PEM - Modelo de Ejecución Paralela basado en Redes de Petri

Wolfmann, Aaron Gustavo Horacio

Tesis para alcanzar el grado de
Doctor en Ciencias Informáticas

Director:
Ing. Armando De Giusti

Facultad de Informática
Universidad Nacional de La Plata

17 de Febrero de 2015

Índice de contenidos

1 Introducción **3**

2 Redes de Petri **11**

2.1 Introducción a las Redes de Petri 11

2.2 Redes de Petri Coloreadas 16

2.3 Estrategias para modelar algoritmos paralelos 23

2.4 Desplegado de Redes de Petri Coloreadas 31

3 Modelo de Ejecución Paralela **35**

3.1 Definición del Modelo de Ejecución Paralela 35

3.2 Diseño del Modelo 41

3.3 Implementación del Modelo 45

3.4 El uso del modelo PEM para el programador 48

3.5 Trabajos relacionados 51

4 Experimentación **61**

4.1 Algoritmo de Factorización de Cholesky 61

4.1.1 Ejecución Simulada 70

4.1.2 Ejecución en el *framework* 76

4.1.3 La máquina con procesadores AMD 76

4.1.4 La máquina con procesadores Intel 80

4.2 Multiplicación de Matrices 84

5 Conclusiones, impacto esperado y líneas abiertas de investigación 103

5.1 Conclusiones e impacto esperado 103

5.2 Líneas abiertas de investigación 107

A Archivos de configuración del *framework* 111

Prefacio de agradecimientos

Una profunda gratitud y agradecimiento a mi esposa Daniela por su eterna paciencia y compañía a lo largo de tantos años de sacrificio. También para mis hijos, a los que les he robado cientos de horas de mi dedicación hacia ellos para poder concluir con esta tarea. A mis padres y abuela, que aunque ausentes, no podría haber llegado hasta aquí sin su apoyo incondicional en etapas más tempranas de este proceso. Al resto de mi familia y amistades, por su apoyo y comprensión.

También un reconocimiento y gratitud a mi director, el Ingeniero Armando de Giusti, por haber eliminado los obstáculos que surgieron a lo largo de este proceso y su confianza ciega en mí para poder llegar a buen puerto. A los profesores a lo largo de la carrera de doctorado, y en especial a Fernando Tinetti, por las enseñanzas que aportaron a lo largo de esta carrera. También a los colegas, particularmente a Orlando Micolini, por el intercambio de ideas que dejaron profundas huellas en este trabajo. Por último, a todo el personal de la Facultad de Informática de la Universidad Nacional de la Plata, y en particular a las integrantes de la Secretaría de Posgrado, por facilitar enormemente cada trámite y tarea que fue necesario realizar, de forma eficiente y generosa hacia mi persona.

Capítulo 1

Introducción

La programación paralela es aquella orientada a generar programas de computación que se ejecuten en más de una unidad de procesamiento en forma simultánea. Su principal objetivo es generar código que tome menor tiempo de ejecución que el ejecutado por un único procesador. Por lo tanto no tiene razón de ser si no produce una mejora en el rendimiento de la ejecución, entendiendo por rendimiento, al tiempo que demora en ejecutarse todo el programa. Obviamente debe ser consistente en cuanto al resultado final de la ejecución, que debe ser el mismo, para uno o varios procesadores corriendo en paralelo.

La disciplina no es nueva. Su surgimiento es prácticamente contemporáneo con el nacimiento de la computación [Des87]. En su nacimiento, los equipos eran tan caros, que disponer de una computadora onerosa como la de aquellos días para ejecutar un solo programa, era ineficiente económicamente, por lo que la multiprogramación era algo natural en aquellos sistemas operativos. Los avances tecnológicos hicieron que hoy en día existan muy pocos equipos de procesamiento que dispongan de un único procesador.

Sin embargo, el objetivo de la mejora en el rendimiento no es simple de alcanzar. No se puede poner un programa hecho para correr en forma secuencial, en una máquina con más de un procesador y que en forma automática empiece a correr en paralelo. Si bien hay intentos en esa dirección, sus resultados no logran alcanzar el potencial que los procesadores paralelos brindan. Es necesaria la intervención de un programador para que transforme un programa secuencial en paralelo, o al menos lo desarrolle de esa forma desde el inicio, para obtener resultados acorde a los recursos utilizados.

El paralelismo, entendido como la capacidad de un programa de ejecu-

tarse en paralelo, no es fácil de lograr. Para ello debe tenerse en cuenta la ejecución simultánea del código en varios procesadores. Dejamos totalmente de lado aquellos programas que pueden dividirse en tareas que pueden realizarse independientemente una de otra. Quizás sea una forma ideal de paralelismo por su baja complejidad, pero no es de interés su investigación ya que, por un lado, es un problema de solución trivial, y por otro lado, es poco frecuente.

El desafío de la programación paralela surge cuando el algoritmo tiene dependencias internas que deben ser respetadas para que el resultado final pueda alcanzarse correctamente. El orden de ejecución de las instrucciones de un programa debe ser alterado o combinado en forma diferente al secuencial para ser ejecutado en paralelo con la restricción de que el resultado original se mantenga. Este es el concepto de división de tareas, una de las fuentes del paralelismo. Las tareas de un programa secuencial se dividen en subtareas que deben ser realizadas por más de un procesador en paralelo, sin alterar el resultado.

Además de la división de tareas existe otra fuente principal de paralelismo, y es la referida a la división de datos. Esta surge cuando el conjunto de datos es lo suficientemente grande para poder ser dividido y procesado por más de un procesador. En esos casos, los procesadores ejecutan el mismo código sobre diferente juego de datos, de forma tal que se gana tiempo de ejecución el poder procesar en paralelo.

De esta forma se llega a la taxonomía de Flynn de la arquitectura de procesadores paralelos [RR10], que está definida a partir de un cuadro de doble entrada, una dimensión para datos y otra para código, que determina cuatro formatos de arquitecturas: SISD, SIMD, MISD, MIMD. 'S' refiere a *single*, 'M' a *multiple*, 'I' a *instruction* y 'D' a *data*. Es claro que SISD no es paralelo, las restantes sí. La analogía entre la arquitectura de procesadores y la de programa paralelos es directa, salvando la distancia en que a nivel de programas, en lugar de instrucciones, se refiere a programas, y para los datos, se refiere a bloques de datos en lugar de un valor individual.

En general, un problema de programación paralela es una combinación de división de datos y división de tareas, es decir MPMD. Si fuera paralelo por uno solo de estos factores, el problema es de resolución relativamente simple. Solo división de tareas, implica analizar la estructura del algoritmo, particionar la ejecución y asignar a cada procesador. En el otro extremo, solo división de datos, o embarazosamente paralelo, tiene una solución del mismo tenor de complejidad, manteniendo el código igual en cada procesador,

asignado un juego de datos diferente a cada procesador.

