hay otros equipos disponibles se pueden utilizar para explorar las ramas restantes.

Andorra-I tiene dos componentes principales: un motor y un preprocesador. El motor ejecuta código de dos tipos: código interpretado representando las cláusulas del programa, y el código para cada procedimiento generado por el preprocesador. El motor integra el trabajo previo en la implementación del paralelismo DAP y OR, mientras que el preprocesador genera secuencias de instrucciones que complementan la definición de los procedimientos, y que son luego utilizadas por el motor para detectar las características determinantes de las metas de manera eficiente.

El paralelismo OR implementado utiliza arreglos de ligaduras (binding arrays, ver Sección 4.2.1). Esto trae algunos problemas ya que los workers dentro de un mismo equipo tienen que compartir el mismo arreglo de ligaduras. La solución es dividir el arreglo en partes para permitir que cada worker asigne las entradas independientemente. En Andorra-I el planificador OR y el planificador AND son completamente independientes: el primero planifica a los workers dentro de un equipo; el segundo planifica a los equipos entre sí. Para el momento en que Santos Costa *et al.* [SCWY91a] escribían su artículo, el número de equipos era fijo. Sería importante poder adaptar el tamaño y el número de equipos dinámicamente para extraer todo el paralelismo que sea posible, quizás a través de un planificador de más alto nivel.

Andorra-I es completamente funcional y soporta paralelismo OR y AND. Sin embargo, se necesita más trabajo relacionado con el manejo de memoria y las facilidades de debugging. El sistema Andorra-I corre en máquinas secuenciales y en multiprocesadores. Andorra-I es capaz de obtener buenos speed-ups a partir del paralelismo OR y el paralelismo AND. En casos adecuados, el speed-up obtenido al explotar ambas formas de paralelismo combinado es mejor que el que se puede conseguir aprovechando sólo uno de ellos. Este lenguaje ha sido utilizado en el desarrollo de sistemas de manejo de tráfico avanzado que usó British Telecom para controlar el flujo de tráfico en sus redes telefónicas [Has95], y en una variedad de aplicaciones de telecomunicaciones [SCWY91b].

4.5.2. CIAO Prolog

CIAO Prolog posee una librería que implementa paralelismo AND independiente: `andprolog` [BCC+97]. Esta librería provee al programador de predicados para codificar

explícitamente el paralelismo presente en su aplicación. Básicamente, las metas son ejecutadas de forma and-paralela, asumiendo que sus argumentos no comparten ligaduras, i. e., que no están ligados a términos que contienen una variable común. Esta librería no está completa, aún se encuentra en desarrollo. Sólo se devuelve la primer solución a la conjunción debido a consideraciones de performance, pero los autores prometen que esta es una restricción que planean remover en un futuro cercano.

CIAO PROLOG tiene su origen en el compilador y motor de Prolog paralelo de &-Prolog [HG91]. Los predicados que se incluyen son:

&/2:

`&(GoalA, GoalB)`

Las metas `GoalA` y `GoalB` son ejecutadas en modo IAP. Esto es sólo un primer ensayo, y es sólo válido para metas determinísticas e independientes. Se puede utilizar como `q:- a & b`, lo que iniciaría a `a` y `b` en threads separados (posiblemente en paralelo si la máquina y el sistema operativo lo permiten), y continúa cuando ambos hayan terminado. Este tipo de ejecución es sólo seguro cuando `a` y `b` son independientes en el sentido de que no compartan variables. Esta condición puede testearse con el predicado `indep/2`.

active_agents/1:

`active_agents(NumberOfAgents)`

Testea o setea el número de agentes que se encuentran activos buscando metas para ejecutar. Estos agentes consumen recursos, aún cuando están buscando trabajo y sin ejecutar ninguna meta.

indep/2:

`indep(X, Y)`

Los argumentos `X` e `Y` son independientes si están ligados a términos que no tienen variables en común. Por ejemplo, `indep(X, Y)` es verdadero para `X=f(Z), Y=g(K)` y para `X=f(a), Y=X`, pero no se cumple para `X=f(Z), Y=g(Z)` y para `X=Y`.

indep/1:

`indep(X)`

El argumento `X` representa una lista de listas de longitud dos, i. e., una lista de la forma `[[T1, T2], [T3, T4], ...]`. Las variables en cada par de la lista `X` son testeadas por independencia utilizando `indep/2`.

Además, CIAO Prolog cuenta con un módulo con primitivas de bajo nivel para concurrencia/multithreading. El mismo provee los mecanismos básicos para utilizar concurrencia e implementar aplicaciones con múltiples metas a ser resueltas. Dispone de medios para que se especifique que metas arbitrarias sean ejecutadas en un conjunto de pilas separadas; en ese caso, se les asigna un identificador de meta con el que se pueden realizar consultas posteriores a la meta (e. g., preguntar por más soluciones). Adicionalmente, en algunas arquitecturas, estas metas pueden ser asignadas a un thread del sistema operativo distinto del que hizo la llamada principal, proveyendo capacidades de concurrencia y, en multiprocesadores, paralelismo.

4.5.3. Parlog

En Parlog [RR88, CG86] las cláusulas se separan mediante el operador de búsqueda paralelo indicado por '.', o el operador de búsqueda secuencial indicado por ';'. Esto quiere decir que, un conjunto de cláusulas separadas por el '.' tradicional se evaluarán en paralelo (paralelismo OR); mientras que un conjunto de cláusulas separadas por ';' se evaluarán en orden secuencial como en Prolog estándar.

Este lenguaje trabaja con *cláusulas de Horn con guarda*, de la forma:

$$H \leftarrow G_1 \ op \ \ldots \ op \ G_n : \ B_1 \ op \ \ldots \ op \ B_m.$$

donde H es un átomo (la cabeza de la cláusula); $G_1, \ldots, G_n$ es un conjunto de átomos que forman la guarda; y $B_1, \ldots, B_m$ es un conjunto de átomos que forman el cuerpo de la cláusula. El símbolo ':' separa la guarda del cuerpo de la cláusula. El símbolo 'op' se reemplaza por el operador para conjunción paralela ',' o para conjunción secuencial '&'. Parlog utiliza la sintaxis tradicional de Prolog para denotar paralelismo e incluye una semántica alternativa para especificar lo que es secuencial.

Dada una meta B, la resolución de los átomos en B que están separados por el operador de conjunción paralela se hace en paralelo (paralelismo AND), y aquellos separados por la conjunción secuencial se realizan secuencialmente. Para un átomo A, la búsqueda de

las cláusulas candidatas para resolver A se hace en paralelo para aquellas cláusulas que están separadas por el operador de búsqueda paralelo y serialmente para las cláusulas separadas por el operador de búsqueda secuencial (paralelismo OR).

Además, en un programa PARLOG cada predicado tiene asociado una declaración de modo, i. e., un conjunto de anotaciones que definen el comportamiento de E/S de los argumentos del predicado [Cia92]. Una variable marcada con '`?`' es una variable de entrada, y una variable marcada con '`^`' es una variable de salida. El único mecanismo de comunicación entre procesos es justamente mediante la unificación de variables compartidas. La declaración de modos es un atributo estático del programa: el efecto es que el proceso lógico correspondiente a ese predicado se suspende si una restricción de entrada no se satisface, i. e., una variable de entrada debe estar unificada antes de intentar evaluar la guarda. Usualmente, una variable compartida tiene un productor y varios consumidores. No se permite que un consumidor cree una ligadura para una variable compartida. Luego, una meta de la forma:

```
?- prod(X), cons(X).
```

debe invocar a un programa `prod(X^)` y a un programa `cons(X?)`. Si el consumidor precisa un valor en uno de sus canales de entrada y el productor aún no ha escrito en el canal, el consumidor se suspende (*stream parallelism*). La sincronización de las tareas paralelas se realiza implícitamente, utilizando la suspensión en las variables lógicas sin ligar.

Se han implementado intérpretes de PARLOG para distintos multiprocesadores con memoria compartida. Strand (*STReam AND-parallelism*), un subconjunto de PARLOG disponible comercialmente, ha sido implementado para sistemas distribuidos (hipercubos y redes).

4.5.4. Quintus Prolog

En QUINTUS PROLOG [Int03] el paralelismo que puede explotarse surge a partir de la comunicación entre procesos, y es algo limitado. Un programa en QUINTUS PROLOG puede utilizar evaluación paralela, utilizando un *esclavo* en otro procesador y comunicándose a través de una red. Esta interface tiene entonces dos lados: el programa en Prolog que realiza la llamada (master) y el programa que es llamado (esclavo). Cada uno debe realizar ciertas funciones que permiten la cooperación.

Una vez que un programa ha sido cargado, sus reglas y hechos residen en la base de datos de Prolog. Es posible salvar este estado de la base de datos en un archivo; a este estado salvado lo denominan *saved-state*. Esto permite restaurar la base de datos a un estado determinado sin tener que recompilar los archivos fuentes.

Para utilizar paralelismo, el programa master debe primero crear su esclavo, iniciando un proceso de Prolog que será el esclavo. El sistema crea un proceso al correr un estado creado previamente por el programador. Después de que el esclavo ha sido creado y está corriendo, el maestro le envía metas para evaluar utilizando `call_servant/1`. Todas las metas enviadas al esclavo son evaluadas en la base de datos del esclavo, que es disjunta de la base del master. Sin embargo, con este predicado el programa master se queda esperando hasta que el proceso esclavo termina de computar y en consecuencia no tenemos un comportamiento paralelo. El predicado `bag_of_all_servant/3` ambos procesos (en distintas máquinas) progresan en simultáneo.

Los predicados principales son:

create_servant/3:

`create_servant(+Machine, +SavedState, +OutFile)`

Antes de que un maestro pueda utilizar un esclavo, el esclavo debe ser iniciado y se deben realizar las conexiones con él. El argumento `Machine` representa el nombre de la máquina donde se correrá el esclavo. `SavedState` es el nombre del archivo que contiene el estado guardado de Prolog en esa máquina, que debe haber sido previamente creado con `save_servant/1`. `OutFile` es el nombre del archivo en el cual se escribirá la salida del esclavo. Este archivo se ubica en la máquina local, y también contendrá información de errores.

call_servant/1:

`call_servant(+Goal)`

Una vez que el esclavo fue creado se le pueden enviar metas para evaluación utilizando esta primitiva. Ésta envía la meta `Goal` al esclavo, que evalúa y envía los resultados. Las soluciones después de la primera se obtienen haciendo backtracking en `call_servant/1`. En realidad, el esclavo computa y envía todas las soluciones al maestro, aún cuando el que realiza la llamada descarte todas menos la primera.

bag_of_all_servant/3:

`bag_of_all_servant(?Template, +Goal, -Bag)`

Ejecuta una meta `Goal` devolviendo los valores a los que unifican las variables incluidas en `Template` dentro el conjunto de todas las soluciones posibles. Luego `Bag` es una lista con los valores de `Template`. La operación exacta depende de la forma de `Goal`. Si la meta a evaluar es una conjunción de la forma `(Goal1, Goal2)` o una disjunción de la forma `(Goal1; Goal2)`, entonces la primer submeta se ejecutará en el esclavo y la segunda en el proceso actual. El sistema tratará de solapar la evaluación local y remota tanto como sea posible (aparentemente este es el único caso de ejecución paralela). Si `Goal` no es una conjunción o disjunción, la meta completa se enviará al cliente para su ejecución.

Hay algunas restricciones en cuanto al uso, a saber: `Goal2` no debe contener ningún *cut*; y `Goal2` no puede requerir servicios del esclavo, es decir, no puede incluir llamadas a `call_servant/1` o `bag_of_all_servant/3`.

4.5.5. PVM-Prolog

PVM-Prolog [Mar96] es una extensión de Prolog que hereda su claridad y lectura declarativa [CM96]. Al expresar el paralelismo, la distribución de tareas, los esquemas de comunicación y el manejo de máquina virtual dentro de PVM-Prolog, el programador se libera de los detalles de bajo nivel y se puede concentrar principalmente en la organización lógica de sus algoritmos.

PVM-Prolog permite la composición de aplicaciones híbridas, basadas en múltiples motores de procesos de PVM-Prolog, PVM-C y PVM-FORTRAN. Estos motores para distintos lenguajes de programación pueden coexistir dentro el mismo ambiente de máquina virtual. Esto es interesante en aplicaciones especializadas, como por ejemplo en sistemas de robótica heterogéneos. También es interesante si queremos desarrollar sistemas inteligentes distribuidos donde los procesos de PVM-Prolog actúen como managers o controladores de otros componentes de software, escritos en diferentes lenguajes.

El sistema de PVM-Prolog consiste de dos componentes diferentes: el motor de procesos (PE, *process engine*) y la máquina virtual (PPVM). El motor de procesos es el bloque base que representa las entidades computacionales en una aplicación de PVM-Prolog particular. Se corresponde con la máquina abstracta que soporta cada semántica

específica de los lenguajes de programación. Cada motor de procesos debe soportar un motor de inferencia específico de Prolog. Múltiples PEs pueden coexistir en el mismo entorno de máquina virtual y cooperar entre ellos.

La máquina virtual (PPVM, Parallel Prolog Virtual Machine) provee primitivas para la activación y control de los motores de procesos, para comunicación y sincronización de los procesos, y para el manejo del ambiente paralelo y distribuido subyacente.

Con el objetivo de mantener la máxima flexibilidad posible en la interface, se decidió mantener toda la funcionalidad de PVM accesible al nivel de Prolog. La mayor inquietud en la definición de la interface fue adaptar las funciones de PVM de acuerdo a la semántica del lenguaje Prolog. Los principales aspectos de diseño fueron:

- Los mensajes son interpretados como términos de Prolog, y las extensiones a las funciones de empaquetado y desempaquetado de PVM se proveen con el objeto de convertir representaciones en aplicaciones híbridas y heterogéneas.
- Todos los predicados exhiben un estricto comportamiento determinístico, i. e., fallan en el backtracking.
- El pasaje de parámetros de entrada y salida aprovecha el mecanismo de unificación de Prolog (las listas de Prolog son convertidas a arreglos de C y viceversa).
- Las situaciones de error en la invocación de predicados son manejadas por falla.

La máquina virtual se organiza en capas, diseñadas por encima de sistemas existentes como el sistema PVM y el motor de Prolog. Estos dos componentes a su vez se apoyan en las capas más bajas de sistema y hardware. Este acercamiento en capas tiene muchas ventajas conocidas, que se cree equilibran a la pérdida de performance. Particularmente mejora la portabilidad, flexibilidad y generalidad. La interface de PPVM la define una capa baja, PPVM0, que consiste de una interface básica a PVM. Sobre ella se provee una capa alta, PPVM1, que consiste de un conjunto de predicados de interface que ofrecen la semántica de alto nivel para control de procesos y comunicación.

Los procesos de PVM-Prolog se corresponden con tareas en PVM. Los identificadores de tareas en PVM se utilizan para identificar a los procesos de Prolog, como nombres de átomos, aunque el usuario puede definir su propio sistema de nombres en una capa superior.

Más allá de la interface con PVM, PVM-Prolog agrega manejo de threads dentro de Prolog, para permitir concurrencia con granularidad más fina [Mar96]. A continuación se describen algunas funciones de ejemplo de PVM al nivel de Prolog.

Creación e Identificación de procesos:

```
pvm_spawn(+progname, +goal)
pvm_spawn(+progname, +goal, +opt_list, +where, +ntasks, -tid_list,
          +StacksSize, +HeapSize)
```

En general, una cantidad `ntasks` de tareas (sólo una en `pvm_spawn/2`) son creadas para resolver una meta `goal` dada. El argumento `progname` es el nombre del archivo que contiene el programa de Prolog. Con respecto a la semántica operacional, primero se crea una nueva tarea de PVM para ejecutar una instancia del motor de inferencia de Prolog (Cunha y Marques [CM96] utilizan NanoProlog). Luego, se consulta el archivo de Prolog especificado y se activa la meta de alto nivel que corresponda. El nuevo proceso es completamente independiente de su padre; el usuario debe controlar las interacciones entre padre e hijo para obtener las soluciones.

```
pvm_mytid(-tid)
```

Como en el caso de PVM, este predicado devuelve el identificador del proceso. Además, si el proceso aún no es una tarea de PVM se convierte en una.

Comunicación:

```
pvm_send(+tid, +msgtag, +term)
pvm_mcast(+tid_list, +msgtag, +term)
pvm_[n]recv(+tid, +msgtag, -msg)
```

En el nivel PPVM1 se decidieron ocultar todas las primitivas para manejo de buffer porque las representaciones de Prolog están basadas en términos. En estos predicados se presenta un identificador de proceso destinatario (`send`) del mensaje o del proceso originador del mismo (`recv`), un tag de mensaje y el término que corresponde al mensaje (`term` y `msg`). El predicado `pcm_recv` realiza una recepción bloqueante, mientras que `pvm_nrecv` realiza una recepción no bloqueante y falla si no hay mensajes pendientes en ese momento. Si el `tid` y/o el `msgtag` contienen un

'-1' indican que se espera recibir un mensaje de cualquier proceso, con cualquier identificador, según sea el caso.

Configuración de PVM:

`pvm_mstat(+host, -mstat)`

Este predicado provee información sobre el estado de una máquina real, `host`. El argumento `mstat` se instancia a 'ok', 'down' o 'unknown'.

`pvm_addhosts(+hostlist, -infolist)`
`pvm_delhosts(+hostlist, -infolist)`

La lista `hostlist` contiene un conjunto de hosts a ser incluidos o eliminados del entorno de PVM (de la máquina virtual), e `infolist` devuelve los códigos de éxito de la operación para cada host.

Notificación de eventos:

`pvm_notify(+about, +msgtag, +tidlist)`

Este predicado permite la implementación una capa de manejo de fallas de alto nivel, encargada de los errores en hosts y tareas. El mismo promueve el envío de un mensaje (con tag `msgtag`) a una lista de hosts (multicast) cuando un evento determinado ocurre –de acuerdo a lo especificado en `about`–, como puede ser la destrucción de una tarea, la eliminación o caída de un host, o la inclusión de un nuevo host.

4.5.6. DeltaProlog

Prolog está basado en lógica de primer orden; los eventos relacionados con el tiempo no son tratados satisfactoriamente en ese marco. DeltaProlog es un marco formal de la programación en lógica concurrente que extiende la lógica pura de cláusulas de Horn con la noción de *eventos*: una forma de comunicación entre procesos relacionada con el tiempo [Cia92].

DeltaProlog [CMCP92] es en realidad una extensión de Prolog: de hecho, las submetas atómicas de una meta son, por defecto, evaluadas secuencialmente. Se agregan:

- una *meta fork* (utilizando el operador '`//`') que expresa paralelismo AND. La coma ('`,`') permite la especificación de una restricción secuencial para la activación de las metas. Es decir, en `a, b//c, d` se activa primero `a`, seguido de una ejecución paralela de `b` y `c` y, tras su finalización correcta, se ejecutaría la meta `d`.
- una meta para elección no-determinística (utilizando el operador '`::`' o una de sus variantes). Esto posibilita a que un proceso realice una elección no-determinística entre varias alternativas, en coordinación con su entorno, i. e., con otros procesos del sistema.

DeltaProlog es el único lenguaje lógico concurrente que ofrece ambas formas de no-determinismo: *don't care* (commited choice) y *don't know* [Cia92]. DeltaProlog incluye un conjunto de metas especiales denominadas *metas de eventos*:

- una meta de evento sincrónico (representado por los operadores binarios '`!`' y '`?`');
- una meta de evento sincrónico sin backtracking (operadores '`^^^`' y '`???`');
- una meta de evento asincrónico (operadores '`^^`' y '`??`');

Una meta de evento es un literal especial que facilita la transmisión de valores entre cláusulas que son evaluadas en paralelo (por procesos diferentes). Una meta de evento sincrónica de la forma `G!E:Cond1` produce un valor para transmitir a una cláusula receptora que tenga una meta de evento de la forma `G?E:Cond2`. En la notación, `G` es el término a ser transferido, `E` es un nombre global distinguiendo esta transacción de cualquier otra que pueda estar llevándose a cabo en el mismo momento, y `Cond1`, `Cond2` son metas de Prolog. Dos procesos que se comunican mediante el evento `E`, deben ejecutar las metas complementarias:

```
proceso1 :-                 proceso2 :-
    Term1!E:Cond1             Term2?E:Cond2
```

Ambas tienen éxito si y sólo si `Term1` unifica con `Term2`, y las condiciones `Cond1` y `Cond2` son satisfechas localmente en cada proceso, y así en consecuencia, el evento `E` tiene éxito. Si la unificación de términos falla, la cláusula emisora es suspendida, mientras que la receptora falla. Si una de las condiciones no es satisfecha, el evento no ocurre y se inicia el backtracking (distribuido) en ambos procesos.

