Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA
Unidad Iztapalapa

División de Ciencias Básicas e Ingeniería

Lenguaje Visual para el Desarrollo de Programas Paralelos

Tesis para obtener el grado de Doctor en Ciencias
(Doctor en Ciencias y Tecnologías de la Información)

M. en C. José Luis Quiroz Fabián

Asesores:

Dra. Graciela Román Alonso
Dr. Miguel Alfonso Castro García

Jurado calificador:

Presidente: Dr. René Luna García ....
Secretario: Dr. Ricardo Marcelín Jiménez ....
Vocal: Dr. Amilcar Meneses Viveros ....
Vocal: Dr. René MacKinney Romero ....
Vocal: Dra. Graciela Román Alonso ....

Ciudad de México
Septiembre 2019

Me gustaría dedicar esta tesis a mi padres, mi esposa y mi hijo ...

Agradecimientos

A mis padres, **María** y **Moisés**, les agradezco su apoyo, su guía y su confianza en la realización de mis estudios. Soy afortunado por contar con su amor y apoyo.

A mi esposa **Adriana**, ella ha sabido aceptar y apoyar estos retos que me he encaminado a pesar de las ausencias que ello conlleva.

A mis hermanos, **Angélica**, **Claudia**, **Israel** y **Moisés**, por estar siempre a mi lado y en especial a mi hermano Moisés a quien tuve la fortuna de conocer 16 años de mi vida.

A mis asesores la **Dra. Graciela Román Alonso** y el **Dr. Miguel Alfonso Castro García**, por su constante apoyo y asesoramiento en todos los aspectos de la investigación y elaboración de esta tesis, así como por la confianza depositada en mi.

Al **Dr. Manuel Aguilar Cornejo**, por el apoyo y consejos brindados durante mi camino en la UAM.

Al **Consejo Nacional de Ciencia y Tecnología (Conacyt)**, por la beca que me otorgó durante mi estancia en el Doctorado.

A la **(UAM) Universidad Autónoma Metropolitana**, por todo el apoyo que siempre he tenido de esta gran institución.

A todos los **integrantes del Posgrado en Ciencias y Tecnologías de la Información**, amigos y profesores, sin su ayuda no estaría donde me encuentro ahora.

En especial me gustaría agradecer a mi hijo, **Damían Moisés**, él me ha motivado a seguir adelante y ha generado en mi un sentimiento que es muy grande, el amor de un padre a su hijo.

Resumen

Hoy por hoy, las arquitecturas paralelas se encuentran en todas partes, desde clusters, computadoras con múltiples unidades de procesamiento, hasta celulares y tabletas. Sin embargo, el programar estos dispositivos de forma eficiente puede ser una tarea compleja. El programar en estas arquitecturas paralelas requiere: que el desarrollador tenga un conocimiento profundo del problema a resolver y los lenguajes, herramientas y bibliotecas de programación (lo anterior se complica cuando los conocimientos del programador son únicamente de programación secuencial). Este problema se puede reducir significativamente mediante el uso de lenguajes visuales, los cuales permiten esconder aspectos teóricos y de implementación relacionados con la especificación de la comunicación y administración de procesos (por ejemplo creación de procesos y grupos).

Esta tesis presenta VPPL (Visual Parallel Programming Language), un lenguaje visual para el desarrollo de programas paralelos que permite especificar un programa a través del modelo de flujos de trabajo (*Workflow*), los cuales se construyen mediante la composición de iconos. Los iconos en VPPL son de diferentes tipos y permiten especificar: flujo de control (organización del flujo de trabajo), operaciones de entrada y salida, comunicación y procesamiento. Los iconos de procesamiento de VPPL incluyen un conjunto de patrones de cómputo paralelo, SPMD (Single Program Multiple Data), MPMD (Multiple Program Multiple Data), Pipeline y Maestro-Esclavo. Los programas en VPPL son descritos por medio de una gramática formal visual.

En la tesis se propone también una ambiente de desarrollo integrado, el cual se denomina VPPE (Visual Parallel Programming Environ-

ment) para construir programas en VPPL. El ambiente tiene una arquitectura definida para un funcionamiento en la nube, la cual para su ejecución solo requiere un navegador web. Cuando un flujo de trabajo se ejecuta, VPPE transparentemente genera código en un lenguaje de alto nivel textual (Java-MPI) el cual se ejecuta en un cluster. Para validar la propuesta, mostramos una comparación cualitativa y de rendimiento. La primera permite diferenciar los pasos de programación de un programa paralelo realizado de forma textual como tradicionalmente se implementa vs realizar el desarrollo del programa con VPPL y VPPE. La segunda comparación permite evaluar el desempeño de programas generados con VPPL.

Índice general

Índice de figuras viii

Índice de tablas x

1. Introducción 1

1.1. Motivación . . . 3
1.2. Objetivos . . . 4
1.2.1. Objetivo general . . . 4
1.2.2. Objetivos particulares . . . 4
1.3. Organización de la tesis . . . 5

2. Estado del arte 6

2.1. Programación con paso de mensajes . . . 6
2.2. Lenguajes visuales . . . 7
2.3. Características de los flujos de trabajo . . . 8
2.3.1. Expresividad . . . 8
2.3.2. Aplicabilidad . . . 11
2.3.3. Nivel de interacción . . . 12
2.3.4. Portabilidad . . . 14

3. Gramática de grafos 17

3.1. Introducción y terminología . . . 17
3.1.1. Hiperarista . . . 17
3.1.2. Hipergrafo . . . 18

3.2. Reemplazo de hiperarista 19
3.3. Gramática de reemplazo de hiperaristas 21
3.4. Definición de un lenguaje de programación paralela mediante gramática de grafos 22

4. Lenguaje VPPL 26
4.1. Introducción 26
4.2. Estructuras visuales e iconos de VPPL 28
4.2.1. Estructuras de flujo de trabajo 28
4.2.2. Iconos de entrada y salida 29
4.2.3. Iconos de procesamiento 30
4.2.4. Iconos de comunicación 31
4.3. Gramática del lenguaje VPPL 35
4.3.1. Elementos de la gramática de VPPL 35
4.3.2. Reglas de producción 37

5. Programación en VPPL 44
5.1. Programa secuencial 44
5.2. Multiplicación de matrices usando el modelo SPMD 45
5.3. Inversa de una matriz 46
5.4. Procesamiento de un lote de imágenes usando el modelo Maestro-Esclavo. 48
5.5. Algoritmo genético iterativo usando SPMD 49

6. Arquitectura del ambiente de programación VPPE 52
6.1. Interfaz de usuario 53
6.2. Traductor 55
6.3. Motor de ejecución 59
6.4. Módulo de persistencia 60

7. Evaluación y resultados 64
7.1. Comparación cualitativa: VPPL vs PBT 64
7.2. Evaluación de desempeño 67

7.2.1. Programas 67
7.2.2. Infraestructura utilizada 68
7.2.3. Resultados 69

8. Conclusiones y trabajo futuro **73**

Bibliografía **77**

Acrónimos **82**

Glosario **83**

Índice de figuras

1.1. Representación visual de un programa paralelo. 2

2.1. Tipos de diagramas visuales más comunes. 9
2.2. Envío y recepción en CODE y HeNCE. 11
2.4. Encapsulamiento en CODE y Trapper. 13

3.1. Hiperarista con 3 tentáculos. 18
3.2. Elementos del hipergrafo H. 19
3.3. Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R. 20
3.4. Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R. 20
3.5. Gramática de reemplazo de hiperarista. 21
3.6. Ejemplo de una derivación en una gramática de reemplazo de hiperarista. 22

4.1. Iconos para un flujo de trabajo en VPPL. 29
4.2. Iconos de entrada y salida en VPPL. 29
4.3. Iconos procesamiento en VPPL. 30
4.4. Tipos de comunicación en VPPL. 32
4.5. Iconos de comunicación de salida: difusión, dispersión y difusión-dispersión. 32
4.6. Parte de un flujo de trabajo representando una operación de dispersión. 33

4.7. Iconos de comunicación de entrada: recolección regular (G_R) e irregular G_I . . . 33
4.8. Parte de un flujo de trabajo de una recolección irregular (a) y parte de un flujo de trabajo con tres variables de una recolección irregular. 34
4.9. Gramática del lenguaje VPPL. . . . 36
4.10. Derivación simple de un flujo de trabajo. . . . 38
4.11. Ejemplo de la derivación de un flujo de trabajo SPMD. . . . 39
4.12. Derivación un flujo de trabajo MPMD. . . . 40
4.13. Ejemplo de un flujo de trabajo anidado. . . . 41
4.14. Flujo de trabajo con un icono SPMD dentro de un ciclo. . . . 41

5.1. Flujo de trabajo secuencial 45
5.2. Flujo de trabajo VPPL para la multiplicación de matrices (lado izquierdo) y su paralelismo subyacente (lado derecho). . . . 46
5.3. Flujo de trabajo VPPL para calcular la inversa de una matriz. . . 47
5.4. Flujo de trabajo VPPL para el procesamiento de imágenes. . . . 49
5.5. Flujo de trabajo VPPL implementando un algoritmo genético. . . 50

6.1. Arquitectura del ambiente VPPE 52
6.2. Un flujo de trabajo por medio de VPPL. . . . 53
6.3. Editor visual VPPE 54
6.4. Estructura VPPE-List 55
6.5. Arquitectura del traductor VPPE 56
6.6. Traducción de un flujo de trabajo a código Java-MPI 57
6.7. Algoritmo para generación de grupos. . . . 58
6.8. Grupo de procesos en un flujo de trabajo. . . . 59
6.9. Traduciendo y ejecutando un flujo de trabajo. . . . 60
6.10. Creación de un archivo XML a partir de un flujo de trabajo. . . . 61
6.11. Proceso de carga de un flujo de trabajo. . . . 62

7.1. Comparación de desempeño para el problema de 17-Reinas: VPPE vs PBT 70
7.2. Comparación de desempeño para el procesamiento de un conjunto de imágenes: VPPE vs PBT 71

Índice de tablas

2.1. Elementos visuales de la expresividad. 10
2.2. Comportamientos del sistema que apoyan el nivel de interacción. . 13
2.3. Características de portabilidad 14
2.4. Comparativo de los lenguajes visuales que siguen el modelo de flujo de trabajo 15

7.1. Comparación cualitativa entre programación con PBT y VPPL. . 65
7.2. Características de la plataforma experimental. 69

Capítulo 1

Introducción

En los últimos años, las arquitecturas paralelas han retomado un papel fundamental en la solución de problemas. Cada vez es más común encontrar procesadores multi-núcleo (multi-cores) y tarjetas gráficas (Graphics Processing Unit (GPU)) en computadoras personales y en dispositivos móviles. Incluso, recursos como los clusters[1] o los Grids[2] que eran impensables para pequeñas instituciones y organizaciones actualmente están disponibles gracias al cómputo en la nube, el cual permite rentar infraestructura de cómputo como un servicio. Sin embargo, a pesar de la accesibilidad de dichas arquitecturas, el desarrollo eficiente de programas paralelos en memoria distribuida para hacer uso de esos recursos sigue siendo un reto. A diferencia de un programa secuencial, donde únicamente se requiere del conocimiento del problema a resolver y del lenguaje de programación con que se implementará, los programas paralelos en memoria distribuida requieren un conocimiento más profundo de otros aspectos que están interrelacionados: el particionamiento del trabajo entre un conjunto de unidades de procesamiento y la comunicación y sincronización entre éstas para compartir datos o código. El particionamiento del trabajo requiere tanto del conocimiento del problema como de la arquitectura paralela a utilizar a fin de tomar la mayor ventaja de los componentes de ésta. Igualmente la comunicación y sincronización requieren un conocimiento del problema, del hardware, de los lenguajes de programación y

[1] Conjunto de computadores conectadas a un red de alta velocidad

[2] Sistema distribuido conformado por un conjunto de clusters.

las bibliotecas a utilizar para ese fin buscando de reducir la latencia, es decir, el retardo o el tiempo necesario que toma una comunicación.

Una forma de simplificar y agilizar el desarrollo de programas paralelos en memoria distribuida es el uso de un Lenguaje de Programación Visual (LPV), el cual permite describir gráficamente tareas simultáneas y la manera en como se comunican y sincronizan [BDH+94] [Doz01] [BD91] [BBG11] [BB11] [CZJz+08] [CCCG04] [CCG06] [CCS03]. En general, los LPVs permiten ocultar detalles respecto a la comunicación y sincronización de tareas para el desarrollo de programas paralelos. Los LPVs le proporcionan a los desarrolladores abstracciones de alto nivel que les ayudan a especificar la estructura de un programa paralelo mediante una representación simple de tareas concurrentes; por ejemplo, las ejecuciones en paralelo se pueden representar gráficamente mediante un árbol, donde cada nodo representa una tarea secuencial y los nodos en el mismo nivel representan tareas que se ejecutan concurrentemente, como se presenta en la Figura 1.1.

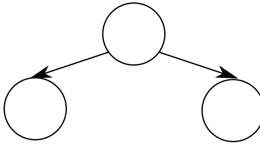


Figura 1.1: Representación visual de un programa paralelo.

Una forma de caracterizar los lenguajes visuales se presenta en [Doz01] en donde se consideran aspectos como: expresividad, aplicabilidad, nivel interacción y portabilidad. La expresividad describe todos los elementos visuales y las funciones que proporciona el lenguaje para ayudar a los programadores al desarrollo de sus programas. La aplicabilidad se refiere al rango de programas que pueden ser implementados con el lenguaje. El nivel de interacción es el número de interacciones que requiere realizar el desarrollador con la herramienta a fin de implementar su algoritmo. La portabilidad es la característica de poder hacer uso del lenguaje en diferentes plataformas (por ejemplo Unix, Linux, Windows, etc.) y con diferentes herramientas y/o bibliotecas (por ejemplo Message Passing Interface (MPI) y Parallel Virtual Machine (PVM) [LR06] [GBD+94]).

1.1. Motivación

Los lenguajes de programación más utilizados para el desarrollo de programas en memoria distribuida son los lenguajes textuales. Estos lenguajes definen operaciones para especificar el particionamiento del trabajo, la comunicación y la sincronización de tareas. Sin embargo, los lenguajes textuales presentan varios inconvenientes, entre los cuales podemos destacar que: no permiten una abstracción adecuada de la concurrencia; se tienen que nombrar a los procesos a fin de establecer código condicional que permita decidir que acción sigue cada proceso, lo cual puede conllevar a errores si se define incorrectamente el código condicional; se tienen varios parámetros en las operaciones de comunicación (por el ejemplo el nombre o identificador del proceso emisor y/o receptor), por lo cual si alguno se establece de forma incorrecta se pueden generar errores; se tiene que definir grupos de procesos, etc.

Como se mencionó previamente, una alternativa para hacer más simple o agilizar el desarrollo de programas paralelos es el uso de LPVs. No obstante, hacer que un LPV cumpla con características adecuadas de expresividad, aplicabilidad, nivel de interacción y portabilidad es una tarea difícil, debido a que cada una de ellas considera diferentes aspectos que originan dependencias entre estas características, la cuales pueden afectar la definición del lenguaje visual; por ejemplo, un lenguaje visual que proporciona un icono simple para resolver un problema particular tiene un nivel de interacción alto, sin embargo su aplicabilidad es limitada debido a que es muy especializado. Un lenguaje muy expresivo puede afectar el nivel de interacción, dado que el programador debe trabajar en un nivel de detalle muy bajo debido al número de pasos o interacciones requeridos de la herramienta. La portabilidad toma en cuenta aspectos como la ejecución en múltiples plataformas lo cual puede limitar el tipo de interacción, y el acceso al lenguaje/entorno sin requerir software o configuraciones adicionales.

Por estas razones es de suma importancia proponer lenguajes visuales que cumplan en mayor medida con dichas características buscando reducir sus dependencias sin afectar la simplicidad en el desarrollo de programas paralelos. Además, considerando que un elemento importante en el cómputo paralelo es el tiempo de ejecución, el tiempo de un programa visual no debe ser muy grande

respecto al tiempo de ejecución del mismo desarrollado mediante los lenguajes textuales de programación paralela.

1.2. Objetivos

1.2.1. Objetivo general

Definición de un nuevo lenguaje visual para el desarrollo de programas paralelos en memoria distribuida.

1.2.2. Objetivos particulares

Para lograr el objetivo general se han de cumplir, entre otras etapas, con los siguientes objetivos particulares:

1. Caracterizar los lenguajes visuales para el desarrollo de programas paralelos, dando especial importancia a los diagramas conocidos como flujos de trabajo por ser intuitivos para describir la ejecución de sistemas concurrentes.

2. Diseño y definición formal de un nuevo lenguaje visual llamado Visual Parallel Programming Languaje (VPPL) considerando las características/aspectos propuestos en [Doz01]: expresividad, aplicabilidad, nivel de interacción y portabilidad.

3. Propuesta de una arquitectura y un ambiente de desarrollo, Visual Parallel Programming Environment (VPPE), para el lenguaje VPPL a fin de llevar a cabo la traducción y ejecución de un programa visual en un cluster.

4. Evaluación del lenguaje VPPL y de su ambiente VPPE para mostrar que VPPL simplifica el desarrollo de programas paralelos y que el desempeño respecto al programa desarrollado de forma textual no se ve fuertemente afectado.

1.3. Organización de la tesis

La presente Tesis Doctoral estructura su contenido en 8 capítulos, los cuales se resumen a continuación.

- En este primer Capítulo, se presenta el planteamiento general de la Tesis Doctoral.
- El Capítulo 2 presenta el estado del arte de los ambientes y lenguajes visuales, describiendo sus características y modelo de programación. Este capítulo permite conocer las ventajas y desventajas de los ambientes y lenguajes existentes.
- El Capítulo 3 presenta una introducción a la gramática de grafos mediante el reemplazo de hiperaristas. Esta parte introduce los conceptos requeridos para entender la gramática propuesta para el lenguaje visual VPPL.
- El Capítulo 4 presenta el lenguaje propuesto VPPL. Se describen y clasifican cada unos de sus elementos gráficos. También se muestra la gramática de VPPL y se describe la función de cada una de sus reglas de producción.
- El Capítulo 5 presenta un conjunto de ejemplos de programas paralelos que se pueden desarrollar mediante el lenguaje VPPL.
- El Capítulo 6 presenta la arquitectura del ambiente VPPE. Se describe la función de cada módulo en VPPE a fin de traducir un programa visual de VPPL a lenguaje Java-MPI y ejecutarlo en un cluster.
- El Capítulo 7 muestra la evaluación de VPPL desde 2 enfoques: un enfoque cualitativo comparando sus características contra las de los lenguajes textuales y un enfoque de desempeño respecto al lenguaje de programación paralela basado en texto Java-MPI.
- El Capítulo 8 presenta las conclusiones y las sugerencias para el trabajo futuro.
- Los Acrónimos y el Glosario de este trabajo se encuentra al final del documento.

Capítulo 2

Estado del arte

En este capítulo se muestra una comparación de los lenguajes y ambientes visuales más representativos para el desarrollo de programas paralelos en memoria distribuida. Como primera parte, se presenta brevemente las desventajas que tenemos al desarrollar programas usando la programación textual tradicional en memoria distribuida, posteriormente se presentan las ventajas de los lenguajes visuales, en particular los que se conocen como flujos de trabajo y se estudian los más representativos en base a su *expresividad*, *aplicabilidad*, *nivel de interacción* y *portabilidad*.

2.1. Programación con paso de mensajes

El desarrollo de programas paralelos en arquitecturas de memoria distribuida tradicionalmente se ha realizado usando lenguajes o bibliotecas textuales. El paso de mensajes es el modelo por omisión en el desarrollo de programas en este tipo de arquitecturas. La programación usando el modelo de paso de mensajes ha sido dominada principalmente por el estándar MPI [Pac96] [Sta96] [GLS14]. MPI es una especificación que define operaciones textuales de paso de mensajes, la cual se incluye en los programas como una biblioteca. MPI proporciona un conjunto de funciones o rutinas de bajo nivel, las cuales se utilizan principalmente en lenguajes como C, Fortran o Java. En MPI cada proceso ejecuta el mismo programa sobre

su propio conjunto de datos (los procesos solo tienen acceso a sus propios datos). La comunicación de procesos se realiza mediante operaciones de envío y recepción de mensajes, las cuales deben ser definidas de forma explicita por el programador. Además, MPI proporciona un conjunto de operaciones o rutinas que permiten al programador:

- Crear grupos de procesos.
- Realizar comunicación colectiva entre grupos de procesos.
- Definir puntos de sincronización, entre otras operaciones.

La principal ventaja que proporciona MPI es la gran diversidad de programas que se pueden realizar debido a la flexibilidad de sus operaciones de bajo nivel que presenta. Sin embargo, el desarrollo de programas en MPI no es trivial. El desarrollador debe preocuparse de elementos como la identificación de los procesos que se deben comunicar, definir el tipo(s) de dato(s) y el dato(s) a transferir, identificar que tipo de comunicación se debe realizar (si es bloqueante o no), entre otros elementos. Estas consideraciones complican el desarrollo de programas de paso de mensajes en MPI, ya que decisiones incorrectas en cualquiera de estos puntos puede originar errores como resultados incorrectos o bien abrazos mortales.

2.2. Lenguajes visuales

Como se mencionó anteriormente, el desarrollo de programas paralelos tradicionalmente se realiza usando usando bibliotecas textuales. Sin embargo, una alternativa para simplificar o agilizar el desarrollo de programas es el uso de lenguajes visuales. Un diagrama visual proporciona una representación del problema y su solución más cercana a la representación mental del programador respecto a la programación textual [GP96] [Doz01].