Lo ciertamente complejo de un programa paralelo es la existencia de dependencias entre tareas combinado con la división de datos, ya que implica sincronismo en la ejecución. Un procesador inactivo no puede iniciar un cómputo hasta que las tareas anteriores, las cuales le aportan datos, hayan finalizado. Esta necesidad de sincronismo entre los procesadores es una de las causas principales de la pérdida de eficiencia en los programas paralelos. Se debe esperar hasta que el antecesor finalice para poder continuar, y dicha espera implica inactividad, justamente lo que se desea evitar para poder lograr un buen rendimiento.

Estudiar y analizar un algoritmo para detectar las dependencias y la forma de reorganizar las instrucciones para ganar en rendimiento, no es una tarea simple, ni existen métodologias desarrolladas para su realización. Si bien existe el clásico análisis de dependencias, RaR, RaW, WaR y WaW [RR10], aplicarlo a un algoritmo con un volumen de datos que justifique su ejecución en paralelo, generalmente miles de valores, resulta prohibitivo, por lo que este criterio debe aplicarse para dividir bloques de datos. Y la forma o patrón en que los datos sean divididos, es por ahora, una actividad dependiente del intelecto de cada persona.

Otro factor para alcanzar un buen rendimiento es el buen uso de la memoria *cache*. Los componentes de *hardware* utilizados para almacenar datos, las memorias, tienen un esquema cuyo tiempo de acceso corre paralelo con el costo de elaboración del componente. Siempre existen tiempos de demora en el acceso de a los datos por parte de un procesador, y es norma que a menor demora, mayor costo. Por lo que el almacenamiento de grandes cantidades de datos se realiza sobre memorias más económicas. Debido a que, como se dijo anteriormente, la programación paralela es conveniente para grandes volúmenes de datos, el esquema de acceso a estos es crucial para un buen rendimiento.

La memoria, como un todo, está compuesta por una jerarquía de componentes, los más rápidos más cercanos al procesador y los más lentos, más distantes, en términos de tiempos de acceso. Las memorias más rápidas se denominan memoria *cache*, ya que contienen copias de los datos existentes en nivel inferior. El esquema de acceso es desde un nivel superior a un nivel inferior. Cada vez que un dato es necesario, se busca en el nivel superior, y si no existe en ese nivel, genera una "falla de *cache*", que es resuelta buscando el mismo en un nivel inferior. Como el nivel inferior es más lento, una falla de *cache* genera inactividad en el procesador, por lo que, si bien las fallas

de *cache* no pueden evitarse, se debe tratar de reducirlas al mínimo posible [RR10].

El problema de las fallas de *cache* es pertinente a la programación en general y no exclusivo de la programación paralela. No obstante, este problema es exaltado cuando dos procesadores que comparten un componente físico de memoria *cache* trabajan sobre conjuntos de datos diferentes, por lo que terminan compitiendo por el uso del mismo. Una buena programación paralela debe tener un cuenta estos factores para obtener los rendimientos aceptables.

Los trabajos anteriores al inicio de las investigaciones que dieron lugar a esta tesis, presentan problemas en el rendimiento por causas del sincronismo. Tanto sea el caso del algoritmo para multiplicación de matrices en entornos de *clusters* de computadoras [WT08, TW08, TW09, WT09] o para equipos con múltiples procesadores [TW08, TW09], como también para el algoritmo de factorización de matrices de Cholesky [Wol10, TW11]. En todas estos trabajos, la fuente principal de inactividad de los procesadores es la espera por el sincronismo planteado en el algoritmo paralelo, se trate de primitivas del tipo *barrier* en memoria compartida [CJP07], o del tipo *broadcast* en memoria distribuida [SOHL+98]. Como se utilizaron bibliotecas de rutinas usuales de álgebra lineal, el código fuente de estas rutinas está fuera del alcance del programador. Lo que este puede controlar es la división de datos, de tareas, y su asignación a procesadores, por lo que el sincronismo es la variable a ajustar para mejorar rendimientos.

No se debe eludir, que la sincronización facilita el desarrollo de programas paralelos. Es más simple plantear que los procesadores trabajen en paralelo sobre diversos juegos de datos o sobre diversas tareas y que avanzan cada uno en su procesamiento hasta un determinado punto de ejecución en donde todos concluyen su tarea y a partir de allí se plantea una nueva distribución de cómputo paralelo. Dado que la programación paralela no resulta natural al programador que hace sus primeras experiencias en el tema, los puntos de convergencia en el procesamiento ayudan al razonamiento. Sin embargo, en trabajos previos, se comprobó que la inactividad por sincronismo se llevaba más del 50% del tiempo total de ejecución del algoritmo de Cholesky [Wol10, TW11]. Este hecho fue determinante para buscar una opción a este forma de programación paralela.

Eliminar el sincronismo en la ejecución paralela implica que el avance de la ejecución en cada procesador paralelo es independiente del avance de sus pares, lo cual implica la necesidad de un elemento coordinador del proce-

samiento para que el algoritmo paralelo como un todo se ejecute correctamente. El coordinador debe determinar que tareas están realizadas, cuales pendientes, cuales pendientes y habilitadas, y cuales pendientes pero a la espera de la resolución de dependencias de datos. El coordinador debe asignar las habilitadas a algún procesador inactivo. El modelo *"Master / Slave"* es el ejemplo más común de esta forma de programación [KGGK94]. En este modelo, existe un proceso llamado *"Master"* encargado de definir y asignar las tareas a cada proceso *"Slave"* para su realización.

Un problema habitual en este modelo es la sobrecarga del proceso *"Master"*, que al ser secuencial, debe atender, de a uno por vez, a cada *"Slave"* inactivo para asignarle la tarea. Además, si los datos no están distribuidos entre los esclavos, aumenta la sobrecarga al tener que comunicar los datos a los procesadores trabajadores para que puedan realizar su trabajo. La concentración en uno o más procesos coordinadores suele generar inactividad por espera en los esclavos. Por otro lado, en caso de disponer pocos procesadores paralelos, puede ser muy gravoso a los fines del rendimiento general, asignar un procesador exclusivamente a la tarea de administración.

Otro modelo asíncrono es el denominado *"Peer to peer"*. Está conformado por una red de computadoras que pueden actuar como servidores o como clientes, proveyendo o solicitando servicios. Su finalidad principal es evitar la concentración de servicios sobre un servidor, que en caso de desperfectos o fallas, produce la caída del mismo. El modelo plantea que al trabajar en forma distribuida, la servicialidad se mantiene, ya que siempre existirá un equipo "par" (*peer*) que pueda suplantar al caído [CLR09]. Es un modelo que tiene una orientación al sostenimiento de servicios más que a la programación y ejecución paralela de algoritmos.