La comunicación asincrónica se soporta en DeltaProlog con los predicados `^^` y `??`. En este caso el nombre del evento individualiza a un canal de comunicación (un buffer) entre los dos procesos:

`Term^^canal` significa "enviar el término al canal y continuar"

`Term??canal` significa "esperar hasta recibir el término del canal, y luego continuar"

No ocurre backtracking distribuido cuando un proceso realiza backtracking sobre una meta de evento asincrónica (es como la semántica del `read` y `write` de Prolog). El principal problema de implementación de DeltaProlog es justamente el backtracking cuando se tienen que manejar metas de eventos. ¿Qué debería suceder cuando en la elección de una meta la falla de la alternativa seleccionada inicia el backtracking? ¿Cómo puede ser distribuido para incluir procesos diferentes que no sean el que falló? Una implementación reciente incluye un mecanismo para un *backtracking distribuido* verdadero: cuando un proceso intenta realizar backtracking en una meta de evento, debe informar a su proceso compañero que él también debe realizar el backtracking en ese evento.

El modelo de programación subyacente de DeltaProlog es muy interesante, porque resulta natural para estructurar una aplicación paralela como una colección de procesos de Prolog que se comunican, cada uno representando una base de datos o un programa de cliente. Sin embargo, este lenguaje no es adecuado para sistemas paralelos de granularidad fina, porque un proceso de DeltaProlog es un proceso con toda la funcionalidad, implementando un intérprete de Prolog. Tal lenguaje es adecuado para aplicaciones distribuidas en donde las comunicaciones sucedan entre agentes (secuenciales) de gran tamaño; por ejemplo, en un sistema distribuido de agentes expertos, cada uno escrito en Prolog, interactuando a través de una red para su coordinación. Un lenguaje como FCP (Flat Concurrent Prolog), GHC (Guarded Horn Clauses) o Parlog es preferible en otros casos, ya que pueden ser utilizados en alto nivel para aplicaciones donde se requiera comunicación y coordinación entre un gran número de procesos livianos.

4.6. Ejemplo comparativo: RankSort

El ejemplo presentado en el Capítulo 3 también se puede implementar con un lenguaje lógico. A continuación se incluye el código para cada uno de los lenguajes estudiados en este capítulo.

Se presentará en primer lugar una versión de RankSort para Prolog estándar (secuencial). Las demás implementaciones serán una versión modificada de este algoritmo principal, o híbridos con la versión imperativa, pero siguen la misma línea de razonamiento. Los predicados que no sufren modificaciones en las sucesivas implementaciones se incluirán sólo la primera vez.

El razonamiento en este ejemplo es simple y directo: se busca el rank del elemento que está primero en el arreglo, se ubica el elemento en la posición que le corresponde y luego se procesa el resto del arreglo de manera análoga hasta que el arreglo no tenga más elementos.

RankSort en Prolog estándar:

```
1  ranksort(A, R):- procesar(A, A, R).
2
3  procesar([], _, _).
4  procesar([X |T], A, R) :-
5          rank(X, A, Pos),
6          ubicar(X, Pos, R),
7          procesar(T, A, R).
8
9  % --- rank/3 ------------------------------------------------------------
10 % rank(X, A, Pos)
11 % cuenta la cantidad de numeros menores a X en el arreglo A y lo
12 % devuelve en Pos.
13 % ---------------------------------------------------------------------
14 rank(X, [Y], 1):- X < Y.
15 rank(X, [Y], 0):- Y =< X.
16 rank(X, [Y |T], Pos):-
17     X < Y,
18     rank(X, T, TPos),
19     Pos is TPos + 1.
20 rank(X, [Y |T], Pos):-
21     Y =< X,
22     rank(X, T, Pos).
23
24 % --- ubicar/3 ----------------------------------------------------------
```

```
25 % ubicar(X, Pos, R)
26 % ubica el elemento X en la posicion Pos del arreglo R, comenzando de 0.
27 % ------------------------------------------------------------------------
28 ubicar(X, Pos, [X |T]):- length(T, Pos).
29 ubicar(X, Pos, [_ |T]):- ubicar(X, Pos, T).
```

RankSort en CIAO Prolog:

```
1  ranksort(A, R):- procesar(A, A, R).
2
3  procesar([], _, _).
4  procesar([X |T], A, R) :-
5          rank(X, A, Pos) & procesar(T, A, R),
6          ubicar(X, Pos, R).
```

RankSort en Parlog:

En PARLOG primero se debe declarar el modo de los argumentos. Se recuerda que en este caso la coma indica ejecución paralela.

```
1  mode ranksort(?, ^),
2  mode procesar(?, ?, ^),
3  mode rank(?, ?, ^),
4  mode ubicar(?, ?, ^).
5
6  ranksort(A, R):- procesar(A, A, R).
7
8  procesar([], _, _).
9  procesar([X |T], A, R) :-
10         rank(X, A, Pos), procesar(T, A, R) &
11         ubicar(X, Pos, R).
```

RankSort en Quintus Prolog:

Antes de ejecutar el programa master debemos correr Prolog en la máquina en que se ejecutará el esclavo, y cargar (esto es, compilar o consultar) todo lo que el

esclavo necesite (la definición del predicado `rank/3`). Luego, se llamará al predicado `save_servant('quintusrank.bd')`.

En el programa master tenemos:

```
init:- create_servant('BBWK01', 'quintusrank.bd', 'outfile01.dat').

ranksort(A, R):- procesar(A, A, R).

procesar([], _, _).
procesar([X |T], A, R) :-
        call_servant(rank(X, A, Pos)),
        procesar(T, A, R),
        ubicar(X, Pos, R).

init.
```

RankSort en PVM-Prolog:

En el programa principal se define:

```
% --- ranksort/2 ----------------------------------------------------------
% ranksort(A, R)
% programa principal que ordena los elementos del arreglo A en el
% arreglo R; crea los workers necesarios, envia los parametros y
% recibe los resultados.
% ------------------------------------------------------------------------
ranksort(A, R):-
    pvm_mytid(_),
    set_opt(route, routeDirect),
    n_workers(NUM_PROCS),
    length(A, CantElementos),
    pvm_spawn(ranksortx, worker, [], [], NUM_PROCS, WorkerTIDS, 1200, 1000),
    pvm_mcast(WorkerTIDS, 1, A),
    enviarElementos(A, WorkerTIDS),
    recibirValores(CantElementos, R),
    pvm_mcast(WorkerTIDS, 2, stop),
```

```
17      pvm_exit.
18
19  % --- enviarElementos/2 ----------------------------------------------------
20  % enviarElementos(A, WorkerTIDS)
21  % envia el elemento del arreglo A que le toca computar a cada uno de los
22  % workers.
23  % ------------------------------------------------------------------------
24  enviarElementos(A, WorkerTIDS):-
25                  enviarElementos(A, WorkerTIDS, WorkerTIDS).
26  enviarElementos(_, [], WorkerTIDS):-
27                  enviarElementos(A, WorkerTIDS, WorkerTIDS).
28  enviarElementos([Elemento |T], [Worker |W], WorkerTIDS):-
29                  pvm_send(Worker, 2, Elemento),
30                  enviarElementos(T, W, WorkerTIDS).
31
32  % --- recibirValores/2 -----------------------------------------------------
33  % recibirValores(CantElementos, R)
34  % recibe los valores computados por cada uno de los workers y va
35  % guardando los elementos en forma ordenada en el arreglo R.
36  % ------------------------------------------------------------------------
37  recibirValores(0, _).
38  recibirValores(CantElementos, R):-
39                 pvm_recv(-1, 3, [Elemento, Pos]),
40                 ubicar(Elemento, Pos, R),
41                 CantElementos1 is CantElementos - 1,
42                 recibirValores(CantElementos1, R).
```

En lo que sería el programa `ranksortx`, que domina el comportamiento de los esclavos o workers, debemos definir:

```
1  % --- worker/0 ------------------------------------------------------------
2  % worker
3  % realiza el computo de los esclavos, recibe parametros, calcula y envia
4  % al proceso padre.
5  % ------------------------------------------------------------------------
6  worker:-
```

```
7          pvm_setop(route, routeDirect),
8          pvm_parent(PTID),
9          pvm_mytid(_),
10         pvm_recv(PTID, 1, A),
11         repeat,
12                pvm_recv(PTID, 2, Orden),
13                worker_do(Orden, PTID)
14         fail.
15
16 % --- worker_do/3 ---------------------------------------------------------
17 % worker_do(Orden, A, Pos)
18 % procesa la orden que envia el proceso principal, que puede ser de
19 % terminar el computo = stop, o de continuar el computo si envia un
20 % elemento valido.
21 % --------------------------------------------------------------------------
22 worker_do(stop, _, _):-
23           pvm_exit, halt.
24 worker_do(Elemento, A, PTID):-
25           rank(Elemento, A, Pos),
26           pvm_send(PTID, 3, [Elemento, Pos]).
```

RankSort en DeltaProlog:

```
1 ranksort(A, R):- procesar(A, A, R).
2
3 procesar([], _, _).
4 procesar([X |T], A, R) :-
5         rank(X, A, Pos) // procesar(T, A, R),
6         ubicar(X, Pos, R).
```

La versión secuencial es la que podría utilizarse con ANDORRA-I para su paralelización automática. En el resto de las implementaciones, el paralelismo es agregado a través de algún operador especial o primitivas un poco más complejas. Los casos de CIAO PROLOG, PARLOG y DELTAPROLOG son muy similares en cuanto a la forma de los predicados;

todos intentan explotar el paralelismo AND que está presente en el ejemplo, buscando solapar el cálculo del rank de un elemento con el procesamiento del resto del arreglo. De alguna manera, este paralelismo de conjunción se asemeja al caso del FORALL en los lenguajes imperativos, ya que el cálculo del rank en subprocesos se realiza en paralelo con la expansión del cómputo en los procesadores disponibles. Lo único que queda afuera es la reubicación de los elementos en el arreglo.

Quintus Prolog es un caso particular, pues nos permite trabajar con una arquitectura cliente/servidor. Aquí se busca crear un esclavo que ejecute en forma concurrente el cálculo del rank. Cabe destacar que según la descripción del comportamiento de `call_servant/1` la ejecución no sería realmente paralela, porque el proceso principal se queda esperando a la finalización del esclavo. Incluso si la ejecución pudiera ser paralela, quizás convendría estructurar el programa de otra forma, como para que el proceso principal y el esclavo se dividan los elementos el arreglo a ordenar -cada uno haría el cálculo del rank de su parte.

Para PVM-Prolog las reglas de la programación declarativa cambian ligeramente. Es el código que más se diferencia del resto de los fragmentos de código que se muestran aquí, pero a su vez el más parecido a los del Capítulo 3 en su comportamiento. Entre el programa principal y los workers se utiliza el intercambio de mensajes para los argumentos de entrada y para los resultados. Esta versión estructura más a la programación en lógica, y le quita parte de su poder declarativo. No obstante, introduce muchas primitivas que aumentan su funcionalidad y posibilitan explotar el paralelismo de forma más completa (siempre explícita).

4.7. Comparación de lenguajes

En esta sección abordaremos un análisis más completo de los lenguajes estudiados. En particular, en el Cuadro 4.3, podemos ver el nivel de abstracción provisto, la forma de descomposición de las tareas, los mecanismos de comunicación y sincronización entre procesos que se utilizan, la interoperabilidad del lenguaje y la granularidad de los procesos.

El nivel de abstracción se considera bajo en Quintus Prolog y PVM-Prolog ya que a la hora de crear un nuevo proceso para su ejecución paralela, el usuario debe proveer información extra como el nombre del archivo de programa, la máquina en que

se desea ejecutar, etc. En los otros lenguajes la descomposición es más automática, dado que implementan alguna forma de paralelismo OR y/o AND.

En todos los lenguajes la descomposición de las tareas es dinámica, i. e., se realiza en tiempo de ejecución por lo que no requiere que se especifique el número de procesos de antemano. Aunque, puede suceder que la cantidad de procesos que se crean tenga una cota superior, por ejemplo en Andorra-I, donde el número y el tamaño de los equipos es fijo.

En los lenguajes de programación en lógica más puros la comunicación generalmente se da a través de la unificación de variables, como en Andorra-I, CIAO Prolog, Parlog y Quintus Prolog. Además, este último provee comunicación a través de archivos de salida (para devolver los resultados). En PVM-Prolog se utiliza el intercambio de mensajes a través de primitivas send/receive y la notificación de eventos. DeltaProlog define eventos sincrónicos y asincrónicos, donde los últimos se comportan como si se tuviera un canal de comunicación entre los procesos.

Parlog obliga a declarar el modo de los argumentos para sincronizar las lecturas y las escrituras de las variables (define quién es el proceso responsable de generar la ligadura y quién la lee). En PVM-Prolog, al estilo PVM, se utilizan mensajes y barreras para sincronización de procesos. Las metas de eventos en DeltaProlog se utilizan para la coordinación de los procesos, como si de mensajes se tratara.

En cuanto a la granularidad de los procesos, en los tres primeros lenguajes la granularidad es fina, ya que a cada uno de los procesos le corresponde una pequeña porción de código para ejecutar, cada vez que se realiza una descomposición (sucede a nivel de predicados). En Quintus Prolog y PVM-Prolog los procesos ya no son tan elementales, pues existe una sobrecarga mayor en la división de las tareas y la creación de los procesos. En DeltaProlog cada proceso implementa un intérprete de Prolog y la granularidad se vuelve más gruesa.

Finalmente, el único lenguaje que presenta interoperabilidad con otros lenguajes es PVM-Prolog a través del concepto de máquina virtual, y de hecho ésta es una de las características más atrayentes de la propuesta.

	Andorra-I	**CIAO Prolog**	**Parlog**	**Quintus Prolog**	**PVM-Prolog**	**DeltaProlog**
Nivel de abstracción	alto	alto	alto	bajo	bajo	alto
Descomposición	dinámica	dinámica	dinámica	dinámica	dinámica	dinámica
Comunicación	unificación de variables	unificación de variables	unificación de variables	unificación de variables y archivos de salida	pasaje de mensajes y notificación de eventos	eventos y canales de comunicación
Sincronización	—	—	declaración de modo de argumentos	—	mensajes y barreras	metas de eventos
Granularidad	fina	fina	fina	mediana	mediana	gruesa
Interoperabilidad	no	no	no	no	si	no

Cuadro 4.3: Comparación de los distintos lenguajes lógicos

4.8. Resumen

El paralelismo en programas lógicos puede verse como una consecuencia directa de su separación entre lógica y control [Kow79]. Dentro de los lenguajes de programación en lógica, la estrategia típica en el desarrollo de sistemas paralelos se basa en la traslación de una o más de las posibilidades de no determinismo presentes en la semántica operacional a computaciones paralelas. Surgen así los tres tipos clásicos de paralelismo: paralelismo de unificación, paralelismo OR y paralelismo AND.

El paralelismo OR y el paralelismo AND son ortogonales entre sí, ya que el primero paraleliza la exploración de las diferentes alternativas en un punto de elección, y el segundo paraleliza la resolución de las distintas submetas. Con ello se podría esperar que la explotación de una de las formas de paralelismo no afecte al aprovechamiento de la otra, e incluso podría ser factible su explotación simultánea. No obstante, en la práctica no es tan fácil integrarlos de manera eficiente. Sistemas que combinan ambos tipos continúan en desarrollo e investigación, como por ejemplo Andorra-I, ACE, ParAKL y DAOS.

Los problemas típicos que se enfrentan al incorporar estas distintas formas de paralelismo implícito a los lenguajes lógicos son, entre otros:

- la presencia de dependencias entre los argumentos de las submetas;
- el manejo de múltiples entornos, donde las variables pueden recibir distintas ligaduras;
- la definición de la estrategia de backtracking que se adopta para obtener la mayor performance posible.

Varias aproximaciones han sido propuestas para abordar el tema de la detección de dependencias de datos. Éstas van desde técnicas puramente en tiempo de compilación a otras completamente en tiempo de ejecución. Existe obviamente un trade-off entre la cantidad de paralelismo explotado y la sobrecarga del análisis de las dependencias de datos incurrido en tiempo de ejecución. La técnica de reordenamiento de sentencias puede producir un incremento en la performance, garantizando la correctitud de los resultados. El renombramiento de variables se utiliza para identificar variables compartidas y permitir que las mismas puedan obtener nombres distintos en cada una de las metas paralelas.

También son muy importantes los algoritmos de planificación para la determinación de las metas que están listas para su ejecución paralela. Éstos buscan garantizar una mejor correspondencia entre la organización lógica de la computación y su distribución física en memoria, con el objetivo de simplificar el backtracking y la performance de la memoria.

Asimismo, la paralelización también puede ocurrir de manera explícita. Esto puede lograrse extendiendo al lenguaje de programación en lógica con constructores explícitos para concurrencia o modificando la semántica del lenguaje lógico de programación de manera adecuada. Los lenguajes de programación en lógica que fueron presentados representan, de alguna manera, las diferentes formas de incorporar el paralelismo.

Se presentó uno de los lenguajes con paralelismo implícito más completo, como es Andorra-I, en el cual el paralelismo es explotado automáticamente, sin intervención del usuario. Dentro de los ejemplos con paralelismo explícito, se utilizan distintas técnicas para obtener el paralelismo. En CIAO Prolog y Parlog se utilizan anotaciones para marcar las metas pasibles de ejecución paralela. Parlog además declara el modo de las variables. En Quintus Prolog el paralelismo se logra a través de primitivas del estilo cliente/servidor para comunicación entre procesos, pero es muy limitado, ya que el programa master puede tener sólo un esclavo. PVM-Prolog extiende el concepto de PVM (que se introdujo en el Capítulo 3) a Prolog, y DeltaProlog agrega manejo de eventos.

Capítulo 5

Programación en paralelo con un modelo de Sistemas Multi-Agente

En este capítulo mostramos cómo las áreas de Sistemas Multi-agente (SMA) y de paralelismo pueden complementarse y ofrecer soluciones a problemas de cada una. En particular se describe cómo utilizar el modelo de SMA para el diseño de programas con paralelismo. Uno de los objetivos es identificar las características que debería poseer un lenguaje de programación con paralelismo que utilice un modelo de Sistemas Multi-agente, cubriendo las componentes de descomposición, comunicación y sincronización.

Los SMA se han concentrado en proveer un modelo abstracto cuya intención es lograr que el programador se involucre más con el modelado del problema y la lógica del programa que con los detalles de la ejecución paralela en sí misma. En cambio, los desarrollos de paralelismo se han orientado más hacia la eficiencia, siendo la mayoría programas de cálculo numérico.

Para cerrar el estudio, se evalúa uno de los pocos lenguajes que existen con estas características. El lenguaje ORGEL constituye una primera aproximación al conjunto de características deseables. El modelo posee una eficiencia y un estilo similar al de la programación secuencial. Asimismo, posibilita la especificación abstracta del paralelismo y las comunicaciones, para no agobiar a los usuarios programadores. Aquí se resaltan las buenas decisiones de diseño y se plantean algunas limitaciones que exhibe el lenguaje, que en su mayoría se deben a que no se ha continuado con la implementación del mismo.

5.1. Agentes y Sistemas Multi-agentes

Los paradigmas de Sistemas Multi-agente abordan las inquietudes que son claves para la próxima generación de aplicaciones de computación: modularidad, tolerancia a fallas, y reusabilidad de componentes.

Un agente puede ser una entidad física o virtual que puede actuar, percibir su entorno –en forma parcial– y comunicarse con otros agentes; es autónomo y tiene habilidades para alcanzar sus objetivos y metas. Un sistema multi-agente contiene un entorno, objetos y agentes, relaciones entre las entidades, un conjunto de operaciones que pueden ser realizadas por las entidades y por cambios en el universo en tiempo y en respuesta a las acciones.

En términos generales podemos definir a un agente como un sistema de hardware o software que posee las siguientes características:

- Autonomía: los agentes funcionan sin la intervención directa de seres humanos o de otros, y tienen cierta clase de control sobre sus acciones y estado interno.
- Capacidad social: los agentes interactúan con otros agentes (y posiblemente seres humanos) mediante un lenguaje de comunicación definido.
- Reactividad: los agentes perciben su ambiente (que puede ser el mundo físico, un usuario vía una interface gráfica, otros agentes, Internet, o quizás todos estos combinados), y responden en una manera oportuna a los cambios que ocurren en ella.
- Proactividad: los agentes no actúan simplemente en respuesta a su ambiente, ellos pueden exhibir comportamiento dirigido a las metas tomando la iniciativa.

Los agentes se comunican con sus pares intercambiando mensajes en un lenguaje de comunicación expresivo del agente. Aunque los agentes pueden ser tan simples como subprogramas, son típicamente entidades más grandes con cierta clase de control persistente. En Inteligencia Artificial se caracteriza a un agente usando estados mentales tales como conocimiento, creencia, intención, y obligación [Sho93].