Aplicar el uso de lenguajes visuales para el desarrollo de programas paralelos no es una idea nueva. La representación visual de un programa en paralelo se ha realizado principalmente mediante diagramas secuenciales (Figura 2.1a), grafos (Figura 2.1b) o flujos de trabajo (Figura 2.1c). El uso de diagramas secuenciales

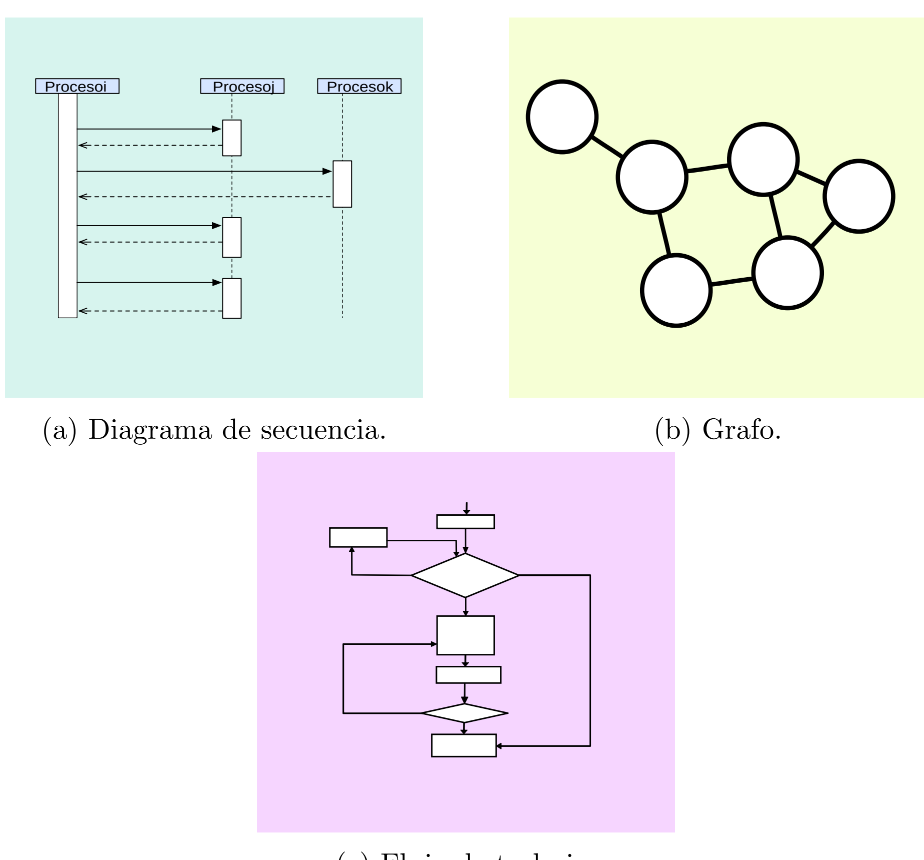
o grafos facilita la comprensión de la interacción y la sincronización de los procesos. Estos se han utilizado en herramientas/lenguajes como VISO [AMA97], Kaira [BBG11][BB11], ParaModel [CZJz+08], VisualGOP [CCCG04][CCG06][CCS03], VisualMPI [FNSW99], y Visper [NK99][SZ97][SZ02]. Sin embargo, entender el comportamiento del programa puede resultar complejo cuando hay varias aristas entre los nodos, más aún si éstas especifican comunicación y sincronización. Además, al utilizar diagramas y grafos secuenciales, la representación del flujo de ejecución en el tiempo es pobre o nula. Por ejemplo, en un grafo no necesariamente se especifica qué proceso inicia la ejecución de un programa o cuál proceso muestra el resultado final. Por el contrario, el uso de flujos de trabajo (ver Figura 2.1c) ofrece una idea más intuitiva acerca de cómo los procesos están trabajando e interactuando en el tiempo. En la siguiente sección se presentan a detalle cinco lenguajes/herramientas representativos que siguen el modelo de flujos de trabajo visuales.

2.3. Características de los flujos de trabajo

Como se mencionó en el capítulo anterior, un flujo de trabajo de un programa paralelo permite describir las ejecuciones simultáneas mediante un grafo que organiza el flujo de datos. Las herramientas visuales más representativas que usan flujos de trabajo son HeNCE [BD91], P-Grade [FK11][KDF96][KDFL99], HiPPO [Lee04], CODE [NB92], y Trapper [SSKF93][SSKF95]. Estos sistemas tienen diferentes tipos de iconos visuales, comportamientos y características que de acuerdo a [Doz01] permiten evaluar cada sistema con respecto a su: expresividad, aplicabilidad, nivel de interacción, y portabilidad.

2.3.1. Expresividad

La *expresividad* incluye todos los elementos visuales y las funciones proporcionadas por el lenguaje para ayudar a los programadores al desarrollo de sus algoritmos. Se distinguen al menos siete elementos visuales (iconos o estructuras) que sustentan la especificación del comportamiento de un programa paralelo



(a) Diagrama de secuencia.

(b) Grafo.

(c) Flujo de trabajo.

Figura 2.1: Tipos de diagramas visuales más comunes.

(véase la tabla 2.1): tarea secuencial (*SC*), bifurcación (*BF*), estructuras de repetición (*ER*), operaciones de entrada (*ENT*), operaciones de salida (*SAL*), patrones de procesamiento especial (*PPE*), y comunicación (*COM*).

Nombre	**Nemonico**	**Función**
Secuencial	SC	Tarea secuencial.
Bifurcación	BF	Ejecución simultánea de tareas.
Estructuras de repetición	ER	Ejecución de un conjunto de iconos siempre y cuando una condición dada sea verdadera.
Operaciones de entrada	ENT	Lectura de un conjunto de datos de entrada como variables o archivos.
Operaciones de salida	SAL	Mostrar o guardar el valor de un conjunto de variables o resultados.
Patrones de procesamiento especial	PPE	Organización de un conjunto de procesos para ejecutar patrones de computación paralela.
Comunicación	COM	Especificación de comunicación punto a punto o colectiva.

Tabla 2.1: Elementos visuales de la expresividad.

La mayoría de los entornos incluyen los siguientes elementos visuales:

- Un icono para representar una tarea secuencial (SC), una estructura para representar la bifurcación y con ello generar ramas simultáneas/concurrentes (BF).
- Una estructura de repetición (ER) para ejecutar más de una vez un conjunto de iconos.
- Al menos dos iconos para representar procesamiento por medio de patrones paralelos (PPE), por ejemplo Single Program Multiple Data (SPMD) y Pipeline.
- Un icono o una representación visual para la comunicación punto a punto y/o la operación colectiva de difusión (COM).

Sin embargo, en la mayoría de los entornos las variables en la comunicación (COM) tienen que ser expresadas en el texto usando nuevas palabras reservadas o símbolos textuales (por ejemplo *input* y *output*), diferenciando el tipo operación (por ejemplo, recibir o enviar) y especificando el nombre de las variables. En la Figura 2.2a se muestra un ejemplo de como se realiza la comunicación en CODE, se definen puertos de comunicación de entrada y salida (X y Y) y se leen y escriben datos sobre los mismos ($v = val(X)$ y $Y = v$). La Figura 2.2b representa como se reciben y envían datos en HeNCE, para lo cual se utilizan textualmente los símbolos $<$ y $>$ respectivamente. Asimismo, los entornos no permiten expresar visualmente operaciones colectivas como la dispersión de datos (*scatter*).

```
                    Puerto entrada
input_ports { X }(recepción)
output_ports { Y }Puerto salida
                  (envío)
reglasDeEntrada{
   disponible(X)=>v=val(X)
}
...
reglasDeSalida{
   true=>Y=v
}
```

(a) CODE

```
entrada
(recepción)
      <  int v=0;
      ...
      > int v;
salida
(envío)
```

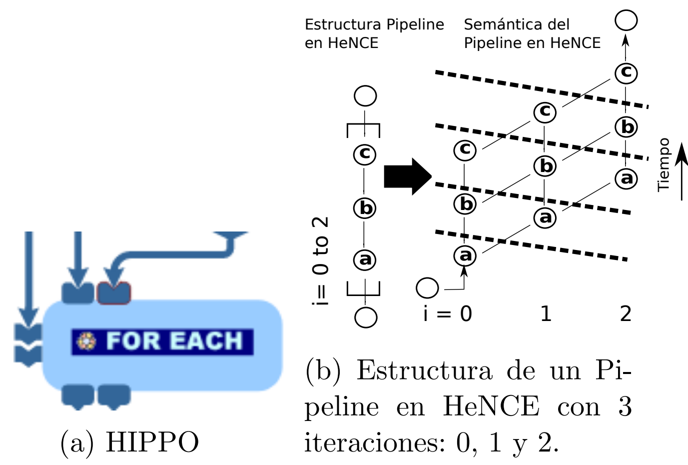
(b) HeNCE

Figura 2.2: Envío y recepción en CODE y HeNCE.

2.3.2. Aplicabilidad

La *aplicabilidad* se refiere al rango de programas que pueden ser implementados haciendo uso del lenguaje. Todos los entornos permiten la construcción de programas mediante patrones de cómputo paralelo a fin de incrementar el rango de problemas a resolver, y no estar limitados a las estructuras que se pueden formar con los flujos de trabajo básicos usando únicamente iconos de procesamiento secuencial y bifurcación. Las patrones más comunes en los entornos son SPMD y Multiple Program Multiple Data (MPMD), no obstante hay entornos (por ejemplo HeNCE) que implementan el patrón Pipeline. En la Figura 2.3a se muestra como en HiPPO se implementa el patrón SPMD mediante un icono llamado "FOR EACH", el rectángulo más grade representa el icono de operación, los rectángulos pequeños representan propiedades del icono para ingresar datos

o bien código; en la Figura 2.3b se muestra como se implemente el Pipeline en HeNCE, la parte izquierda representa la estructura utilizada en el lenguaje y la parte derecha la semántica al momento de la ejecución.



(a) HIPPO

(b) Estructura de un Pipeline en HeNCE con 3 iteraciones: 0, 1 y 2.

Si bien todos los entornos tienen al menos un patrón de procesamiento paralelo, el número y el tipo de éstos es muy limitado. Además, no se consideran patrones que permitan trabajar con programas que utilicen un conjunto de datos dinámicos, por ejemplo el patrón Maestro-Esclavo.

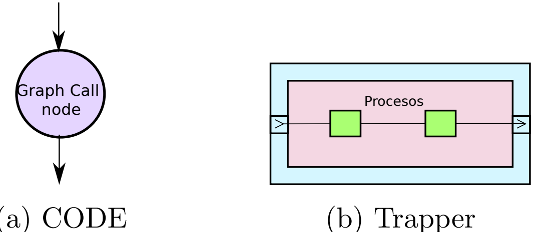
2.3.3. Nivel de interacción

El *nivel de interacción* es el número de interacciones que requiere realizar el desarrollador con la herramienta a fin de implementar su algoritmo. Hay cinco aspectos fundamentales o comportamientos a considerar (véase Tabla 2.2): interfaz de arrastrar y soltar (*drag and drop*) (*IAS*), encapsulación (*ECS*), re-ordenamiento de iconos (*ROI*), acercamiento y alejamiento del flujo de trabajo (*AAW*) y acceso a la nube (*ACN*).

Comportamiento	Abreviación	Descripción
Interfaz de arrastrar y soltar	IAS	Selecciona un icono "agarrándolo" y arrastrándolo a una ubicación diferente.
Encapsulamiento	ECS	Agrupa un conjunto de iconos como un nuevo icono.
Re-ordenamiento de iconos	ROI	Los iconos en un flujo de trabajo son automáticamente re-organizados.
Acercamiento y alejamiento del flujo de trabajo	AAW	Los iconos son ajustados en base a la resolución de la pantalla.
Acceso a la nube	ACN	La interacción con el ambiente desde cualquier lugar usando un navegador web.

Tabla 2.2: Comportamientos del sistema que apoyan el nivel de interacción.

La interfaz de arrastrar y soltar es un aspecto muy importante; HeNCE, P-Grade y HiPPO especifican que este comportamiento está incluido. En general, todos los entornos incluyen encapsulamiento (ECS). La Figura 2.4 muestra como se realiza este comportamiento en CODE y Trapper. En CODE se tiene un nodo (vértice) especial llamado *Graph Call Node* el cual permite invocar o ejecutar un grafo previamente definido. En Trapper los contenedores de un grafo se denominan $sub-sistemas$ y se representan con un rectángulo con doble marco.



(a) CODE (b) Trapper

Figura 2.4: Encapsulamiento en CODE y Trapper.

Aunque los entornos incluyen el encapsulamiento, ninguno de ellos incluye re-ordenamiento de iconos (ROI), alejamiento y acercamiento del flujo de trabajo (AAW) y acceso a la nube (ACN). Lo anterior dificulta que los diagramas sean claros de leer cuando hay un número grande iconos, tener una perspectiva global del programa cuando el diagrama (programa) crece tanto que no es posible

visualizarlo en pantalla y tener acceso al entorno requiriendo únicamente de un navegador web lo cual es una tendencia actual.

2.3.4. Portabilidad

La *portabilidad* define la característica de un software de ejecutarse en diferentes plataformas. Como se menciona en [Doz01], la pobre o nula portabilidad en la implementación de los entornos visuales es una de las principales razones de su poca distribución y/o popularidad. La portabilidad se evalúa considerando elementos como la interfaz de usuario (IU), la capa del acceso a datos portable ó *back-end* (CADP) y la nube (NB), ver Tabla 2.3. Debido a que la mayoría de los ambientes se desarrollaron en los años 90, sus interfaces se implementaron utilizando herramientas nativas y bibliotecas que no los hacen portables. Respecto al acceso por medio de la nube, estos entornos se desarrollaron con tecnologías antiguas por lo que no presentan esta funcionalidad.

Característica/Aspecto	**Abreviación**	**Descripción**
Interfaz de Usuario	IU	La habilidad de que la interfaz pueda ser transferida de una máquina/sistema a otra.
Capa portable de acceso a los datos	CADP	La habilidad de que la capa de acceso a datos pueda ser transferida de una maquina/sistema a otro.
Nube	NB	Implementación usando tecnologías web

Tabla 2.3: Características de portabilidad

Resumen

En este capítulo se ha presentado un estudio de lenguajes y ambientes visuales para construcción de programas paralelos en base al tipo de diagrama visual que presentan: grafos, diagramas de secuencia, y flujos de trabajo. Se ha considerado un conjunto de características que debe tomar en cuenta un lenguaje visual de acuerdo a los aspectos propuestos en [Doz01]: expresividad, aplicabilidad, nivel de interacción y portabilidad. Debido a que los diagramas mediante flujos de trabajo son más intuitivos para representar la concurrencia y el flujo de datos, nos hemos centrado principalmente en ellos realizando una comparación de un grupo de lenguajes y ambientes que usan este modelo: HeNCE, P-Grade, HiPPO, CODE, y Trapper. Dicha comparación se puede resumir en la Tabla 2.4. La primer columna indica la herramienta/lenguaje estudiado, de la segunda a la quinta columna se hace referencia a las características con las que cuenta dicha herramienta de acuerdo a los aspecto en [Doz01].

Herramienta/Leng.	**Expresividad**	**Aplicabilidad**	**Nivel de Interacción**	**Portabilidad**
HeNCE [BD91]	SC, BF, ER, PPE, COM	SPMD, MPMD y Pipeline (Tuberías)	IAS, ECS	No portable.
Editor de flujos de trabajo P-Grade [FK11][KDF96][KDFL99]	SC, BF, PPE, COM	SPMD y MPMD	IAS, ECS	Interfaz Portable
HiPPO [Lee04]	SC, BF, ER, ENT, SAL, PPE, COM	SPMD y MPMD	IAS, ECS	Interfaz Portable
CODE [NB92]	SC, BF, ER, ENT, PPE, COM	SPMD y MPMD	ECS	No portable
Trapper [SSKF93] [SSKF95]	SC, BF, ER, PPE, COM	SPMD y MPMD	ECS	No portable

Tabla 2.4: Comparativo de los lenguajes visuales que siguen el modelo de flujo de trabajo

En general, de dicha tabla podemos concluir los siguientes puntos:

1. La mayoría de los lenguajes no definen iconos visuales para representar la entrada y la salida de datos.
2. La única operación colectiva que representan visualmente es la difusión.
3. La mayoría de los lenguajes únicamente ofrecen los patrones de procesamiento paralelo el SPMD y MPMD. El ambiente/lenguaje que proporciona más patrones es HeNCE, el cual cuenta con SPMD, MPMD y procesamiento Pipeline.
4. El *nivel de interacción* y la *portabilidad* están limitadas debido a que estos lenguajes presentaron su mayor apogeo en los años 90's, cuando las interfaces gráficas y el web eran muy pobres.

Capítulo 3

Gramática de grafos

La definición de nuevos lenguajes gráficos o visuales se puede apoyar de mecanismos formales como ocurre con los lenguajes de programación textuales. En este capítulo se presenta una breve introducción a la Gramática de Grafos (GG), en particular al formalismo conocido como Gramática de Grafos mediante el Reemplazo de Hiperaristas (GGRH) [DKH97] [Roz97a]. Este formalismo puede ser utilizado para definir la gramática los lenguajes gráficos de programación paralela; en particular el lenguaje gráfico propuesto en este proyecto fue definido con el formalismo GGRH. Inicialmente se definen los conceptos fundamentales de esta teoría y posteriormente se define lo que es una GGRH.

3.1. Introducción y terminología

3.1.1. Hiperarista

Una *hiperarista* es un elemento (abstracto) representado por medio de un *cuadrado* y un conjunto de *tentáculos* (enlaces) numerados. Para cada hiperarista su *tipo* es igual al número de sus tentáculos. En el ejemplo de la Figura 3.1 se muestra una hiperarista con 3 tentáculos $\Rightarrow tipo(hiperarista) = 3$.

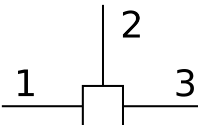


Figura 3.1: Hiperarista con 3 tentáculos.

3.1.2. Hipergrafo

Teniendo como base el concepto de una hiperarista se puede definir un hipergrafo de la siguiente forma. Sea C un conjunto de etiquetas, un *hipergrafo* H sobre C es una 4-tupla (V, E, lab, att) donde:

- V es un conjunto de nodos o vértices.
- E es un conjunto de hiperaristas tal que $V \cap E = \emptyset$.
- lab es una función que etiqueta cada hiperarista, $lab : E \to C$.
- att es un mapeo que contiene el conjunto de nodos que están asociados (conectados) a una *hiperarista* $e \in E$ y satisface $tipo(e) = |att(e)|$, $att : E \to V^*$.

Para ilustrar lo anterior, considere el hipergrafo de la Figura 3.2a sobre $C = \{A, B\}$. La especificación de cada uno de los componentes de éste hipergrafo se muestran en las Figuras 3.2b, 3.2c , 3.2d y 3.2e. La Figura 3.2b muestra los nodos del hipergrafo (representados por círculos). Los nodos de un hipergrafo se pueden clasificar en dos tipos: externos (los círculos con fondo negro y números en blanco) e internos. Los nodos externos permiten conectar a dos hipergrafos y los nodos internos son los típicos nodos en un grafo. El tipo de un hipergrafo H ($tipo(H)$) se define como número de nodos externos de H. En la Figura 3.2a el $tipo(Hipergrafo) = 1$. La Figura 3.2c muestra el componente E de las hiperaristas (y sus tentáculos de cada una). La Figura 3.2d describe la función de etiquetado lab. Esta función sirve para nombrar a cada hiperarista asignando dentro de su cuadrado una etiqueta; en el documento nos referiremos a una hiperarista nombrando a su etiqueta asociada. La Figura 3.2e muestra el mapeo att de

cada hiperarista del hipergrafo, el cual muestra los nodos que están conectados con los tentáculos de cada hiperarista.

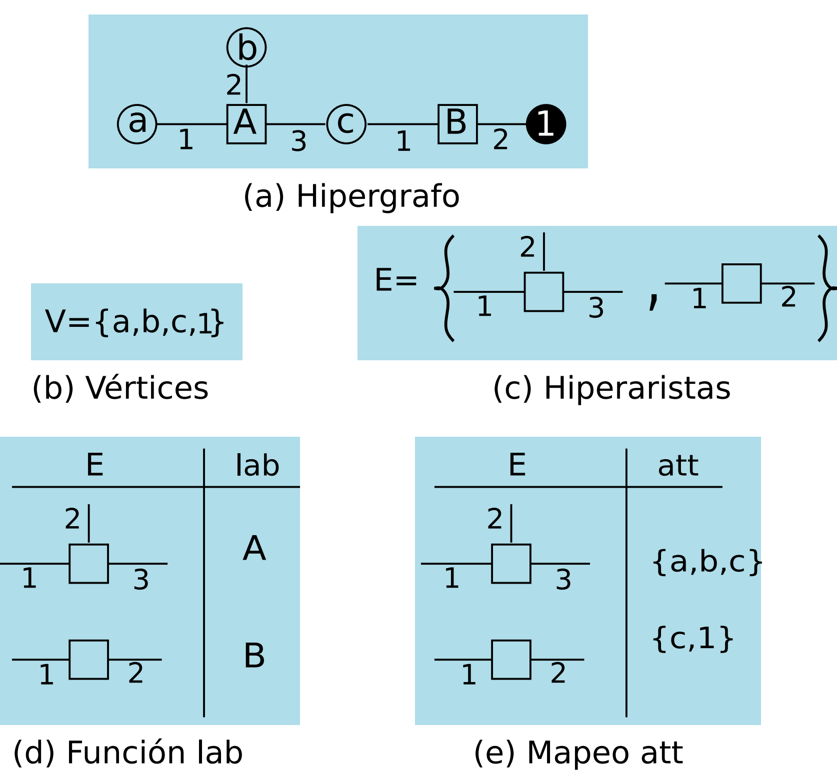
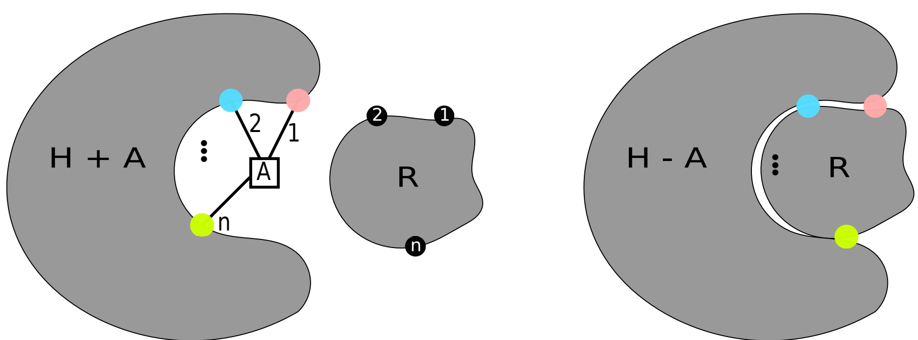


Figura 3.2: Elementos del hipergrafo H.