La teoría de la planificación o *Scheduling* como es más conocida, estudia la asignación de recursos limitados a actividades en el tiempo, con la finalidad de optimizar en algún sentido, el uso de los recursos. Existe una rama que estudia el *scheduling* de procesos de cómputos, entendiendo por recurso a asignar, no solo a los procesadores, sino no también a los componentes de sistemas de comunicaciones [RV09]. Las herramientas desarrolladas en este dominio alcanzan un óptimo en el uso de los recursos, lo cual tiene un costo computacional. Si este costo es inaceptablemente alto, se buscan técnicas que permitan alcanzar un subóptimo.

Desde el punto de vista del autor, la teoría de *scheduling* obtiene resultados que son puntualmente insuperables, pero es una teoría poco útil para la mayoría de los casos de programación paralela. Para alcanzar el óptimo

es necesario precisar, el algoritmo (el uso de los recursos, en términos de *scheduling*) y los recursos. La realidad, es que para un algoritmo dado, con un tamaño de datos dado, con una máquina paralela dada, con un número dado de procesadores, determinando la velocidad del reloj de estos, el ancho de banda de la red que interconecta las computadoras y una cierta cantidad de memoria distribuida en un número fijo de bancos, la teoría de *scheduling* puede determinar el óptimo. Es decir, es necesario fijar todas las condiciones particulares para calcularlo. En términos de tiempo de ejecución, es un *scheduler* estático.

Lo normal es que las condiciones no sean estáticas y que los algoritmos sean distintos, el volumen de datos a computar por el algoritmo cambie, la partición de estos también, la máquina tenga distinto número de procesadores, los canales de memoria y de comunicaciones tengan distintos anchos de banda, etc. Es decir, las condiciones que permiten calcular un óptimo, no son estables, por lo que el cálculo del óptimo es materialmente imposible para todos y cada uno de estos casos.

Es esperable que el *scheduler* de tareas sea dinámico y que evite una sobrecarga de procesamiento tal que no justifique su utilización. Siguiendo este criterio, es probable que no se alcance el óptimo, ya que se dejan de lado los elementos necesarios para calcularlo, pero desde un punto de vista práctico, un *quasi*-optimo rápidamente factible es mejor que un óptimo con un costo computacional imposible.

Otra técnica usual en la programación paralela es el uso de los diagramas de dependencia. Estos parten de una descomposición del algoritmo en tareas y generan un grafo dirigido cuyos vértices representan tareas y las aristas, la dependencia que existe entre dos tareas. Como el grafo es dirigido, la arista va en sentido de la tarea habilitada [BLKD07]. Esta estructura de datos significa una mejora respecto de utilizar otras estructuras para representar a un algoritmo, como es el caso de las tablas. El uso de estas para determinar dependencias es engorroso cuando el número de tareas es elevado. Sin embargo, el mecanismo de *scheduling* de las tareas habilitadas entre los procesadores inactivos no forma parte de este modelo de representación del algoritmo.

Tomando en cuenta todos estos antecedentes, se estudió la posibilidad de utilizar Redes de Petri [Dia09] para modelar un algoritmo paralelo. Este tipo de redes se destacan para modelar sistemas concurrentes, ya que en su semántica, la ocurrencia de eventos que modifican el estado del modelo, puede ser en simultáneo, a diferencia de las máquinas de estado finito (FSM), cuyo cambio de estados sucede de a uno por vez [HMU03]. Otra ventaja que

presentan la Redes de Petri, es su representación gráfica, que semánticamente es clara, y permite visualizar gráficamente una red compleja con simplicidad. También, como es una herramienta analítica, es posible analizar la correctitud del modelo con los requerimientos y objetivos del mismo.

Las Redes de Petri también permiten identificar dentro de su modelo, lo que son los eventos, por medio de las Transiciones y por otro lado, lo que son condiciones para que dicho evento puede ocurrir, las Plazas. De esta forma, un algoritmo puede ser representado con mayor claridad a un grafo de dependencias, ya que se presentan y distinguen claramente los elementos que lo componen, a saber, las tareas, por medio de la Transiciones, y los datos, por medios de las Plazas. Estas últimas, a su vez, pueden actuar de nexo entre dos tareas, si son el resultado de una tarea y la entrada de otra. De esta forma, la dependencia de datos se representa naturalmente. Si a estas facilidades le sumamos su inmejorable capacidad de representar concurrencia, utilizar estas redes para el modelado de algoritmos paralelos es indiscutido.

Sin embargo, el camino de representar algoritmos paralelos con Redes de Petri no está exento de inconvenientes. El primero, y más sobresaliente, es el rápido crecimiento de la red al agregar tareas. Una red extensa es difícil de interpretar gráfica y analíticamente. Es habitual que los algoritmos puedan dividirse en numerosas tareas, y más aún si hay división de datos en su ejecución, por lo que el número de tareas a representar crece rápidamente. El modelo del algoritmo debe ser lo suficientemente claro para poder ser entendido y controlado por parte del programador, de lo contrario, no es útil.

La ejecución de la red, o algoritmo para los fines de esta tesis, tampoco es implícito en la Red de Petri. La semántica de ejecución, puede ser paralela, pero también puede ser secuencial, dependiendo del modo en que quiera ser utilizada. Este factor, junto con el presentado en el anterior párrafo, fueron los principales desafíos que hubo que enfrentar en el desarrollo de esta tesis para lograr que el modelado del algoritmo pueda ser simple, y a su vez, pueda servir de base para una ejecución paralela.

Es por todo esto que el **objetivo de la tesis** es la definición de un modelo de ejecución paralelo que basado en la representación de un algoritmo paralelo con Redes de Petri, permita a un conjunto flexible de procesadores independientes entre sí, ejecutar el algoritmo en forma asíncrona con altos rendimientos y que el programador tenga capacidad de ajustar los parámetros de ejecución en vista de mejoras de rendimiento.

Los fundamentos son claros: se desea contar con una herramienta de eje-

cución de programas paralelos que permita modelar el algoritmo, y pasar del modelo a la ejecución asíncrona preservando el modelo. Las Redes de Petri son la herramienta básica e indiscutiblemente pertinente para lograr el objetivo. Un desafío es cubrir la brecha o *gap* existente entre el modelado y una ejecución del programa paralelo de rendimientos aceptables y escalables. Para ello, debe existir una vinculación del modelo con un conjunto de unidades de procesamiento que corran en paralelo.

En la bibliografía vigente no fueron hallados ejemplos que usen a las Redes de Petri como herramienta de modelado y como motor de ejecución sobre procesadores reales de computación en forma simultánea. Existen numerosos desarrollos de simulación, pero no de ejecución directa en máquinas físicas que puedan obtener un alto rendimiento. Para alcanzar estos rendimientos, es necesario además, que el *scheduling* de las tareas planteadas en el modelo sea dinámico y flexible por razones de eficiencia.