Los Sistemas Multi-agente (SMA) son sistemas en los cuales varios agentes interactúan para conseguir algún objetivo o realizar alguna tarea. En estos sistemas el control

es distribuido, los datos están descentralizados y la computación es inherentemente asincrónica. Además, cada agente posee información incompleta y capacidades limitadas. La implementación de SMA requiere de un soporte, i. e., una herramienta o un lenguaje que permita especificar la estructura, el comportamiento y el razonamiento de los agentes. Uno de los lenguajes más populares que se utilizan para programar agentes es Java, dada la cantidad de librerías en existencia. La mayor desventaja que tiene este lenguaje es que fue concebido para la programación orientada a objetos, y falla cuando se trata con la autonomía del agente. Actualmente se han desarrollado una variedad de plataformas de agentes, cuyo objetivo es proveer un nivel mayor de abstracción para que los SMA estén al alcance de un público más amplio, a través de una herramienta útil de creación ágil y metódica. En ejecución, un SMA se puede plantear naturalmente como un ambiente donde los agentes se ejecutan en paralelo, pero este paralelismo debe ser programado explícitamente en la mayoría de los sistemas.

El paralelismo se ha convertido en una de las herramientas computacionales de mayor relevancia, ya que permite dividir el problema en partes, de modo tal que cada parte pueda resolverse en un procesador diferente, simultáneamente con las demás. De igual manera, los SMA proveen un modelo para resolver problemas que no pueden ser resueltos de manera satisfactoria con técnicas clásicas, ya sea por el tamaño o la naturaleza del problema.

A pesar de esta similitud existen grandes diferencias. Los SMA se han concentrado en proveer un modelo abstracto, cuya intención es lograr que el programador se involucre más con el modelado del problema y la lógica del programa que con los detalles de la ejecución paralela en sí misma. Los desarrollos de paralelismo se han orientado más hacia la eficiencia, siendo la mayoría programas de cálculo numérico.

5.2. Programación en paralelo utilizando un modelo de SMA

Como se anticipó en la Sección 1.4, la programación en paralelo ofrece una herramienta computacional imprescindible para aprovechar el uso de múltiples procesadores y en la resolución de problemas que no pueden resolverse mediante técnicas clásicas. En el proceso de diseño de programas paralelos hay que tener en cuenta lo siguiente [HH03]:

1. *Descomposición*: involucra el proceso de dividir el problema y la solución en partes más pequeñas. Es decir, determinar qué parte del software realiza qué tarea.
2. *Comunicación*: se debe determinar cómo se lleva a cabo la comunicación entre los distintos procesos o computadoras, cómo sabe un componente de software cuando otro terminó o falló, cómo se solicita un servicio a otro componente, qué componente debe iniciar la ejecución, etc.
3. *Sincronización*: se debe determinar el orden de ejecución de los componentes, si todos los componentes inician su ejecución simultáneamente, o alguno debe esperar mientras otros trabajan, etc.

A partir de estos tres puntos se pueden identificar los obstáculos que se encuentran comúnmente al programar en paralelo en la Figura 5.1.

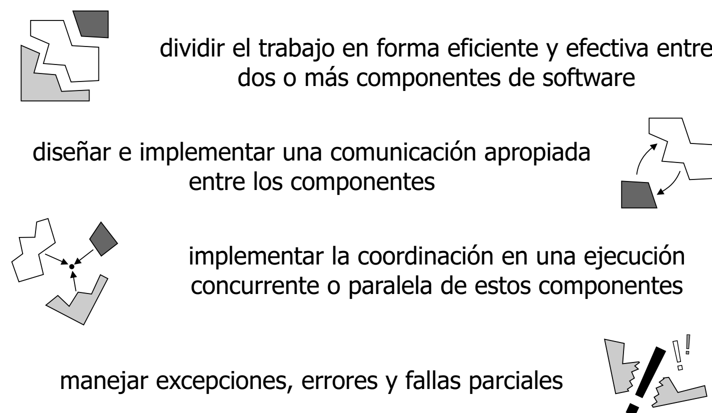

Figura 5.1: Obstáculos de la programación en paralelo

Para enfrentar estos obstáculos podemos utilizar diferentes lenguajes y librerías de programación. El método más popular es el uso de librerías de pasaje de mensajes a bajo nivel (PVM y MPI) o sistemas multithreaded (Java y Pthread), sobre un lenguaje secuencial como C o Fortran. Este modelo es más fácil de aprender para los usuarios por su familiaridad con el lenguaje. Sin embargo, el orden y el número de los mensajes en las comunicaciones produce disparidades potenciales al enviar y recibir, y en consecuencia se vuelve más propenso a errores. Por ello justamente, existen lenguajes de alto nivel que pueden ayudar a aliviar estos problemas.

A continuación mostraremos sintéticamente cómo estas tres soluciones clásicas, en sus distintos niveles de abstracción (ver Figura 5.2), se utilizan para superar los obstáculos

mencionados. Luego, se describirá cómo pueden solucionarse utilizando un SMA de una forma mucho más natural.

5.2.1. Soluciones clásicas

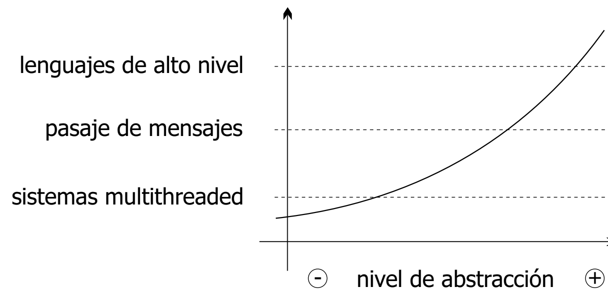

Figura 5.2: Nivel de abstracción en las soluciones clásicas

Sistemas multithreaded

En un lenguaje de programación de bajo nivel multithreaded, como por ejemplo los threads de Java o C+Pthread, los obstáculos anteriores los resuelve el programador intentando asegurar performance y correctitud. Por ejemplo, en la descomposición del trabajo, el programador va a crear nuevos threads cuando sea necesario para asignarles una tarea. En la comunicación entre threads lo más común es utilizar algún área de memoria compartida. Para sincronizar el acceso a los datos compartidos, se implementa la exclusión mutua mediante *semáforos* y sus primitivas para adquirir y liberar el acceso (`lock` y `unlock` respectivamente).

Si en cambio, la comunicación es entre procesos, lo más común es dejar abierto un puerto P determinado y utilizar un thread para sensar los mensajes que arriban. La sincronización de los procesos puede hacerse también mediante el envío y recepción de mensajes. El manejo de excepciones y fallas depende del lenguaje, en Java por ejemplo, se tienen las primitivas de `throw`/`catch`/`try`.

Pasaje de mensajes

Existen otras herramientas de bajo nivel que se basan en un sistema de pasaje de mensajes, como es el caso de la librería de funciones de PVM [GBD+94] o MPI [ERL97] (ver Secciones 3.2.3 y 3.2.4).

La tarea de descomposición se resuelve mediante funciones que crean automáticamente uno o más procesos (`spawn`) o desde la línea de comandos cuando se invoca al programa (según el número de procesadores disponibles). La comunicación se lleva a cabo mediante el intercambio de mensajes con las operaciones de `send` y `receive`. La coordinación se logra sincronizando los procesos cuando llegan a cierto punto en su código, a través de barreras, con mensajes u otras operaciones colectivas. El manejo de excepciones, errores y fallas es responsabilidad del programador, quien debe chequear la información de estado que devuelven las funciones en algún parámetro asociado (generalmente como un número entero).

Lenguajes de alto nivel

También existen lenguajes de más alto nivel, donde el programador participa en algunas etapas del paralelismo, pero se esconden la mayoría de los detalles de la ejecución paralela de bajo nivel. Este es el caso de lenguajes como HPF [Hig97] y OpenMP [Ope02] (ver Secciones 3.2.1 y 3.2.2), entre otros. En esta categoría también entran los lenguajes declarativos vistos en el Capítulo 4, que incluyen paralelismo explícito mediante algún operador o eventos para ejecución concurrente de metas.

En este tipo de lenguajes existen rutinas que paralelizan los bucles, de manera que se crean procesos que serán los encargados de la ejecución paralela de las sentencias dentro del mismo. También se cuenta con la primitiva `fork` para crear procesos. La comunicación se realiza con una abstracción mayor, mediante canales que se definen entre los procesos. Los canales son variables que se comportan como una cola FIFO de valores cierto tipo y también pueden utilizarse en operaciones de sincronización. En lenguajes declarativos como BinProlog, la comunicación y sincronización pueden realizarse mediante un *blackboard* (espacio de tuplas) compartido de mensajes, donde los procesos leen y escriben (modelo Linda [CG89]).

5.2.2. Solución de SMA

En sistemas multi-agente estos obstáculos se resuelven naturalmente, ya que salvo el item que corresponde a la comunicación, los demás son manejados implícitamente por las capacidades de los agentes. Al utilizar agentes, el programador puede concentrarse en el correcto modelado de la tarea que debe realizar el agente, en vez de preocuparse por el control explícito del paralelismo en el programa. La descomposición ocurre simplemente por el hecho de asignar al agente una o más tareas. Debido a que los límites de los agentes son claros y bien definidos, la comunicación entre agentes es más fácil que entre módulos anónimos (procesos o threads). En cuanto a la sincronización, los problemas se ven reducidos debido a que la racionalidad propia del agente le indica cuándo debe y/o puede realizar una acción. En un SMA es importante que los agentes puedan comunicarse y coordinar sus tareas. Para esto, el lenguaje de implementación del sistema debe proveer primitivas de comunicación y coordinación.

Cabe destacar que el empleo de agentes resulta, finalmente, en un programa en paralelo que es más fácil de *mantener y mejorar*. Corresponde a un importante cambio de paradigma, dado que simplifica la tarea del programador a la hora de pensar la solución del problema. Por todos estos motivos, los SMA han sido vistos como una forma eficiente de programar.

5.3. Características deseables de un lenguaje basado en SMA

Un lenguaje de programación en paralelo basado en el modelo de Sistemas Multi-agente debería reunir como mínimo las siguientes características:

- Descomposición: el lenguaje debe proveer la funcionalidad necesaria para permitir la creación de agentes en distintas unidades de ejecución.

- Comunicación: se deben proveer primitivas de comunicación que resuelvan los protocolos de transporte, que permitan optar por operaciones sincrónicas (donde el agente se bloquea esperando por un mensaje) o asincrónicas, primitivas para la identificación de los agentes con independencia de su localización (Sistema de

Manejo de Agentes), y de comunicación entre grupos de agentes (manejar adhesiones a grupos e identificación), entre otras operaciones posibles.

- Sincronización: en problemas de paralelismo usualmente se necesita sincronizar la ejecución de varias tareas en cierto punto de la computación. Para tener un mecanismo similar entre los agentes se podría utilizar un *Agente Barrera*, al que los agentes que necesitan sincronizarse le envían un mensaje para que este agente especial les avise cuándo pueden continuar (una vez que todos estén esperando en el mismo punto). Otro tipo de sincronización puede manejarse con operaciones de comunicación o protocolos de interacción.

- Performance: la sobrecarga que surge a partir de la creación de agentes y su comunicación debe ser la menor posible, a fin de no afectar la ganancia que se obtiene de una ejecución en paralelo.

Si el lenguaje de programación reúne las características antes mencionadas, automáticamente se obtiene la capacidad de representar el paralelismo a alto nivel, facilitando a su vez la tarea del programador mediante un lenguaje con un estilo similar al de la programación secuencial. Hay otras características que podrían incluirse al disponer de un modelo de sistemas multi-agente, tales como:

+ Soportar movilidad de agentes.
+ Implementar algún lenguaje de comunicación entre agentes (ACL) [Lab01].
+ Utilizar tecnología "estandarizada" (e. g., FIPA[1], KQML [FFMM94], entre otros).
+ Otorgar flexibilidad en los lenguajes en que pueden programarse los agentes (multi-lenguaje).
+ Facilitar el desarrollo de distintos tipos de agentes (colaborativos, deliberativos, BDI, de propósito general, etc).
+ Brindar una interface gráfica al usuario.

La movilidad de agentes es importante tanto para la descomposición del problema como para la performance, ya que se podría cambiar el mapeo a los procesadores en tiempo de ejecución. El agente podría recibir una nueva tarea a resolver, para lo cual le

[1]Foundation for Intelligent Physical Agents (FIPA): `http://www.fipa.org`.

conviene migrar de procesador y así disminuir el costo de las comunicaciones o estar más cerca de los recursos que necesita.

Un lenguaje de comunicación estándar permite que los agentes puedan comunicarse conociendo los tipos de mensajes que pueden enviar y recibir. Los protocolos de interacción son una manera de definir normas para la coordinación de actividades entre agentes, e introducen un contexto a los mensajes transmitidos durante una conversación, facilitando así su interpretación.

Si se utiliza tecnología estandarizada y se tiene flexibilidad en cuanto al lenguaje de programación de los agentes, entonces se puede pensar en la interoperabilidad de los agentes desarrollados por distintos grupos de programadores, e incluso, empleando lenguajes de programación diferentes. Además, si el lenguaje cuenta con tipos de agentes predefinidos, estos actuarían como un constructor de alto nivel para el programador (ya podrían tener parte de la lógica implementada). Por último, si se presenta una interface gráfica al usuario el lenguaje se torna más amigable, reduciendo un poco la curva de aprendizaje y poniéndose al alcance de un público más general.

5.4. Caso de estudio

En esta sección se presentará en primer lugar uno de los pocos lenguajes de programación que utilizan el modelo de agentes para programar con paralelismo. Luego, en la Sección 5.4.3 se presentará el ejemplo de RankSort desarrollado con este lenguaje. Para cerrar, en la Sección 5.4.4, se realizará una evaluación de este lenguaje contrastándolo con las características elaboradas en la Sección 5.3.

El paradigma de programación de ORGEL [OYON00] plantea un ambiente de múltiples *agentes* conectados por canales de comunicación abstractos llamados *streams*. Los agentes corren en paralelo, intercambiando mensajes por el stream. En ORGEL la descomposición ocurre al inicio del programa, cuando el agente `main` crea a todos los demás agentes. La comunicación y sincronización se llevan a cabo a través de los streams.

ORGEL fue desarrollado alrededor del año 2000, por investigadores de la Universidad de Tecnología de Toyohashi, Japón. Los autores buscaron un modelo con estilo similar al que exhibe la programación secuencial y con un manejo de alto nivel de las comunicaciones, para que estuviera al alcance de una amplia gama de programadores. En este capítulo se

resaltan estas buenas decisiones de diseño y se plantean algunas limitaciones que exhibe el lenguaje, que en su mayoría se deben a que no se ha continuado con la implementación del mismo.

5.4.1. Resumen de Orgel

La sintaxis de Orgel está basada en C. Se agregan declaraciones de streams, agentes, mensajes y conexiones de red; sentencias para creación, transmisión y dereferencia de mensajes, terminación de agentes y servicios o funciones de agentes. Además, se reemplazan a las variables globales por variables propias de cada agente (lo cual favorece al encapsulamiento).

El comportamiento de streams y agentes está definido por los *tipos* de *streams* y de *agentes*. Cuando un programa en Orgel es ejecutado, se crea y se corre un agente `main`. Si este agente contiene definiciones de variables de tipo agente o stream, sus instancias son automáticamente creadas, los streams se conectan a los agentes y los agentes comienzan su ejecución en paralelo.

Un stream es un canal de mensajes abstracto, que tiene la dirección del flujo de mensajes, y uno o más agentes pueden estar conectados a cada extremo. La declaración es la siguiente:

```
stream StreamType [inherits StreamType1 [, ...]] {
    MessageType[(Type Arg: Mode [, ...])];
    ...
};
```

En la declaración se enumeran los tipos de mensajes que el stream *StreamType* está preparado para manejar. Un tipo de mensaje en Orgel toma la forma de una función: *MessageType* es usado como identificador del mensaje, y una lista de argumentos con tipos y modos de entrada/salida (`in/out`).

Un agente es una unidad de ejecución activa, que envía mensajes a otros agentes mientras realiza su computación. La forma de la declaración de un tipo de agente es similar a la declaración de una función de C; la sintaxis es:

```
agent AgentType([ StreamType StreamName: Mode [, ...]]) {
    member function prototype declarations
```

```
    member variable declarations
    connection declarations
    initial Initializer;
    final Finalizer;
    task TaskHandler;
    dispatch (StreamName) {
        MessageType: MessageHandler;
        ...
    };
};
```

Los argumentos de un tipo *agente* son sus streams de entrada/salida con tipos y modos. Las funciones de agente se declaran de la misma forma que una función en C++, siguiendo la convención *AgentType*::*FunctionName*. El entorno de las variables del agente se extiende dentro de la declaración de agente y de las funciones de servicio. Las declaraciones de conexión especifican cómo conectar a los agentes y streams definidos como variables de agente.

Los últimos cuatro elementos: `initial`, `final`, `task` y `dispatch`, definen manejadores de eventos. Los handlers son código C secuencial, con extensiones para manejar mensajes. Los procedimientos *Initializer* y *Finalizer* son ejecutados en la creación y destrucción del agente, respectivamente. *TaskHandler* define la computación propia del agente; es ejecutado cuando el agente no está procesando mensajes. La declaración `dispatch` especifica el handler para cada tipo de mensaje procedente de un stream de entrada *StreamName*, enumerando los tipos *MessageType* de mensajes aceptables y el código del handler *MessageHandler*. Esta declaración funciona como un framework para el procesamiento asincrónico de mensajes.

5.4.2. Detalles de Orgel

En esta sección incluiremos algunos detalles para completar la descripción de ORGEL, con el objetivo de que este trabajo sea autocontenido. De todas formas, para mayor información puede consultar a los autores en [OYON00]. Para finalizar se incluye un ejemplo que ilustra el uso de las primitivas presentadas.

Declaración de conexión

La conexión entre agentes y streams puede ser especificada de la siguiente forma:

`connect` [*Agent0.S0 Dir0*] *Stream Dir1 Agent1.S1* ;

La declaración toma una variable de tipo stream, *Stream*, y los streams de entrada/salida *S0*, *S1* de las variables de agente *Agent0*, *Agent1*. La palabra reservada `self` puede ser utilizada en lugar de una variable de agente para conectar al agente que contiene la declaración de conexión. *Dir0* y *Dir1* son especificadores de dirección (`==>` o `<==`) que indican el flujo del mensaje.

Si la variable de agente o stream es un arreglo, se necesita un especificador de subíndice de la forma [*SubcriptExpression*]. Si la expresión entre corchetes es una constante, el argumento representa un elemento del arreglo. Si la expresión se omite, el argumento representa todos los elementos del arreglo. La expresión del subíndice puede contener un identificador llamado *pseudo-variable*, que funciona como una variable cuyo entorno es local a la declaración de la conexión, y representa todos los valores enteros, con la restricción de que no se excedan los límites del arreglo.

Si el stream tiene múltiples receptores, en el momento de enviar un mensaje se realiza un *multicast* a todos los receptores. Los mensajes desde múltiples emisores son ordenados en forma secuencial no determinísticamente. Dado que la declaración de conexión se define estáticamente, el compilador puede hacer un análisis preciso para optimizar la planificación y las comunicaciones.

Mensajes

Utilizando los tipos definidos en la declaración de tipos de stream, se pueden definir variables para los mensajes. Una variable de mensaje actúa como una variable lógica en los lenguajes de programación en lógica: puede estar ligada o sin ligar. En su estado inicial se encuentra sin ligadura (unbound), y recibe una ligadura cuando se le asigna un objeto de tipo mensaje u otra variable de mensaje. El estado de ligadura (bound) tiene dos casos: si a la variable se le asigna un objeto mensaje, el estado es *instanciado*; si la variable se asigna a otra variable que está sin instanciar, el estado es *sin instanciar*. En el último caso, una asignación posterior puede cambiar el estado de todas las variables relacionadas a instanciadas.

Crear y enviar mensajes

Para crear un objeto de tipo mensaje, se describen en forma funcional el tipo de mensaje y los argumentos actuales. El tipo de un argumento de mensaje debe ser cualquier tipo de dato de C (excepto el tipo puntero) o cualquier otro tipo de mensaje.

En el primer caso, el argumento es almacenado en el mismo objeto mensaje. Si el argumento es un arreglo, el argumento actual es un puntero indicando la cabeza del arreglo de datos del tamaño declarado.

En el caso de que un argumento sea de tipo mensaje, el modo puede ser `in` o `out`. Si el argumento es de entrada, el parámetro actual puede estar sin instanciar en la creación del mensaje, pero debe ser instanciado por el emisor en su momento. Si el argumento es de salida, el receptor es quien instancia el argumento, y por lo tanto el parámetro actual debe ser *una variable de mensaje sin instanciar*.