3.2. Reemplazo de hiperarista

Cada hiperarista de un hipergrafo puede ser reemplazada por otro hipergrafo. Lo anterior se ilustra en las Figuras 3.3a y 3.3b. El hipergrafo H de la Figura 3.3a tiene una hiperarista A con n tentáculos. La hiperarista A es reemplazada en el hipergrafo H por el hipergrafo R (notar que para que se de el reemplazo el número de tentáculos de la hiperarista A debe ser igual el número de nodos externos del hipergrafo R) manteniendo los nodos de H (coloreados con rojo, azul y verde) y desapareciendo los nodos externos de R (ver Figura 3.3b).

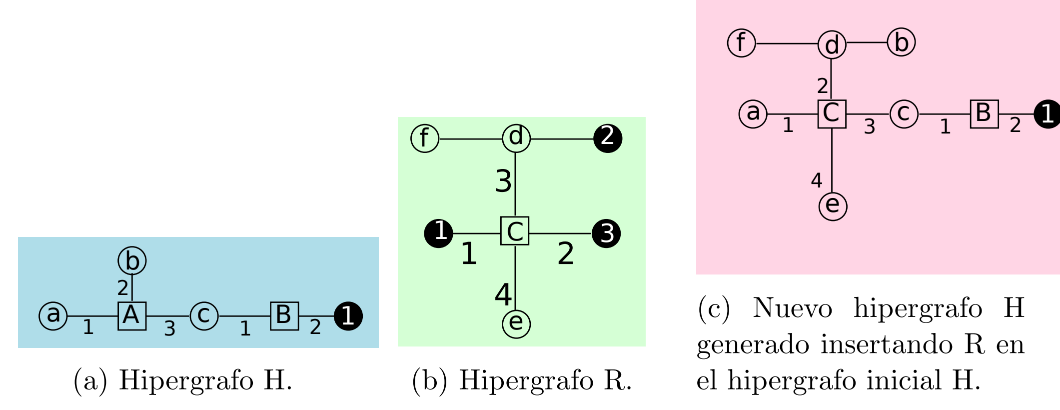


(a) Hipergrafo H con su hiperarista A e hipergrafo R.

(b) Reemplazo de A por el hipergrafo R.

Figura 3.3: Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R.

Un ejemplo particular del reemplazo de una hiperarista se muestra en la Figura 3.4. Se tiene un hipergrafo H que incluye una hiperarista A donde el $tipo(A) = 3$ (ver Figura 3.4a). Cada tentáculo de la hiperarista está conectado a un nodo del hipergrafo H. Además, se tiene un hipergrafo R con 3 nodos externos ($tipo(R) = 3$), ver Figura 3.4b. El hipergrafo R puede reemplazar la hiperarista A en H direccionando/orientando cada nodo externo i con el correspondiente nodo que se encuentra conectado al tentáculo i de la hiperarista A, lo que genera un nuevo hipergrafo (ver Figura 3.4c).



(a) Hipergrafo H.

(b) Hipergrafo R.

(c) Nuevo hipergrafo H generado insertando R en el hipergrafo inicial H.

Figura 3.4: Reemplazo de la hiperarista A en el hipergrafo H por el hipergrafo R.

3.3. Gramática de reemplazo de hiperaristas

Considerando los conceptos de hiperarista y de hipergrafo se puede definir una gramática de grafos mediante el reemplazo de hiperaristas. Esta gramática es muy similar a las gramáticas textuales, se tienen un conjunto de no terminales (representados por las etiquetas de las hiperaristas), terminales (representados por los nodos) y un conjunto de reglas de producción que permite identificar que hipergrafos pueden reemplazar una determinada hiperarista. La definición formal de una gramática de reemplazo de hiperaristas es la siguiente. Sea C un conjunto de etiquetas, una Grámatica de Grafos de Reemplazo de Hiperaristas (GGRH) sobre C es una tupla $\boldsymbol{G = (N,\Sigma,R,S)}$, donde:

- N, $\Sigma \subseteq C$ son conjuntos de etiquetas de no terminales (hiperaristas) y terminales (nodos internos) finitos y disjuntos ($N \bigcap \Sigma = \emptyset$)
- $S \epsilon N$ es el no terminal inicial.
- R es un conjunto de reglas de la forma $A ::= H$ con $A \in N$ y H es un hipergrafo tal que, $tipo(A) = tipo(H)$.

La Figura 3.5 muestra un ejemplo de GGRH con cuatro reglas. En este ejemplo $C = \{S, A, B, a, b, c, d\}$, $N = \{S, A, B\}$, $\Sigma = \{a, b, c, d\}$. La hiperarista inicial es S, las reglas de producción son *S::=Hipergrafo1*, *A::=Hipergrafo2*|*Hipergrafo3* (la cual representa dos reglas *A::=Hipergrafo2 y A::=Hipergrafo3*) y *B::=Hipergrafo4*.

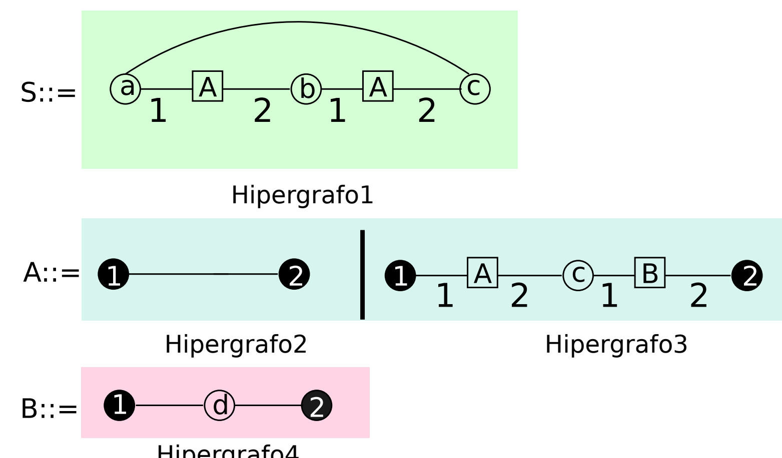


Figura 3.5: Gramática de reemplazo de hiperarista.

Un ejemplo de una derivación usando la gramática de la Figura 3.5 se muestra en la Figura 3.6. Como se observa en dicha figura, se inicia con la hiperarista S obteniendo el Hipergrafo1. Posteriormente se reemplaza la hiperarista A más a la izquierda por la regla Hipergrafo3. Después la nueva hiperarista A más a la izquierda es reemplazada por la regla Hipergrafo2, en seguida la hiperarista B se reemplaza aplicando la regla Hipergrafo4. Finalmente se deriva la última hiperarista A usando la regla Hipergrafo2 y se obtiene un grafo final.

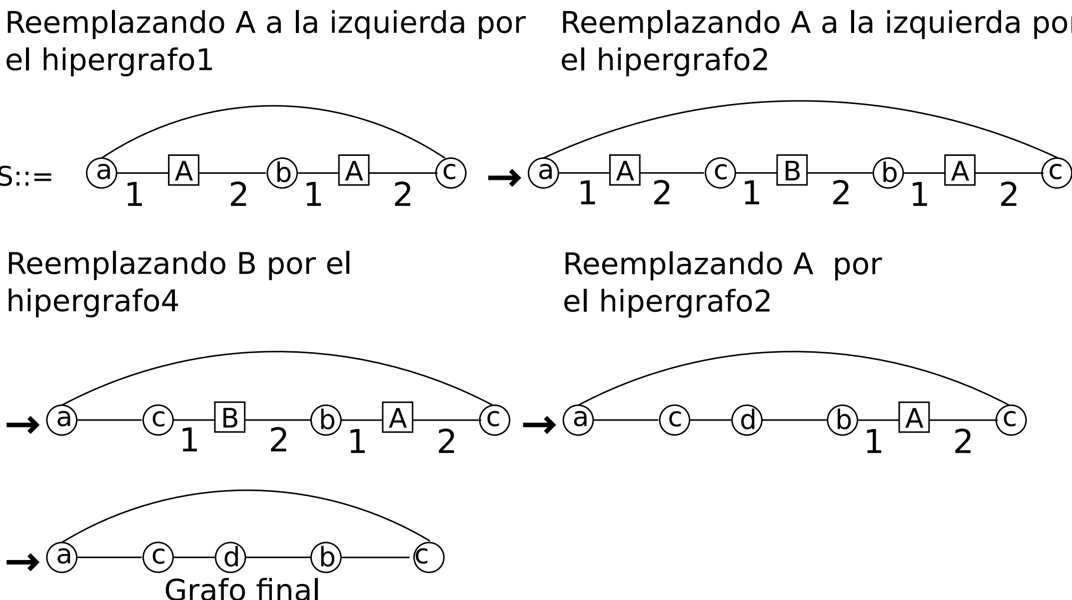


Figura 3.6: Ejemplo de una derivación en una gramática de reemplazo de hiperarista.

3.4. Definición de un lenguaje de programación paralela mediante gramática de grafos

Desde su aparición en los años 60 las gramáticas de grafos (GG) se han aplicado a diferentes áreas donde es más fácil o natural representar datos como diagramas o grafos que como cadenas de texto. Algunas de estas áreas son: especificación y desarrollo de software, reconocimiento de patrones, sistemas de bases de datos,

especificación de tipos de datos, biología, etc. En ciencias de la computación, la definición de un lenguaje visual paralelo mediante una gramática de grafos provee varias ventajas, entre las que podemos destacar:

1. Descripción visual de la concurrencia.
2. Aprendizaje rápido del lenguaje.
3. Descripción formal y visual del lenguaje.
4. Identificación de programas (diagramas) que no pertenecen al lenguaje.
5. Extensión de nuevas características al lenguaje.
6. Generación de programas mediante un desarrollo dirigido por sintaxis.

La *descripción visual de la concurrencia* permite identificar las tareas que se efectúan simultáneamente, lo que es complicado en los lenguajes textuales. Contar con un conjunto pequeño de iconos como representación de las operaciones del lenguaje permite un *aprendizaje rápido del lenguaje* ya que una imagen es más fácil de recordar respecto a una instrucción textual con sus parámetros. El uso de una gramática formal (al igual que en los lenguajes textuales) permite conocer las reglas que definen al lenguaje, por lo que utilizar una gramática de grafos permite una *descripción formal y visual del lenguaje* paralelo propuesto. El contar con una GG permite conocer si un programa no cumple con las reglas de su lenguaje, es decir, permite la *identificación de programas (diagramas) que no pertenecen al lenguaje*. Conocer las reglas de una gramática, permite en un futuro la *extensión* de alguna de ellas a fin de ampliar las operaciones o funcionalidades del lenguaje. Finalmente, el conocer las reglas de una gramática permite realizar la *generación de programas mediante un desarrollo dirigido por sintaxis* donde al programador únicamente se le permite realizar conexiones válidas entre elementos gráficos.

Considerando estas ventajas, en el siguiente capítulo se define formalmente el lenguaje propuesto VPPL siguiendo una GG, y en particular una de tipo GGRH. La GGRH de VPPL define un conjunto de reglas que separan la estructura general de un programa, las operaciones de entrada y salida, el procesamiento y la concurrencia. Proponer la gramática de VPPL mediante una GGRH permitirá

describir qué tipos de grafos pertenecen al lenguaje VPPL y conocer sus alcances y limitaciones.

Resumen

En este capítulo hemos presentado la teoría general de la gramática de grafos mediante el reemplazo de hiperaristas (GGRH). Este formalismo es muy similar al de las gramáticas textuales. Sin embargo, los lenguajes que describen las GGRH han sido utilizados para diagramas/lenguajes visuales. Utilizar una GGRH para describir un lenguaje visual provee de varias ventajas respecto a la visualización de la concurrencia y el desarrollo de programas válidos. En este trabajo de investigación se propone un GGRH para describir el lenguaje VPPL. En el siguiente capítulo se describen cada unos de los elementos de la gramática VPPL y su semántica.

Capítulo 4

Lenguaje VPPL

El desarrollo de programas paralelos no es una tarea trivial. Los desarrolladores deben considerar diferentes aspectos como son: el particionamiento del trabajo o tarea, la comunicación sincronización de procesos, un conocimiento profundo del problema a resolver, etc. A fin de agilizar y simplificar el desarrollo de programas paralelos en esta tesis se ha propuesto VPPL, un lenguaje visual que permite del desarrollo de programas paralelos mediante la composición de iconos. En este capítulo presentamos los iconos de VPPL, la semántica de cada uno ellos, y la gramática GGRH que representa su lenguaje.

4.1. Introducción

VPPL es un lenguaje cuyo diseño busca simplificar el desarrollo de programas paralelos representando visualmente la interacción de procesos. Este diseño considera diferentes desafíos que se presentan en los lenguajes visuales [Doz01]: Expresividad, Aplicabilidad, Nivel de interacción y Portabilidad.

Expresividad. La expresividad en VPPL se garantiza mediante un conjunto de estructuras e iconos que se clasifican en: estructura de flujo de trabajo y estructura cíclica, iconos de entrada/salida, de procesamiento y de comunicaciones. La composición y organización de estos iconos permite la representación de flujos de trabajo con varias ramas, las cuales a su vez son flujos de trabajo. La con-

currencia subyacente de un programa paralelo puede ser expresada visualmente mediante estos elementos.

Aplicabilidad. La composición de iconos de un flujo de trabajo permite la representación de procesamientos secuenciales, patrones de cómputo paralelo (por ejemplo, MPMD, SPMD, Pipeline, o Maestro-Esclavo), o una combinación de ambos. Lo anterior permite el desarrollo de una variedad de programas paralelos.

Nivel de interacción. El lenguaje visual de VPPL permite que los desarrolladores generen sus propios flujos de trabajo y patrones paralelos por medio de la composición de iconos o bien pueden utilizar los patrones ya implementados. Por ejemplo, un Pipeline puede ser implementado por una secuencia lineal de procesamientos secuenciales insertados en una estructura cíclica, o bien usando el patrón Pipeline proporcionado por VPPL. Además, VPPL extiende su nivel de interacción mediante el diseño de un ambiente de desarrollo propuesto para el propio lenguaje el cual se llama VPPE (Visual Parallel Programming Environment). VPPE permite la creación de flujos de trabajo mediante una interfaz Drag-and-Drop, la encapsulación (representación de un conjunto de iconos como uno solo, el cual puede ser insertado en otro flujo de trabajo), el re-ordenamiento de iconos (los iconos son automáticamente re-ordenados para proporcionarle al desarrollador una presentación más clara de su flujo de trabajo), alejamiento y acercamiento (*zoom*) de un flujo de trabajo (los iconos de un flujo de trabajo pueden ser ajustados en base a la resolución del monitor del desarrollador así como a el tamaño de la ventana del navegador), y acceso cloud (los desarrolladores pueden interactuar con el ambiente desde cualquier parte usando únicamente un navegador web.). En el capítulo 4 se explora a detalle el ambiente de desarrollo VPPE.

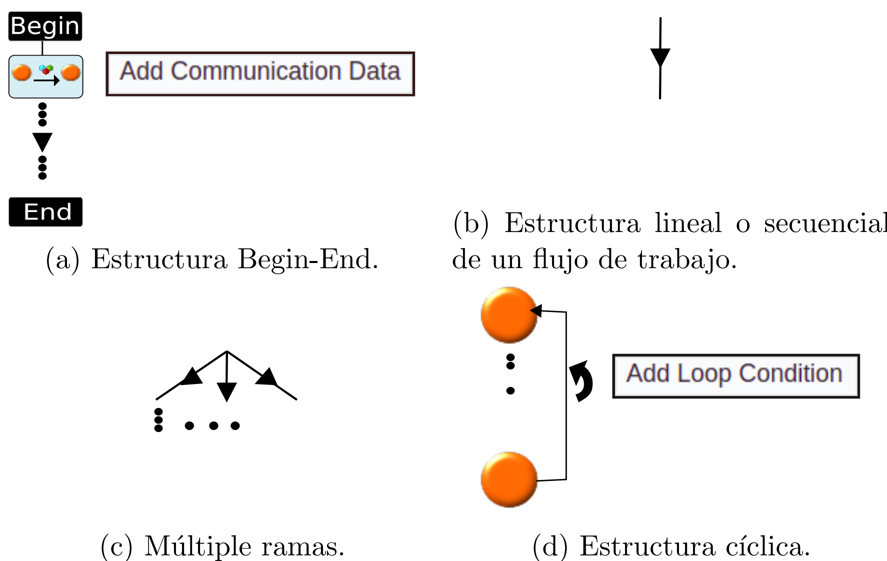
Portabilidad. La portabilidad de VPPL se orienta principalmente a su ambiente VPPE. VPPE usa para su interfaz (front-end) tecnologías web estándar, los desarrolladores no necesitan instalar software adicional para tener acceso al ambiente. Con respecto al back-end, VPPL actualmente utiliza el lenguaje de programación Java (y la biblioteca MPI) como lenguaje final en la traducción de un flujo de trabajo. Sin embargo, éste se pueden adaptar fácilmente para generar código para otros lenguajes de programación como son C++, Python, entre otros.

4.2. Estructuras visuales e iconos de VPPL

Los componentes de VPPL pueden ser divididos en cuatro grupos; estructuras de flujo de trabajo, iconos de entrada/salida (e/s), procesamiento y comunicaciones. Dichos componentes o iconos tienen un conjunto de parámetros por medio de los cuales se ingresa información de acuerdo al tipo de icono; por ejemplo, por medio de un parámetro se ingresa el código que ejecutarán los iconos de procesamiento y los iconos de comunicación tienen un parámetro para especificar el nombre de las variables que tienen los datos que se transmitirán. A continuación se describen las componentes e iconos de VPPL así como sus parámetros.

4.2.1. Estructuras de flujo de trabajo

Las estructuras que representan un flujo de trabajo se muestran en la Figura 4.1. La *estructura Begin-End* representa el inicio y el fin de un flujo de trabajo (Figura 4.1a). El icono en forma de rectángulo debajo del icono *Begin* tiene un parámetro llamado *Add Communication Data* para definir las variables que se requieren en la comunicación (envío o recepción) en el flujo de trabajo (Figura 4.1a, derecha). Dentro del cuerpo principal, la flecha punteada se puede cambiar a un flujo de trabajo con *estructura lineal* (Figura 4.1b) o bien, a un flujo de trabajo con *múltiples ramas* paralelas para especificar un procesamiento *(MPMD)* (Figura 4.1c), o una *estructura cíclica* (representada por un flecha curvada) para especificar que un conjunto de estructuras serán ejecutadas mientras una condición se cumpla (Figura 4.1d). Para este último icono, la condición se agrega por un parámetro llamado *Add Loop Condition* (Figura 4.1d, derecha)



(a) Estructura Begin-End.

(b) Estructura lineal o secuencial de un flujo de trabajo.

(c) Múltiple ramas.

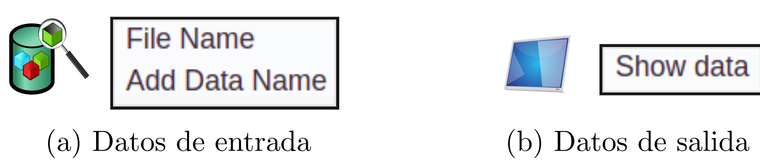
(d) Estructura cíclica.

Figura 4.1: Iconos para un flujo de trabajo en VPPL.

4.2.2. Iconos de entrada y salida

Los *datos de entrada* se representan mediante un icono en forma de cilindro (Figura 4.2a). Por medio de éste icono es posible realizar la lectura de un archivo cuyo nombre se ingresa por medio de un parámetro de su menú llamado *File Name* (4.2a, derecha); además, por medio de un parámetro llamado *Add Data Name* debe ser definido un conjunto de variables, sus nombres y tipos para almacenar los valores almacenados en el archivo.

La *salida* se representa mediante un icono en forma de pantalla (Figura 4.2b). Los valores de un conjunto de variables pueden ser escritos en un archivo o desplegados en pantalla por medio de un diálogo que se muestra a través del parámetro *Show Data*.



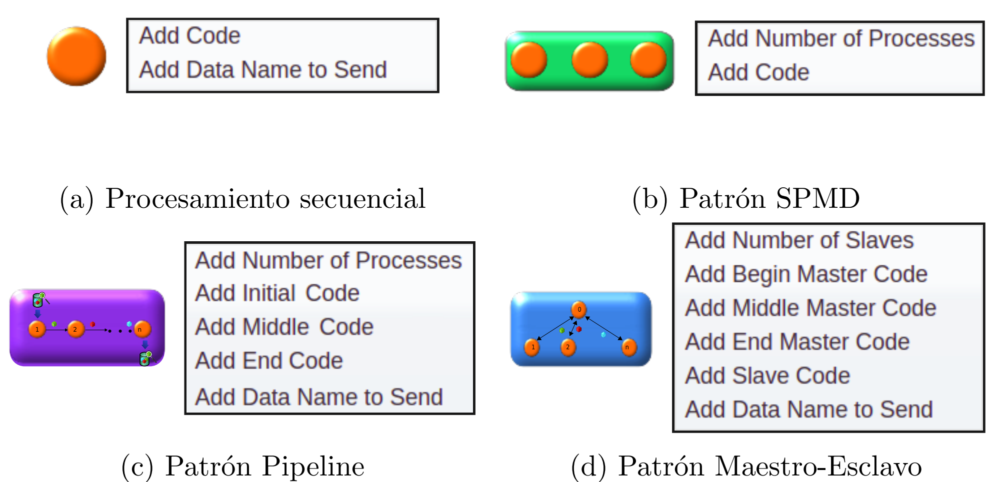
(a) Datos de entrada

(b) Datos de salida

Figura 4.2: Iconos de entrada y salida en VPPL.