El contenido de la tesis es dividido en pocos capítulos: en el capítulo 2 se presentan los conceptos básicos de Redes de Petri utilizados a lo largo de este documento, las características del modelado de algoritmos con este tipo de redes y un ejemplo sobre un algoritmo específico, el cual será utilizado posteriormente. En el capítulo 3 es completado el modelo de programación paralela planteado, agregando componentes orientados a la ejecución paralela del algoritmo modelado. En el capítulo 4 se presentan los experimentos y trabajos realizados que convalidan el modelo planteado, con diferentes algoritmos, bibliotecas de rutinas y máquinas paralelas, y finalmente, en el capítulo 5 se presentan las conclusiones, el impacto esperado y la líneas de investigación abiertas de esta tesis.

Capítulo 2

Redes de Petri

El presente capítulo presenta los conceptos básicos sobre Redes de Petri. No ha sido realizado con la intención de ser un tratado completo del tema, y muchas propiedades importantes de las Redes de Petri no han sido incluidas aquí. Son presentados solo los conceptos necesarios para el resto del trabajo. Se incluye además una sección que expone la estrategia utilizada para modelar algoritmos con alto nivel de abstracción, y sobre cómo convertir este modelo abstracto en otro modelo más simple que permita ser ejecutado en paralelo facilmente.

2.1 Introducción a las Redes de Petri

Una Red de Petri Net (PN) es un grafo dirigido bipartito compuesto por nodos llamados Plazas y Transiciones. Usualmente las Plazas representan "estados" y las Transiciones "acciones". El conjunto de Plazas se denota como P y el conjunto de Transiciones como T. El grafo es bipartito ya que los conjuntos P y T son disjuntos, $P \cap T = \emptyset$. Los arcos del grafo siempre conectan una Plaza con una Transición o vice versa. El conjunto de arcos A es definido como $A \subseteq (P \times T) \cup (T \times P)$. Los arcos son denominados de entrada o de salida dependiendo si la dirección del arco es de una Plaza a una Transición o en sentido contrario, respectivamente [Dia09, IA06].

La PN posee *tokens* que existen solamente en las Plazas, y generalmente representan "hechos". La función de marcado μ es una función, $\mu : P \to \mathbb{N}$ que determina el numero de *tokens* existentes en cada Plaza. El Vector de Marcado es un vector $M \in \mathbb{Z}^{1\times|P|}$ donde $M[i] = \mu(i)$ y que denota el numero

de *tokens* en cada Plaza. En este modelo básico, todos los *tokens* son iguales y fungibles. El estado inicial de un Vector de Marcado es denominado como M_0.

El *preset* de una Transición t es el conjunto de Plazas de entrada de t, y es definido como $\bullet t = \{p \in P : \forall(p,t) \in A\}$. De igual manera es definido el *postset* de t como $t\bullet = \{p \in P : \forall(t,p) \in A\}$. Ambos conceptos determinan el conjunto de Plazas de entrada y de salida de una Transición t, respectivamente.

Una Transición t es llamada "habilitada" cuando todos sus Plazas de entrada tienen *tokens*, es decir, $M[p] > 0$, para todo $p \in \bullet t$. Hasta aquí ha sido implícito que los arcos del grafo tienen un peso de 1, tanto si el arco es arco (p,t) o (t,p). El peso de un arco $a \in A$ es definido por la función $w(a) : A \to \mathbb{N}$, lo cual determina que el peso puede ser un número natural positivo. Un valor mayor que uno representa que es necesario más de un *token* para habilitar la Transición, tantos como sea el peso. Así, una Transición t queda habilitada si $M[p] \geq w(p)$, para todo $p \in \bullet t$.

El estado general de la red evoluciona cuando una Transición es "disparada". En el modelo básico de PN, una Transición es automáticamente disparada si está habilitada. Esto implica movimiento de *tokens* desde las Plazas de entrada hacia las Plazas de salida. Este movimiento es realizado mediante las operaciones de absorción e inyectado de *tokens* en las Plazas de entrada y salida respectivamente.

Cuando una Transición t es disparada, el Vector de Marcado es actualizado de la siguiente forma:

$$M'[p] = \begin{cases} M[p] - w(p) & p \in \bullet t \\ M[p] + w(p) & p \in t\bullet \\ M[p] & default \end{cases}$$

lo cual es expresado con la notación $M \xrightarrow{t} M'$.

El conjunto de estados de M que se puede alcanzar a partir de los disparos realizados desde el estado inicial M_0 es llamado Marcado de Alcanzabilidad (*Reachability Marking*), $\Re(M_0)$, y es definido como:

$$\Re(M_0) = \{M' | M_0 \xrightarrow{*} M'\} \tag{2.1}$$

donde $\xrightarrow{*}$ representa la operación de clausura transitiva de disparos sobre

el conjunto de Vectores de Marcado. El Grafo de Alcanzabilidad es el grafo formado por la relación de disparo entre Vectores de Marcado. En otras palabras, es el grafo que representa todos los valores alcanzables para M, por sucesivos disparos a partir de un estado inicial M_0.

Las Redes de Petri tienen muchas propiedades matemáticas importantes. Se destacan tres de ellas:

1. El problema de la *alcanzabilidad* es decidir si el marcado M'' puede ser alcanzado desde el estado inicial M_0 por una serie de disparos.

2. El problema de la sobrevivencia (*Liveness*). Un marcado sin Transiciones habilitadas es llamado muerto. Una PN es pseudo-viva, si $\forall M \in \Re(M_0) \exists t \in T : t$ está habilitada, lo que significa que todos los marcados alcanzables tienen una Transición que puede ser disparada. Una PN es quasi-viva si $\forall t \in T, \exists M \in \Re(M_0) : t$ está habilitada en M, lo que significa que todas las Transiciones tienen al menos un marcado en que están habilitadas. Una PN es viva si $\forall M \in \Re(M_0)$ y $\forall t \in T, \exists M' \in \Re(M) : t$ está habilitada M', lo que significa que para todos los marcados alcanzables, todas las Transiciones quedarán habilitadas en al menos un marcado posterior.

3. El problema de la delimitación *Boundedness*. Una PN es delimitada si todas sus Plazas tienen a lo máximo k *tokens* en todos los marcados alcanzables, (también denominado *k-bounded*). Una PN es segura si es *1-bounded*.

La teoría de grafos enseña que el conjunto de arcos A puede ser representado por medio de una matriz [AHU83], llamada la Matriz de Incidencia. Una Matriz de Incidencia $D \in \mathbb{N}^{|P|\times|T|}$, representa cada Plaza en una fila y cada Transición en una columna, y los valores en la matriz son $w(a)$ o cero según el arco $a = (p, t)$ o $a = (t, p)$, pertenezca o no a A.