La sentencia *send* tiene la forma: *Stream* `<==` *Message*`;`

Recibir mensajes

Un agente conectado al extremo receptor de un stream, recibe mensajes y los procesa de acuerdo a la declaración `dispatch` de su tipo de agente. Cuando un mensaje de algún tipo permitido es recibido, el valor de los argumentos del mensaje se almacena en las variables correspondientes especificadas en el `dispatch`. De manera similar que en el caso de la creación de un mensaje, la variable que corresponde a un argumento de tipo arreglo se ve como un puntero, y se copia el área del tamaño del arreglo.

Si el mensaje tiene argumentos de salida, serán variables sin instanciar. Estas variables se ligan a las correspondientes variables del emisor, y cuando se instancian las variables con objetos de tipo mensaje, los objetos son enviados de vuelta al emisor.

Los mensajes que aparecen como argumentos de otros mensajes pueden obtenerse mediante una operación de *dereferencia*:

Variable `?==` *MessageType*`[(`*Arg1*`[, ...])];`

Si una variable de mensaje *Variable* no está instanciada, el agente que ejecuta esa expresión se suspende, y se restaura cuando otro agente instancia la variable, i. e., es bloqueante.

Ejemplo de aplicación

Para ilustrar el uso de las primitivas y declaraciones de tipos y variables de ORGEL, se programará una solución para el conocido problema de *Productor-Consumidor*. En este caso tenemos un agente productor y `N` agentes consumidores. El agente productor genera `CANT` elementos de tipo entero y los envía dentro de alguno de los tres tipos de mensajes. Dependiendo del tipo de mensaje que contenga al entero, variará el procesamiento que reciba el mismo. Las declaraciones de constantes y tipos de streams con los mensajes que maneja, será:

```
#define N 10
#define CANT 15

//Stream utilizado por el Productor para enviar los elementos generados
stream SProducts {
    //tipos de elementos que maneja el stream
    Elem1(int num: in);
    Elem2(int num: in);
    Elem3(int num: in);
}
```

El agente `main` declara las variables de tipo stream, productor y consumidor, y realiza las conexiones de los streams. Cada stream del productor se conecta al stream de uno de los consumidores. Nótese la ausencia del subíndice del arreglo en la declaración de conexión. Esto vale porque, como se dijo anteriormente, si no se especifica un subíndice, se reemplaza por todos los valores enteros permitidos.

```
agent main(void) {
    AProducer Producer;     //Agente Productor
    AConsumer Consumer[N];  //N Agentes Consumidores
    SProducts Products[N];  //Streams que conectan al Prod. con los Cons.
    connect Producer.prod[] ==> Products[] ==> Consumer[].prod;
}
```

A continuación se transcribe el código del agente productor y del agente consumidor. Los distintos tipos de mensajes se generan de acuerdo a un número aleatorio. El agente

consumidor implementa las funciones de agente `ProcesarElemX(int)` de acuerdo a cada tipo de mensaje.

```
agent AProducer(SProducts prod[N]: out) {
//Genera CANT productos de distintos tipos para cada consumidor
    int i,j,Rnd;
    task {
        for (i=1; i<CANT; i++) {
            for (j=0; j<N; j++) {
                Rnd = random() MOD 3;
                if (Rnd==0) prod[j] <== Elem1(i);
                if (Rnd==1) prod[j] <== Elem2(i);
                if (Rnd==2) prod[j] <== Elem3(i);
            }
        }
        terminate;
    }
}

agent AConsumer(SProducts prod: in) {
//Consume productos de distintos tipos
    ProcesarElem1(int n){...}
    ProcesarElem2(int n){...}
    ProcesarElem3(int n){...}

    dispatch (prod) {
        Elem1(num): {ProcesarElem1(num);}
        Elem2(num): {ProcesarElem2(num);}
        Elem3(num): {ProcesarElem3(num);}
    }
}
```

5.4.3. Ejemplo comparativo: RankSort

Luego de esta introducción al lenguaje Orgel vamos a desarrollar nuestro ejemplo integrador mediante la declaración de agentes. Para esta implementación se plantean

dos tipos de agentes: el *agente principal* (master) y los *agentes esclavos* que serían los encargados de computar el rank de los elementos del arreglo. Si disponemos de un arreglo a ordenar de N elementos tendremos entonces N agentes realizando los cálculos. Cada uno de estos agentes se conecta al agente principal a través de dos streams: uno de ellos es utilizado para recibir el arreglo de elementos y el índice a computar; el otro –en dirección inversa– para enviar los resultados al agente principal.

La alternativa de no enviar el arreglo completo a cada uno de los agentes esclavos implicaría declarar más streams entre cada par de agentes y aún mantendríamos las otras conexiones para recibir el índice que le toca computar y enviar los resultados, por lo tanto es más costosa aunque igualmente válida.

```
1  #define N 10
2
3  //tipos de streams
4  stream SElements {
5      ArrayElement(TElem[] array:in, int index:in);
6  }
7  stream SResults {
8      Rank(int num:in, int index:in);
9  }
10
11 //agente principal
12 agent main() {
13     AWorker worker[N];  //N agentes esclavos
14     SResults result;    //conecta a los esclavos con el agente principal
15     SElements data[];   //conectan al agente principal con los esclavos
16     TElem source[N], target[N];
17     int cant=0;
18
19     connect self ==> data[] ==> worker[].channel;
20     connect worker[].result ==> result ==> self;
21
22     initial {
23         (* ...inicializar el arreglo source aqui... *)
24         for (int i=0; i<N; i++)
25             data[i] <== ArrayElement(source, i);
```

```
26      }
27      final {
28          for (int i=0; i<N; i++)
29              writeln(target[i]);
30      }
31      dispatch (result) {
32          Rank(num_rank, i): {
33              target[num_rank]= source[i];
34              cant++;
35              if (cant==N) then terminate;
36          }
37      }
38  }
39
40  //agentes esclavos
41  agent AWorker(SElements channel:in, SResults result:out)
42  {
43      int RankElement(TElem elem, TElem[] source) {
44          int rank=0;
45          for (int i=0; i<N; i++)
46              if (source[i] < elem) then rank++;
47          return rank;
48      }
49      dispatch (channel) {
50          ArrayElement(array, index): {
51              int rank = RankElement(array[index], array);
52              result <== Rank(rank, index);
53          }
54      }
55  }
```

La declaración de los agentes esclavos y las conexiones de los streams se realiza en el agente *principal*. Asimismo, este agente cuando es creado se encarga de inicializar el arreglo y enviar los datos a los esclavos. Al completar la tarea del envío de mensajes, se queda esperando a recibir los mensajes con el resultado. Una vez que recibió los resultados de todos los agentes termina su ejecución, realizando como último paso la

impresión del arreglo ordenado. Los agentes *esclavos* son creados y a partir de allí esperan recibir mensajes para computar el rank de un elemento. Si se desea se puede agregar una instrucción `terminate` al agente esclavo para que sólo procese un mensaje y luego termine su ejecución.

También se podría haber utilizado sólo un stream con un argumento de salida en el mensaje `ArrayElement`, pero la operación de dereferencia es bloqueante, lo que suspendería la ejecución del agente principal antes de que termine de enviar todos los datos de cómputo a sus esclavos.

La solución presentada es bastante similar en cuanto a su estrategia a las implementaciones con PVM y MPI, sobre todo porque utiliza envío y recepción de mensajes. Sin embargo, la implementación es más sencilla ya que las instrucciones para manejo de streams son de alto nivel, quedando los detalles de su implementación no visibles al programador.

5.4.4. Evaluación

Algunas de las características planteadas en la Sección 5.3 están presentes en Orgel. Por ejemplo, el lenguaje estudiado exhibe una eficiencia y estilo similar al de la programación secuencial. Asimismo, adopta un modelo de programación más efectivo, donde el programador puede contribuir en el desarrollo de una implementación eficiente mediante la especificación abstracta del paralelismo y las comunicaciones. [OYON00]

El tipo de los agentes soportados es amplio y flexible, pero queda como responsabilidad del programador la definición de modelos de agentes colaborativos, deliverativos o BDI, los cuales suelen estar incluidos en *plataformas de agentes* (ver [MG03]).

Una interface gráfica al usuario podría acercar el lenguaje a más programadores. Sobre todo, cuando se considera la declaración de agentes, streams y tipos de mensajes, y la conexión de los streams. El diseño es bastante abierto: al estar programado sobre C, es posible que el programador defina sus propias librerías y que las utilice en sucesivos desarrollos. La declaración de los agentes y su funcionalidad, al estar autocontenida en la misma declaración (como si fuese un objeto), es totalmente reutilizable.

Funcionalidad de streams y mensajes

La abstracción de los canales de comunicación que provee el stream es muy útil desde el punto de vista del programador. Sin embargo, aún para transferir tan sólo un entero, se debe construir un mensaje, declarar un tipo de mensaje y enviar el mensaje vía stream. Esto puede ser visto como un punto en contra, pero se alivia si se tiene en cuenta que el overhead que se presenta en este caso no es tanto mayor cuando se transfieren más datos. Además, se tiene la funcionalidad de manejar los mensajes como si de eventos se tratara, definiendo las funciones handler que se utilizan para el procesamiento de cada tipo de mensaje.

Orgel permite la definición de argumentos de salida en los mensajes. Esto implica que se enviará el mensaje con una variable sin instanciar, y que el receptor del mensaje al instanciar la variable con un objeto mensaje envía el objeto de vuelta al emisor. La semántica no es clara en esta parte, pues quedaría implícito el stream en la otra dirección.

El modelo de comunicación que plantea el lenguaje es estático, quedando definido a través de las declaraciones de `connect`. Es decir, toda comunicación que suceda entre los agentes, al realizarse a través de los canales, debe estar prevista en tiempo de compilación. Esta característica aparta a Orgel del modelo de SMA clásico, donde hay dinamismo en las comunicaciones.

Performance

Con respecto a la performance, en [OYON00] se presenta una evaluación del desempeño de Orgel contrastado con la librería Pthread estándar ante dos ejemplos de prueba. Los resultados muestran que programando en Orgel se pueden obtener buenos speed-ups, aunque dependiendo del ejemplo se pueden llegar a necesitar ciertas optimizaciones estáticas. Existe una sobrecarga del 5-11 % cuando se utiliza Orgel en lugar de C. La sobrecarga de conmutación de threads en Orgel es un 22 % mayor que con C y Pthread.

Como reporta [OYON00], se supone que Orgel debería tener menos sobrecarga que otras alternativas de programación con semánticas de ejecución paralela, pero no hay pruebas al respecto.

Implementación

La implementación de Orgel existente utiliza Pthread y soporta ejecución concurrente en una máquina con un único procesador o ejecución en paralelo en máquinas multiprocesador (en máquinas SPARC con Solaris). El paquete consiste de un compilador Orgel y unas librerías de soporte para el manejo de los agentes y los streams de comunicación.

El compilador cumple principalmente la función de un pre-procesador. Es en realidad un traductor a código C de las sentencias de Orgel, luego compila automáticamente el código generado con el compilador de C, y finalmente es vinculado con las librerías de Orgel para generar el ejecutable.

Para suprimir el número de threads por eficiencia, y para permitir una creación demorada de agentes, cada instancia de agente se representa en un registro de agente. Correspondiendo a la creación lógica de agentes, se crean los registros de agentes con los parámetros adecuados. El ambiente de ejecución de Orgel planifica agentes utilizando estos registros, y crea nuevos threads en caso de que sea necesario. Este modelo es interesante porque posibilita que el planificador realice un balance de carga entre los procesadores.

La implementación actual se basa en un ambiente multithreaded, es por eso que se compara con Pthread. No obstante, cuando se trata de sistemas multi-agente, una ejecución concurrente en un procesador no es del todo real. Si cada agente es una entidad independiente, sería bueno que más de uno de ellos esté activo al mismo tiempo, e incluso que se encuentren dispersos geográficamente.

Los autores de Orgel tenían planeado el desarrollo de una implementación para procesadores con memoria distribuida, pero aparentemente no se ha llevado a cabo. Esto limita mucho las capacidades de experimentación con el lenguaje, pues no siempre se cuenta con una máquina de múltiples procesadores y memoria compartida. Es más fácil la experimentación con un lenguaje como PVM o MPI, porque siempre se dispone de una red de computadoras donde correr los programas.

Para un lenguaje de agentes, sería interesante que el código de los mismos pueda ser distribuido en varias máquinas, y la configuración pueda ser controlada mediante una interface gráfica remota. Un paso mayor sería que la configuración pueda incluso ser

cambiada en tiempo de ejecución, moviendo agentes de una máquina a otra cuando sea necesario, al estilo de Jade (para más información ver [MG03]).

5.5. Resumen

En este capítulo hemos planteado un conjunto de características deseables que deberían estar presentes en un lenguaje de programación en paralelo orientado a agentes. Estas características constituyen la funcionalidad básica que debe proveer el lenguaje de programación para hacer posible la codificación de programas con paralelismo.

El proceso de diseño de programas paralelos incluye tres tareas básicas: descomposición, comunicación y sincronización de las partes. Mediante la utilización de un sistema multi-agente estas complicaciones se resuelven naturalmente, ya que salvo el item que corresponde a la comunicación, los demás son manejados implícitamente por las capacidades de los agentes. De este modo, el programador puede concentrarse en el correcto modelado de la tarea que debe realizar el agente, en vez de preocuparse por el control explícito del paralelismo en el programa.

Se resumieron los distintos lenguajes de programación de alto y bajo nivel que fueron presentados en capítulos anteriores, con un enfoque particular en los problemas de paralelismo, para contrastarlos con la opción de Sistemas Multi-agente. Asimismo, se evaluó uno de los pocos lenguajes de programación que utilizan el modelo de agentes para programar con paralelismo. Orgel, gracias a su estructura de diseño, se adapta a la autonomía e independencia de los agentes facilitando un desarrollo ágil, y el mantenimiento y reuso de los componentes de software. En Orgel la descomposición ocurre al inicio del programa, cuando el agente `main` crea a todos los demás agentes. La comunicación y sincronización se llevan a cabo mediante los streams. La sincronización entre agentes de un sistema multi-agente se considera de más alto nivel, por lo que se asocia con políticas de conversación o protocolos de interacción, que pueden quedar fuera del lenguaje y en un futuro ser incorporados mediante funciones de librería.

El conjunto de propiedades mencionadas como deseables, tanto las básicas como las adicionales, están en concordancia con las que incluye Huget en su *Desiderata* [Hug02]. Además este autor menciona:

- Seguridad

- Arquitectura orientada a eventos

Tanto la seguridad como la movilidad de agentes pueden incluirse en una versión posterior del lenguaje, ya que no son suficientemente generales y los desarrolladores pueden incluirlas por su cuenta si lo creen necesario. El tema de la arquitectura orientada a eventos combina perfectamente con el diseño de ORGEL, esto es, se pueden definir agentes que reaccionen ante estímulos que ocurren en el ambiente implementando los manejadores de eventos apropiados. Claramente, en este acercamiento se deben simular los eventos como mensajes.

Capítulo 6

Tendencias e Integraciones

A lo largo de los capítulos anteriores hemos introducido varios conceptos relacionados con el paralelismo e incluso se han analizado las alternativas que tenemos en software para incorporar el paralelismo en la ejecución de nuestros programas. La Figura 6.1 resume los temas mencionados y, en cierta forma, los ordena en categorías de lo más general a lo más específico como son el hardware y el software.

Como se ilustra en la Figura 6.1, el paralelismo nace a partir del deseo inicial de trabajar con Computación de Alta Performance. Justamente las aplicaciones adecuadas para explotar el paralelismo son aquellas que requieren gran cantidad de cálculos y/o trabajan sobre enormes conjuntos de datos. De ello surgen dos tipos de paralelismo: paralelismo de datos y paralelismo de tareas; cada uno de los cuales se enfoca más sobre un requerimiento o el otro, optimizando algunos factores de acuerdo al caso.

A partir de la investigación en esta rama, se han definido parámetros que permiten medir la eficiencia del paralelismo, como el speedup, la cantidad y utilización de procesadores, entre otros. A su vez, se han desarrollado otros resultados teóricos como la Ley de Amdahl, que determina los límites del paralelismo. Todo ello se apoya, de alguna manera, en la Clasificación de Flynn y tiene aplicación especialmente en lo que se denominan multiprocesadores y multicomputadoras (ambas MIMD).

Distintos mecanismos de comunicación y sincronización son necesarios para coordinar las tareas. En este punto surgen algunas variantes en cuanto a la estrategia a implementar (multiprocesamiento, threads, sincronización en el acceso a variables compartidas, coherencia de memoria en la replicación de datos, localidad). Estos mecanismos se hacen

general	Computación de Alta Performance
tipos	Paralelismo de datos Paralelismo de tareas
teoría	Speedup Ley de Amdahl Taxonomia de Flynn
coordinación	Multiprocesamiento Multithreading Multitarea Coherencia de memoria Sincronización Barreras Computación Distribuida SMA Grid Computing
programación	Modelo de programación Paralelismo implícito Paralelismo explícito
APIs	POSIX Threads OpenMP MPI PVM CUDA
lenguajes	HPF Parlog CIAO Prolog Andorra I Quintus Prolog DeltaProlog Orgel
hardware	SMP NUMA COMA Memoria compartida Memoria distribuida MPP GPU

Figura 6.1: Temas incluidos en esta Tesis

presentes no sólo en los lenguajes de programación que hemos visto, sino también en nuevas tendencias, como son los sistemas multiagentes (SMA), Grid Computing y GPGPU (programación de propósito general en la GPU). En este capítulo presentaremos estas dos últimas tendencias mencionadas para completar algunos de los puntos indicados en la Figura 6.1.

Sobre el centro de la figura, aparecen los dos paradigmas para programación en paralelo: implícito y explícito. Dentro del paralelismo implícito se describieron varios lenguajes, la mayoría de programación en lógica, que permiten la explotación de esta forma de paralelismo. En lo que a paralelismo explícito refiere, se repasaron lenguajes imperativos y algunos declarativos. Se eligieron los lenguajes y librerías (APIs) que iban a permitir contar con una amplia muestra de características para enriquecer la comparación. A lo largo de los capítulos previos, se resaltaron los puntos débiles y fuertes de cada uno.

Hacia el final de este capítulo se hará una integración de los modelos de paralelismo estudiados, para intentar trasladar aspectos interesantes de un paradigma al otro.

6.1. Tendencias actuales

El paralelismo hoy en día se aplica en nuevas oportunidades de desarrollo, como por ejemplo las Grids comunitarias y la computación en el GPU (Graphics Processing Unit). Quizás estas dos tendencias son las que surgen como más prometedoras y las que concentran mayor atención. A continuación se comentará brevemente de qué se trata cada una.

6.1.1. Grid Computing

Según Foster, uno de los pioneros en el campo, un sistema de Grid Computing[1] (GC) debe contar con tres características básicas [Fos02]:

- ✓ coordinar recursos que no están sujetos a un control centralizado;
- ✓ utilizar protocolos e interfaces estándares, abiertos y de propósito general; y,
- ✓ ofrecer calidades de servicio no triviales.

Expresado de otra manera por IBM, GC es un sistema paralelo y distribuido en el cual los recursos están dispersos a lo largo de distintos dominios administrativos y se pueden seleccionar, compartir e integrar basándose en reglas comunes que se acuerdan. Consiste en compartir recursos heterogéneos –basados en distintas plataformas, arquitecturas, sistemas operativos, software y lenguajes de programación–, situados en distintos lugares y pertenecientes a diferentes dominios y organizaciones sobre una red que utiliza estándares abiertos. Dicho brevemente, consiste en virtualizar los recursos informáticos.

GC es un nuevo paradigma de computación distribuida en el cual todos los recursos de un número indeterminado de computadoras son englobados para ser tratados como un único superordenador de manera transparente.

Este paradigma ha sido diseñado para resolver problemas demasiado grandes, mientras se mantiene la flexibilidad de trabajar en multiples problemas más pequeños. Por lo tanto, la computación en grid es naturalmente un entorno multi-usuario; por ello, las técnicas

[1] Utilizaremos el término en inglés por ser más preciso. Algunas traducciones que se encontraron son: "computación distribuida", "computación en grilla" o "informática en rejilla".

de autorización segura son esenciales antes de permitir que los recursos informáticos sean controlados por usuarios remotos.

La realidad es que hoy, GC es un trabajo en progreso, con la tecnología subyacente aún en una fase de prototipo, y que está siendo desarrollada por cientos de investigadores e ingenieros de software de todo el mundo. GC está atrayendo mucha atención porque su futuro, aunque todavía es incierto, es potencialmente revolucionario. Existen muchas Grids en evolución, algunas privadas, otras públicas, algunas dentro de una región o país, algunas con dimensiones realmente globales. Unas tantas dedicadas a problemas científicos particulares, otras de propósito general.