4.2.3. Iconos de procesamiento

VPPL actualmente cuenta con cuatro iconos de procesamiento (ver Figura 4.3): El icono de *procesamiento secuencial* (Figura 4.3a) especifica un conjunto de instrucciones que serán ejecutadas secuencialmente (por un solo proceso), el icono del patrón SPMD (*Single Program Multiple Data*, ver Figura 4.3b) define un conjunto de procesos que ejecutan el mismo código de forma simultánea sobre diferentes datos. El icono *Pipeline* (ver Figura 4.3c) define un conjunto de procesos ordenados en un Pipeline donde ejecutan el código de acuerdo a su posición en el mismo (primero, medios, último). Finalmente, el icono del patrón *Maestro-Esclavo* (Figura 4.3d) define un proceso maestro y un conjunto de procesos esclavos para realizar un conjunto de tareas que pueden ser estáticas o se pueden generar dinámicamente. Cada que el maestro recibe una solicitud de trabajo de un esclavo le asigna una tarea. Cuando todas las tareas son hechas, el maestro recolecta el resultado global. Los iconos SPMD, Pipeline, y Maestro-Esclavo generan un paralelismo implícito, donde el número de procesos participantes es indicado por medio de un parámetro.



(a) Procesamiento secuencial

(b) Patrón SPMD

(c) Patrón Pipeline

(d) Patrón Maestro-Esclavo

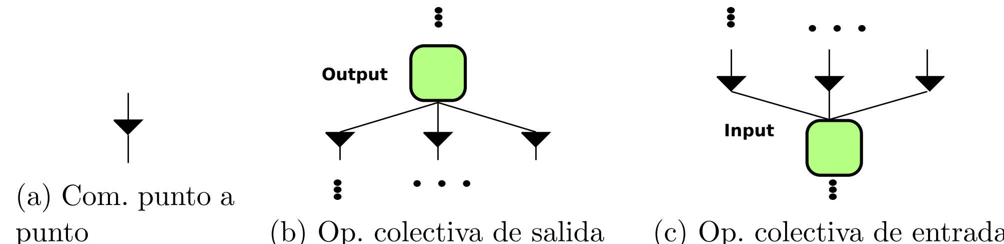
Figura 4.3: Iconos procesamiento en VPPL.

Los principales parámetros de los iconos de procesamiento se muestran en la misma Figura 4.3. El icono de procesamiento secuencial le permite al desarrollador agregar código secuencial (*Add Code*) y especificar el nombre de las variables

de comunicación (*Add Data Name to Send*) cuyos valores serán enviados al siguiente icono de procesamiento en el flujo lineal (estas variables deben haber sido declaradas dentro la estructura Begin-End). El patrón SPMD requiere el número de procesos SPMD en el patrón (*Add Number of Processes*) y el código secuencial para todos los procesos SPMD (*Add Code*). El patrón Pipeline maneja cinco principales parámetros: el número de procesos en el Pipeline (*Add Number of Processes*, mayor de 3); tres parámetros para ingreso de código especificando el comportamiento del primer proceso en el Pipeline (*Add Initial Code*), los procesos medios (*Add Middle Code*), y el últino proceso (*Add End Code*); y la última opción en el menú especifica el nombre de las variables de comunicación (*Add Data Name to Send*, con la misma función que en el icono del proceso secuencial). El patrón Maestro-Esclavo tiene seis parámetros principales, el número de procesos esclavos (*Add Number of Slaves*); cuatro parámetros para ingreso de código: el código de inicio del proceso Maestro (*Add Begin Master Code*) , el código que se ejecuta después de que un esclavo envía un resultado parcial (*Add Middle Master Code*), el código ejecutado por el maestro cuando todos los esclavos han terminado todas las tareas (*Add End Master Code*) y el cuarto código es el que desempeñará un esclavo al recibir una tarea (*Add Slave Code*); la última opción en el menú especifica el nombre de las variables de comunicación (*Add Data Name to Send*). Cabe mencionar que el Pipeline y el Maestro-Esclavo requieren parámetros adicionales para especificar el nombre de las variables a ser transferidas entre los procesos internos de cada patrón.

4.2.4. Iconos de comunicación

VPPL permite tres tipos comunicación: unidireccional punto a punto (Figura 4.4a), colectivas de salida (Figura 4.4b) y de entrada (Figura 4.4c).



(a) Com. punto a punto (b) Op. colectiva de salida (c) Op. colectiva de entrada

Figura 4.4: Tipos de comunicación en VPPL.

La estructura punto a punto define un enlace de comunicación entre dos procesos consecutivos en un flujo de trabajo. La operación colectiva de salida incluye tres tipos (Figura 4.5): Difusión (*Broadcast*, *B* en la Figura 4.5a), le permite a un proceso propagar los mismos datos a un conjunto de procesos; la dispersión (*Scatter*, *S* en la Figura 4.5a) le permite a un proceso distribuir diferentes datos entre un conjunto procesos (esta operación únicamente trabaja con variables de tipo array) y Difusión-Dispersión (*BS-cast*, *BS* en la Figura 4.5b) el cual le permite a un proceso tener ambas operaciones colectivas, una difusión seguida por una operación de dispersión.

Los iconos B y S tienen el parámetro *Add Data Name* para especificar el nombre de las variables de comunicación (Figura 4.5a, derecha). El icono BS-cast tiene dos parámetros, en el primero se especifica el nombre de las variables para la operación de difusión y en el segundo el nombre de las variables de comunicación para la operación de dispersión (Figura 4.5b, derecha). Los procesos receptores que participan en cualquiera de estas comunicaciones de salida automáticamente manejan los mismos nombres de variables en la recepción.

(a) Comunicación colectiva difusión (B) y dispersión (S). (b) Comunicación colectiva difusión-dispersión (BS).

Figura 4.5: Iconos de comunicación de salida: difusión, dispersión y difusión-dispersión.

La Figura 4.6 muestra un ejemplo de una operación S para distribuir un arreglo a que contiene seis elementos. Después de realizar la operación de dispersión,

cada uno de los procesos 1-3 recibe un arreglo a el cual es re-dimensionado a dos elementos. Para la operación B las variables recibidas mantienen el mismo nombre y el mismo tamaño para cada uno de los receptores.

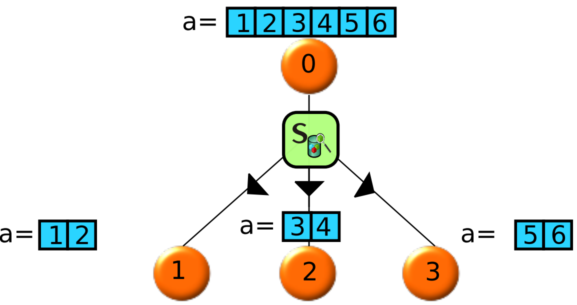


Figura 4.6: Parte de un flujo de trabajo representando una operación de dispersión.

(a) Comunicación colectiva recolección regular

(b) Comunicación colectiva recolección irregular

Figura 4.7: Iconos de comunicación de entrada: recolección regular (G_R) e irregular G_I

La operación colectiva de entrada (recolección o *gather*) se utiliza para especificar que un proceso secuencial recolecta datos de un conjunto de procesos. Esta puede ser de dos tipos (Figura 4.7): Recolección regular (G^R) e irregular (G^I). La operación G^R (Figura 4.7a) hace referencia a una instrucción de recolección donde un proceso obtiene diferentes valores de diferentes procesos, únicamente trabaja con arreglos. La Figura 4.8a muestra un ejemplo del G^R donde los procesos 1-3 tienen un arreglo local con el mismo nombre a con dos elementos. Después de realizar el G^R, el arreglo a utilizado en la recepción por el proceso 4 es automáticamente re-dimensionado a seis elementos.

La operación G^I (Figura 4.7b) representa una recolección similar a su versión regular; sin embargo, en esta operación es posible utilizar diferentes tipos de datos. La Figura 4.8b muestra un ejemplo de G^I donde el proceso 4 recibe un

entero del proceso 1, un arreglo de dos elementos del proceso 2 y un arreglo de 3 elementos del proceso 3.

El icono de recolección regular tiene el parámetro *Add Data Name* para especificar el nombre de las variables de comunicación (Figura 4.7a, derecha). Para el icono de recolección irregular, el nombre de las variables de comunicación son especificados en el último proceso de cada rama del patrón MPMD.

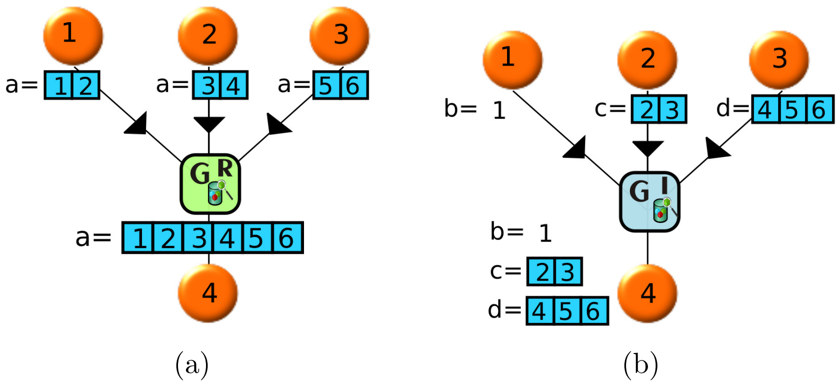


Figura 4.8: Parte de un flujo de trabajo de una recolección irregular (a) y parte de un flujo de trabajo con tres variables de una recolección irregular.

Debido a que los tamaños de las variables transferidas (buffers) pueden cambiar durante la ejecución, VPPL controla transparentemente el redimensionamiento de los buffers de recepción de acuerdo al tipo de comunicación. Internamente lo anterior se realiza de la siguiente forma: primero los emisores VPPL calculan el tamaño del buffer a enviar, y envían este valor (entero) a los receptores; los receptores, en turno, crean el buffer en base al entero (tamaño) recibido. En un segundo mensaje se envía el contenido del buffer de los emisores a los receptores, asegurando que la cantidad de datos recibidos es igual al tamaño del buffer.

Por cada mensaje que realiza un programa hay otro mensaje que lleva el tamaño del mensaje respectivamente. Particularmente, para programas de granularidad fina donde se tiene que realizar un gran número de comunicaciones, este protocolo podría afectar ligeramente el desempeño. Sin embargo, aunque esta desventaja puede aparecer, hay más ventajas que respaldan la programación en VPPE, tales como:

- VPPL evita que los desarrolladores especifiquen explícitamente el tamaño de las variables a enviar/recibir.

- Hay un uso eficiente de la memoria en las comunicaciones; únicamente la memoria requerida.

- No hay limitaciones con respecto a considerar un tamaño máximo de mensaje durante el desarrollo y ejecución de un programa (por su puesto, siempre y cuando se tenga memoria disponible).

- El comportamiento correcto de funciones u operaciones (utilizando el redimensionamiento de arreglos recibidos) se mantiene; por ejemplo, es posible usar atributos o estructuras de control de Java tales como "length" o "foreach".

4.3. Gramática del lenguaje VPPL

La definición de un lenguaje gráfico de programación paralela debe ser soportada por mecanismos formales, como es el caso de los lenguajes textuales donde una gramática libre de contexto permite analizar y traducir un programa a código máquina. Como se muestra en [QFRABC+13], un lenguaje gráfico puede ser definido empleando el formalismo GGRH [Hab92] a[Roz97b], el cual se estudió en el capitulo anterior. Dicho formalismo ha sido usado para definir el lenguaje propuesto en esta tesis: VPPL.

4.3.1. Elementos de la gramática de VPPL

La gramática de VPPL se define como la tupla e (N, Σ, R, S) donde:

- N representa el conjunto de hiperaristas (no terminales) del lenguaje; las cuales se muestran como palabras de color azul encerradas en un rectángulo. N = {Program, Body, Computation, Parallel, SpecializedPattern, GeneralPattern, GeneralPatternList, Pattern, MPMD, Communication, Gather, Input, y Output.}. Se considera el tipo(H)=2, $\forall$ H $\in N$.

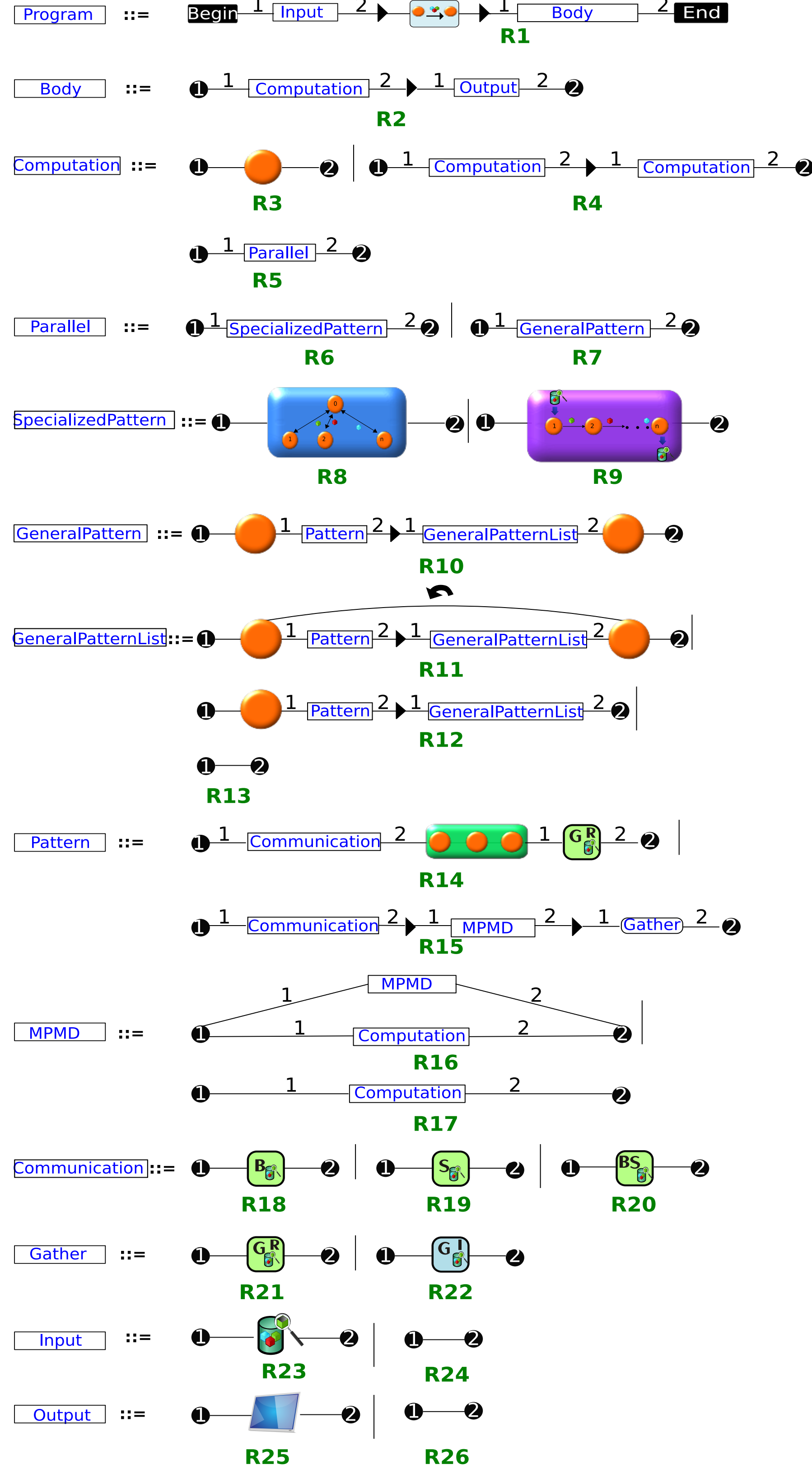


Figura 4.9: Gramática del lenguaje VPPL.

- Σ representa los iconos coloreados, las líneas, y la punta de flecha en medio de las líneas (nodos internos); los círculos negros numerados con 1 y 2 corresponden a los nodos externos.

- El conjunto de reglas de VPPL se muestran en la Figura 4.9. Las reglas son identificadas por la etiqueta Ri, donde i es el número de regla. Cada regla tiene dos extremos, en el extremo izquierdo del símbolo ::= hay siempre una hiperarista de tipo=2 y en el lado derecho un hipergrafo. El hipergrafo del lado derecho puede ser un hipergrafo terminal (sin grafo) o no.

- S denota el símbolo no terminal inicial que debe ser utilizado para iniciar la generación el programa gráfico; en esta gramática $S = Program$.

El conjunto de hipergrafos generados por la gramática VPPL es el conjunto de flujos de trabajo que pueden ser generados. En los siguientes párrafos se describe el comportamiento de cada regla en VPPL.

4.3.2. Reglas de producción

La regla *Program* (R1) permite la generación de un flujo de trabajo VPPL, definiendo los limites/extremos del mismo por los nodos *begin* y *end*. Una flecha negra entre dos nodos puede representar comunicación o flujo de ejecución. El icono representado con un rectángulo azul, y entre la hiperarista $Input$ y *body* define las variables utilizadas en las comunicaciones.

La regla *Body* (R2) permite la generación de los iconos de procesamiento que forman el flujo de trabajo, y termina con un icono definido en la regla $Output$.

La regla *Computation* (R3, R4 y R5) permite especificar una secuencia de varias tareas de cómputo, cada una de ellas desempeñando una ejecución secuencial o paralela.

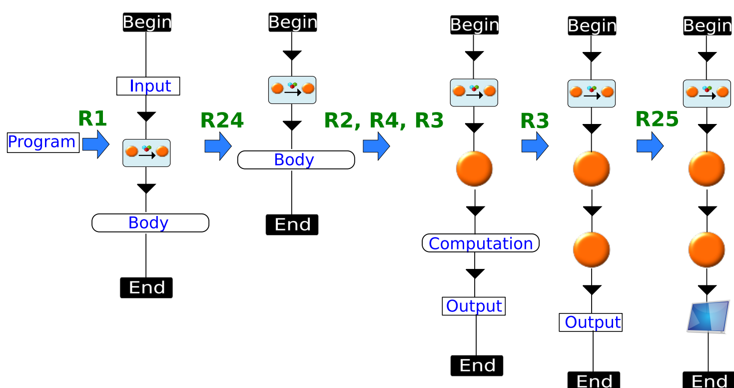


Figura 4.10: Derivación simple de un flujo de trabajo.

La regla *Parallel* (R6 y R7) permite la especificación de patrones de computo paralelo (que consideran un conjunto de procesos ejecutándose en paralelo). Los patrones pueden ser de dos tipos: especializados (R6) o patrones generales (R7). La Figura 4.10 muestra una derivación de un flujo de trabajo lineal con dos procesos secuenciales usando esta regla. El orden de aplicación de las reglas fueron: R1 → R24 → R2 → R4 → R3 → R3 y R25.

La regla *Specialized Pattern* define los patrones especializados. La versión actual de VPPL incluye dos patrones especializados: Maestro-Esclavo y Pipeline.

La regla *General Pattern* permite generar una secuencia de tareas paralelas, delimitadas por dos iconos naranjas (tareas secuenciales). El primer icono secuencial se utiliza para realizar pre-cálculos que pueden ser requeridos antes que se ejecuten el conjunto de procesos en paralelo. El segundo icono es utilizado para recolectar y procesar los resultados generados por uno o más procesos.

La regla *General Pattern List* permiten la generación de una lista de iconos de patrones de cómputo paralelo. Por medio de esta regla es posible definir ciclos, donde un procesamiento paralelo puede ser repetido mientras una condición sea verdadera.

La regla *Pattern* especifica un cómputo paralelo mediante un patrón general. Actualmente hay dos patrones generales: SPMD y MPMD; el patrón SPMD es especificado directamente usando su icono correspondiente, mientras que el patrón MPMD se define mediante la regla MPMD. Antes de la ejecución de estos patrones se especifica un tipo de comunicación para propagar o dispersar datos. Cuando los patrones SPMD o MPMD terminan, hay una tarea secuencial que recolecta

los diferentes resultados del conjunto de procesos. En las Figura 4.11 se presenta una derivación para generar un patrón SPMD. Las reglas aplicadas son R1 → R23 → R2 → R5 → R7 → R10 → R13 → R14 → R18 y R26.

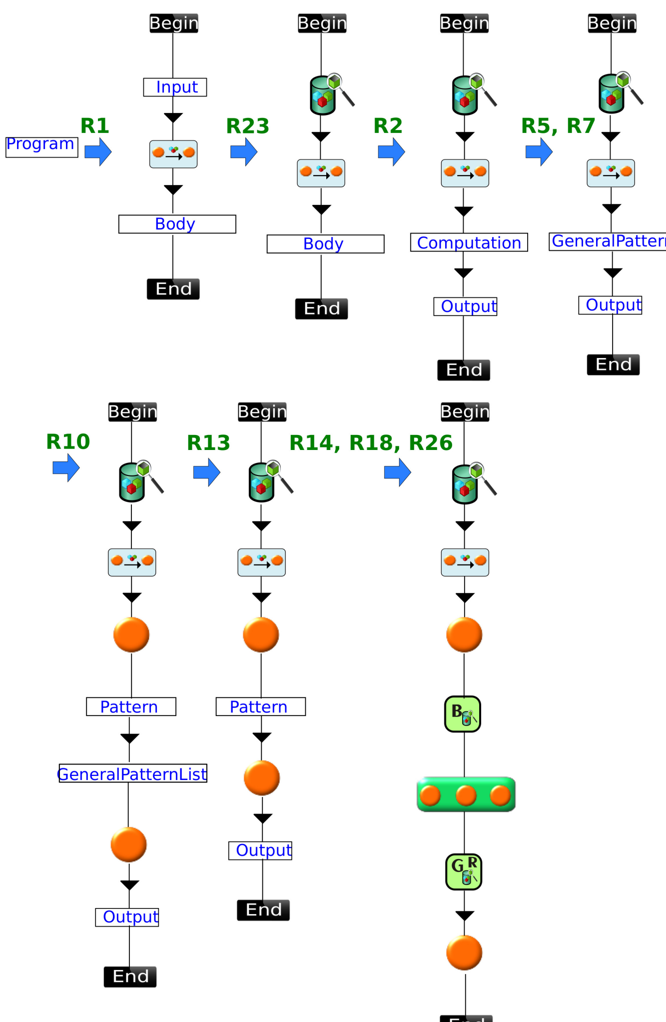


Figura 4.11: Ejemplo de la derivación de un flujo de trabajo SPMD.