Se definen dos Matrices de Incidencia: D^- and D^+, llamadas Matriz de Incidencia Negativa y Positiva respectivamente, y representan los valores de los pesos de las Plazas de entrada y de salida . Extendiendo la notación, $a^-_{(i,j)}$ es el arco de la Plaza i en $\bullet j$ y $a^+_{(i,j)}$ es el arco i en $j\bullet$. De esta forma, los valores en D^- y D^+ son definidos como:

$$D^-_{(i,j)} = w(a^-_{(i,j)})$$
$$D^+_{(i,j)} = w(a^+_{(i,j)})$$

Frecuentemente se define una Matriz de Incidencia única $D = D^+ - D^-$, pero para el resto de este documento, se considerará al par de matrices D^+ y D^-, por razones que serán vistas más adelante.

Existen varias definiciones matemáticas de una Rede de Petri [IA06, Dia09, JK09], de las cuales se elije la más apropiada a los fines del trabajo. Por ello se define a una *Token Petri Net* (TPN) como la n-upla:

$$TPN = (P, T, D^-, D^+) \tag{2.2}$$

siendo P, T, D^- y D^+ lo definido anteriormente. Una Red de Petri Marcada es el par (TPN,M).

Siguiendo la definición basada en Matrices de Incidencia, para determinar el conjunto de Transiciones habilitadas en un marcado M, solo se necesitan operaciones elementales del álgebra lineal. En efecto, si D_j^- y D_j^+ representan la j-ava columna (Transición) en D^- y D^+ respectivamente, la Transición j está habilitada si la diferencia vectorial $I_j = M - D_j^-$ no tiene ningún valor negativo, es decir, todas las Plazas de entrada de la Transición j tiene suficientes *tokens* para satisfacer su disparo.

El Vector de Marcado M determina el estado de la red. Dado un estado particular, el conjunto de Transiciones habilitadas E es definido como:

$$E = \{j | I_j = M - D_j^- \wedge \forall k = 1 \ldots |P| : I_j(k) \geq 0\} \tag{2.3}$$

El comportamiento de la red puede ser visto por medio de la secuencia de disparos de Transiciones. Varias semánticas han sido definidas para los disparos, según el número de Transiciones que puedan ser disparadas en forma simultánea. En un extremo de estas opciones solo una Transición del conjunto E puede ser disparada a la vez, y en el extremo opuesto, se disparan todas en paralelo. Las opciones intermedias también son válidas.

La semántica que permite disparar solo una Transición del conjunto E define un comportamiento serial de la red, anulando la propiedad má importante de las PN: la modelización de la concurrencia. Por ello, la semántica más frecuentemente usada es la del disparo de todas las Transiciones habilitadas en simultáneo. Muchas de la propiedades matemáticas de las PN son obtenidas a partir de esta semántica.

El modelo matemáticas de las Redes de Petri puede ser visto como una evolución de los Autómatas de Estado Finito (*Finite State Automata - FSA*)

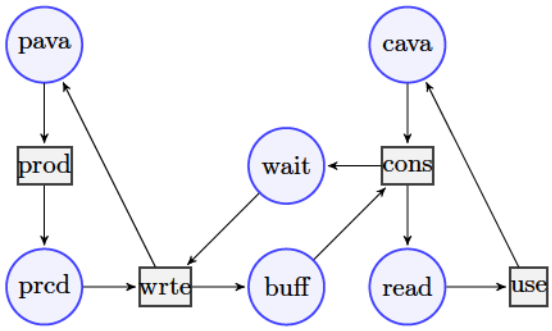


Figure 2.1: Red de Petri que modela un sistema de un productor y un consumidor, con un *buffer* limitado intermedio.

[HMU06]. El FSA permite modelar un conjunto de estados por medio de un conjunto de símbolos de entrada que cambian el estado del modelo a partir de una función de transición. La principal diferencia con las TPN es que los cambios en una FSA son hechos de a uno a la vez y la única forma de representar concurrencia es usando el producto cartesiano de todos los estados posibles. Es evidente la complejidad de un análisis de un modelo con un número grande de estados, dado el carácter exponencial del número de combinaciones posibles. El rol del *token* en las TPN es de simplificar este número combinatorio de posibles estados por medio de su representación.

Un ejemplo de una TPN es graficado en la Fig. 2.1, que representa un modelo típico de productor / consumidor con un *buffer* intermedio. La Plaza *pava* representa el estado de disponibilidad del productor, y similarmente para la Plaza *cava*, para el consumidor. La Transición *prod* representa la acción de producir, cuya Plaza de salida representa la existencia de un bien producido. A continuación, la Transición *wrte* representa el hecho de transmitir el bien producido al *buffer*, cuyas salidas representan el retorno al estado de disponibilidad para el productor y el objeto en el *buffer*. El consumidor, estando disponible, espera a que el productor envíe el objeto al *buffer*. Una vez sucedido esto, el consumidor lo toma por medio del disparo de la Transición *cons*, y luego lo usa y vuelve al estado de disponibilidad.

La TPN recién descripta permite tener un *buffer* de tamaño mayor a uno. En efecto, si la Plaza *wait* tiene n *tokens*, entonces el productor puede enviar hasta n objetos antes de que el consumidor comience a tomarlos. Un impacto diferente es causado en la TPN si se dispone de más de un productor o consumidor sobre el mismo *buffer*. La Fig. 2.2 ilustra el caso de una TPN con dos productores y dos consumidores que envían y toman del

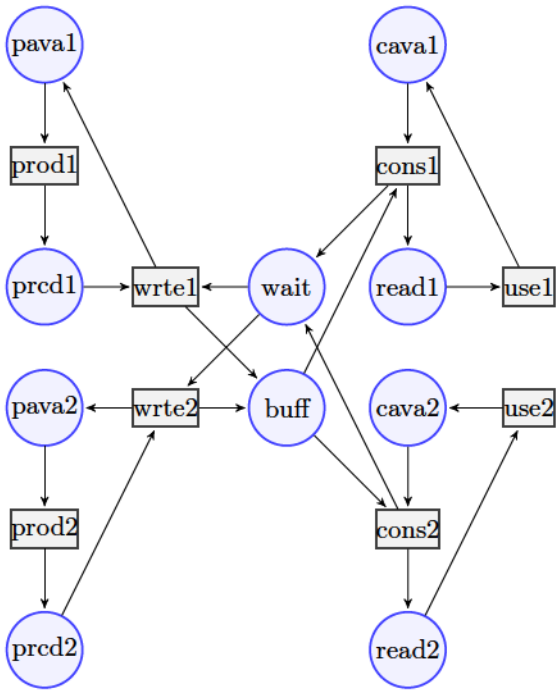


Figure 2.2: TPN que modela un sistema de dos productores y dos consumidores, con un *buffer* limitado intermedio.

mismo *buffer*. Esta red tiene semejanzas con la anterior, pero difiere en que hay tantas Transiciones *wrte* y *cons* como productores y consumidores sean modelados, respectivamente. Es fácil extrapolar este modelo a una única cola de impresión, alimentada por varias computadoras y con varias impresoras disponibles para la cola.