Usualmente, una única computadora, un cluster de computadoras estándar o aún una supercomputadora con propósitos especiales, no es suficiente para los cálculos que los científicos quieren realizar. Por supuesto que las computadoras evolucionan increíblemente rápido. No obstante, no pueden seguir el ritmo a lo que los científicos demandan.

Típicamente, los científicos llegan al límite cuando se encuentran con situaciones donde:

- La cantidad de información que necesitan es muy grande y los datos están almacenados en diferentes instituciones, e. g., las imágenes satelitales de la tierra.
- La cantidad de computaciones similares que el científico debe hacer es enorme, e. g., simular el efecto de moléculas de droga en una proteína para tratar alguna enfermedad.
- Un equipo de científicos con miembros en distintas partes del mundo quiere compartir grandes cantidades de datos y realizar un análisis complejo online de los datos rápidamente y al mismo tiempo, mientras se discuten los resultados en una video conferencia.

Quizás uno de los ejemplos más conocidos de GC es el proyecto SETI@home. Con base en la Universidad de California, Berkeley, SETI@home es una supercomputadora virtual que analiza los datos del radio telescopio Arecibo, en Puerto Rico, buscando signos de inteligencia extraterrestre. Utilizando Internet, SETI junta la capacidad de procesamiento de más de 3 millones de computadoras personales de alrededor del mundo, y ya ha utilizado un equivalente a **600.000 años de cómputo de una computadora personal**.

Otro de la misma serie que se ha vuelto muy popular es Folding@Home, que estudia el plegamiento proteico normal y anormal, la agregación proteica y las enfermedades relacionadas. A marzo del 2008, Folding@home alcanzaba picos de 1502 teraflops sobre más de 270.000 máquinas.

Arquitectura de una Grid

La arquitectura de una Grid está usualmente descrita en términos de capas, donde cada capa tiene una función específica. Las capas más altas son en general centradas en el usuario, mientras que las más bajas están más orientadas al hardware.

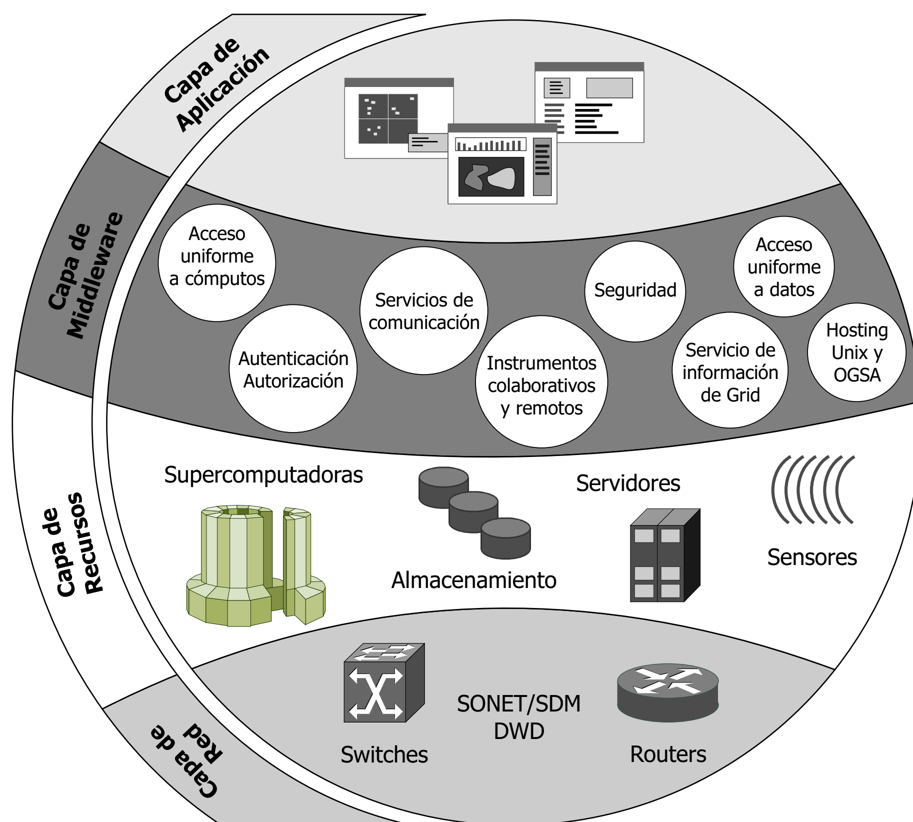

Figura 6.2: Arquitectura de Grid [CER]

Observando la Figura 6.2, en lo más bajo nos encontramos con la capa de red, que conecta a los recursos de la Grid. Sigue la capa de recursos que consta de los recursos reales de la Grid, como las computadoras, los sistemas de almacenamiento, los catálogos electrónicos de datos, sensores y telescopios que están conectados a la red. Luego tenemos

la capa de *middleware*, cuya función es proveer las herramientas que permiten que los distintos elementos (los recursos) participen en una Grid. Esta capa es el cerebro detrás de GC y por lo tanto su componente crucial. Apoyada sobre todas las anteriores tenemos a la capa más alta, que es la capa de aplicación, e incluye a las aplicaciones de ciencia, ingeniería, negocios, finanzas y más, así como también a los portales y toolkits de desarrollo que soportan a las aplicaciones.

El middleware automatiza todas las interacciones 'máquina a máquina' (M2M) y crea una Grid, organizando e integrando los recursos. Un ingrediente clave del middleware es la *metadata*: el dato que describe a los datos. La metadata incluye información sobre cómo y cuándo los datos son recolectados, cómo los datos están formateados y dónde se almacenan.

Algunos programas de middleware actúan como *agentes* y otros como *intermediarios*. Juntos, negocian acuerdos donde los recursos son intercambiados, pasando de un proveedor de recursos a un usuario de recursos en la grid. Los programas agentes presentan metadatos sobre los usuarios, datos y recursos. Los programas intermediarios emprenden las negociaciones requeridas para la autenticación y autorización, y luego cierran los negociados de acceso y facturación por recursos y/o datos específicos.

Cuando el acuerdo está firmado, el intermediario planifica las actividades computacionales y supervisa las transferencias de datos requeridas. Al mismo tiempo, agentes especiales optimizan los ruteos de red y monitorean la calidad de servicio.

Su importancia para el paralelismo

La computación paralela es la disciplina detrás de los sistemas de cluster y grid computing, que permite que las aplicaciones intensivas computacionalmente escalen a los requerimientos del mundo real.

A primera vista, las características de Grid Computing pueden sugerir que este paradigma es sólo adecuado para cálculos con granularidad gruesa (de acuerdo a lo que fue descripto en la Sección 2.5.3). Sin embargo, muchos de los problemas interesantes de la ciencia requieren una combinación de granularidad fina y gruesa, y es aquí donde en particular una Grid puede ser muy poderosa.

Existen dos tipos de problemas en GC relacionados con paralelismo: problemas centrados en la computadora y problemas centrados en los datos. Los problemas centrados

en la computadora son del dominio de HPC (computación de alta performance). El usuario necesita tantos "teraflops"[2] como sea necesario. Muchas aplicaciones centradas en la computadora pueden beneficiarse de GC para combinar grandes recursos computacionales a fin de abordar problemas que no pueden ser resueltos en un sistema único, o al menos para realizarlo mucho más rápidamente.

Los problemas centrados en los datos (también llamados problemas intensivos en los datos, i. e., *data-intensive*) son la fuerza conductora primaria bajo el Grid en el presente y en el futuro. GC será utilizada para recolectar, almacenar y analizar los datos mantenidos en repositorios distribuidos geográficamente, librerías digitales y bases de datos. Algunos ejemplos incluyen: experimentos futuros de física de alta energía, datos fotográficos astronómicos y sistemas modernos de predicción meteorológica.

6.1.2. Computación de propósito general en la GPU

Desde una perspectiva técnica, la GPU (Graphic Processing Unit) es un procesador dedicado exclusivamente al procesamiento de gráficos que se utiliza para aligerar la carga de trabajo del procesador central en aplicaciones 3D interactivas, como por ejemplo los videojuegos. La computación de propósito general en la GPU (GPGPU) es una tendencia reciente a utilizar la GPU para realizar cómputos en vez de la CPU. La adición de escenarios programables y aritmética de alta precisión para los pipelines de rendering permiten que la GPU sea utilizado por los desarrolladores en aplicaciones no relacionadas con lo gráfico. Actualmente podemos encontrar GPGPU en simulaciones físicas, procesamiento de señales, geometría computacional, manejo de bases de datos, biología, finanzas, visión por computadora, entre otras.

Muchas de las arquitecturas para computadoras personales o estaciones de trabajo modernas incluyen un GPU para desvincular la tarea de renderizar objetos gráficos de la CPU. Una GPU posee gran cantidad de unidades funcionales que se pueden dividir en dos: aquellas que procesan **vértices** y aquellas que procesan **píxeles**. Inicialmente se transmite información de vértices a la GPU. El primer paso es aplicar el proceso de *vertex shader*, donde se realizan transformaciones de traslación y rotación en las figuras. Luego, en el *clipping*, se calcula qué parte de estos vértices será visible, y a partir de allí los vértices se transforman en píxeles mediante el proceso de *rasterización*. Estas etapas no

[2]TFLOPS: trillones de operaciones de punto flotante por segundo.

imponen grandes desafíos para la GPU. Donde sí se presenta una carga relevante es en el proceso de *pixel shader*, que constituye el siguiente paso. Aquí es el lugar donde se realizan las transformaciones referentes a los píxeles, tales como la aplicación de texturas. Además, se aplican algunos efectos de *antialising*, *blending* y *fogging* antes de guardar los píxeles en memoria caché.

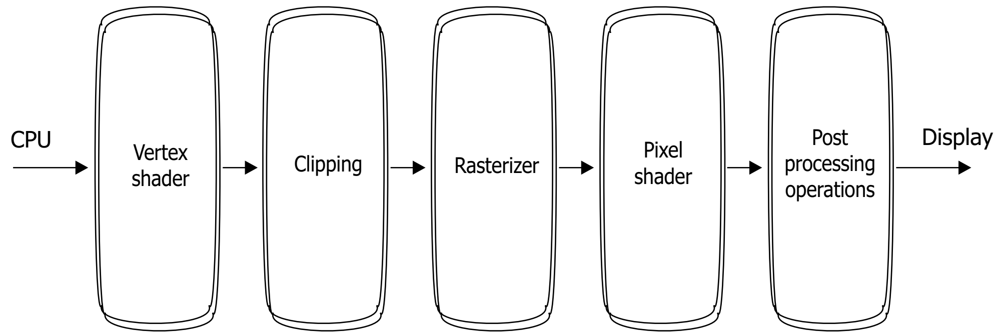

Figura 6.3: Pipeline tradicional de GPU

Finalmente, otras unidades funcionales se encargan de tomar la información de la caché y preparar los píxeles para su visualización. Asimismo pueden ser responsables de aplicar algunos efectos. Toda esta salida se almacena en el frame buffer, de donde se tomará para dibujarla en pantalla. El artículo de Fatahalian y Houston [FH08] explica con más detalle las etapas del pipeline gráfico.

Diferencias con la CPU

En la actualidad las GPU son tan potentes que pueden incluso superar la frecuencia de reloj de una CPU antigua ($>$ 500MHz), aunque obviamente no es posible reemplazar una por la otra. Este incremento en la potencia se debe a que la GPU es un componente altamente especializado, ya que como realiza una sola tarea, se puede optimizar su diseño para que sea más eficiente en los cómputos predominantes como son las operaciones de punto flotante.

Los modelos actuales de GPU en general contienen una docena de unidades para procesamiento de vértices y el doble o triple de procesadores para píxeles, dado que es un cálculo más demandante. Son denominadas ‘manycore’ en contraste con las CPUs ‘multicore’ (dual/quad). Existen varias presentaciones:

- Tarjetas gráficas dedicadas
- Soluciones gráficas integradas
- Híbridos
- Stream processing

Las tarjetas dedicadas se corresponden con la clase más potente de GPU, las cuales usualmente se conectan a un slot de expansión del motherboard, tales como PCIe (PCI express) o AGP (Accelerated Graphics Port). También existen los componentes integrados, que utilizan una porción de la memoria RAM del sistema en vez de una memoria dedicada a los gráficos. Estas soluciones son más económicas de implementar, pero no tienen tanta capacidad de cómputo –sobre todo cuando se prueban con video juegos 3D de alto calibre. Dado que la actividad de la GPU es intensiva en el uso de memoria, las soluciones integradas se ven compitiendo por el acceso a la memoria RAM del sistema. En cuanto al ancho de banda, la memoria del sistema se mueve dentro de los rangos de 2GB/s - 12.8GB/s, en contraste con los 10GB/s - 100GB/s que tienen las GPU dedicadas según el modelo. No obstante, los puntos medios siempre están presentes: existen soluciones híbridas, que comparten la memoria del sistema, pero también tienen una memoria pequeña integrada, que es utilizada en los accesos frecuentes con el objetivo de disminuir la latencia de la RAM.

El último tipo se refiere a stream processing, que aplica una forma limitada de paralelismo. Se presenta en ciertas aplicaciones cuando una serie de operaciones son aplicadas masivamente sobre un vector de datos (stream), que al ejecutarse en una GPU arroja una performance varios órdenes de magnitud mayor. Recientemente, NVidia comenzó a producir tarjetas que soportan una API –extensión de C– llamada CUDA (Compute Unified Device Architecture) que permite que las funciones especificadas en un programa normal de C sean ejecutadas sobre los procesadores paralelos de la GPU. Esto da la posibilidad de que los programas en C sean capaces de tomar ventaja de las habilidades de la GPU para operar sobre grandes matrices en paralelo, y al mismo tiempo puede hacer uso de la CPU cuando sea apropiado. CUDA es la primer API que permite un acceso a la GPU para aplicaciones de propósito general sin las limitaciones que surgen cuando se utiliza una API gráfica como OpenGL o Direct3D.

Arquitectura de una GPU actual

La GPU es una componente altamente segmentada, i. e., posee gran cantidad de unidades funcionales para realizar los cálculos. Pero no todo reside en los procesadores; ocupando un lugar no menos importante, se encuentra la memoria que es utilizada para almacenar los resultados intermedios de las operaciones y las texturas que se utilizan. Es por ello que la velocidad de la memoria puede tener un gran impacto en la performance general del componente.

Introducida en noviembre del 2006, Tesla, la arquitectura unificada para gráficos y cómputos de NVidia, extiende en forma significativa a la GPU al ser masivamente multithreaded [LNOM08]. Escala tanto el número de procesadores como las particiones de memoria, y se expande a un mercado de amplio rango. Tesla se encuentra construida sobre un arreglo escalable de SM (streaming multiprocessors) multithreaded. Cada SM contiene a su vez a 8 SP (streaming processors o procesadores paralelos). La Figura 6.4 muestra una GPU con 14 SMs interconectados con 4 particiones externas de memoria DRAM. Cuando un programa en CUDA en la CPU invoca al kernel, la unidad de distribución de trabajos de cómputo enumera los bloques de la grilla y empieza a distribuirlos en los SM disponibles. Los threads de un bloque se ejecutan concurrentemente en un SM.

Además de los SP, cada SM contiene dos SFU (unidades de funciones especiales), una MT IU (unidad de instrucciones multithreaded) y una memoria compartida en el chip (ver ampliación hacia la derecha de la Figura 6.4). El SM crea, administra y ejecuta hasta 768 threads concurrentes en hardware con una sobrecarga de planificación nula. Implementa una sincronización con barreras de CUDA de forma intrínseca en una única instrucción. Esta rápida sincronización por barreras sumada a una liviana creación de threads y cero sobrecarga en la planificación, soportan un paralelismo de granularidad muy fina, permitiendo crear un thread para el cómputo de cada vértice, píxel y punto de dato.

Esta arquitectura combina un procesamiento SIMD (single instruction, multiple data) de múltiples núcleos o multicore, que resulta en una variante que se conoce como SIMT (single instruction, multiple thread). Un SM ejecuta threads en grupos de 32 threads paralelos, llamados *wraps*. Los threads individuales que componen un wrap comienzan en la misma dirección del programa, pero son libres para saltar de locación y ejecutarse independientemente. No obstante, el wrap ejecuta una instrucción a la vez, por lo que

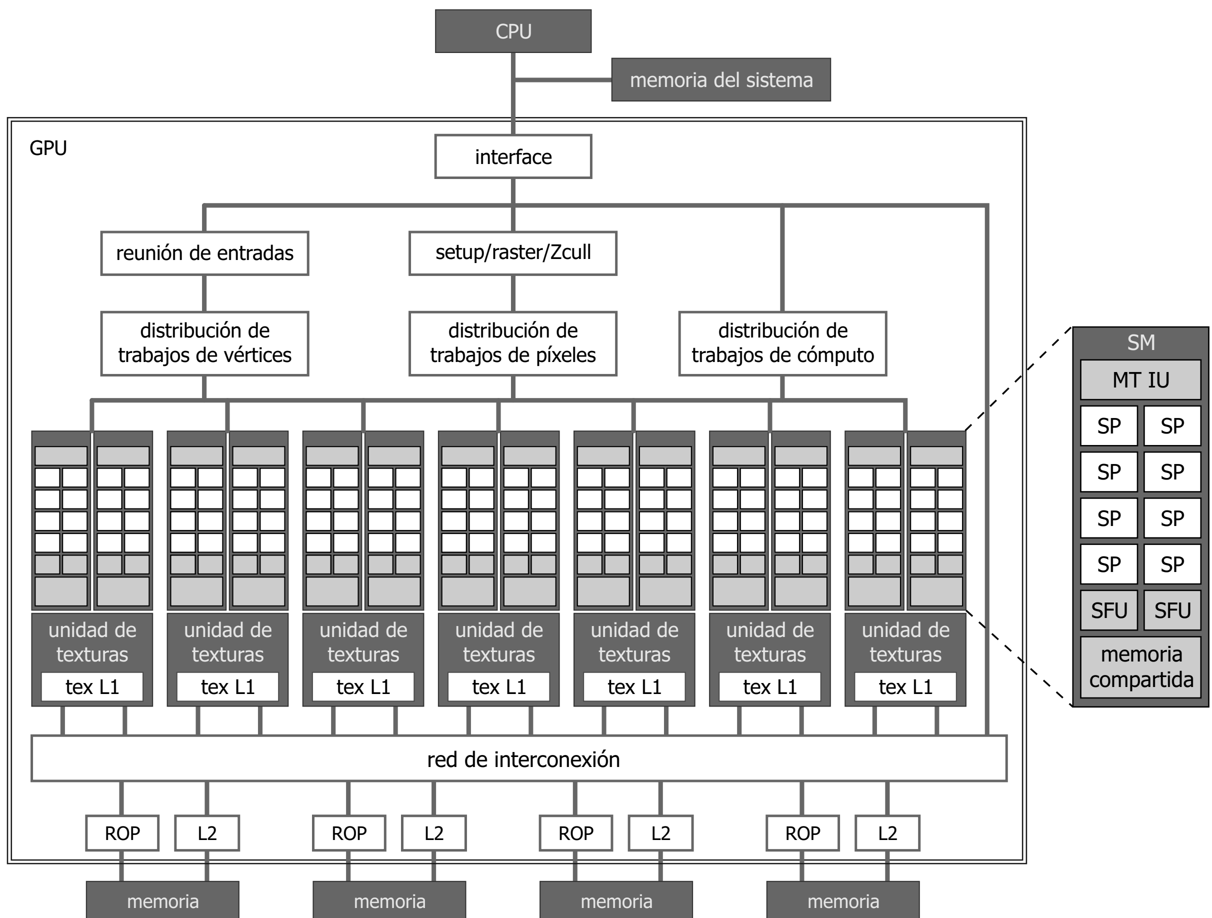

Figura 6.4: GPU NVIDIA Tesla con 112 procesadores paralelos [NBGS08]

es más eficiente cuando todos los threads coinciden en el camino de ejecución. Cada SM administra un pool de 24 wraps de 32 threads por wrap, lo que da un total de 768 threads.

Las aplicaciones en CUDA son performantes en una arquitectura Tesla porque el paralelismo, la sincronización, la memoria compartida y la jerarquía de grupos de threads de CUDA mapean eficientemente a características de la arquitectura de la GPU [NBGS08].

El modelo CUDA para programación en paralelo

Con el advenimiento de CPUs multicore y GPUs manycore[3], en líneas generales, los chips de procesamiento se han convertido en sistemas paralelos. El desafío está en

[3] *Manycore* significa que tiene muchos núcleos, análogamente a *multicore* pero haciendo referencia a una cantidad con un orden de magnitud mayor.

desarrollar software de aplicaciones que escale en su paralelismo de forma transparente, para balancear de esa manera el número creciente de núcleos de procesamiento, similarmente a como las aplicaciones de gráficos 3D escalan su paralelismo a manycore GPUs. Desde que Nvidia lanzó CUDA en el año 2007, rápidamente los desarrolladores han producido programas paralelos escalables para un gran número de aplicaciones. CUDA es un compilador de C y un conjunto de herramientas de desarrollo que permiten a los programadores usar el lenguaje C para codificar algoritmos que se ejecutarán en la GPU.