La regla *MPMD* permite la especificación de uno o más flujos de trabajo (ramas) ejecutados en paralelo. En la Figura 4.12 se muestra un ejemplo generando un programa MPMD. Las reglas aplicadas para generar este flujo de trabajo fueron R1 → R23 → R2 → R5 → R7 → R10 → R13 → R15 → R16 → R17 → R3 → R3 → R3 → R18 → R22 → y finalmente R25.

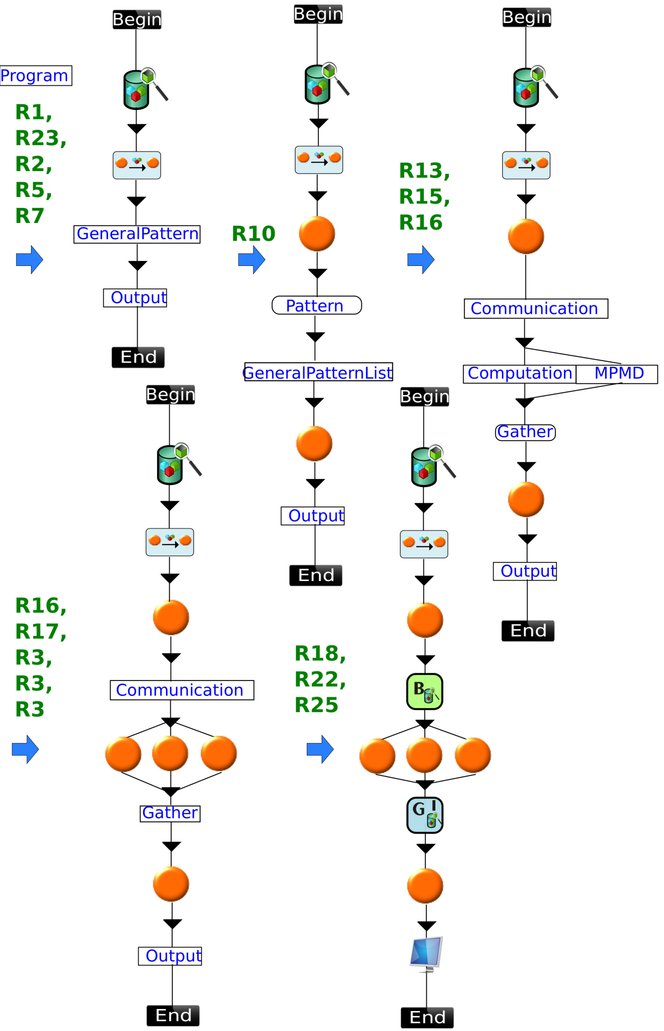


Figura 4.12: Derivación un flujo de trabajo MPMD.

Cabe mencionar que la regla *MPMD* permite definir ramas paralelas anidadas, como se muestra en la Figura 4.13, en este ejemplo la hiperarista *Computation* es remplazada por la regla *Parallel*, entonces mediante la aplicación de R7, R10, R11 o R12, y R15 se puede generar otra hiperarista *MPMD* (o SPMD si se requiere). Además, ambos patrones, SPMD o MPMD, se pueden definir dentro de un ciclo por la regla R11.

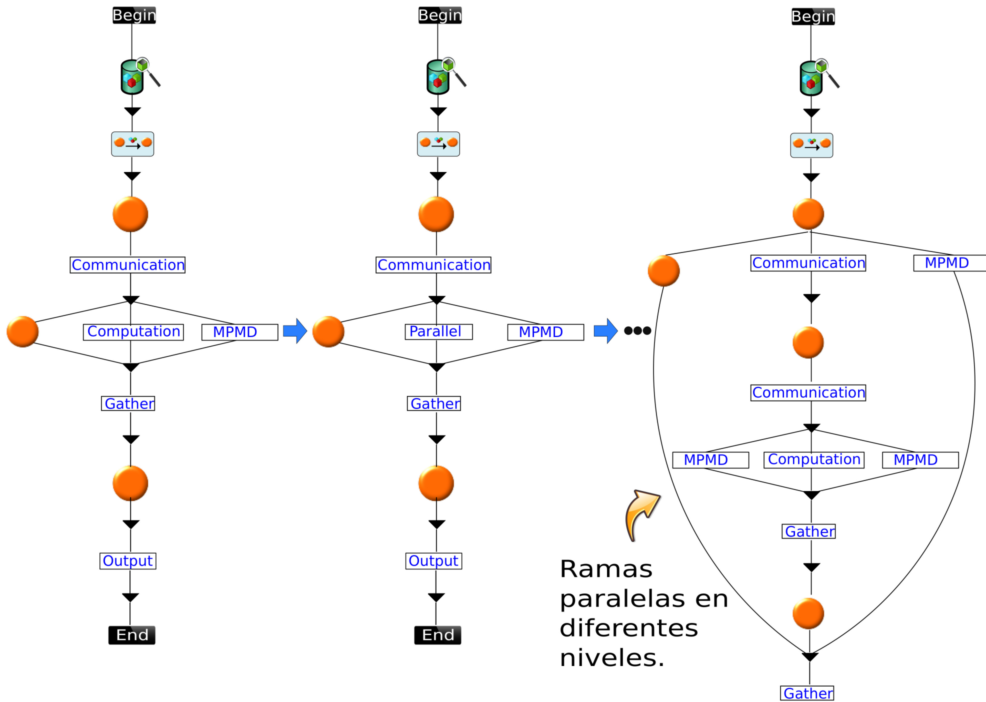


Figura 4.13: Ejemplo de un flujo de trabajo anidado.

La Figura 4.14 muestra un ejemplo donde el patrón SPMD se repite hasta que una condición sea falsa.

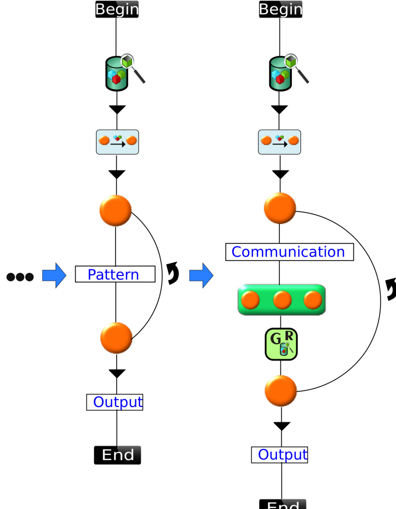


Figura 4.14: Flujo de trabajo con un icono SPMD dentro de un ciclo.

Las regla *Communication* permite la definición de diferentes tipos de comunicaciones colectivas de salida que se realizan antes de un patrón paralelo. Consideramos tres tipos de comunicación difusión (B), dispersión (S), y difusión-dispersión (BS). Los tipos de datos considerados para estas comunicaciones son *int*, $float$, *double*, *char* y arreglos de una o dos dimensiones.

Las reglas *Gather* permiten la especificación de dos tipos de recolección: *Regular* e $Irregular$. El primer caso se utiliza cuando los datos recolectados tienen el mismo tipo, en caso contrario se utiliza el irregular.

La regla *Input* permite la definición de variables con valores iniciales que pueden ser leídos de un archivo.

La última regla *Output* ofrece la posibilidad de utilizar un icono que permita desplegar los resultados finales del procesamiento a los desarrolladores.

Para ilustrar el desarrollo de programas paralelos usando VPPL, en la siguiente sección se presentan cinco ejemplos representativos de flujos de trabajo VPPL.

Resumen

En este capítulo hemos presentado el lenguaje VPPL. Por medio de VPPL se pueden desarrollar programas paralelos mediante el modelo de flujo de trabajo. VPPL presenta un pequeño conjunto de estructuras e iconos que se clasifican en flujo de trabajo, entrada y salida, procesamiento y comunicación. Por medio de la composición de estas estructuras e iconos se expresan elementos fundamentales del paralelismo como son la concurrencia, sincronización y comunicación de procesos.

Respecto a los iconos de procesamiento, se cuenta con un icono que representa un proceso secuencial y otros que representan patrones paralelos. Éstos últimos definen un conjunto de procesos que ya cuentan con un protocolo de comunicación implementado, por lo que el desarrollador únicamente tiene que ingresar bloques de códigos secuenciales y el nombres de las variables involucradas en la comunicación, el paralelismo ya es subyacente.

Las reglas de cómo se realiza la composición (programas) de las estructuras e iconos de VPPL se define mediante el formalismo *Gramática de Grafos mediante Reemplazo de hiperaristas* (GGRH). Este formalismo es simple y tiene una semejanza a las gramáticas libres de contexto para lenguajes textuales.

Capítulo 5

Programación en VPPL

Existe una variedad de programas que se pueden desarrollar mediante VPPL, un conjunto de ellos (presentados en este capítulo) son los siguientes:

1. Secuencial.
2. Multiplicación de matrices usando el procesamiento SPMD.
3. Calcular la inversa de una matriz usando procesamiento MPMD.
4. Procesamiento de un lote de imágenes empleando el modelo Maestro-Esclavo.
5. Un algoritmo genético usando SPMD.

Cada flujo de trabajo (excepto el primero) contiene etiquetas de la forma *Paso* i, con i un entero que describen el flujo de ejecución que sigue el flujo de trabajo al ejecutarse.

5.1. Programa secuencial

En la Figura 5.1 se muestra el flujo de trabajo que representa a un programa secuencial. Como se muestra, el procesamiento en el flujo de trabajo radica en el icono de procesamiento secuencial. En este programa no se requiere que los desarrolladores escriban el método *main* como en los lenguajes de programación

tradicionales (Java, C, C++, etc.). Únicamente se requiere que indique el código para el icono.

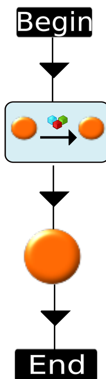


Figura 5.1: Flujo de trabajo secuencial

5.2. Multiplicación de matrices usando el modelo SPMD

La multiplicación de matrices es una operación binaria que toma dos matrices (A y B), y genera otra matriz (C). El algoritmo secuencial más utilizado para esta operación realiza la multiplicación renglón por columna para generar cada entrada de la matriz C, esto es, la entrada (i, j) se obtiene multiplicando el renglón i de A por la columna j de B y sumando los resultados.

Un algoritmo paralelo para la multiplicación de matrices particiona la matriz A en bloques de renglones, de manera que cada proceso mantiene un bloque. Cada bloque es multiplicado por la matriz B usando el algoritmo secuencial, de manera que cada proceso genera una parte de la matriz C (una solución parcial).

La Figura 5.2 muestra el flujo de trabajo para la multiplicación de matrices (lado izquierdo) y su paralelismo subyacente (lado derecho). Las matrices A (de tamaño $m \times n$), B (de tamaño $n \times p$) y C (de tamaño $m \times p$) son declaradas usando el icono de *comunicaciones* (paso 1) para ser consideradas en las comunicaciones. Por medio de un parámetro se ingresa el código para inicializar A y B (paso 2). A continuación, el icono BS-cast se utiliza para *difundir* y *dispersar* la matriz B y A respectivamente (paso 3). Como resultado, las filas de A se distribuyen de

forma automática y de manera equitativa entre los procesos SPMD (k filas por proceso). El código ingresado como parámetro al icono SPMD es ejecutado por todos los procesos. Entonces, cada proceso realiza una multiplicación secuencial de la matriz A local (de tamaño $k \times n$) y B (de tamaño $n \times p$), obteniendo una matriz local C de tamaño $k \times p$ (paso 4). El icono G^R indica la recolección de cada matriz C local de los procesos SPMD (paso 5). Las matrices C locales son concatenadas para construir la matriz C final de tamaño $m \times p$, la cual se envía a un proceso secuencial (paso 6). Por ultimo, la matriz C obtenida se muestra (paso 7) y almacena en un archivo de salida.

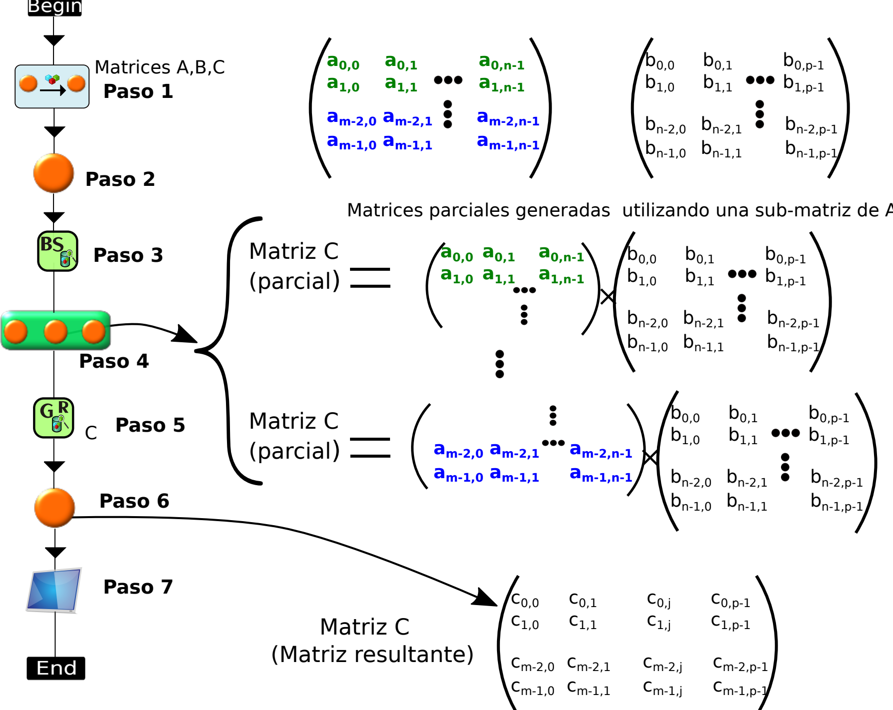


Figura 5.2: Flujo de trabajo VPPL para la multiplicación de matrices (lado izquierdo) y su paralelismo subyacente (lado derecho).

5.3. Inversa de una matriz

La inversa de una matriz A se puede determinar por la fórmula:

$$A^{-1} = \frac{1}{|A|} \times (A_{adj})^t$$

donde, $|A|$ es el determinante de A y $(A_{adj})^t$ es la matriz transpuesta de la adjunta de A. Podemos observar que los dos términos necesarios para encontrar A^{-1} se pueden calcular de forma independiente. La Figura 5.3 muestra el flujo de trabajo VPPL propuesto y su paralelismo subyacente. Primero se declara la matriz A de este flujo de trabajo usando el icono de *comunicaciones* (paso 1). A continuación, un proceso secuencial inicializa A (paso 2). Posteriormente, la matriz A se difunde a dos procesos que siguen el modelo MPMD (paso 3). Un proceso calcula $\frac{1}{|A|}$, mientras que el otro calcula simultáneamente $(A_{adj})^t$ (paso 4). A continuación, por medio del un icono G^I se recolectan los resultados parciales (paso 5), y se envían a un proceso secuencial (paso 6) que se encarga de llevar a cabo la multiplicación de los resultados parciales recolectados para obtener la matriz inversa.

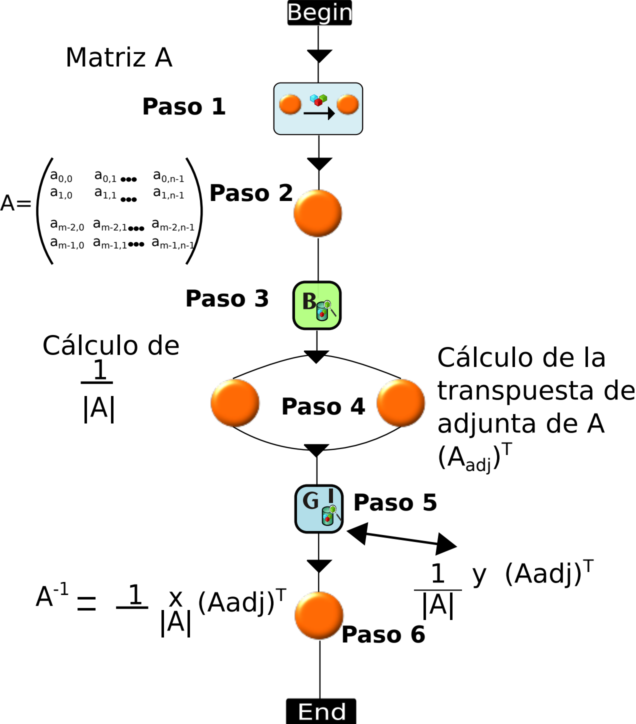


Figura 5.3: Flujo de trabajo VPPL para calcular la inversa de una matriz.

5.4. Procesamiento de un lote de imágenes usando el modelo Maestro-Esclavo.

El balance de carga es una alternativa conveniente para procesar una gran cantidad de imágenes que requieren diferentes tiempos de procesamiento. La Figura 5.4 muestra un flujo de trabajo VPPL para procesar un lote de imágenes en paralelo utilizando el modelo Maestro-Esclavo. En esta solución, se declara un contador mediante el icono *comunicaciones* (Paso 1). En el Paso 2, el contador se inicializa con el total de imágenes a procesar en el código de inicialización del Maestro. Por cada petición de trabajo de un Esclavo, se envía el valor del contador para posteriormente decrementarlo. Cuando un Esclavo recibe este valor entero del contador, el cual representa el número de una imagen, realiza el procesamiento correspondiente y almacena la imagen resultante en la carpeta personal del desarrollador, entonces se envía una nueva petición al Maestro para obtener otro número de imagen para procesarla; en este caso, no se requiere enviar un resultado parcial al Maestro. El Maestro termina a los Esclavos cuando el contador es 0. Cuando todas las tareas se llevan a cabo, se obtiene un nuevo lote de imágenes.

Cabe mencionar que este problema podría también haber sido resuelto mediante el uso de un modelo SPMD; sin embargo, el enfoque SPMD utiliza una partición estática de tareas y no tienen en cuenta un balance de carga, el cual es necesario cuando el procesamiento de cada imagen es diferente ya que algunos procesadores podrían no tener nada para procesar mientras que otros pueden estar muy sobrecargados.

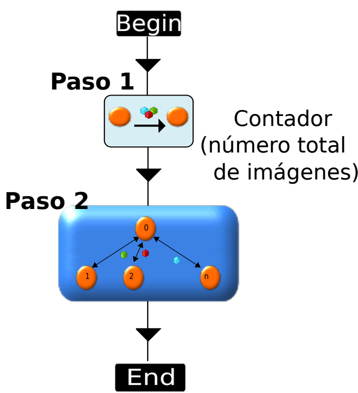


Figura 5.4: Flujo de trabajo VPPL para el procesamiento de imágenes.

5.5. Algoritmo genético iterativo usando SPMD

Los algoritmos genéticos son métodos eficientes de búsqueda basados en los principios de la selección natural y la genética [SGK05]. Se han aplicado con éxito para encontrar soluciones óptimas a problemas en los negocios, la ingeniería y la ciencia. Cuando los algoritmos genéticos utilizan grandes poblaciones o consideran una función aptitud compleja que requiere grandes tiempos de procesamiento, el computo paralelo podría ser una alternativa adecuada para reducir los tiempos de ejecución. La Figura 5.5 muestra un flujo de trabajo para implementar un algoritmo genético. En este ejemplo se usa el modelo SPMD dentro de un ciclo. Además, se considera que se conoce la organización o estructura que debe tener la solución óptima. La idea principal de este enfoque es distribuir la población global en un conjunto de sub-poblaciones procesadas por un conjunto de procesos SPMD. Primero, la variable utilizada para almacenar el mejor cromosoma se declara utilizando un icono *comunicaciones* (paso 1); esta variable está vacía al principio. Posteriormente, un proceso secuencial difunde el mejor individuo entre los procesos SPMD (paso 2 y 3). Los procesos SPMD ejecutan el código que se les ingreso como parámetro, en cual consiste en lo siguiente: cuando un proceso SPMD recibe un cromosoma vacío (sólo en la primera iteración), genera aleatoriamente su sub-población inicial (paso 4). Los procesos SPMD aplican los operadores genéticos (mutación, selección y cruce) en su sub-población e identifican su mejor cromosoma local. Posteriormente, un proceso secuencial recolecta

todos los mejores cromosomas locales, y selecciona el mejor cromosoma global (paso 5 y 6). Si este no presenta la estructura del óptimo (la condición de la estructura de cíclica no se ha cumplido), envía el mejor cromosoma global (paso 7) al proceso secuencial del paso 2 al comienzo del flujo de trabajo. Cuando un proceso SPMD recibe un cromosoma que no está vacío, reemplaza su peor cromosoma local por el cromosoma que recibió y continúa el procedimiento evolución. Cada iteración del ciclo define una nueva generación de la población. Si el proceso en el paso 6 identifica que el mejor cromosoma global es la solución óptima, el ciclo termina y la solución se muestra al usuario (paso 8).