Cuando el número de productores y consumidores crece, se agrava el modelado con este tipo de Redes de Petri, complicando además la lectura e interpretación del grafo. En la próxima sección es presentado un tipo de Redes de Petri que simplifica este problema, permitiendo generalizar el modelo.

2.2 Redes de Petri Coloreadas

Las TPN tienen la propiedad de que todos los *tokens* son equivalentes, por lo que no existen variaciones entre ellos. Sin embargo, hay muchas situaciones donde es necesario distinguir un *token* de otro porque representas diferentes hechos. Este caso puede ser resuelto con TPN usando Plazas diferentes para

diferenciar los *tokens*, definiendo tantas Plazas como diferentes *tokens* deben ser representados. Esta estrategia genera redes con un gran número de Plazas y Transiciones, dificultando la comprensión y el análisis sin la ayuda de una herramienta informatizada.

Las Redes de Petri Coloreadas (*Coloured Petri Net - CPN*) han evolucionado como una clase de Redes de Petri de Alto Nivel [Dia09] que permiten modelar problemas complejos en forma compacta. La propiedad que más resalta de las CPN es que los *tokens* pueden tener diferencias entre ellos. El concepto matemático de dominio es necesario para definir el "color" en las redes. En CPN cada Plaza se relaciona con un dominio, y así, los colores representan el dominio asociado a la Plaza, por lo que los *tokens* presentan un color. De esta forma, los *tokens* dejan de ser fungibles como en TPN, y un color simboliza un valor que cada *token* tiene en su dominio.

El hecho de que cada Plaza tenga un dominio relacionado causa que el marcado sea diferente al de las TPN. Aquí es necesario determinar, no solo cuantos *tokens* tiene una Plaza, sino también que color tiene cada *token*, y el número de repeticiones para cada color. La teoría matemática de *multiset* o bolsa es usada para representar los *tokens* en una Plaza. Sea S un conjunto no vacío, $S = s_1, s_2, \ldots, s_n$. Un *multiset* es una función $m : S \to \mathbb{N}$, que relaciona cada elemento de S con un número natural; $m(s) \in \mathbb{N}$ es llamado el número de repetición de s. Cada Plaza p tiene una función de *multiset* asociada, m_p.

Un concepto adicional en las CPN es el de la guarda de cada Transición. En TPN una Transición está habilitada si tiene suficientes *tokens* en sus Plazas de entrada. En CPN, como cada Plaza tiene su dominio asociado con múltiples valores, la condición que habilita una Transición debe ser más específica. La guarda es una expresión lógica que determina cuando una Transición queda habilitada y cuales son los *tokens* que deben ser absorbidos de cada Plaza de entrada.

La definición de una Red de Petri Coloreada (CPN) usada ha sido adaptada de los autores Jensen [JK09] y Diaz [Dia09]. Una Red de Petri Coloreada es una n-upla:

$$CPN = (P, T, A, \Sigma, V, C, G, E) \tag{2.4}$$

siendo:

- P es un conjunto finito de Plazas.

- T es un conjunto finito de Transiciones, cumpliendo $P \cap T = \emptyset$.
- A es un conjunto de arcos dirigidos, $A \subseteq P \times T \cup T \times P$.
- Σ es un conjunto finito de dominios no vacíos (colores).
- V es un conjunto finito de variables con tipo v, siendo el tipo de v: $type(v) \in \Sigma$.
- $C : P \to \Sigma$ es la función de coloreado, que asigna un color a una Plaza.
- $G : T \to Expr_v$ es la función de guardas, que asigna una expresión de tipo "booleano" $Expr_v$ a una Transición t. $Expr_v$ denota una expresión que utiliza un lenguaje de inscripciones que decora la red.
- $E : A \to Expr_v$ es la función de expresiones que asocia cada arco con una expresión $Expr_v$, cumpliendo $type[E(a)] = C(p)$, el color de la Plaza que conecta el arco a.

Es necesario definir algunos conceptos adicionales. Sea $C(p)_{ms}$ el conjunto de todos los *multisets* del color de la Plaza p.

- El marcado es una función μ que relaciona cada Plaza con un *multiset* de *tokens*, $\mu(p) \subseteq C(p)_{ms}$.
- μ_0 define el marcado inicial de la red.
- Las variables de la Transición t son denotadas como $Var(t) \subseteq V$, que son las variables libres que aparecen en la guarda de t y en las expresiones de todos los arcos conectados a t.
- La atadura (*binding*) de una Transición t es una función b que mapea cada variable de $v \in Var(t)$ con un valor $b(v) \in Type(v)$. Una atadura es denotado por $\langle v_1 = val_1, v_2 = val_2, \ldots, v_n = val_n \rangle$, para las n variables de la Transición t que componen $Var(t)$. $B(t)$ denota todas las ataduras una Transición t.
- Un elemento de atadura (*binding element*) es un par (t, b) tal que $t \in T$ y $b \in B(t)$. El conjunto de todos los elementos de atadura de la Transición t es definido como $BE(t) = \{(t, b) | b \in B(t)\}$.
- BE define el conjunto de todos los elementos de atadura en una CPN.
- Un paso $Y \in BE_{ms}$ es un *multiset* finito y no vacío de elementos de atadura.

Para determinar cuando una Transición está habilitada, deben cumplirse dos condiciones. Primero, la guarda de la Transición debe ser cumplida. Esto es representado por $G(t)\langle b\rangle$, la guarda de t, evaluada con la atadura b debe ser verdadera. Segundo, deben existir suficientes *tokens* en cada Plaza de entrada que satisfagan la atadura b. Sea (p,t) que representa un arco a que es un arco de entrada de t desde p, y (t,p) un arco de salida desde t hacia p. $E(p,t)\langle b\rangle$ especifica el *multiset* de *tokens* requerido en p para que t quede habilitada por medio de la atadura b. Ambas condiciones son expresadas como:

$$G(t)\langle b\rangle = true \tag{2.5}$$

$$\forall p \in P : E(p,t)\langle b\rangle \ll= \mu(p) \tag{2.6}$$

donde $\ll=$ representa la relación de menor o igual en *multisets*. En otras palabras, debe existir al menos una cantidad de *tokens* en todas las Plazas de entrada de t que satisfaga la expresión de arco $E(p,t)$ evaluada con la atadura b.