CUDA dispone de tres abstracciones claves –una jerarquía de grupos de threads, memorias compartidas y sincronización por barreras– que proveen una estructura paralela clara para el código en C convencional de un thread de la jerarquía. Múltiples niveles de threads, memorias y su sincronización proveen paralelismo de datos de granularidad fina y paralelismo de threads, anidados dentro de un paralelismo de datos de granularidad gruesa y paralelismo de tareas. Las abstracciones guían al programador hacia el particionamiento del problema en subproblemas que podrán ser resueltos en paralelo, y luego en piezas más finas que serán resueltas colaborativamente en paralelo. Un programa de CUDA compilado es capaz de ejecutarse en cualquier número de procesadores; sólo el sistema de ejecución necesita conocer el número actual de procesadores físicos.

Este modelo de programación es una extensión mínima de C y C++. El programador escribe una serie de programas paralelos que se denominan *kernels*, y que pueden ser funciones simples o programas enteros. Un kernel se ejecuta en paralelo a lo largo de un conjunto de threads paralelos. El programador organiza estos threads en una jerarquía de grillas de bloques de threads. Un bloque de threads es un conjunto de threads concurrentes que pueden cooperar entre ellos por medio de sincronización de barreras y compartir el acceso a una memoria privada al bloque. Una grilla es un conjunto de bloques que pueden ser ejecutados independientemente.

Cuando se invoca a un kernel, el programador especifica el número de threads por bloque y el número de bloques que forman la grilla. CUDA soporta bloques de threads conteniendo hasta 512 threads. Por conveniencia, los bloques de threads y las grillas pueden tener una, dos o tres dimensiones, accedidas por índices `.x`, `.y` y `.z`.

La ejecución paralela y el manejo de threads es automática. Toda la creación de threads, planificación y terminación es administrada por el sistema subyacente en beneficio del programador. Como si fuera poco, en una GPU Testa todo este manejo de threads se realiza en hardware. Los threads en un bloque se sincronizan mediante barreras; tras

superar la barrera también está garantizado que los threads verán lo que sus pares threads han escrito en memoria antes de alcanzar la barrera. Dado que los threads comparten la memoria local y utilizan el mecanismo de barreras, estos residirán en el mismo procesador físico o multiprocesador. El número de bloques puede, sin embargo, exceder el número de procesadores.

Los distintos bloques no tienen medios directos de comunicación, aunque pueden coordinar sus actividades utilizando operaciones atómicas en la memoria global visible a todos los threads. Los bloques de threads se planifican en cualquier orden a lo largo de un número arbitrario de núcleos de procesamiento. Esto ayuda a evitar la posibilidad de un deadlock.

Una aplicación puede ejecutar múltiples grillas tanto dependiente como independientemente. Las grillas independientes se pueden ejecutar concurrentemente si se tienen los recursos de hardware suficientes. Las grillas dependientes se ejecutan secuencialmente, con una barrera inter-kernel implícita entre ellas, garantizando que los bloques de la primer grilla se completen antes de que cualquier bloque de la segunda grilla sea iniciado.

Como se puede apreciar en la Figura 6.5, cada thread tiene su propia memoria privada; cada bloque tiene una memoria compartida visible a todos los threads del grupo; y todos los threads tienen acceso a la misma memoria global. Los programas pueden declarar variables en la memoria compartida y en la memoria global mediante los calificadores `_shared_` y `_device_` respectivamente. En la arquitectura Tesla estos espacios de memoria corresponden a memorias físicamente separadas: la memoria de bloques es una RAM on-chip de baja latencia; la memoria global reside en la rápida DRAM de la placa gráfica. El programa maneja el espacio de memoria global por medio de las primitivas `cudaMalloc()` y `cudaFree()`.

Para soportar una arquitectura de sistemas heterogénea donde se combina una CPU y una GPU, cada uno con su propio sistema de memoria, un programa en CUDA debe copiar los datos y los resultados entre la memoria del host y la del dispositivo. La sobrecarga de la interacción CPU-GPU y de las transferencias de datos es minimizada utilizando motores de transferencia de bloques de DMA y rápidas interconexiones. Por supuesto, los problemas suficientemente grandes que necesitan un plus de performance de GPU, amortizan la sobrecarga mejor que los problemas pequeños.

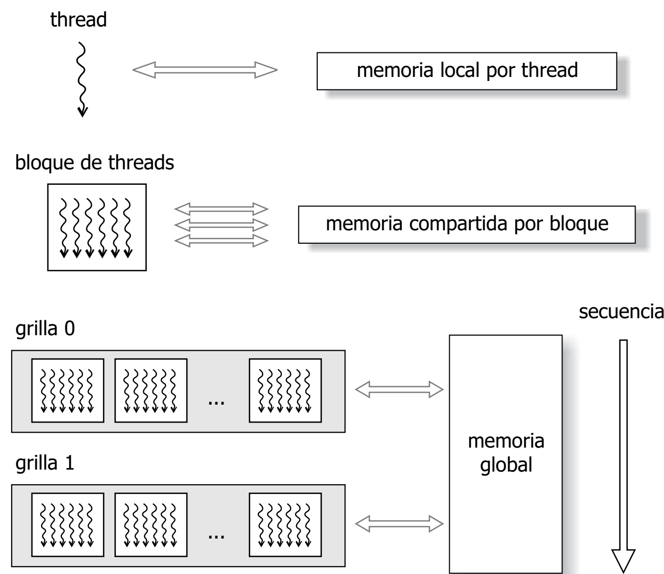

Figura 6.5: Niveles de granularidad paralela y repartición de memoria [NBGS08]

Una de las limitaciones de CUDA es que requiere una GPU capacitada para su ejecución, disponibles únicamente a partir de Nvidia (GeForce 8 series y superiores, Quadro and Tesla).

AMD también tiene su propio modelo de programación en paralelo, se llama Brook+ y es una implementación de Brook en la capa de abstracción de cómputo de la GPU de AMD con algunas mejoras (procesamiento de enteros y números de doble precisión, junto con operaciones de scatter) [Adv07]. Brook es una extensión al lenguaje C para stream programming originalmente desarrollada en la Universidad de Stanford.

6.2. Integraciones

Uno de los objetivos de esta Tesis es investigar la posibilidad de obtener un modelo de programación que permita reunir las ventajas de los dos tipos de paralelismo mencionados (explícito e implícito). Algunos de los objetivos parciales planteados al comienzo del estudio fueron:

- Analizar distintos lenguajes declarativos con paralelismo implícito, a fin de identificar qué aspectos podrían ser trasladados a lenguajes procedurales.

- Considerar distintos paradigmas procedurales con paralelismo explícito, a fin de estudiar cuáles son las primitivas de programación en paralelo más comunes, y cuáles se pueden combinar con las herramientas de lenguajes declarativos.
- Definir un modelo que combine los elementos identificados en los lenguajes de programación declarativa con paralelismo implícito junto con las primitivas de lenguajes procedurales con paralelismo explícito.

Luego de recorrer un variado espectro de lenguajes de programación con paralelismo en los capítulos anteriores, intentaremos responder en esta sección tres preguntas centrales:

i. *¿Qué primitivas deberíamos incluir si quisiéramos definir nuestro propio lenguaje para programar con paralelismo explícito?*
ii. *¿Qué aspectos del paralelismo implícito pueden trasladarse a lenguajes imperativos?*
iii. *¿Qué aspectos del paralelismo explícito pueden trasladarse a lenguajes declarativos?*

Para abordar los puntos ii. y iii., se considerarán los elementos estudiados en los capítulos anteriores para evaluar cómo los distintos enfoques se pueden trasladar a otro tipo de lenguajes. Esto es, trabajando con paralelismo implícito se verá como las herramientas de lenguajes declarativos pueden adaptarse a lenguajes imperativos. Análogamente, al tratar con paralelismo explícito se evaluarán qué elementos de los lenguajes imperativos pueden ser ajustados para el caso de lenguajes declarativos.

Lo que se intenta es aplicar las propiedades o características interesantes de los lenguajes o librerías que son más fuertes en cada tipo de paralelismo a los lenguajes que se encuentran en el extremo opuesto de la clasificación.

6.2.1. Primitivas básicas para paralelismo explícito

A continuación se intentarán resumir, a modo de referencia, las primitivas que deberían estar disponibles en un lenguaje para programar con paralelismo explícito. Muchas de ellas son comunes a varios lenguajes. No obstante, hay otras más específicas (y no tan fundamentales), que colaboran con la claridad y la facilidad de programación. Para esta tarea nos enfocamos en las funciones principales que necesitamos dentro de cada grupo:

Creación de procesos

En cuanto a creación de procesos necesitaremos funciones para crear y terminar procesos, para identificar a cada uno de los procesos del sistema y para poder esperar la finalización de un proceso determinado.

FORALL	para creación masiva de procesos que ejecutan tareas similares sobre un gran conjunto de datos.
FORK/SPAWN	para la creación individual de procesos hijos.
JOIN	para esperar por la terminación de un proceso hijo particular.
IDs	identificadores para distinguir entre los distintos procesos.

Comunicación

Para implementar la comunicación entre procesos necesitamos un mecanismo para enviar y recibir mensajes, en modo bloqueante y no bloqueante, a un proceso o a un conjunto de ellos.

SEND	enviar un mensaje.
RECV	recibir un mensaje.
MCAST	multicast (a un grupo de procesos).
BCAST	broadcast (a todos los procesos).

Sincronización

Con respecto a la sincronización de los accesos a regiones críticas, necesitamos formas de adquirir un permiso para acceder a la región de código, y luego otra para liberarlo cuando hayamos terminado de realizar la operación deseada. También es muy útil el concepto de barreras para sincronización masiva (grupos de procesos).

LOCK	adquiere el acceso a la sección crítica.
UNLOCK	libera el acceso a la sección crítica.
BARRIER	una barrera para sincronizar un grupo de procesos en un punto determinado de la ejecución.

Computación global

Para realizar cálculos globales es útil contar con una primitiva de reducción de datos. Si bien se puede realizar algo similar mediante el pasaje de mensajes a un proceso raíz o coordinador, la opción de tener esta primitiva implementada resulta más eficiente (por ejemplo para análisis de resultados).

REDUCE	aplica un operador asociativo a un conjunto de datos.

Transferencia de datos

Las primitivas que se estudiaron de transferencia de datos, si bien son útiles, no se consideran básicas (puede obtenerse el mismo resultado mediante alguna secuencia de las demás operaciones). Podrían incluirse sólo si justifican una mejor performance al estar optimizadas.

6.2.2. Ideas para la integración de modelos con paralelismo implícito

Generalmente, las tareas que se relacionan con la incorporación implícita de paralelismo a un programa las realiza el compilador mismo o un preprocesador. Este compilador/preprocesador debe ser capaz de detectar el paralelismo posible en el programa, y codificarlo de cierta manera para que pueda ser aprovechado en ejecución. Las tareas involucran: detección de dependencias de datos, privatización de arreglos, reconocimiento de idioma y análisis simbólico, entre otras.

En lo que sigue se estudiará cómo los distintos tipos de paralelismo (de unificación, or y and) pueden trasladarse a lenguajes imperativos.

Asignaciones

Se puede plantear una transformación del paralelismo de unificación en lenguajes lógicos a una forma de paralelismo implícito en lenguajes procedurales. Este tipo de paralelismo se daría en las asignaciones de estructuras complejas, tanto en sentencias

simples como al corresponder los parámetros de procedimientos o funciones. En el ejemplo, trabajando con `MiTipo` se tiene la posibilidad de asignar el arreglo y las matrices al mismo tiempo:

```
TYPE
  MiTipo = RECORD
    arreglo1, arreglo2: ARRAY [1..r] OF REAL;
    matriz: ARRAY [1..n, 1..m] OF INTEGER;
    cadena: ARRAY [1..t] OF CHAR;
    valor: BOOLEAN;
    END;
```

Este comportamiento es probablemente el que siguen los lenguajes que soportan asignación de arreglos en una sola sentencia, sólo que ahora la serie de asignaciones se realiza en paralelo.

Este tipo de paralelismo se aplicaría más a arquitecturas con memoria compartida (cuidando que no haya contención) o a aplicaciones con memoria distribuida donde los distintos datos residan en distintos procesadores (para que haya una porción local en cada uno de ellos).

Una situación semejante puede darse en las expresiones de condicionales y comparaciones. Por ejemplo:

```
VAR A, B, C: MiTipo;
    X, Y: ARRAY [1..t] OF CHAR;
....
IF ((A<>B) AND (X<Y)) OR
   ((A<>C) AND (X>Y) AND (X>A.cadena))
    THEN ...
    ELSE ...
```

Este tipo de paralelismo es similar al paralelismo and/or analizado para el caso de programación en lógica.

Análisis de Dependencias

Para establecer las porciones de código que pueden ser paralelizadas, es necesario un análisis de dependencias de datos. Se puede dividir el programa en bloques y realizar un grafo de dependencias. A partir de los bloques independientes se pueden crear distintos threads.

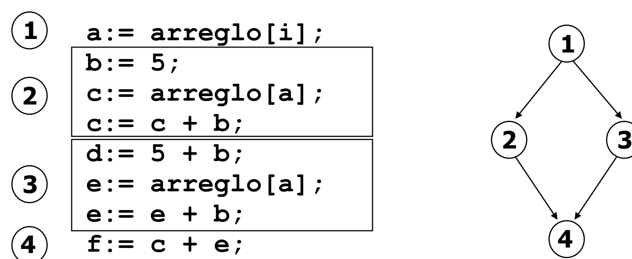

Figura 6.6: Un grafo de dependencias

Para determinar los bloques de código se deben identificar los distintos tipos de dependencias (RAW, WAR, WAW). Por ejemplo, en la Figura 6.6, los bloques de código 2 y 3 necesitan el valor de `a` como índice del arreglo. Esta es una dependencia RAW (read after write) con el bloque 1 que es donde se escribe la variable.

Los bloques se podrían terminar de delimitar mediante un análisis de costo de computación, para no ocasionar overheads innecesarios en ejecución (teniendo en cuenta, por ejemplo, el tiempo de creación de threads).

Si tenemos condicionales entre las sentencias, se podría aplicar algún tipo de predicción, o se podrían evaluar los bloques del 'then' y del 'else' en paralelo, realizando modificaciones provisionales hasta que se decida el valor de la condición.

También se puede plantear un *look ahead* de instrucciones que busque una sentencia costosa en términos de tiempo de ejecución y que sea independiente de las sentencias que la preceden, e inicie su ejecución de antemano. Cuando se llega a ese punto ya hay gran parte del trabajo hecho.

Esta forma tiene mucha similitud con lo que se presentó en cuanto a paralelismo AND dependiente, por las variables que pueden estar compartidas y por la precedencia entre la ejecución de las distintas sentencias.

Bucles y Array Privatization

Es una técnica para aumentar la performance de los programas paralelos generados automáticamente. Es el pie para aplicar paralelismo con FORALL.

Si la asignación de variables sobrescribe el mismo arreglo de elementos en diferentes iteraciones de un bucle L, la asignación es dependiente de sí misma en el bucle L. Una asignación es privatizable en el bucle L si y sólo si el valor generado por ella no es utilizado a lo largo de las iteraciones de L (es loop-independent); luego, cada iteración podría trabajar con una copia del arreglo (un arreglo privado).

Si desplegamos las iteraciones de este bucle en sentencias consecutivas, tendríamos un caso similar al de análisis de dependencias donde identificaríamos varios bloques que son independientes, y por lo tanto podrían ejecutarse en paralelo.

6.2.3. Ideas para la integración de modelos con paralelismo explícito

El paralelismo explícito en lenguajes lógicos se puede obtener de distintas maneras:

- definir operadores para especificar los tipos de paralelismo vistos anteriormente, como un & para el paralelismo AND y un || para el paralelismo OR (ahorra chequeos de dependencias).
- agregar predicados que emulen las directivas/funciones de los lenguajes procedurales: forall, fork, send/recv, lock, ...

El problema es que éstos métodos no garantizan una mejor performance que en los modelos con paralelismo implícito. Una tercera opción sería agregar nuevos predicados para que el programador brinde más especificaciones acerca de la forma de su programa, e. g., al indicar qué variables dependen de otras (símil anotaciones).

En algunos Prologs (DeltaProlog [CMCP92]) se utilizan eventos para incluir paralelismo. Estos eventos permiten la interacción entre procesos (sincronización por rendez-vous).

Básicamente, se definen dos operadores nuevos:

X?E → un proceso se queda esperando por un evento.
Y!E → indica la ocurrencia del evento, unifica con un X?E ↔ X unifica con Y.

Paralelismo condicional

El mismo efecto que se lograba en OpenMPcon ‘parallel if’ se puede hacer en Prolog con guardas. Para ello se introduce un operador especial “&:”:

```
a(X):- <guarda>&: b(X, Y) & c(Y) & d(X).
```

Si el predicado de guarda tiene éxito, entonces las submetas del cuerpo de la regla se ejecutan en paralelo (con paralelismo and). Caso contrario, la ejecución de las submetas se realiza como en la versión secuencial (se trata como una conjunción ordinaria). Equivale a:

```
a(X):- <guarda>, b(X,Y) & c(Y) & d(X).
a(X):- NOT <guarda>, b(X,Y), c(Y), d(X).
```

Comunicación

Si tenemos memoria compartida podemos utilizar un blackboard (espacio de tuplas) compartido de mensajes, donde los procesos escriben y leen. El modelo Linda [CG89] puede utilizarse para la comunicación de procesos en programas lógicos y también para sincronización.

Las escrituras en el *blackboard* no son bloqueantes; tienen el efecto de agregar la tupla al blackboard. Las lecturas quitan una tupla existente que unifique con los argumentos presentados; si no existe ninguna, el proceso se bloquea. Se puede hacer una lectura condicional.

Existen compiladores de Prolog que implementan este modelo. BinProlog [Bin] es una implementación de Prolog que permite la ejecución en paralelo de tareas sobre una red [BT93]. Para la comunicación y sincronización de los procesos define un área de memoria compartida (blackboard) donde se puede acceder en forma remota utilizando las operaciones del modelo Linda. Jinni [Tar98] es un compilador de Prolog basado en Java, liviano y multithreaded, pensado para ser usado como una herramienta de scripting flexible para unir componentes que procesan conocimiento y objetos Java en aplicaciones cliente/servidor sobre una red, así como también sobre applets en la Web. De Java extrae las facilidades para programar aplicaciones portables, y de Prolog utiliza el razonamiento basado en reglas, por lo que Jinni se vuelve ideal para construir Agentes Inteligentes en

Sistemas Multi-agente [Tar05]. Los blackboards de Jinni interoperan con la última versión de BinProlog y soportan transacciones sincronizadas entre usuarios.

En otro caso, la comunicación puede implementarse al estilo de PVM o MPI, con predicados para send/recv.

6.3. Resumen

En este capítulo hemos presentado dos tendencias muy acreditadas hoy en día que no fueron abarcadas en capítulos anteriores. Grid Computing y GPGPU pueden ser justamente algunas de las líneas de investigación por las que se profundice en el estudio del paralelismo a futuro. Grid Computing apunta en sí misma a compartir la potencialidad que existe en distintas ubicaciones geográficas, controladas por distintos dominios administrativos. En cambio, el objetivo de GPGPU es aprovechar la potencia de cómputo creciente de las placas de video, uno de los componentes más sofisticados y con más crecimiento en los últimos años.

Habiendo descrito un importante fragmento de lo que es el paralelismo en la ciencia, se realizó una integración de los paradigmas. Inicialmente se plantearon las características que necesariamente creemos que deben estar presentes en un lenguaje con paralelismo explícito, dividiendo su clasificación según los cinco aspectos que se fijaron con anterioridad: creación de procesos, comunicación, sincronización, computación global y transferencia de datos. Estas primitivas ponen a disposición del programador las herramientas básicas para la tarea de codificación, y propician a un código claro de escribir y entender.

Por último, se presentaron ideas para trasladar características interesantes de los lenguajes con paralelismo implícito a lenguajes imperativos. Ello motivó estudiar cómo los tipos de paralelismo de unificación, OR y AND -incluso dependiente- podían ser implementados en las estructuras comunes de los lenguajes procedurales, como asignaciones, condicionales y bucles. También se realizó la tarea complementaria de evaluar las propiedades de lenguajes con paralelismo explícito que podían ser migradas a lenguajes declarativos. Las más interesantes son el manejo de eventos, las guardas en los predicados y los mecanismos de comunicación que pueden soportarse, como ser un espacio de tuplas compartido.