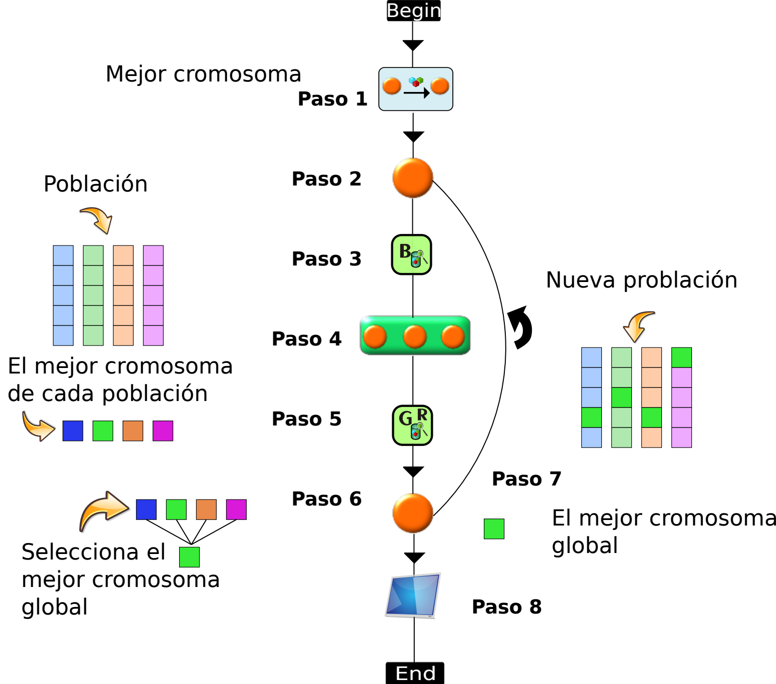


Figura 5.5: Flujo de trabajo VPPL implementando un algoritmo genético.

Resumen

En este capítulo hemos presentado un conjunto de ejemplos de programas paralelos desarrolladas con VPPL. Estos ejemplos describen desde como realizar un programa secuencial en VPPL hasta como implementar un algoritmo genético paralelo. En conclusión, observamos que mediante VPPL se busca un lenguaje que permita el desarrollo de programas paralelos de forma más simple aprovechando la expresividad de los lenguajes visuales. El número de iconos y estructuras de VPPL es muy pequeño a fin de no hacer complejo en lenguaje. Además, los patrones permiten desarrollar programas que por su naturaleza no sería posible realizarlas mediante el flujo lineal en un flujo de trabajo.

Capítulo 6

Arquitectura del ambiente de programación VPPE

El desarrollo de de un nuevo lenguaje visual esta muy de la mano con el ambiente donde será ejecutado. Tener un lenguaje visual simple puede resultar no práctico si el ambiente donde se ejecuta es muy complejo. En esta tesis hemos propuesto un ambiente de desarrollo para el nuevo lenguaje VPPL al cual hemos llamado VPPE (Visual Parallel Programming Environment). VPPE ha sido diseñado como un ambiente que se ejecute como una aplicación enriquecida de internet. La arquitectura de VPPE esta compuesta de 4 módulos (Figura 6.1). La interfaz de usuario, el traductor, el módulo de persistencia, y el motor de ejecución. En este capítulo exploraremos cada unos de los módulos de VPPE.

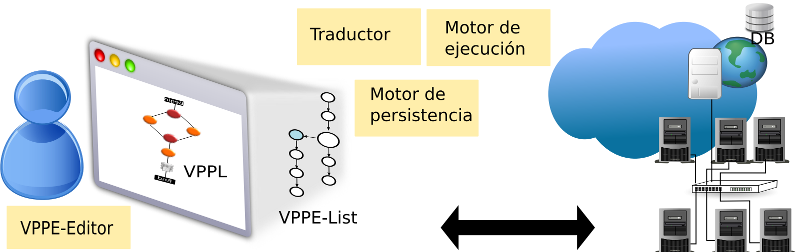


Figura 6.1: Arquitectura del ambiente VPPE

6.1. Interfaz de usuario

La interfaz de usuario esta compuesta de dos elementos: un nuevo lenguaje visual, VPPL y un editor visual, el VPPEd. VPPL le proporciona al desarrollador un conjunto de iconos para especificar operaciones de entrada y salida, comunicación y procesamiento. Los iconos no se pueden insertar en cualquier punto del flujo de trabajo, por lo que VPPE verifica que los iconos se inserten en posiciones válidas. En la Figura 6.2 se muestra un ejemplo de un flujo de trabajo usando VPPL. Como se mencionó en la Sección 4.2, los iconos del lenguaje tienen un conjunto de parámetros por medio de los cuales se ingresa información de acuerdo al tipo de icono.

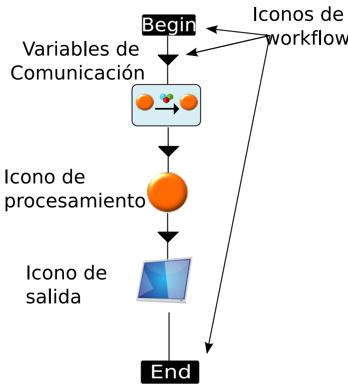


Figura 6.2: Un flujo de trabajo por medio de VPPL.

El editor de VPPE (VPPEd) permite la creación y edición de flujos de trabajo. En nuestra propuesta el editor fue definido mediante un diseño portable y una interfaz simple, sin muchos menús. Se ejecuta como una aplicación enriquecida de internet [1] sobre los navegadores web más populares (Chrome, Firefox, Opera, Safari, etc); de esta manera, el desarrollador no tiene que instalar ningún software o biblioteca adicional. El VPPEd se muestra en la Figura 6.3, las etiquetas azules (círculos) en la figura se utilizan para describir cada parte de la interfaz.

[1]Aplicación web que tiene la mayoría de las características de las aplicaciones de escritorio tradicionales.

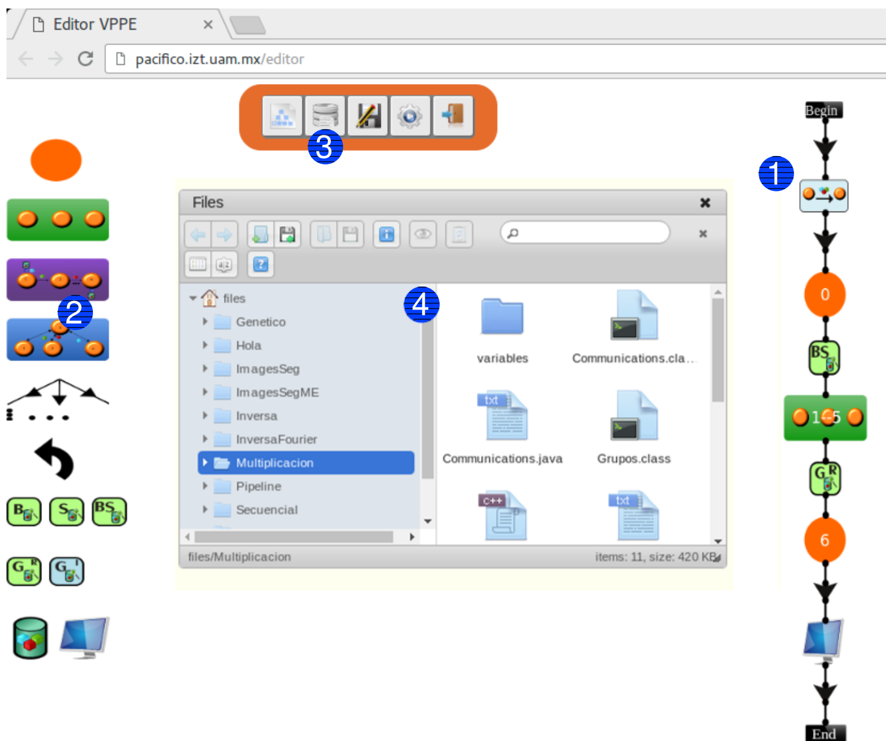


Figura 6.3: Editor visual VPPE

La etiqueta con el número uno hace referencia al área de edición, donde los flujos de trabajo son creados o editados. Los iconos de la parte izquierda de la imagen (etiqueta dos) corresponden a diferentes iconos que están disponibles para incluirlos en un flujo de trabajo; estos iconos se pueden agregar por medio de un click en el área de edición o bien mediante operaciones *drag and drop*.

En la parte superior del editor (etiqueta tres) hay una barra de herramientas que tiene botones para crear, abrir, guardar o ejecutar un programa gráfico. El explorador de archivos (etiqueta cuatro) se despliega cuando se presiona el botón abrir; se mantiene un directorio personal donde los archivos del ambiente VPPE se guardan y donde los programas secuenciales pueden ser almacenados para ser incluidos como parte de una ejecución paralela.

Internamente un flujo de trabajo se organiza como una lista de listas llamada VPPE-List. Cada nodo de la lista se relaciona con un icono o un conjunto de iconos cuando representa a un patrón paralelo, por ejemplo, los patrones SPMD o MPMD (ver el cuadro purpura con trazas de la Figura 6.4). Además, dentro de

cada nodo son almacenados los parámetros de los iconos.

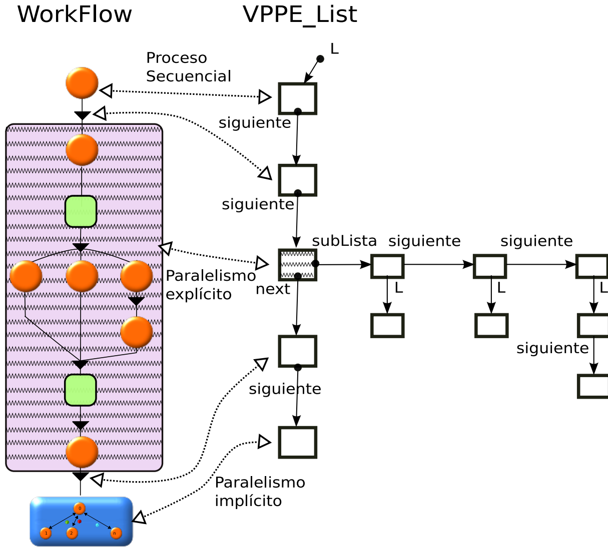


Figura 6.4: Estructura VPPE-List

La VPPE-List se actualiza en tiempo de ejecución al modificar el flujo de trabajo; al cambiar el valor de un parámetro o al mover, agregar o eliminar un icono. A fin de encontrar el nodo asociado al icono se utiliza un algoritmo recursivo para realizar un recorrido en la lista.

6.2. Traductor

El módulo traductor (ver Figura 6.5) traduce un flujo de trabajo (nuestro código fuente) en código en lenguaje de programación de alto nivel (Java, nuestro código objeto). La traducción es hecha por un componente interno llamado traductor. Este componente puede generar código objeto en más de un lenguaje de programación; sin embargo, actualmente únicamente genera código en Java, el cual es un lenguaje de programación ampliamente usado, eficiente y portable. El código objeto generado para cada icono de VPPE es almacenado en un repositorio, en un archivo con la extensión del lenguaje utilizado. El traductor utiliza una función llamada *getLanguaje* para identificar el tipo de código objeto que generará, y después, dependiendo del icono a traducir, selecciona e inserta el

código objeto correspondiente en el archivo del programa.

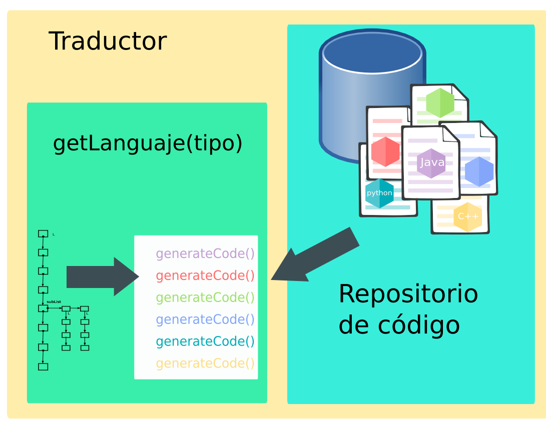


Figura 6.5: Arquitectura del traductor VPPE

Un ejemplo de la traducción de un flujo de trabajo a código Java-MPI se muestra en la Figura 6.6. La parte izquierda de la figura muestra un flujo de trabajo en VPPE (código fuente) y la parte derecha el código Java-MPI generado (código objeto). Para describir el flujo de trabajo, cada icono de procesamiento es etiquetado con un único número positivo llamado *id*. Los ids se utilizan para generar (en el programa objeto) bloques de código condicional que definen el bloque de código que cada proceso ejecutará (lineas 11-47, 12-21, 22-33, 34-35 y 36-46). Cada fecha continua de un icono a un bloque de color azul en el código Java-MPI representa el código ingresado por el desarrollador mediante un click derecho en el icono (usando los parámetros del icono); el código ingresado se concatena en el programa objeto que se genera. En este ejemplo, el primer bloque (lineas 5-6) define la declaración de variables (array y partialResults) que se utilizan en la comunicación de procesos. Los otros bloques representan código que será ejecutado por los procesos (lineas 13-16, 28-31 y 40-45).

Una línea punteada de un icono a uno o mas bloques punteados representa código que es generado automáticamente por VPPE para la comunicación de procesos (lineas 17-20, 23-27, 32, y 37-39). El icono de comunicación colectiva S representa la dispersión de una o más variables de comunicación (en el ejemplo, array) y el icono etiquetado con G^r representa la recolección de una o más variables de comunicación (en el ejemplo, partialResults). Como podemos observar, para los iconos de comunicación únicamente es necesario ingresar el nombre de

las variables involucradas en la comunicación a través de una ventana emergente (diálogo). Note que el código objeto generado (java) por el traductor de VPPE utiliza métodos en tiempo de ejecución propios de VPPE, los cuales internamente utilizan métodos de Java-MPI después de una contabilidad de datos. Por ejemplo, el método de comunicación dispersión de VPPE (línea 27 en la Figura 6.6), utiliza el método *scatter* de Java-MPI únicamente después de que todos los procesos están de acuerdo (después de un mensaje adicional) de la cantidad de datos que serán transferidos de manera que los receptores reciben en un arreglo o buffer del tamaño exacta para la cantidad de datos recibidos. Las ventajas de esto las discutimos en la última parte de la Sección 4.2.

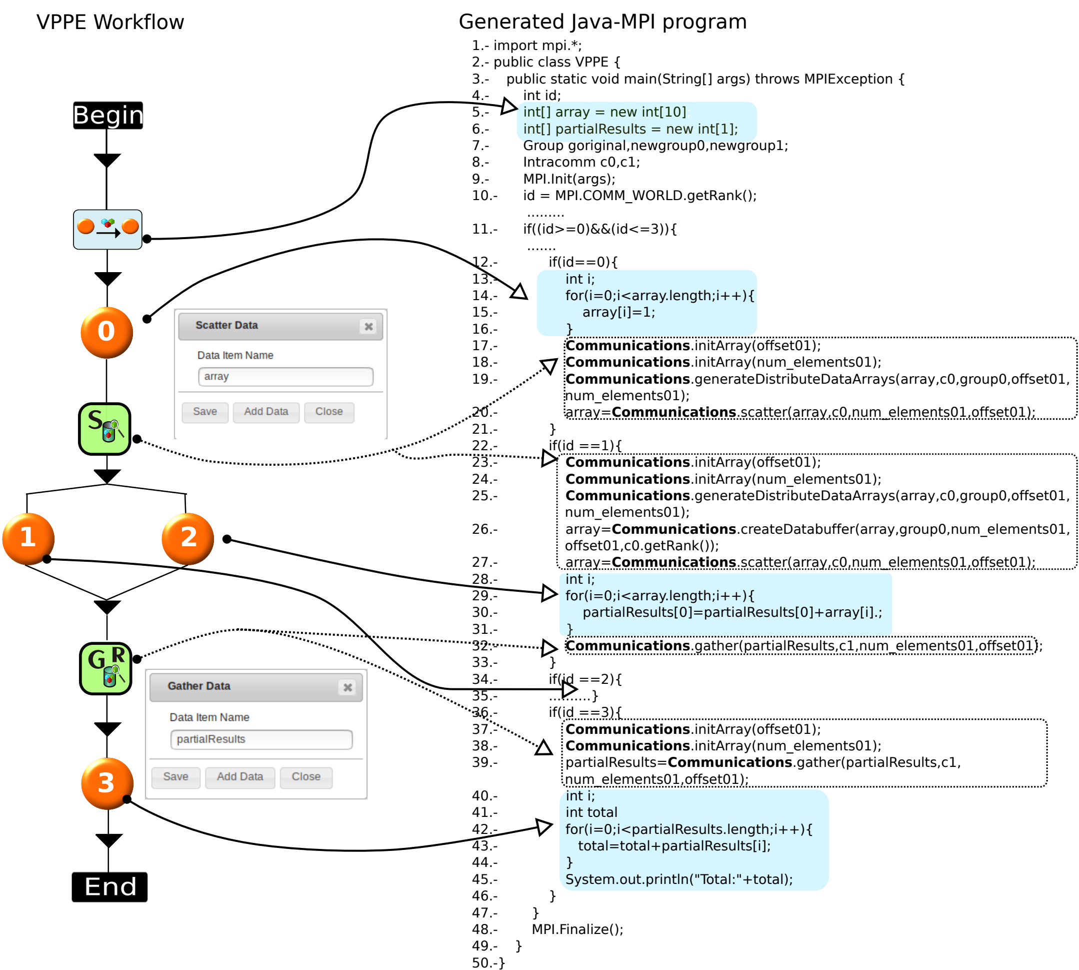


Figura 6.6: Traducción de un flujo de trabajo a código Java-MPI

Como se mencionó en la sección previa, un flujo de trabajo se representa internamente como una lista llamada VPPE-List. En la traducción, esta lista es la que se utiliza para generar el código objeto (Java-MPI). La VPPE-List se recorre mediante un algoritmo recursivo creando grupos de procesos, y concatenando el código de cada icono dentro de bloques de código condicional los cuales consideran los id's generados previamente. La creación de grupos únicamente es llevada a cabo en dos casos:

1. En un icono de procesamiento que utiliza paralelismo implícito.
2. En un conjunto de ramas de flujos de trabajo.

La Figura 6.7 muestra el algoritmo recursivo diseñado para la generación de grupos. Las llamadas recursivas en el algoritmo únicamente se requieren para los nodos que contienen otras sub-listas de tipo flujo de trabajo.

```
1:  Función grupos(VPPE_List L)
2:      Si L ≠ null Entonces
3:          Si es_procesamientoParalelo(L) Entonces
4:              Según sea tipoDeProcesamiento(L)) Hacer
5:                  Caso Implicito: creaGrupo(L);
6:                  Caso Explicito:
7:                      ptr = L → SubLista;
8:                      Mientras ptr ≠ null Hacer
9:                          Groupos(ptr → L);
10:                         ptr = ptr → siguiente;
11:                     Fin Mientras
12:                     creaGrupo(L);
13:             Fin Según
14:             Grupos(L → siguiente);
15:         Fin Si
16:     Fin Si
17: Fin Función
```

Figura 6.7: Algoritmo para generación de grupos.

En la Figura 6.8 se muestra un ejemplo de creación de grupos de la parte del flujo de trabajo que se ilustra en la Figura 6.4. Como se observa, son creados tres grupos. El primer grupo (grupo 1) corresponde a 4 procesos secuenciales, donde un proceso realiza una operación colectiva de salida. El segundo grupo (grupo 2) corresponde a 4 procesos secuenciales, donde un proceso recolecta datos de los tres procesos restantes, finalmente el tercer grupo (grupo 3) corresponde a los procesos que integran el patrón Maestro-Esclavo.

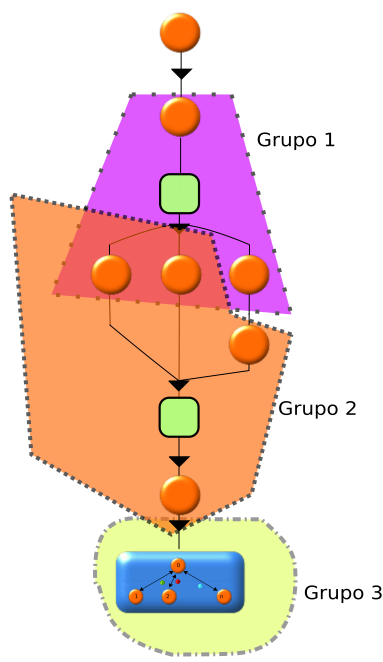


Figura 6.8: Grupo de procesos en un flujo de trabajo.

6.3. Motor de ejecución

El motor de ejecución de VPPE (VPPE-Ex) recibe el código fuente generado por el traductor, y compila y ejecuta el programa fuente en un cluster (ver Figura 6.9).

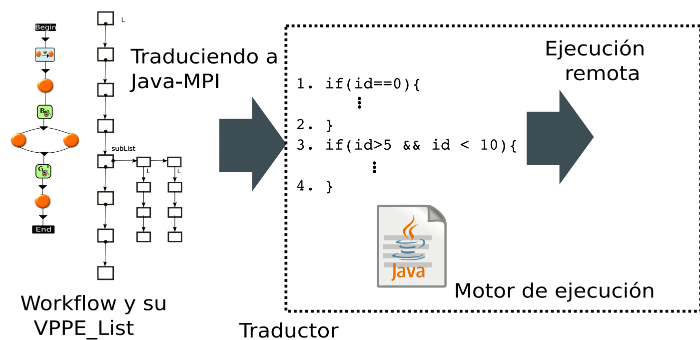


Figura 6.9: Traduciendo y ejecutando un flujo de trabajo.

En este proceso son generados dos archivos: log y out. El primero almacena los mensajes de error que se presentan al momento de la compilación, y el segundo almacena la salida estándar que genera el programa cuando es ejecutado. Si el programa genera otros archivos (propios del programa), estos son almacenados en el *home* del desarrollador.

6.4. Módulo de persistencia

El módulo de persistencia tiene dos funciones principales: guardar un flujo de trabajo como un archivo XML y cargar un flujo de trabajo de un archivo XML. El formato XML se utilizó para guardar un flujo de trabajo porque éste se acopla de manera adecuada a la estructura árbol que representa un flujo de trabajo, lo que permite guardarlo y recuperarlo fácilmente (Figure 6.10). Para construir un archivo XML se recorre la estructura VPPE-List, y por cada nodo se crea una o un conjunto de etiquetas. Algunas de las etiquetas que son definidas para caracterizar un flujo de trabajo se muestran en la Figura 6.10 (*program*, *workflow*, *wf*, *components*).