Un disparo sobre una Transición habilitada produce un nuevo marcado μ' definido como:

$$\mu'(p) = (\mu(p) - -E(p,t)\langle b\rangle) + +E(t,p)\langle b\rangle \tag{2.7}$$

para todo p, donde $--$ y $++$ denotan las operaciones de sustracción y de adición sobre *multiset* respectivamente, $E(p,t)\langle b\rangle$ representa los *tokens* eliminados en todas las Plazas de entrada de t, y $E(t,p)\langle b\rangle$ los *tokens* inyectados en sus Plazas de salida, ambos evaluados para la atadura b.

Como se señaló anteriormente, un paso Y es un *multiset* de elementos de atadura. Es requerido que cada elemento de atadura (t,b) incluido en Y cumpla con la guarda de t, y además, debe ser capaz de eliminar todos los *tokens* sin interferir con los *tokens* de otros elementos de atadura del paso. Cuando ocurre un paso Y, un nuevo marcado es producido, lo que se denota como:

$$\mu(p) \rightarrow \mu'(p) \tag{2.8}$$

y también, cuando ocurre una secuencia finita de pasos, $Y_1, Y_2, \ldots, Y_n$, el estado del marcado es:

$$\mu \rightarrow \mu' \ldots \rightarrow \mu^n \tag{2.9}$$

lo cual define el conjunto de marcados alcanzables desde μ:

$$\Re(\mu_0) = \mu_i \rightarrow \mu_{i+1} \tag{2.10}$$

para $i = 0 \ldots n-1$.

Los conceptos presentados hasta este punto son solo una pequeña parte del conocimiento desarrollado sobre las CPN, pero son suficientes para justicar la importancia de este tipo de redes en la modelización de algoritmos paralelos. Dado que la partición de datos y la división de tareas son los conceptos centrales en la programación paralela, y que los altos rendimientos de procesamiento son alcanzados por el uso de cientos o miles de procesadores, sea hace necesario una herramienta de modelización que permita administrar un gran número de divisiones de datos y tareas.

Las funcionalidades de las CPN descriptas hasta aquí hacen que este tipo de redes sean una herramienta excelente para modelar un algoritmo, y también para analizar las características paralelas presentes en el, permitiendo una representación compacta del mismo.

Estos factores motivan el uso del modelo matemático de CPN para representar los algoritmos. Dos conceptos son fundamentales en esta tesis. Uno es la representación de la división de datos por medio de los colores. Así, los valores de los dominios son las etiquetas que identifican los bloques de datos usados en un programa paralelo. El segundo concepto es la representación del procesamiento por medio de las Transiciones, las cuales son utilizadas para representar las rutinas del algoritmo.

De esta manera, las Plazas de una CPN y los dominios son utilizados para representar datos, y los colores, sus divisiones. Sin importar cuantas divisiones sean necesarias, la CPN que represente al algoritmo será la misma, difiriendo solamente en el valor del parámetro que defina el número de divisiones. Las Transiciones que conformen la CPN son independientes del esquema de división de datos. Las funciones de guarda, que representan las condiciones de los algoritmos, también pueden ser paramétricas. De esta forma, la estructura del algoritmo es invariante al número variable divisiones de datos.

La dependencia de datos entre tareas se representa por medio de las Plazas que tienen el rol de ser salida y entrada de Transiciones, de forma tal que

Figure 2.3: Red de Petri Coloreada que modela un sistema de múltiples productores / consumidores, con un *buffer* limitado intermedio.

el dato producido por una tarea necesario para otra es representado por un *token* en una Plaza, que tiene un doble rol, al convertirse de salida de una tarea en entrada de otra, generando una relación de dependencia entre ambas tareas. Las tareas que no tengan dependencia entre sí, pueden ser disparadas (o ejecutadas en términos de algoritmos), en paralelo por el modelo en forma natural.

Un ejemplo de CPN es presentado en la Fig. 2.3, que modela el caso del productor / consumidor de la sección previa, usando ahora CPN, lo cual permite parametrizar el número de productores y de consumidores en el mismo modelo. El principal cambio es la definición de los dominios de cada Plaza, que son la clave para la extensión del modelo de TPN en CPN. Existen ahora tres dominios, uno para productores, $< x >$, otro para consumidores, $< y >$, y el último para el tamaño del *buffer*, $< z >$, cada uno de los cuales es implementado como un número natural desde 1 to p, c y b respectivamente, denotando el número de integrantes de cada dominio. No es necesario el uso de repeticiones (*multiset*) en este caso.

En resumen, los dominios usados para cada Plaza son:

- Plazas *pava* y *prdc*: $X = \{< x >\}, x = 1 \dots p$
- Plazas *cava* y *read*: $Y = \{< y >\}, y = 1 \dots c$
- Plazas *wait* y *buff*: $Z = \{< z >\}, z = 1 \dots b$

La dinámica de la red es la siguiente. La Plaza *pava* tiene inicialmente p *tokens*, numerados como $1 \dots p$ y que representan a cada productor. Igual para la Plaza *cava*, de $1 \dots c$, y para la Plaza *wait* para cada *buffer* disponible. Una vez que el productor x produce un objeto, el *token* x va a la Plaza *prcd*; cuando se dispara *wrte* el *token* x vuelve a la Plaza *pava* y z es enviado a la Plaza *buff*. Esto permite disparar *cons* que mueve el *token* z hacia la Plaza *wait* nuevamente, utilizando un consumidor disponible y, el cual es inyectado en *read*. Luego de su uso, el *token* y retorna a la Plaza *cava*, concluyendo el ciclo.

Este caso es elemental y no necesita del uso de guardas, pero en un caso hipotético, por ejemplo si hubiera una restricción en la combinación de pro-

ductor / consumidor, la guarda debería estar en la Transición *urte* expresando la restricción requerida.

Es destacable la simplicidad en la notación gráfica de la CPN, facilitando el análisis de la red, independientemente del número de integrantes que formen el modelo. Comparado con la TPN expuesta en la Fig.2.2 que solo tiene dos productores y dos consumidores, la CPN es más simple, clara y extensible.

A esta altura del desarrollo es fácil de resaltar una de las principales facultades de las CPN: el alto nivel de expresividad que permite modelar situaciones complejas en una representación compacta. Esto es particularmente relevante para modelar algoritmos con bucles, cuyo número de ciclos y componentes es paramétrico.

La llave del modelado con CPN es la definición de los dominios y de la asignación de estos a las Plazas. En efecto, si se define correctamente el dominio de cada Plaza, el modelo se simplifica. Es la tarea más desafiante en el proceso de modelado, la cual, si es hecha en forma lúcida, un modelo complejo se vuelve simple.