Capítulo 7

Conclusiones

El paralelismo se ha convertido en una de las herramientas computacionales de mayor relevancia en una gran parte de los ámbitos científicos, donde deben resolverse problemas computacionales muy complejos o con altos requerimientos en tiempo y recursos. Para su aplicación no sólo es necesario contar con hardware que permita realizar cómputo paralelo, sino también con lenguajes de programación para el desarrollo de aplicaciones con paralelismo.

Como se ha mostrado a lo largo de esta Tesis, afortunadamente existe un conjunto bastante poblado de lenguajes de programación que permiten la implementación de sistemas con paralelismo. No obstante, la elección del lenguaje apropiado no es una tarea trivial. Es por este motivo que en esta Tesis se presentó un estudio y análisis detallado de un amplio abanico de lenguajes y librerías a fin de mostrar las distintas herramientas que se encuentran a disposición hoy en día para programar en paralelo.

Persiguiendo un interés por cubrir el espectro más amplio posible, se han estudiado lenguajes imperativos, lenguajes declarativos, lenguajes orientados al desarrollo de sistemas multi-agentes, y nuevas tendencias como la programación en Grid y los lenguajes de propósito general para la GPU (Graphic Processing Unit).

A continuación se lista un conjunto de puntos destacados que han surgido de la investigación realizada en esta Tesis:

- La correctitud y performance son puntos centrales en paralelismo.

- En los tiempos actuales, las computadoras multiprocesador se están volviendo un estándar, mientras que los aumentos de velocidad en procesadores individuales han disminuido. Por lo tanto la clave para obtener aumentos en performance es justamente aprovechar –mediante paralelismo– los múltiples procesadores al ejecutar un programa.

- Una estructura de programa simple favorece a la especificación del problema, y por lo tanto, es más probable que un compilador realice eficientemente una división del problema en partes.

- En los lenguajes procedurales –como los que se vieron en el Capítulo 3– la explotación implícita del paralelismo impone una serie de chequeos de dependencias y, en general, la granularidad del paralelismo explotado será pequeña.

- En lenguajes lógicos –como los que se expusieron en el Capítulo 4– el paralelismo surge naturalmente de manera implícita porque no hay restricción de orden en la ejecución de las sentencias. Sin embargo, también se requieren ciertos procedimientos de análisis de dependencias de datos.

- Se han desarrollado muchos sistemas de programación, pero en la práctica, pocos de ellos son utilizados por programadores de aplicación. Wilson *et al.* [WIS95] remarcaron que los programadores se atan a sistemas de bajo nivel como extensiones de paralelismo o librerías de pasaje de mensajes. De lo estudiado en las Secciones 3.2.3 y 3.2.4, esto tiene sentido si se observa lo ampliamente utilizadas que son algunas librerías de paralelismo como son MPI o PVM, que presentan extensiones para el lenguaje C.

- Como fue presentado en el Capítulo 2 y en parte del Capítulo 6, la diversidad del hardware disponible afecta seriamente las posibilidades de portabilidad para el software paralelo. Las capacidades de reuso de código y de operaciones de entrada/salida son pobres en los lenguajes paralelos, y los prototipos son usualmente abandonados por nuevos desarrollos, en vez de recibir soporte adecuado.

- Una buena opción es plantear un paralelismo de programas lógicos que sea un híbrido entre una implementación implícita y una explícita: implícito con anotaciones, según lo visto en las Secciones 4.5 y 6.2.3. El programador, al proveer información extra

sobre el comportamiento del problema, ahorra chequeos en tiempo de ejecución que afectan la performance.

- Se destaca la propuesta de extender un lenguaje de programación con una librería, adaptando las ideas existentes presentes en PVM o MPI (ambos portables). De hecho, existen trabajos relacionados con PVM-Prolog [CM96] como se detalló en la Sección 4.5.5, pero no se ha encontrado nada respecto a un posible MPI-Prolog. La idea es brindar al programador de Prolog acceso completo al entorno de PVM/MPI.

- El uso de eventos en programas lógicos puede brindar resultados interesantes si está bien implementado. La idea es que los procesos que se encuentran resolviendo tareas en paralelo interactúen mediante la emisión y recepción de eventos definidos por el programador al momento de codificar la solución. Así, los eventos son despachados en forma remota y los procesos se sincronizan y colaboran para el avance de la computación. Este mecanismo se pudo observar en los lenguajes de las Secciones 4.5.6 y 5.4.

Paralelismo implícito

Podemos afirmar que uno de los mayores problemas a resolver cuando se intenta paralelizar código secuencial es el análisis de dependencias para identificar el paralelismo potencial. Otro problema relevante es la evaluación de los costos de comunicación para decidir cómo distribuir las iteraciones de un bucle. Es por ello que, encontrar el conjunto más chico de relaciones de dependencia que preserve la correctitud y la terminación del programa es el objetivo de construcción más importante del compilador o intérprete.

Muchos investigadores han analizado el campo para encontrar soluciones prácticas para el desarrollo de compiladores que optimicen código (e. g., ver [AS92, GdlB94, PP96, KP04, Wol95]). El paralelismo completamente automático se torna muy difícil por las siguientes razones:

- el análisis de dependencias es duro para código que utiliza punteros, recursión y direccionamiento indirecto;
- muchos bucles pueden tener un número indefinido de iteraciones;

- el acceso a recursos globales es difícil de coordinar en términos de manejo de memoria, E/S y variables compartidas.

Generalmente es difícil predecir la performance de un código paralelizado de manera automática. Por ello, el poco éxito logrado por la paralelización automática (e. g., ver [CLS96, HSYHSODS00, KP93]) llevó al desarrollo de diferentes acercamientos en los cuales el programador comparte la responsabilidad de descubrir el paralelismo y la disposición de los datos.

Paralelismo explícito

En estas propuestas, como HPF y OpenMP (ver Capítulo 3), se les solicita a los programadores que provean anotaciones para ayudar a que el compilador cree una implementación eficiente. La mayor crítica a este tipo de paralelismo es que el programa paralelo óptimo para resolver un problema dado puede ser significativamente distinto a su versión secuencial. En este caso, los compiladores y las anotaciones del programador fallarían.

Los lenguajes explícitos de alto nivel tienen propiedades interesantes en cuanto a la programabilidad: proveen abstracción, son expresivos, facilitan el desarrollo de software correcto por construcción, son más fáciles de programar que el pasaje de mensajes y son más eficientes que los compiladores que paralelizan código. La portabilidad de código es provista teóricamente, mientras que la portabilidad de la performance es completamente ignorada.

Los sistemas de programación en paralelo de bajo nivel, como PVM y MPI (ver Capítulo 3), aún siendo más difíciles de usar, exhiben características interesantes en cuanto a efectividad: proveen una performance aceptable, justificando el paralelismo y preservan la complejidad del software. De hecho, los sistemas de pasaje de mensajes son más eficientes en arquitecturas de memoria distribuida y los sistemas multithreaded en arquitecturas de memoria compartida.

Nuevas tendencias

Fuera de la línea clásica de programación, aparecen lenguajes nuevos que surgen gracias a las tendencias de investigación actuales o a avances en el hardware con nuevas

características que se desean aprovechar. Los otros dos lenguajes que se estudiaron son Orgel (modelo de Sistemas Multi-agente en el Capítulo 5) y CUDA (programación en la GPU en la Sección 6.1.2). Asimismo, en la Sección 6.1.1 se describió el modelo de Grid Computing, con características que lo pueden clasificar como propicio exclusivamente para cálculos de granularidad gruesa. Sin embargo, una buena porción de problemas científicos interesantes requieren una combinación de granularidad fina y gruesa, y es aquí donde en particular una Grid puede ser muy poderosa.

Un modelo de Sistemas Multi-agente tiene muchos puntos a favor al programar con paralelismo. Para empezar, la idea de agentes en sí y la descomposición que conlleva promueve que naturalmente pensemos nuestro problema en paralelo, sin atarnos a nuestra costumbre de programar serialmente. Luego, el nivel de abstracción es muy bueno debido a que la mayoría de los obstáculos de la programación paralela se resuelven implícitamente gracias a las capacidades de los agentes. Además, los programas resueltos en base a este modelo son más fáciles de modificar y mantener, aunque a veces se pueda argumentar que su performance no es comparable con las que obtenemos mediante las aproximaciones tradicionales de otros lenguajes.

Para completar el análisis de las nuevas tendencias, se puede manifestar que a través de CUDA o Brook es posible acceder a una capacidad de cómputo en alza, como es la de la GPU. Aquí se pueden soportar programas paralelos de granularidad muy fina, ya que los modelos sobre los que se sostienen plantean sobrecargas muy bajas en la creación de procesos y costos de planificación prácticamente nulos.

Finalmente, como un argumento más a favor de la programación en paralelo y expresando un apoyo al crecimiento sobre esta línea, se encuentra que Microsoft recientemente ha lanzado sus Parallel Extensions al .NET Framework [Mic08]. Ellas permiten expresar el paralelismo en tres formas:

1. Paralelismo de datos declarativo, a través de un lenguaje de consultas integrado que utiliza paralelismo en su ejecución (Parallel LINQ o PLINQ [DE07]).
2. Paralelismo de datos imperativo, a través de mecanismos para expresar operaciones orientadas a los datos (como por ejemplo un FORALL).
3. Paralelismo de tareas imperativo, utilizando expresiones y sentencias para crear tareas livianas y planificarlas de acuerdo a los recursos disponibles.

Como trabajo futuro sería interesante evaluar estas tres extensiones presentes hasta el momento para contrastarlas con los lenguajes preexistentes. La ventaja que inmediatamente se resalta es que las pruebas se pueden ejecutar fácilmente dentro de un entorno de desarrollo muy conocido en la actualidad, y que puede correr en nuestra propia computadora DualCore o QuadCore.

Índice de lenguajes y librerías mencionadas

&-Ace, 103, 104
&-Prolog, 103, 104

ACE, 103, 104, 128
Andorra-I, 3, 16, 101, 103, 104, 107–109, 124, 126–129
Aurora, 104

BinProlog, 136, 173, 174
Brook, 166, 179
Brook+, 166

C$\star$, 40
CIAO Prolog, 3, 16, 104, 105, 107, 109–111, 121, 124, 126, 127, 129
Cilk, 42
Concurrent Prolog, 41, 104
CS-Prolog, 104
CUDA, 3, 161–166, 179

DAOS, 103, 128
DASWAM, 101, 104
DeltaProlog, 3, 16, 104, 106, 107, 117–119, 124, 126, 127, 129, 172

Emerald, 104
Epilog, 102

FCP, 119

GHC, 104, 119

HPF, 3, 16, 39, 43–50, 67, 72–75, 136, 178

Java threads, 42, 134, 135
Jinni, 173

Linda, 104, 136, 173

MPI, 3, 16, 39, 41, 43–45, 60–62, 64–66, 70, 72–76, 134, 136, 148, 150, 174, 176–178
MPI-2, 44, 45, 60, 61, 73
Muse, 103, 104

NESL, 40

Occam, 41
OpenMP, 3, 16, 39, 43–45, 50–52, 54, 55, 68, 72–75, 136, 173, 178, 198
Orca, 104
Orgel, 3, 16, 131, 139–141, 144, 145, 148–152, 179

P3L-SKIE, 41
ParAKL, 103, 128
Parlog, 3, 16, 41, 101, 104, 105, 107, 111, 112, 119, 121, 124, 126, 127, 129
PLoSys, 103

Pthread, 134, 135, 149, 150
PVM, 3, 16, 39, 41, 43–45, 55–57, 60, 66, 68, 72–76, 106, 107, 115–117, 126, 129, 134, 136, 148, 150, 174, 176–178
PVM-Prolog, 3, 16, 104, 106, 107, 114–116, 122, 125–127, 129, 177

Quintus Prolog, 3, 16, 104, 106, 107, 112, 121, 125–127, 129

ROPM, 102

SCL, 41
SkelML, 41
Skil, 41
SR, 104
STRAND, 112

Bibliografía

[Adv07] Advanced Micro Devices. AMD Stream Computing: Software Stack. Whitepaper, Nov. 2007. http://ati.amd.com/technology/streamcomputing/firestream-sdk-whitepaper.pdf.

[AGL+98] Alverson, G. A., Griswold, W. G., Lin, C., Notkin, D., and Snyder, L. Abstractions for Portable, Scalable Parallel Programming. *IEEE Transactions on Parallel and Distributed Systems 9*, 1 (1998), 71–86.

[AK91] Aït-Kaci, H. *Warren's abstract machine, a tutorial reconstruction.* MIT Press, 1991.

[AK02] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, San Francisco, USA, 2002.

[Akl97] Akl, S. G. *Parallel Computation: Models and Methods.* Prentice Hall, 1997.

Introduce los distintos modelos de computadoras paralelas y presenta los distintos métodos para diseñar algoritmos paralelos (dividir y conquistar, broadcasting con reducción, list ranking, entre otros). Contrasta los modelos y métodos con una serie de problemas computacionales (numéricos, de búsqueda y ordenamiento, entre otros).

[And91] Andrews, G. R. *Concurrent Programming: Principles and Practice.* Benjamin/Cummings Publishing Company, Inc., Redwood City, CA., 1991.

Presenta una introducción más lógica y matemática al tema de concurrencia. Discute los mecanismos de comunicación y sincronización (monitores, semáforos, pasaje de mensajes).

[AR97] Araujo, L., and Ruz, J. J. A Parallel Prolog System for Distributed Memory. *The journal of Logic Programming 33*, 1 (1997), 49–79.

[AS92] Austin, T. M., and Sohi, G. S. Dynamic dependency analysis of ordinary programs. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture* (New York, NY, USA, 1992), ACM, pp. 342–351.

[Bac92] Bacon, J. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems.* Addison-Wesley, 1992.

[Bal91] Bal, H. A Comparative Study of Five Parallel Programming Languages. In *EurOpen Spring Conference on Open Distributed Systems* (Tromso, May 1991), pp. 209–228.

[BCC+97] Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., and Puebla, G. The CIAO Prolog System. Reference Manual. Tech. Rep. CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), Madrid, Aug. 1997. Available from http://www.clip.dia.fi.upm.es/.

[BCP+98] Bacci, B., Cantalupo, B., Pesciullesi, P., Ravazzolo, R., Riaudo, A., and Vanneschi, L. Skie user guide. Tech. rep., QSW Ltd., Rome, Dec. 1998.

[BD93] Burns, A., and Davies, G. *Concurrent Programming.* Addison-Wesley, 1993.

Provee a los lectores de una comprensión intuitiva de lo que es concurrencia y de los problemas clásicos que se encuentran en la programación concurrente. Trata más acerca del software (y de lenguajes de programación) que de arquitecturas de computadoras; expone ejemplos en Pascal-FC, poniendo énfasis en el modelo de pasaje de mensajes y en las primitivas de sincronización y comunicación.

[BDO+93] BACCI, B., DANELUTTO, M., ORLANDO, S., PELAGATTI, S., AND VANNESCHI, M. P^{3}L: a structured high-level paralllel language and, its structured support. Tech. Rep. HPL-PSC-93-55, Pisa Science Center, Hewlett-Packard Laboratories, May 1993.

[BDO+95] BACCI, B., DANELUTTO, M., ORLANDO, S., PELAGATTI, S., AND VANNESCHI, M. Summarising an experiment in parallel programming language design. In *High-Performance Computing and Networking*, B. Hertzberger and G. Serazzi, Eds., vol. 919. 1995, pp. 8–13.

[Bec05] BECKER, D. Why MPI Is Inadequate. Linux Magazine, Nov. 2005.

[Bes96] BEST, E. *Semantics of Sequential and Parallel Programs*. Prentice Hall, 1996.

Describe la semántica operacional de programas secuenciales y paralelos de varios tipos por medio de fórmulas matemáticas, secuencias que modelan las ejecuciones o fórmulas lógicas que expresan los efectos de la ejecución.

[BG94] BURKHART, H., AND GUTZWILLER, S. Steps towards reusability and portability in parallel programming. In *Programming Environments for Massively Parallel Distributed Systems*, K. M. Decker and R. M. Rehmann, Eds. Birkhauser, 1994, pp. 147–157.

[Bin] BINNET CORPORATION. BinProlog. http://www.binnetcorp.com.

[BK96] BOTOROG, G. H., AND KUCHEN, H. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Procs. of the 5th International Symposium on High Performance Distributed Computing (HPDC-5)* (1996), IEEE Computer Society, pp. 243–252.

[BKT92] BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering 18*, 3 (1992), 190–205.

[Bra93] BRATVOLD, T. A. A Skeleton-Based Parallelising Compiler for ML. In *Procs. of the 5th International Workshop on Implementation*

of Functional Languages (Nijmegen, the Netherlands, Sept. 1993), R. Plasmeijer and M. van Eekelen, Eds., pp. 23–33.

[BT93] Bosschere, K. D., and Tarau, P. Blackboard-based extensions for parallel programming in BinProlog. In *ILPS '93: Proceedings of the 1993 international symposium on Logic programming* (Cambridge, MA, USA, 1993), MIT Press, p. 664.

[CER] CERN. Grid Café: the place for everybody to learn about the Grid. http://gridcafe.web.cern.ch/gridcafe/index.html.

[CG86] Clark, K., and Gregory, S. PARLOG: parallel programming in logic. *ACM Trans. Program. Lang. Syst. 8*, 1 (1986), 1–49.

[CG89] Carriero, N., and Gelernter, D. Linda in context. *CACM 32*, 4 (1989), 444–458.

[CG90] Carriero, N., and Gelernter, D. *How to Write Parallel Programs, A First Course*. The MIT Press, 1990.

Los autores se concentran en técnicas de programación para máquinas de propósito general asincrónicas o MIMD. Utilizan C (computing language) y Linda (coordination language).

[CH86] Ciepielewski, A., and Hausman, B. Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs. In *Procs. of the Symposium on Logic Programming* (Los Alamitos, CA, 1986), IEEE Computer Society, pp. 246–257.

[CH01] Cabeza, D., and Hermenegildo, M. Distributed WWW Programming using (Ciao-)Prolog and the PiLLoW library. *Theory and Practice of Logic Programming 1*, 3 (2001), 251–282.

[Cia92] Ciancarini, P. Parallel Programming with Logic Languages: A Survey. *Computer Languages 17*, 4 (1992), 213–239.

[CLS96] Chow, J.-H., Lyon, L. E., and Sarkar, V. Automatic parallelization for symmetric shared-memory multiprocessors. In *Procs. of the 1996 conference of the Centre for Advanced Studies on Collaborative research (CASCON '96)* (1996), IBM Press, p. 5.

[CM96] Cunha, J. C., and Marques, R. F. P. PVM-Prolog: A Prolog Interface to PVM. In *Proceedings of the 1st. Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)* (Budapest, Hungria, Oct. 1996), Hungarian Academy of Sciences, pp. 173–182.

[CMCP92] Cunha, J. C., Medeiros, P. D., Carvalhosa, M. B., and Pereira, L. M. Delta prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In *Implementations of Distributed Prolog*, P. Kacsuk and M. Wise, Eds. Wiley & Sons, New York, 1992, pp. 333–356.

[CRH01] Clark, K., Robinson, P., and Hagen, R. Multi-threading and message communication in Qu-Prolog. *Theory and Practice of Logic Programming 1*, 3 (May 2001), 283–301.

[CSG99] Culler, D. E., Singh, J. P., and Gupta, A. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1999.

Explica la reciente convergencia entre las arquitecturas de memoria compartida, pasaje de mensajes, data-parallel y data-driven. Plantea un framework y trabaja a lo largo del libro con cuatro ejemplos de aplicaciones. Examina las consideraciones de diseño que son críticas para una arquitectura paralela.

[DDdSL02] D'Ambra, P., Danelutto, M., di Serafino, D., and Lapegna, M. Advanced environments for parallel and distributed applications: a view of current status. *Parallel Computing 28* (Sep. 2002), 1637–1662.

[DE07] Duffy, J., and Essey, E. Parallel LINQ: Running Queries on Multi-Core Processors. *MSDN Magazine* (Oct. 2007).

[DFH+93] Darlington, J., Field, A. J., Harrison, P., Kelly, P., Sharp, D., Wu, Q., and While, R. L. Parallel programming using skeleton functions. *PARLE'93, Parallel Architectures and Languages Europe 694* (June 1993).

[DGTY95] DARLINGTON, J., GUO, Y., TO, H. W., AND YANG, J. Parallel skeletons for structured composition. *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press* (1995).

[DPV95] DANELUTTO, M., PELAGATTI, S., AND VANNESCHI, M. Restricted parallel models. In *General Purpose Parallel Computers; Architectures, Programming Environments, and Tools*, G. Balbo and M. Vanneschi, Eds. ETS, Pisa, 1995, pp. 155–208.