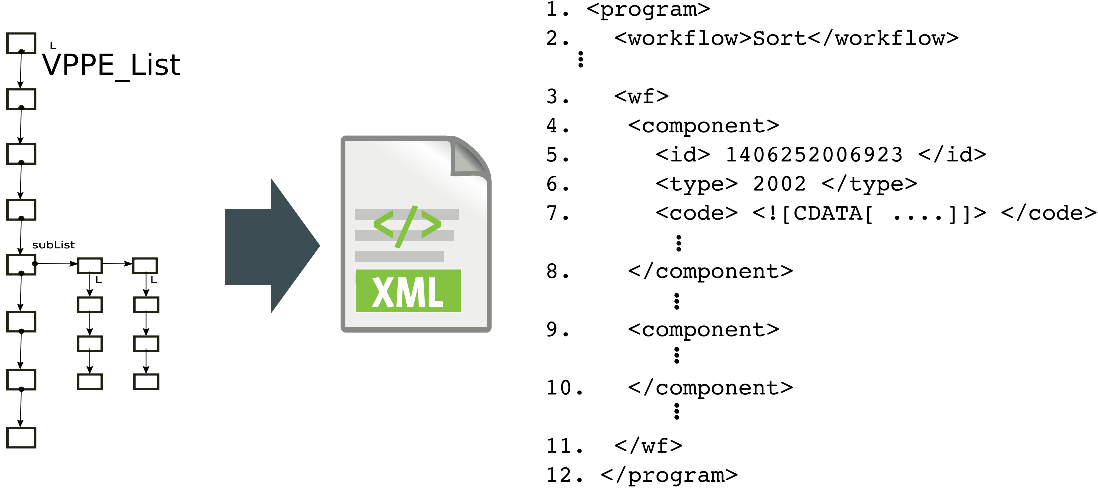


Figura 6.10: Creación de un archivo XML a partir de un flujo de trabajo.

Las etiquetas $< program >$ y $< /program >$ (lineas 1 y 12) representan el inicio y el fin de un flujo de trabajo; las etiquetas $< workflow >$ y $< /workflow >$ (linea 2) almacenan el nombre del flujo de trabajo. Las etiquetas $< wf >$ y $< /wf >$ (lineas 3 y 11) delimitan una rama paralela (flujo de trabajo), y contienen la información de un conjunto de iconos. Las etiquetas $< component >$ y $< /component >$ representan la definición de un icono e incluyen la especificación de sus respectivos parámetros (*id*, *type*, *code*, x and y position, etc.).

Para generar un flujo de trabajo de un archivo XML (Figure 6.11), primero se crea la estructura VPPE-List por medio de un analizador sintáctico que toma los valores de cada etiqueta del archivo XML. Cuando la estructura es generada, el ambiente VPPE lee la estructura VPPE-List y dibuja cada icono en el área de edición tomando en cuenta su posición x y y.

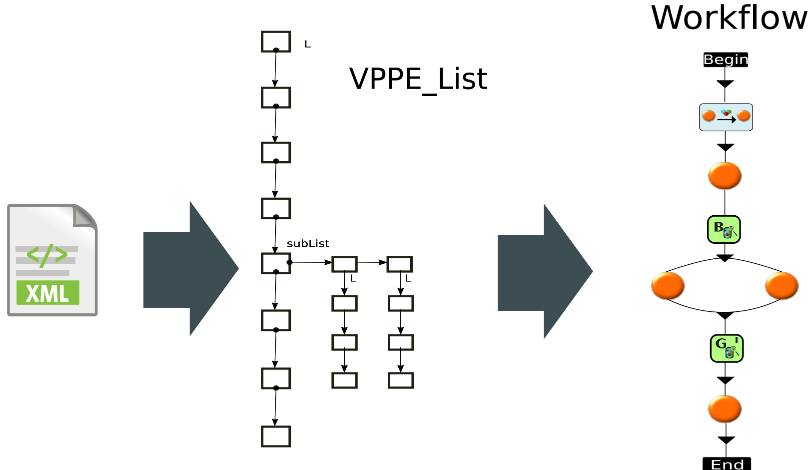


Figura 6.11: Proceso de carga de un flujo de trabajo.

Resumen

En este capítulo hemos presentado el ambiente VPPE. El ambiente fue diseñado a fin de facilitar la interacción con el lenguaje VPPL. El ambiente de VPPE esta conformado por 5 módulos los cuales tiene tareas bien definidas:

1. La Interfaz de usuario: Permite la interacción con VPPL.
2. El traductor: Realiza la traducción de un flujo de trabajo a un programa en Java-MPI.
3. El módulo de persistencia: Permite abrir y guardar un flujo de trabajo.
4. El motor de ejecución: Permite ejecutar un flujo de trabajo en un cluster.

El diseño de de VPPE permite implementarlo como una aplicación de escritorio o web. En nuestra implementación se utilizaron tecnologías web estándar a fin que se el acceso al ambiente sea simple sin requiere alguna instalación extra.

En conclusión, mediante VPPE el uso del lenguaje VPPL es más simple. VPPE permite extender fácilmente el lenguaje ya que cada módulo tiene una tarea bien definida.

Capítulo 7

Evaluación y resultados

La programación visual mediante VPPL es en principio más simple que la glspbt. Intuitivamente, esto se debe a que los iconos visuales de VPPL corresponden a abstracciones de alto nivel de las tareas de procesamiento, que ocultan detalles de programación, permitiendo a los programadores concentrarse en la solución del problema en cuestión, en lugar de en las herramientas de software y su apropiado uso para resolver el problema. Sin embargo, ¿cuánto más simple es la programación con VPPL que la Programación Basada en Texto (PBT), y de qué manera es más sencillo?, también, ¿qué limitaciones impone la programación con VPPL en el desarrollo de programas?.

Esta capítulo presenta una comparación cualitativa de la programación VPPL y PBT que provee algunas respuestas a estas preguntas. Además, se presenta una comparación de rendimiento de dos programas desarrollados tanto con VPPL y PBT a fin de proporcionar evidencia de la calidad del código generado por VPPE en términos de rendimiento respecto a PBT.

7.1. Comparación cualitativa: VPPL vs PBT

VPPL gestiona varios tipos de iconos visuales para especificar: i) estructuras de flujo de trabajo, ii) entrada/salida, iii) procesamiento y iv) comunicación. Todos estos iconos (cuando se utilizan) deben configurarse con unos pocos

parámetros relacionados con las variables en la comunicación y el código secuencial ejecutado por cada proceso. Otros aspectos, como el tiempo de la ejecución de cada proceso, la ejecución simultánea de un conjunto de procesos, incluyendo su interacción, se representan visualmente por cada posición de los iconos en el flujo de trabajo.

Por lo tanto, los iconos de VPPL y su posición en el flujo de trabajo ocultan el código necesario a actividades referentes a la administración de procesos, como son: identificación de procesos, creación de procesos, administración de grupos de procesos, comunicación de procesos, gestión de procesos y depuración de comunicaciones y patrones de procesamiento paralelo (ver Tabla 7.1).

Características	**PBT**	**VPPL**
Identificación de procesos.	Explicita.	No necesaria.
Creación de procesos.	Explicita.	Implícita/Explicita.
Admón. de grupos de procesos.	Explicita.	Implícita.
Comunicación de procesos.	Explicita, General.	Flecha o iconos de comunicación colectiva.
Admón. y Depuración de procesos.	Tarea difícil, creación estática o dinámica.	Libre de administración de procesos y errores de comunicación, creación estática.
Patrones de paralelismo.	Código condicional de acuerdo a los id, implementación desde cero.	Patrones SPMD, Maestro-Esclavo y Pipeline.

Tabla 7.1: Comparación cualitativa entre programación con PBT y VPPL.

La ***Identificación de procesos*** en PBT es explícita, administrada por el programador, generalmente usando código condicional para asignar la tarea que debe ser ejecutada por cada proceso e identificar los procesos que participan en la comunicación. Por el contrario, la asignación de tareas a procesos en VPPL es

implícita a través de sus iconos de procesamiento. Cuando un icono representa un patrón de procesamiento paralelo, una tarea puede estar asociada con un conjunto de procesos; en cuyo caso es posible administrar los identificadores de procesos con el fin de identificar y gestionar un proceso particular. Del mismo modo, la comunicación de procesos en VPPL no requiere que el programador especifique los identificadores de los procesos. Cada proceso conoce su emisor/receptor según su posición en el flujo de trabajo. Si un proceso forma parte de un patrón, envía o recibe mensajes según su función en el patrón.

En la ***creación de procesos***, con PBT es necesario conocer los comandos o instrucciones particulares para crear procesos; esto podría hacerse estática y dinámicamente cuando sea conveniente. Usando VPPL únicamente es posible la creación estática; esto se especifica visualmente durante el desarrollo de un flujo de trabajo cuando el desarrollador inserta un icono de procesamiento. Además, si el icono es un patrón de procesamiento en paralelo (SPMD, Pipeline o Maestro-Esclavo), un número entero especifica la cantidad de procesos en el patrón.

Si un programa requiere de ***administración de grupos de procesos***, usando PBT generalmente es necesario administrar/especificar un conjunto de instrucciones para: definir qué identificadores de procesos pertenecen a un grupo específico, administración de los identificadores del grupo y definir la comunicación inter-grupal. Con VPPL la administración de un grupo de procesos es transparente para los desarrolladores; la necesidad de creación de un grupo es detectada automáticamente cuando se inserta un patrón de procesamiento en paralelo en un flujo de trabajo, por ejemplo MPMD o SPMD.

La ***comunicación de procesos y la depuración de procesos*** con PBT es una tarea compleja. Si un desarrollador no utiliza la sintaxis correcta, semántica o los identificadores de los procesos en instrucciones relacionadas específicamente con la comunicación y sincronización de procesos podrían aparecer varios errores en la pantalla; algunos de ellos no son muy específicos, los errores más difíciles están relacionados con bloqueos o terminaciones abruptas de ejecución. A la inversa, utilizando un lenguaje como VPPL, la comunicación se expresa visualmente, los desarrolladores no son conscientes de los detalles de la comunicación ya que el sistema está a cargo de esta parte; se evitan errores en la administración del grupo de procesos, especificación de comunicación o bloqueo.

Los ***patrones paralelos*** como el SPMD o MPMD en PBT utilizan una biblioteca donde su funcionamiento se considera predeterminado. Sin embargo, los identificadores de los procesos deben ser controlados por el desarrollador para asignar las funciones respectivas en el patrón. Los patrones Maestro-Esclavo y Pipeline no se incluyen en PBT, el desarrollador debe diseñar e implementar el protocolo de interacción entre los procesos. Por el contrario, VPPL ofrece los patrones paralelos SPMD, Maestro-Esclavo y Pipeline mediante tres iconos especiales donde los protocolos de comunicación ya están implementados; los desarrolladores sólo tienen que especificar las tareas secuenciales a realizar y el número de procesos que participan en el patrón. Con respecto al patrón MPMD, se considera la ejecución de diferentes programas cuando varias ramas provienen del mismo proceso. Además, si es necesario se pueden construir flujos de trabajo que utilicen una combinación de varios patrones. El trabajo futuro incluye el desarrollo de otros patrones o modelos paralelos como Map-Reduce [1] [LHL+16].

7.2. Evaluación de desempeño

Se realizaron dos comparaciones de rendimiento para evaluar VPPE mediante dos programas que se desarrollaron en VPPL y PBT. Estas comparaciones permiten evaluar la calidad del código generado por VPPE. A continuación se presentan los programas utilizados, luego la infraestructura utilizada y finalmente los resultados de las comparaciones.

7.2.1. Programas

Los programas incluyen: el problema de las N-Reinas (NAQ por sus siglas en inglés) y el procesamiento de un lote de imágenes. Estos programas fueron elegidos porque ambos tienen un conjunto de datos que se pueden procesar independientemente y tienen tiempos de ejecución de procesamiento diferentes. Por consiguiente, se debe considerar el balance de carga (que es una tarea de codificación más desafiante) para lograr un buen rendimiento.

[1] MapReduce es un modelo de programación sobre grandes colecciones de datos en clusters.

El problema de las N-Reinas consiste en encontrar el número de soluciones en las que se pueden colocar N reinas en un tablero de ajedrez de $N \times N$, de modo que ninguna reina ataque otra [BD75]. El espacio de búsqueda de las N-Reinas puede ser modelado con un árbol de búsqueda de grado N. Las soluciones se encuentran explorando parcialmente el árbol de búsqueda, eliminando los sub-árboles que no son válidos (en nuestros experimentos usamos $N = 17$). En este programa, un proceso Maestro genera todos los sub-árboles de búsqueda parcial con altura 5, que podrían generar posibles soluciones. Este valor de altura se establece considerando la capacidad de memoria del servidor donde se ejecuta el programa. Todos los sub-árboles están representados como un arreglo de tamaño 17, $Tablero$, donde $Tablero[i]$ ($0 \leq i < 17$) contiene el número de columna donde se ubica la reina de la fila i; sólo las primeras cinco ubicaciones de cada arreglo son inicializadas por el maestro logrando que ninguna de las reinas ataque a otra. Los arreglos generados se insertan en una lista (se insertan más de 1,400,000 elementos). Un conjunto de Esclavos envía peticiones al Maestro para obtener un elemento de lista; para cada $Tablero$ recibido el Esclavo completa la búsqueda para encontrar todas las posibilidades de colocar las restantes 12 reinas en las ubicaciones (filas) de 5 a 16. El número de soluciones encontradas se envía al Maestro.

El procesamiento de un lote de de imágenes con el método Mean-Shift (MSM) reconstruye las imágenes cerebrales 3D [JAMBYS06] aplicando dicho método a cada píxel en varias imágenes 2D (cada imagen de 217x181 píxeles). El método es costoso para la gran cantidad de imágenes a ser procesadas (165 en este caso) y la alta resolución requerida. El número de imágenes es conocido, pero el costo de procesamiento de cada imagen varía según el costo de MSM que depende de la intensidad (nivel de grises) de cada píxel.

7.2.2. Infraestructura utilizada

La plataforma experimental es un cluster compuesto de 42 núcleos en 9 nodos; las especificaciones de hardware y el sistema operativo se muestran en la Tabla 7.2. Todos los nodos son conectados a través de un switch Gigabit Ethernet.

Las especificaciones del software son: versión del compilador gcc 4.8.5, versión

Nodo	Núcleos físicos por nodo	Velocidad	Memoria	Sistema operativo
nodo 1-3	4	2.4GHz	4 GB	Centos 7.4
nodo 4-5	4	3.0GHz	2 GB	Centos 7.4
nodo 6-8	4	3.4GHz	16 GB	Centos 7.4
nodo 9	10	3.3GHz	32 GB	Centos 7.4

Tabla 7.2: Características de la plataforma experimental.

de OpenMPI 1.10.2, y SDK java 1.8.0_25.

7.2.3. Resultados

Como se mencionó anteriormente, ambos programas requieren balance de carga para mostrar un buen desempeño. El protocolo Maestro-Esclavo trabaja bien respecto al balance de carga en programas donde algunos esclavos pueden desempeñar más trabajo/tareas que otros, dependiendo de la velocidad o frecuencia del procesador o los tiempos de ejecución que requiere cada tarea. Para PBT (considerando un experto en programación paralela), el protocolo Maestro-Esclavo es implementado desde cero utilizando Java-MPI. Las Figuras 7.1 y 7.2 muestran los tiempos de ejecución en segundos (en escala logarítmica) obtenidos por VPPE y PBT en ambos programas utilizando diferente número de procesos/esclavos.

La Figura 7.1 muestra los tiempos de ejecución obtenidos por VPPE, PBT y el programa secuencial para el problema N-Reinas utilizando 4-40 esclavos. Observamos en general que el desempeño de PBT es ligeramente mejor que el de VPPE (en aproximadamente 10 %). Esto es debido a que VPPE agrega una sobrecarga de comunicaciones que afecta el desempeño su desempeño; en PBT el programador genera únicamente un mensaje por explorar un tablero mientras que en VPPE se requieren dos.

NAQ-PBT administra un buffer de comunicación cuyo tamaño es ajustado durante la ejecución al tamaño de tablero y por tanto en todos los tableros transferidos del maestro a los esclavos el experto directamente aplica el tamaño del buffer de comunicación (de recepción y envío) a 17, requiriendo únicamente un mensaje para transferir un tablero. Por el contrario, como se presentó al final de la Sección 4.2.4, en VPPE para realizar la transmisión de un simple tablero

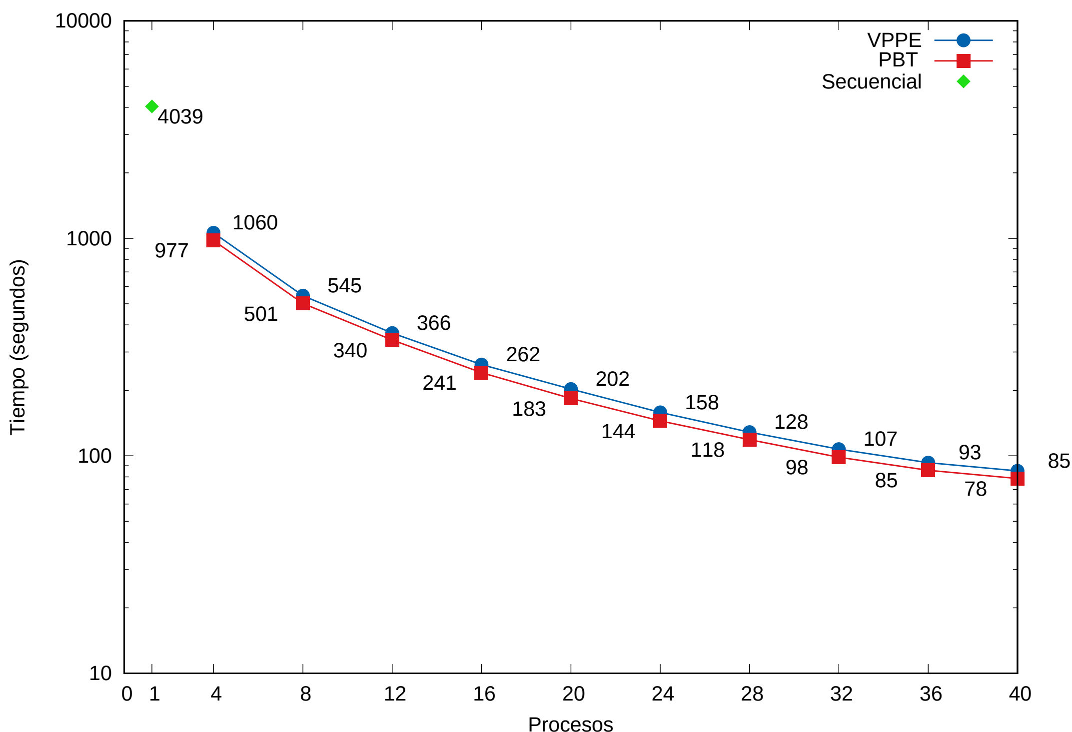


Figura 7.1: Comparación de desempeño para el problema de 17-Reinas: VPPE vs PBT

del Maestro a un Esclavo se requieren dos mensajes; el primer mensaje contiene el tamaño del tablero a ser transferido (17), y el segundo mensaje almacena el tablero. Después de recibir el primer mensaje, el proceso Esclavo crea un buffer de tamaño 17 y almacena el tablero del segundo mensaje en este buffer.

Aunque VPPE incurre en una sobrecarga de comunicación al usar un mensaje adicional por cada mensaje del programa, presenta mejores tiempos que el programa secuencial y ofrece varias ventajas (como se explica al final de la Sección 4.2.4).

La Figura 7.2 compara los tiempos de ejecución del procesamiento de un conjunto de imágenes utilizando el icono de SPMD de VPPE (representado como VPPE-SPMD en la figura), un icono Maestro-Esclavo en VPPE (representado como VPPE-M-Esclavo) y un Maestro-Esclavo en PBT (representado como PBT-M-Esclavo). Si el desarrollador no es un experto, el icono SPMD de VPPE podría ser seleccionado a fin de dividir equitativamente el procesamiento de las imágenes

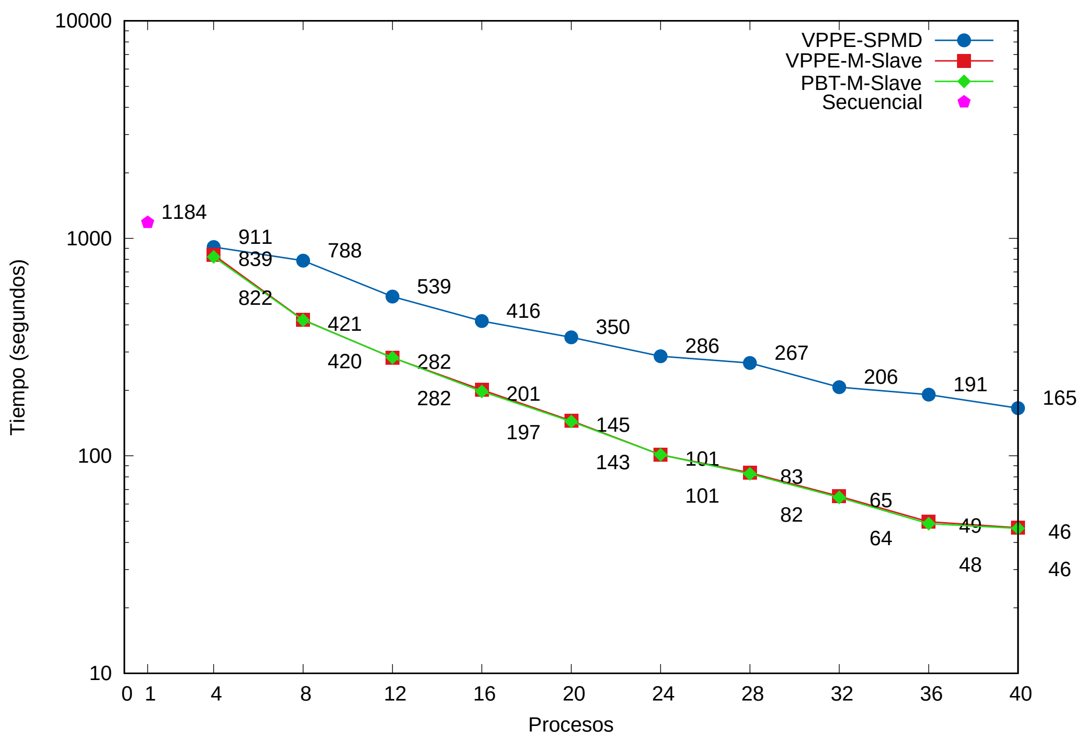


Figura 7.2: Comparación de desempeño para el procesamiento de un conjunto de imágenes: VPPE vs PBT

entre el conjunto de procesadores. Sin embargo, esta solución incurre en altos tiempos de ejecución (en el peor de los casos más de 250 %) comparado a las otras versiones. Esto es porque este programa tiene una granularidad gruesa (el tiempo de procesamiento de cada imagen esta en el orden de $O(n \times m)$, donde n y m es el tamaño de la imagen en píxeles) y el cluster donde es ejecutada el programa es heterogéneo (con procesadores de diferentes velocidades o frecuencias). En este contexto, el protocolo Maestro-Esclavo es más conveniente (respecto al SPMD) a fin de distribuir dinámicamente las imágenes entre los procesadores de acuerdo a la frecuencia de cada procesador. Si el desarrollador VPPE tiene este conocimiento de los procesadores y selecciona el icono de Maestro-Esclavo, el resultado del programa en VPPE tendrá mucho mejor desempeño, casi tan bueno como los obtenido por la versión Maestro-Esclavo en PBT.