[Dut94] DUTRA, I. Strategies for Scheduling And- and Or-Parallel Work in Parallel Logic Programming Systems. M. Bruynooghe, Ed., MIT Press, pp. 289–304.

[ERL97] EL-REWINI, H., AND LEWIS, T. G. *Distributed and Parallel Computing.* Manning Publications Co., Hardbound, 1997.

Clasifica a los sistemas distribuidos de acuerdo a su granularidad: large-grained, medium-grained y fine-grained y a su arquitectura: tightly coupled, coupled, loosely couped y shared database. Introduce distintas medidas de performance para evaluar las limitaciones del paralelismo y las mejoras provistas por las aplicaciones.

[FD90] FAGIN, B. S., AND DESPAIN, A. M. The performance of parallel Prolog programs. *IEEE Transactions on Computer 39*, 12 (1990), 1434–1445.

[FD96] FAGG, G., AND DONGARRA, J. PVMPI: An Integration of the PVM and MPI Systems. *Calculateurs Parallèles 8*, 2 (1996), 151–166.

[FD00] FAGG, G., AND DONGARRA, J. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *Euro PVM/MPI User's Group Meeting* (2000), 346–353.

[FFMM94] FININ, T., FRITZSON, R., MCKAY, D., AND MCENTIRE, R. KQML as an Agent Communication Language. In *Procs. of the 3rd Intl Conference on Information and Knowledge Management (CIKM'94)* (Gaithersburg, MD, USA, 1994), N. Adam, B. Bhargava, and Y. Yesha, Eds., ACM Press, pp. 456–463.

[FH08] Fatahalian, K., and Houston, M. GPUs: A Closer Look. *ACM Queue 6*, 2 (2008), 19–28.

[FHW00] Falaschi, M., Hicks, P., and Winsborough, W. Demand tranformation analysis for concurrent constraint programs. *The Journal of Logic Programming 42*, 3 (March 2000), 185–215.

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

Define Grid Computing y el campo de investigación asociado, propone una arquitectura de Grid y discute las relaciones entre las tecnologías en Grid y otras tecnologías contemporáneas.

[Fly95] Flynn, M. J. *Computer Architecture: Pipelined and Parallel Processor Design*. Jones and Bartlett, 1995.

[Fos95] Foster, I. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. http://www-unix.mcs.anl.gov/dbpp/.

Enfatiza los cuatro requerimientos fundamentales del software paralelo: concurrencia, escalabilildad, localidad y modularidad. Trabaja con varios modelos como el de tarea/canal, pasaje de mensajes, paralelismo de datos y memoria compartida. Utiliza varios lenguajes: CC++, FM , HPF y MPI.

[Fos02] Foster, I. What is the Grid? A Three Point Checklist. GRIDToday, July 2002.

[Gar97] García, A. J. La Programación en Lógica Rebatible, su definición teórica y computacional. Master's thesis, Departamento de Ciencias de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina, 1997.

[GBD+94] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. *PVM: Parallel Virtual Machine, A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[GdlB94] Garcia de la Banda, M. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming.* PhD thesis, Universidad Politecnica de Madrid, 1994.

[GJ93] Gupta, G., and Jayaraman, B. Analysis of Or-Parallel Execution Models. *ACM Transactions on Programming Languages and Systems (TOPLAS) 15*, 4 (Sept. 1993), 659–680.

[GKP96] Geist, G. A., Kohla, J. A., and Papadopoulos, P. M. PVM and MPI: A Comparison of Features. *Calculateurs Paralleles 8*, 2 (1996), 137–150.

[GKT91] Goff, G., Kennedy, K., and Tseng, C. Practical Dependence Testing.

[GL97] Gropp, W., and Lusk, E. L. Why Are PVM and MPI So Different? In *Procs. of the 4th European PVM/MPI Users' Group Meeting* (Poland, Nov. 1997), Lecture Notes on Computer Science 1332, Springer, pp. 3–10.

[GLDS96] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing 22*, 6 (1996), 789–828.

[GPA+01] Gupta, G., Pontelli, E., Ali, K. A. M., Carlsson, M., and Hermenegildo, M. V. Parallel execution of prolog programs: a survey. *ACM Transactions on Programming Languages and Systems (TOPLAS) 23*, 4 (2001), 472–602.

[Gup94] Gupta, G. *Multiprocessor Execution of Logic Programs.* Kluwer Academic Publishers, 1994.

Es una edición revisada de su Tesis de Ph.D. de 1992. El libro se concentra en la ejecución paralela de lenguajes lógicos como Prolog, siendo el tema principal la explotación automática del paralelismo en lenguajes lógicos en máquinas multiprocesador comerciales de granularidad mediana o gruesa.

[Han95] Hansen, P. B. *Studies in computational science: Parallel Programming Paradigms.* Prentice Hall, 1995.

Discute cinco paradigmas de programación para computación entre todos los pares (producto cruzado), multiplicación de tuplas, dividir y conquistar, ensayos de Monte Carlo y autómatas celulares. Los programas están escritos en Super-Pascal, y están pensados para multicomputadoras que se comunican por pasaje de mensajes.

[Har90] HARIDI, S. A logic programming language based on the Andorra model. *New Generation Computing 7*, 2/3 (1990), 109–125.

[Has95] HASENBERGER, J. Modelling and Redesign the Advanced Traffic Management System in Andorra-I. In *Proceedings of the Workshop on parallel Logic Programming Systems* (1995), Univesity of Porto.

[HB84] HWANG, K., AND BRIGGS, F. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

Se estudian arquitecturas de computadoras avanzadas, observando ejemplos comerciales concretos. Las computadoras paralelas se clasifican en tres configuraciones: pipelined computers, array processors y sistemas multiprocesador.

[Her87] HERMENEGILDO, M. Relating Goal Scheduling, Precedence, and Memory Management in And-parallel Execution of Logic Programs. MIT Press, pp. 556–575.

[HF93] HAINS, G., AND FOISY, C. The data-parallel categorical abstract machine. *PARLE'93, Parallel Architectures and Languages Europe 694* (June 1993).

[HG91] HERMENEGILDO, M., AND GREENE, K. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing 9*, 3,4 (1991), 233–257.

[HH03] HUGHES, C., AND HUGHES, T. *Parallel and Distributed Programming Using C++.* Addison Wesley, Aug. 2003.

[Hig97] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification, version 2.0. Tech. Rep. CRPC-TR92225, Rice University, Houston, Texas, 1997.

[HN86] Hermenegildo, M., and Nasr, R. I. Efficient Management of Backtracking in AND Parallelism. In *Procs. of the 3rd International Conference on Logic Programming* (1986), pp. 40–54.

[Hro97] Hromkovic, J. *Communication Complexity and Parallel Computing.* Springer, 1997.

[HSYHSODS00] Hong-Soog, K., Young-Ha, Y., Sang-Og, N., and Dong-Soo, H. ICU-PFC: an automatic parallelizing compiler. vol. 1, pp. 243–246.

[Hug02] Huget, M. P. Desiderata for Agent Oriented Programming Languages. Tech. Rep. ULCS-02-010, Agent ART Group, University of Liverpool, 2002.

Este paper recorre un conjunto de características que pueden estar presentes en un lenguaje de programación orientado a agentes. Las mismas se clasifican en dos grandes grupos: las características que se obtienen a partir de un lengueje de programación con concurrencia (tipos, arreglos, bucles, excepciones, exclusión mutua, sincronización, etc) y las que involucran a los sistemas multiagentes (BDI, soporte para comunicación en alto nivel, soporte para mantener el conocimiento, convivencia de formalismos, estándares, etc). Se contrasta Java con este conjunto de propiedades, intentando demostrar por qué es el preferido en la comunidad de agentes.

[Int03] Intelligent Systems Laboratory. *Quintus Prolog User's Manual.* Swedish Institute of Computer Science (SICS), Sweden, June 2003. Release 3.5.

[Kal85] Kalé, L. *Parallel Architectures for Problem Solving.* PhD thesis, SUNY Stony Brook, Dept. Computer Science, 1985.

[KE01] Kim, S., and Eigenmann, R. The Structure of a Compiler for Explicit and Implicit Parallelism. In *Procs. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'01)* (Cumberland Falls, KY, Aug. 2001).

[KGGK94] Kumar, V., Grama, A., Gupta, A., and Karypis, G. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, Addison-Wesley Publishing Company, 1994.

Provee un estudio profundo de las técnicas para el diseño y análisis de algoritmos paralelos. Tiene una buena introducción al paralelismo y a los modelos de computadoras paralelas. La cobertura incluye algoritmos para búsqueda y de recorrido de grafos, técnicas de optimización discreta y aplicaciones de computación científica. El último capítulo discute algunos paradigmas de programación en paralelo.

[Kow79] KOWALSKI, R. *Logic for Problem Solving.* Elsevier North-Holland, Amsterdam, 1979.

[KP93] KEßLER, C. W., AND PAUL, W. J. Automatic Parallelization by Pattern-Matching. In *Procs. of the Second International ACPC Conference on Parallel Computation* (London, UK, 1993), Springer-Verlag, pp. 166–181.

[KP04] KYRIAKOPOULOS, K., AND PSARRIS, K. Data dependence analysis techniques for increased accuracy and extracted parallelism. *Int. Journal of Parallel Programming 32*, 4 (2004), 317–359.

[Lab01] LABROU, Y. Standardizing Agent Communication. In *Proceedings of the Advanced Course on Artificial Intelligence (ACAI'01)* (2001), Springer-Verlag.

[Les93] LESTER, B. P. *The Art of Parallel Programming.* Prentice-Hall, Inc., 1993.

[Lew94] LEWIS, T. G. *Foundations of parallel programming: a machine-independent approach.* IEEE Computer Society Press, 1994.

Este libro se involucra con el diseño de programa paralelos que utilizan un pequeño número de constructores fundamentales que son suficientemente poderosos como para expresar cualquier algoritmo paralelo. Además, introduce medidas analíticas de performance para cada uno de estos constructores de paralelismo. Al final de cada capítulo provee una buena definición de palabras claves y conceptos que aparecen una y otra vez cuando se habla de paralelismo.

[Li92] Li, Z. Array Privatization for Parallel Execution of Loops. In *ICS '92: Procs. of the 6th International Conference on Supercomputing* (Washington, D. C., United States, 1992), ACM Press, pp. 313–322.

[LNOM08] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), 39–55.

[Mac94] MacColl, W. F. An architecture independent programming model for scalable parallel computing. In *Portability and Performance for Parallel Processors*, J. Ferrante and A. J. G. Hey, Eds. Wiley, New York, 1994.

[Mar96] Marques, R. Um Modelo de Programação Paralela e Distribuída Para o Prolog. Master's thesis, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nove de Lisboa, Lisboa, 1996.

[MAS+02] McDonald, S., Anvik, J., Szafron, D., Schaeffer, J., Bromling, S., and Tan, K. From pattern to frameworks to parallel programs. *Parallel Computing 28* (Sep. 2002).

[MB00] Marowka, A., and Bercovier, M. A Scalable Portability Model for Parallel Computing. *Parallel and Distributed Computing Practices 3*, 3 (Sep. 2000).

[MBGdlBH99] Muthukumar, K., Bueno, F., García de la Banda, M., and Hermenegildo, M. Automatic compile-time parallelization of logic programs for restricted, goal level, independent and parallelism. *The Journal of Logic Programming 38*, 2 (Feb. 1999), 165–218.

[MC99] Marques, R., and Cunha, J. C. *PVM-Prolog: Programmers Reference Manual*. Faculdade de Ciências e Tecnologia, Universidade Nove de Lisboa, Lisboa, 1999.

[Mes] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-1.1/mpi-report.htm.

[MG03] Marchetti, T., and García, A. J. Evaluación de Plataformas para el Desarrollo de Sistemas Multiagente. In *IX Congreso Argentino de*

Ciencias de la Computación (CACiC'03) (La Plata, Buenos Aires, 2003), pp. 625–636.

[Mic08] MICROSOFT CORPORATION. Parallel Computing Development Center, 2008. http://msdn.microsoft.com/en-us/concurrency/default.aspx.

[Mor98] MORGAN, R. *Building an Optimizing Compiler.* Butterworth-Heinemann Publishers, Boston, USA, 1998.

[MS96] MILLER, R., AND STOUT, Q. F. *Parallel Algorithms for Regular Architectures.* The MIT Press, 1996.

El punto central de este libro es desarrollar algoritmos óptimos para resolver problemas en conjuntos de procesadores configurados como mesh o pyramid.

[NBGS08] NICKOLLS, J., BUCK, I., GARLAND, M., AND SKADRON, K. Scalable Parallel Programming with CUDA. *ACM Queue 6*, 2 (2008), 40–53.

[Ope02] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP C and C++ Application Program Interface*, March 2002. version 2.0.

[OYON00] OHNO, K., YAMAMOTO, S., OKANO, T., AND NAKASHIMA, H. Orgel: A Parallel Programming Language with Declarative Communication Streams. In *Procs. of the 3rd International Symposium on High Performance Computing (ISHPC'00)* (2000), vol. 1940 of *Lecture Notes in Computer Science*, Springer, pp. 344–354.

[Par] PARAWIKI. A Wiki resource for the Parallel Programming Community. http://parawiki.plm.eecs.uni-kassel.de/.

[PG95] PONTELLI, E., AND GUPTA, G. On the Duality Between And-Parallelism and Or-Parallelism. S. Haridi and P. Magnusson, Eds., Springer-Verlag, pp. 43–54.

[PH99] PUEBLA, G., AND HERMENEGILDO, M. Abstract multiple specialization and its application to program parallelization. *The Journal of Logic Programming 41*, 2-3 (Nov./Dec. 1999), 279–316.

[PP96] Petersen, P. M., and Padua, D. A. Static and Dynamic Evaluation of Data Dependence Analysis Techniques. *IEEE Transactions on Parallel and Distributed Systems 7*, 11 (1996), 1121–1132.

[Ros97] Roscoe, A. W. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

Explica CSP (Communicating Sequential Processes), una notación para describir sistemas concurrentes cuyos procesos interactúan con otros vía comunicación, como también un conjunto de modelos matemáticos para utilizar la notación.

[RR88] Richard, G., and Rizk, A. Semantics of the concurrent logic programming language PARLOG. Tech. Rep. RR-0848, INRIA Rocquencourt, France, May 1988.

[Sch97] Schneider, F. B. *On Concurrent Programming.* Springer, 1997.

Introduce al lector a temas básicos como: lógica proposicional y de predicados, lógica temporal y de Hoare para programas secuenciales. Discute razonamiento y derivación sobre programas concurrentes.

[SCWY91a] Santos Costa, V., Warren, D. H. D., and Yang, R. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (April 1991), ACM Press, pp. 83–93.

[SCWY91b] Santos Costa, V., Warren, D. H. D., and Yang, R. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Proceedings of the Int. Conf. on Logic Programming* (1991), MIT Press, pp. 825–839.

[Seh92] Sehr, D. C. *Automatic parallelization of Prolog programs.* PhD thesis, University of Illinois, Urbana-Champaign, Illinois, 1992.

[Sha87] Shapiro, E. A subset of concurrent prolog and its interpreter. In *Concurrent Prolog: Collected Papers (Volume I)*, E. Shapiro, Ed. MIT Press, London, 1987, pp. 27–83.

[She92] Shen, K. *Studies in And/Or Parallelism in Prolog.* PhD thesis, University of Cambridge, 1992.

[Shi98] Shires, D. MPI and HPF Performance in a DEC Alpha Cluster. Tech. Rep. ARL-TR-1668, Army Research Laboratory, May 1998.

[Sho93] Shoham, Y. Agent-oriented programming. *Artificial Intelligence 1*, 60 (1993), 51–92.

[Ski90] Skillicorn, D. B. Architecture-independent parallel computation. *IEEE Computer 23*, 12 (Dec. 1990), 38–51.

[SL05] Süß, M., and Leopold, C. Evaluating the state of the art of parallel programming systems. Tech. Rep. 1, University of Kassel, Research Group Programming Languages / Methodologies, Kassel, Germany, March 2005.

[SS96] Szafron, D., and Schaeffer, J. An experiment to measure the usability of parallel programming systems. *Concurrency: Practice and Experience 8*, 2 (1996), 147–166.

[ST98] Skillicorn, D., and Talia, D. Models and Languages for Parallel Computation. *ACM Computing Surveys 30*, 2 (June 1998), 123–169.

[SW00] Silva, F., and Watson, P. Or-parallel prolog on a distributed memory architecture. *The Journal of Logic Programming 43*, 2 (May 2000), 173–186.

[Tal94] Talia, D. Parallel logic programming systems on multicomputers. *Programming Languages 2*, 1 (March 1994), 77–87.

[Tan98] Tanenbaum, A. S. *Structured Computer Organization*, 4th edition ed. Prentice Hall, 1998.

[Tar98] Tarau, P. Jinni: a Lightweight Java-based Logic Engine for Internet Programming. In *Implementation Technology for Programming Languages based on Logic* (1998), pp. 1–15.

[Tar05] TARAU, P. Agent oriented logic programming in Jinni 2004. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing* (New York, NY, USA, 2005), ACM, pp. 1427–1428.

[TBG+02] TIAN, X., BIK, A., GIRKAR, M., GREY, P., SAITO, H., AND SU, E. Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology. *Intel Technology Journal 6*, 1 (Feb. 2002), 36–46.

Este paper presenta la arquitectura del compilador y las técnicas aplicadas para la paralelización por medio de las directivas y los pragmas de OPENMP. También se describen las optimizaciones realizadas y una evaluación de performance, a partir de distintos gráficos que muestran los resultados de un conjunto de benchmarks y aplicaciones.

[TF86] TAKEUCHI, A., AND FURUKAWA, K. *3rd International Conference on Logic Programming.* London, July 1986, ch. Parallel Logic Programming Languages, pp. 242–254.

[TL98] TONG, B., AND LEUNG, H. Data-parallel concurrent constraint programming. *The journal of Logic Programming 35* (1998), 103–150.

[To95] TO, H. W. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, University of London Imperial College of Science, Technology and Medicine, Department of Computing, 1995.

[Ued89] UEDA, K. Parallelism in Logic Programming. In *IFIP* (1989), no. Generation, pp. 957–964.

[Van02] VANNESCHI, M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing 28* (Sep. 2002).

[VR90] VAN ROY, P. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, Dec. 1990.

Provee una prueba constructiva de que el lenguaje Prolog puede ser implementado con una eficiencia tal que se puede comparar a la de lenguajes

imperativos como C para una clase significativa de problemas. El diseño de Aquarius Prolog, el sistema resultante, se basa en reducir la granularidad de las instrucciones, explotar el determinismo, especializar la unificación y realizar análisis del flujo de datos.

[War83] WARREN, D. H. D. An Abstract Prolog Instruction Set. Tech. Rep. 309, SRI International, Menlo Park, CA, Oct. 1983.

[War88] WARREN, D. H. D. The Andorra Principle. Seminar given at Gigalips Workshop, Sweden, 1998.

[Wei99] WEISS, G. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence.* MIT Press, 1999.

[WF] WIKIMEDIA FOUNDATION, I. Wikipedia, The Free Encyclopedia. http://www.wikipedia.org/.

[WG04a] WEINBACH, N. L., AND GARCÍA, A. J. Event Management for the Development of Multi-agent Systems using Logic Programming. In *X Congreso Argentino de Ciencias de la Computación (CACiC 2004)* (Universidad Nacional de La Matanza, San Justo, Buenos Aires, Argentina, Oct. 2004), pp. 1689–1700.

[WG04b] WEINBACH, N. L., AND GARCÍA, A. J. Una Extensión de la Programación en Lógica que incluye Eventos y Comunicación. In *VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004)* (Universidad Nacional del Comahue, Neuquén, Argentina, Mayo 2004), pp. 379–383.

[Wis86] WISE, M. J. *Prolog Multiprocessors.* Prentice-Hall, 1986.

[WIS95] WILSON, G., IRVIN, R., AND SUKUL, A. Assessing and Comparing the Usability of Parallel Programming Systems. Tech. Rep. CRSI-321, Department of Computer Science University of Toronto, March 1995.

[Wol95] WOLFE, M. *High-performance Compilers For Parallel Computing.* Pearson Education Canada, June 1995.

[WTG05] Weinbach, N. L., Tucat, M., and García, A. J. Programación en Paralelo Utilizando un Modelo de Sistemas Multi-agente. In *XI Congreso Argentino de Ciencias de la Computación (CACiC 2005)* (Universidad Nacional de Entre Ríos, Concordia, Entre Ríos, Argentina, Oct. 2005).

[Zav99] Zavanella, A. *Skeletons and BSP: Performance Portability for Parallel Programming.* PhD thesis, Università degli Studi di Pisa, Dipartimento di Informatica, 1999.