Resumen

En este capítulo hemos presentado una evaluación cualitativa del lenguaje VPPL y una evaluación de desempeño del código generado por VPPE. Respecto a la evaluación cualitativa, podemos destacar que el lenguaje VPPL simplifica el desarrollo de programas paralelos ocultando elementos fundamentales en la programación paralela como son: la identificación de procesos, la creación de procesos, la administración de grupos, la comunicación, codificación del protocolos para patrones de cómputo como SPMD, Maestro-Esclavo o Pipeline. Respecto al desempeño, se comparó el código generado por VPPE y el código elaborado en PBT, ambos códigos en Java-MPI, en un cluster dedicado. En general, observamos que aunque PBT puede obtener ligeramente mejor desempeño que VPPE (por ejemplo, como en el programa de las N-Reinas) se requiere conocimiento especializado en programación paralela. Por el contrario, los desarrolladores de VPPE no necesariamente tienen que ser expertos en programación paralela e invirtiendo un tiempo de desarrollo pequeño pueden obtener tiempos de respuesta muy aceptables, incluso cercanos a los obtenidos en PBT. Además. al igual que en la gráfica anterior, se observa que VPPE presenta mejores tiempos que en la versión secuencial. En conclusión, VPPL y VPPE permiten desarrollar programas de forma más simple (no se requiere ser experto en cómputo paralelo) y con un buen desempeño.

Capítulo 8

Conclusiones y trabajo futuro

El desarrollo de programas paralelos en memoria distribuida cada vez es más factible y accesible debido a los bajos costos de las computadoras. Sin embargo, la complejidad subyacente en el desarrollo de este tipo de programas origina una barrera para que los programadores definan la solución de un problema mediante este enfoque. Como se menciona en el Capítulo 2, a fin de simplificar o agilizar el desarrollo de este tipo de programas en la literatura se han propuesto una variedad de lenguajes y herramientas visuales, las cuales se ven limitas en su expresividad, aplicabilidad, nivel de interacción y portabilidad.

Por lo anterior, el objetivo general de este proyecto doctoral fue la definición de un nuevo lenguaje visual para el desarrollo de programas paralelos considerando un mayor nivel de expresividad en sus elementos visuales o iconos, una mayor aplicabilidad con la finalidad de ampliar el rango de programas que se puedan desarrollar usando el lenguaje propuesto, un nivel de interacción alto a fin de simplificar y agilizar el desarrollo de programas y la portabilidad en el desarrollo y ejecución de programas. Para cumplir con este objetivo, se propuso en el Capítulo 4 el lenguaje visual VPPL y en el Capítulo 6 el entorno de desarrollo VPPE. El lenguaje VPPL y el ambiente VPPE incorporan en conjunto: expresividad, aplicabilidad, nivel de interacción y portabilidad en el desarrollo de programas.

Mediante VPPL el programador desarrolla un programa siguiendo el modelo de flujo de trabajo. La expresividad de VPPL se basa en la disponibilidad de iconos visuales para especificar las estructuras de repetición, los flujos de trabajo y las tareas de procesamiento y comunicación. Además, VPPL tiene una amplia

aplicabilidad ya que puede utilizarse para problemas que pueden resolverse bajo los patrones de procesamiento SPMD, MPMD, Maestro-Esclavo, Pipeline o combinaciones de los mismo.

A diferencia de los lenguajes visuales paralelos disponibles, el lenguaje VPPL fue especificado por medio de la teoría de la Gramática de Reemplazo de Hiperaristas (ver Capítulo 4, Sección 4.3), un formalismo que guía la construcción de representaciones visuales válidas. Usando este formalismo el lenguaje VPPL se puede extender fácilmente incorporando nuevas reglas que permitan nuevos elementos de procesamiento, entrada/salida y comunicación.

El ambiente VPPE se organizó bajo una arquitectura compuesta de cuatro componentes: interfaz de desarrollador, traductor, motor de persistencia y motor de ejecución. Este diseño ofrece un *alto nivel de interacción* y facilidad de uso, ya que muchos aspectos de la administración de programas (relacionados con crear, abrir, guardar, editar o ejecutar) son transparentes para los desarrolladores. Además, las operaciones de creación/edición de flujos de trabajo visuales que se llevan acabo mediante arrastrar y soltar, y el encapsulamiento de iconos, etc., también contribuyen a una interacción alto.

VPPE fue diseñado con una interfaz portable implementada sobre tecnologías web estándar. Esto permite que el desarrollo, actualización, almacenamiento y ejecución de los programas se pueda realizar desde cualquier navegador web sin requerir que el desarrollador tenga que instalar herramientas adicionales. El traductor actual de VPPE genera código para Java-MPI; en particular, Java permite facilitar la integración de otras herramientas implementadas en el mismo lenguaje, como Hadoop[1] o Spark[2]. El traductor VPPE puede extenderse fácilmente para permitir la generación de código en otros lenguajes de programación, si se desea.

Con la finalidad de evaluar el lenguaje VPPL y su ambiente VPPE, en el Capítulo 7 se presentó una evaluación cualitativa y una evaluación de rendimiento. Respecto a la evaluación cualitativa (Sección 7.1), sin bien es subjetivo decir que el desarrollo de un programa es “fácil” o “dificil”, esta evaluación permite ver intuitivamente que en principio la programación con VPPL y VPPE es más

[1]Apache Hadoop es un framework de software que soporta aplicaciones distribuidas bajo una licencia libre.

[2]Apache Spark es un framework computación en clúster de código abierto que se diferencia con Hadoop en el uso de operaciones en memoria divididas en varias fases de procesamiento.

simple que la programación basada en texto (la cual es mayormente utilizada en el desarrollo de programas paralelos) debido a que los iconos visuales de VPPL corresponden a abstracciones de alto nivel de las tareas/operaciones de procesamiento y comunicación ocultando detalles de programación textual, y con ello permitiendo a los programadores concentrarse en la solución del problema en cuestión, en lugar de en las herramientas de software y su apropiado uso para resolver el problema.

Respecto a la evaluación de rendimiento (Sección 7.2), se mostró que aunque los programas textuales generados por VPPE incurren en una sobrecarga de comunicación al usar un mensaje adicional (a fin de conocer el tamaño de los datos transferidos), generando con ello ligeramente mayores tiempos de ejecución, los desarrolladores en VPPL y VPPE no necesariamente tienen que ser expertos en programación paralela, pero que al invertir un tiempo de desarrollo pequeño pueden obtener tiempos de respuesta muy aceptables.

El trabajo a futuro en VPPL y VPPE es muy amplio e incluye temas como son:

1. Definición de nuevos patrones o servicios para cómputo paralelo. El lenguaje VPPL actualmente soporta un número muy reducido de patrones de procesamiento (únicamente 4), no obstante estos se pueden extender actualizando dos de las reglas de la gramática de VPPL. El agregar nuevos patrones de cómputo paralelo permite aumentar aún más la aplicabilidad, por ejemplo, se puede incorporar un icono para el patrón o modelo Map-Reduce.

2. Manejo de nuevas arquitecturas. Actualmente se tiene una variedad de arquitecturas de procesamiento, máquinas multi-núcleo (multi-core), tarjetas gráficas, dispositivos móviles, etc. VPPE pueden ser extendido a fin de permitir la ejecución es las arquitecturas antes mencionadas. Para lo anterior, el motor de ejecución y el traductor deben extenderse generando un nuevo módulo que permita identificar el tipo de infraestructura y las herramientas y bibliotecas de comunicaciones disponibles para hacer uso de ellas.

3. Transferencia de objetos. VPPL y VPPE están limitados en los tipos de datos que actualmente transfieren, únicamente se puede transferir datos o

arreglos de tipos primitivos (int, double, float, etc). El transferir objetos permite hacer más simple la solución de un problema y reducir el número de mensajes al encapsular un conjunto de datos como un solo objeto. El traductor y motor de ejecución de VPPE puede integrar nuevas herramientas o bibliotecas que permitan hacer la transferencia de objetos.

4. Interfaz adaptable. Con el auge de los dispositivos móviles, es muy común acceder a los recursos en la nube haciendo uso de un teléfono o una tablet. El diseño actual de VPPL y VPPE están pensados para ejecutarlos en un navegador web de una computadora debido a su resolución (para un manejo más fácil de los diagramas). Sin embargo, la interfaz de usuario puede ser extendida o actualizada a fin de hacer uso de VPPL en dispositivos con pequeñas resoluciones considerando otro tipo de interacciones del desarrollador (con el touch). Un diseño adaptable a la resolución permite ampliar las formas de acceder al lenguaje VPPL y a su ambiente.

5. Distribución de procesos. Actualmente la distribución de los procesos de VPPL es ciega, es decir, no se considera el tipo de icono de procesamiento o la infraestructura de procesamiento. Sin embargo, en un flujo de trabajo de VPPL hay procesos que por su posición generalmente realizan poco procesamiento, es deseable tener un protocolo que permita analizar el “tipo”de trabajo de cada proceso y realizar una asignación más “inteligente”. Lo anterior se puede realizar haciendo un análisis previo del flujo de trabajo y del cluster donde se ejecutará a fin de generar un mapeo proceso→nodo.

Bibliografía

[AMA97] Muhammed Al-Mulhem and Shahid Ali. Visual occam: Syntax and semantics. *Computer Languages*, 23(1):1 – 24, April 1997. 8

[BB11] Stanislav Bohm and Marek Bhálek. Kaira: Modelling and generation tool based on petri nets for parallel applications. In *Proceedings of the 2011 UKSim 13th International Conference on Modelling and Simulation*, UKSIM '11, pages 403–408, Washington, DC, USA, March 2011. IEEE Computer Society. 2, 8

[BBG11] Stanislav Böhm, Marek Běhálek, and Ondřej Garncarz. Developing parallel applications using kaira. In *Digital Information Processing and Communications: International Conference 2011*, ICDIPC 2011, pages 237–251, Berlin, Heidelberg, July 2011. Springer Berlin Heidelberg. 2, 8

[BD75] A. Bruen and R. Dixon. The n-queens problem. *Discrete Mathematics*, 12(4):393 – 395, 1975. 68

[BD91] Adam Beguelin and Jack J. Dongarra. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 435–444, New York, NY, USA, November 1991. ACM. 2, 8, 15

[BDH+94] James C. Browne, Jack Dongarra, Syed I. Hyder, Keith Moore, and Peter Newton. Visual programming and parallel computing. Technical report, Knoxville, TN, USA, 1994. 2

[CCCG04] Chan, J. N. Cao, A. T. S. Chan, and M. Y. Guo. Programming support for MPMD parallel computing in ClusterGOP. *IEICE Transactions on Information and Systems*, E87D(7):1693–1702, July 2004. 2, 8

[CCG06] Fan Chan, Jiannong Cao, and Minyi Guo. *ClusterGOP: A High-Level Programming Environment for Clusters*, pages 1–19. John Wiley & Sons, Inc., 2006. 2, 8

[CCS03] Fan Chan, Jiannong Cao, and Yudong Sun. High-level abstractions for message-passing parallel programming. *Parallel Comput.*, 29(11-12):1589–1621, November 2003. 2, 8

[CZJz+08] Yu Ce, Xu Zhen, Sun Ji-zhou, Meng Xiao-jing, Huang Yan-yan, and Wu Hua-bei. Paramodel: A visual modeling and code skeleton generation system for programming parallel applications. *SIGPLAN Not.*, 43(4):4–10, April 2008. 2, 8

[DKH97] Frank Drewes, Hans-Jörg Kreowski, and Annegret Habel. Hyperedge replacement, graph grammars. In *Handbook of Graph Grammars*, chapter 2, pages 95–162. World Scientific, Printed in Singapore, 1997. 17

[Doz01] Gabor Dozsa. Parallel program development for cluster computing. chapter Visual Programming to Support Parallel Program Design, pages 17–44. Nova Science Publishers, Inc., Commack, NY, USA, 2001. 2, 4, 7, 8, 14, 15, 26

[FK11] Zoltan Farkas and Péter Kacsuk. P-grade portal: A generic workflow system to support user communities. *Future Generation Comp. Syst.*, 27(5):454–465, May 2011. 8, 15

[FNSW99] Dariusz Ferenc, Jaroslaw Nabrzyski, Maciej Stroinski, and Piotr Wierzejewski. Visual mpi - a knowledge-based system for writing efficient mpi applications. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, European PVM/MPI 1999, pages 257–264, London, UK, September 1999. Springer-Verlag. 8

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994. 2

[GLS14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. 6

[GP96] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131 – 174, 1996. 7

[Hab92] Annegret Habel. *Introduction to Hyperedge-Replacement Grammars*, pages 5–42. Springe, New York, USA, 1992. 35

[JAMBYS06] J.R. Jimenez-Alaniz, V. Medina-Banuelos, and O. Yanez-Suarez. Data-driven brain mri segmentation supported on edge confidence and a priori tissue information. *Medical Imaging, IEEE Transactions on*, 25(1):74–83, January 2006. 68

[KDF96] Péter Kacsuk, Gábor Dózsa, and Tibor Fadgyas. Designing parallel programs by the graphical language grapnel. *Microprocess. Microprogram.*, 41(8-9):625–643, April 1996. 8, 15

[KDFL99] Péter Kacsuk, Gábor Dózsa, Tibor Fadgyas, and Róbert Lovas. The {GRED} graphical editor for the {GRADE} parallel program development environment. *Future Generation Computer Systems*, 15(3):443 – 452, 1999. 8, 15

[Lee04] M. D.; Parastatidis S. Lee, P. A.; Hamilton. Visual language for parallel, object-oriented programming. Technical Report 200511, Newcastle upon Tyne Univ. (England). Dept. of Computing Scienc, Newcastle, UK, February 2004. 8, 15

[LHL+16] Ren Li, Haibo Hu, Heng Li, Yunsong Wu, and Jianxi Yang. Mapreduce parallel programming model: A state-of-the-art survey. *International Journal of Parallel Programming*, 44(4):832–866, 2016. 67

[LR06] Thuy T. Le and Jalel Rejeb. A detailed mpi communication model for distributed systems. *Future Gener. Comput. Syst.*, 22(3):269–278, February 2006. 2

[NB92] Peter Newton and James C. Browne. The code 2.0 graphical parallel programming language. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 167–177, New York, NY, USA, July 1992. ACM. 8, 15

[NK99] Stankovic Nenad and Zhang Kang. Visual programming for message-passing systems. *International Journal of Software Engineering and Knowledge Engineering*, 9(3):397–423, August 1999. 8

[Pac96] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. 6

[QFRABC+13] J. L. Quiroz-Fabián, G. Román-Alonso, Jorge Buenabad-Chávez, Miguel Alfonso Castro-García, M. Aguilar-Cornejo, and Jorge Buenabad-Chávez. A graphical language for development of parallel applications. In *PDPTA-2013: International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA'13, pages 672–678, USA, July 2013. CSREA Press. 35

[Roz97a] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997. 17

[Roz97b] Grzegorz Rozenberg. *Hyperedge Replacement Graph Grammars*, pages 95–162. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997. 35

[SGK05] Kumara Sastry, David Goldberg, and Graham Kendall. *Genetic Algorithms*, pages 97–125. Springer US, Boston, MA, 2005. 49

[SSKF93] Christian Scheidler, Lorenz Schäfers, and Ottmar Krämer-Fuhrmann. Trapper: A graphical programming environment for industrial high-performance applications. In *Lecture Notes in Computer Science*, PARLE '93, pages 403–413. Springer Science + Business Media, June 1993. 8, 15

[SSKF95] Lorenz Schäfers, Christian Scheidler, and Ottmar Krämer-Fuhrmann. Trapper: A graphical programming environment for parallel systems. *Future Generation Computer Systems*, 11(4–5):351 – 361, August 1995. 8, 15

[Sta96] Staff. Using mpi-portable parallel programming with the message-passing interface, by william gropp. *Sci. Program.*, 5(3):275–276, August 1996. 6

[SZ97] N. Stankovic and Kang Zhang. Visual parallel programming with visper. In *High Performance Computing on the Information Superhighway*, HPC Asia '97, pages 541–546. IEEE, April 1997. 8

[SZ02] N. Stankovic and K. Zhang. A distributed parallel programming framework. *IEEE Trans. Softw. Eng.*, 28(5):478–493, May 2002. 8

Acrónimos

GG Gramática de Grafos.

GGRH Gramática de Grafos mediante el Reemplazo de Hiperaristas.

GPU Graphics Processing Unit.

LPV Lenguaje de Programación Visual.

MPI Message Passing Interface.

MPMD Multiple Program Multiple Data.

PBT Programación Basada en Texto.

PVM Parallel Virtual Machine.

SPMD Single Program Multiple Data.

VPPE Visual Parallel Programming Environment.

VPPL Visual Parallel Programming Languaje.

Glosario

GG Se refiere a la técnica de crear nuevos grafos.

GGRH Es una generalización de la las gramáticas libres de contexto para lenguajes visuales.

GPU Coprocesador dedicado al procesamiento de gráficos u operaciones de coma flotante.

Java-MPI Implementación de MPI en Java.

LPV Lenguaje que permite describir gráficamente tareas simultáneas y el cómo se comunican y sincronizan éstas.

Maestro-Esclavo Patrón de cómputo paralelo donde dos o más procesos se pueden ejecutar a diferentes velocidades.

MPI Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes.

MPMD Técnica para que múltiples procesadores autónomos ejecuten simultáneamente diferentes programas.

Pipeline Patrón de cómputo paralelo que permite un flujo lineal de datos entre un grupo de procesos.

PVM Biblioteca para el cómputo paralelo en un sistema distribuido de computadoras.

SPMD Técnica para que múltiples procesadores autónomos ejecuten simultáneamente el mismo programa en puntos independientes.

VPPE La herramienta Visual Parallel Programming Environment (VPPE) es un ambiente de desarrollo para generar programas VPPL.

VPPE-Ex Motor de ejecución del ambiente VPPE.

VPPE-List Estructura de datos que representa a un flujo de trabajo VPPL.

VPPEd Editor de VPPE.

VPPL El lenguaje Visual Parallel Programming Languaje (VPPL) es un lenguaje gráfico para desarrollar programas paralelos mediante la especificación de flujos de trabajo.

Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

ACTA DE DISERTACIÓN PÚBLICA

No. 00010
Matrícula: 2121800782

Lenguaje Visual para el Desarrollo de Programas Paralelos.

En la Ciudad de México, se presentaron a las 11:00 horas del día 4 del mes de septiembre del año 2019 en la Unidad Iztapalapa de la Universidad Autónoma Metropolitana, los suscritos miembros del jurado:

DR. RENE LUNA GARCIA
DRA. GRACIELA ROMAN ALONSO
DR. RENE MACKINNEY ROMERO
DR. AMILCAR MENESES VIVEROS
DR. RICARDO MARCELIN JIMENEZ

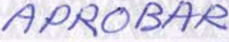
JOSE LUIS QUIROZ FABIAN
ALUMNO

Bajo la Presidencia del primero y con carácter de Secretario el último, se reunieron a la presentación de la Disertación Pública cuya denominación aparece al margen, para la obtención del grado de:

DOCTOR EN CIENCIAS (CIENCIAS Y TECNOLOGIAS DE LA INFORMACION)

DE: JOSE LUIS QUIROZ FABIAN

y de acuerdo con el artículo 78 fracción IV del Reglamento de Estudios Superiores de la Universidad Autónoma Metropolitana, los miembros del jurado resolvieron:



Acto continuo, el presidente del jurado comunicó al interesado el resultado de la evaluación y, en caso aprobatorio, le fue tomada la protesta.

REVISÓ

MTRA. ROSALIA SERRANO DE LA PAZ
DIRECTORA DE SISTEMAS ESCOLARES

DIRECTOR DE LA DIVISIÓN DE CBI
DR. JESUS ALBERTO OCHOA TAPIA

PRESIDENTE
DR. RENE LUNA GARCIA

VOCAL
DRA. GRACIELA ROMAN ALONSO

VOCAL
DR. RENE MACKINNEY ROMERO

VOCAL
DR. AMILCAR MENESES VIVEROS

SECRETARIO
DR. RICARDO MARCELIN JIMENEZ