Sin embargo, la alta expresividad de las CPN produce un impacto negativo en el objetivo de este trabajo. La ejecución de un algoritmo paralelo representado por una red no puede ser hecha en forma eficiente directamente desde el modelo de CPN. El rendimiento se ve afectado por la sobrecarga computacional que el modelo expresivo agrega. La representación de dominios en un programa de computación es compleja dada la generalidad de elementos que un dominio puede contener, haciendo necesario el uso de estructuras adicionales, como por ejemplo los diccionarios. Además, las funciones de guardas de Transiciones con más de una Plaza de entrada deben ser evaluadas en todos las combinaciones posibles que los *tokens* puedan formar, y elegir una que evalúe verdadera a la expresión, suma otra una carga computacional.

La implementación computacional de una CPN implica una tarea compleja que es comparable con la definición e implementación de un lenguaje de programación: es necesario definir tipos y realizar *parsing* y evaluaciones de expresiones. Esta complejidad de funcionamiento genera una sobrecarga computacional en tiempo de ejecución, que debe ser eludida para lograr obtener rendimientos altos en la ejecución.

A pesar de estas características negativas, desarrollar un modelo con CPN respetando ciertas restricciones tiene ciertas ventajas que pueden ser utilizadas para transformar una CPN en una TPN, y reducir los problemas ex-

puestos. Las dos próximas secciones presentan las restricciones y condiciones para modelar un algoritmo con CPN que facilite la ejecución con niveles de rendimiento compatibles con los esperados en una ejecución paralela de gran volumen de datos.

2.3 Estrategias para modelar algoritmos paralelos

En esta sección se presentan los lineamientos para representar un algoritmo paralelo usando el formalismo de CPN. Estos lineamientos permiten definir una CPN que representa un algoritmo paralelo y que en un paso posterior, puede ser transformado en una TPN, que a su vez, es utiliza para ejecutar el algoritmo, lo cual será expuesto en el capítulo siguiente.

Considere que el programador paralelo ha analizado el algoritmo y ha determinado las tareas y el número de divisiones de datos a fin de que sea ejecutado en paralelo. El algoritmo puede ser representado con una CPN haciendo:

- Representar cada tarea del algoritmo por medio de una y solo una Transición.
- Representar cada argumento de datos de cada tarea, por una única Plaza de entrada en la Transición respectiva. En otras palabras, cada Transición tiene tantas Plazas de entrada como parámetros de datos tenga la rutina representada.
- No utilizar otras Plazas ni Transiciones.
- Conectar cada Transición con sus respectivas Plazas de salida, que deben ser las mismas Plazas antes definidas. Este hecho representa la dependencia de datos ya que la salida de una Transición es usada como entrada de otra.
- Determinar los dominios de cada Plaza a fin de satisfacer las condiciones del algoritmo en la red definida.
- Definir el marcado inicial, μ_0, que tendrá posiciones en cero, las correspondientes a estados intermedios del algoritmo, y posiciones con la etiqueta de los bloques de datos que representan los valores iniciales

usados en el algoritmo. Dicho en forma equivalente, el marcado inicial tiene sus posiciones en cero, excepto para las posiciones que representen las Plazas que contienen los valores iniciales.

Un tema central en la estrategia es la correcta definición de los dominios de cada Plaza y las funciones de guarda, las cuales establecen las condiciones del algoritmo. Como cada Plaza representa un parámetro de entrada de una tarea, el dominio de cada Plaza debe ser capaz de representar la división de datos de la cual el parámetro es extraído. El dominio puede tener restricciones impuestas por condiciones que la tarea impone al parámetro.

Estos conceptos son ilustrados con la ayuda de un ejemplo. Considere el algoritmo de factorización de Cholesky. Este es un problema clásico del álgebra lineal, en el cual una matriz cuadrada, simétrica y definida positiva A, con rango r, es factorizada como $A = L*L^T$, siendo L una matriz triangular.

Los valores de L son definidos por las siguientes fórmulas:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad 1 \leq j < i \leq r \tag{2.11}$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad 1 \leq i \leq r \tag{2.12}$$

Las ecuaciones (2.11) y (2.12) imponen una dependencia de datos fuerte, ya que, para computar todos los valores en una fila i, por ejemplo l_{ii}, siguiendo la ec. 2.12, debe haberse computado previamente los valores l_{ik} de esa fila, $k < i$, siguiendo a la ec. 2.11, y luego de ello, quedan disponibles los valores necesarios para computar el valor de la diagonal principal l_{ii}. A su vez, para calcular cada l_{ik} es necesario computar primero todos los valores de las filas i y k, hasta la columna $k - 1$. Este esquema de computación define una dependencia de datos con restricciones importantes vista su ejecución paralela, lo cual será expuesto en el capítulo 4.

Suponga que la matriz es dividida en forma de bloques cuadrados (*tiles*) de $r/n \times r/n$ datos, como es sugerido por la "LAPACK Working Notes" 191 (LAWN 191) [BLKD07], y el procesamiento es hecho por bloques con la ayuda de rutinas definidas en las bibliotecas BLAS [BLA01] y LAPACK [ABB+99]. Otro supuesto es que, como el algoritmo de Cholesky trabaja sobre una matriz simétrica, se utiliza la parte triangular inferior para la factorización.

En el procesamiento del algoritmo de Cholesky, los bloques de la diagonal

principal son computados usando las rutinas **xpotrf** y **xsyrk**, y los restantes bloques con el uso de **xgemm** y **xtsrm**, siendo $x = \{d|s\}$, dependiendo si se usa doble o simple precisión numérica. El algoritmo es expuesto en la siguiente tabla, y la ejecución es graficada en la Fig.(2.4), suponiendo $n = 5$.

1	$i = 1$
2	mientras $(i \leq n)$
3	compute el bloque de la diagonal ppal. l_{ii} llamando a **xsyrk** y luego a **xpotrf**
4	compute los bloques l_{ji}, $j > i$ llamando a **xgemm** y luego a **xtsrm**
5	$i = i + 1$
6	salte a 2

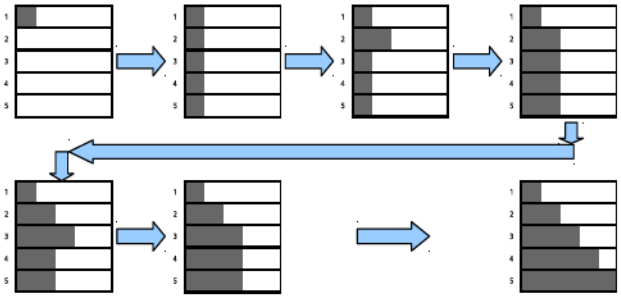


Figure 2.4: Pasos en la ejecución del algoritmo de Cholesky, con $n = 5$

Se destaca que, de las 4 rutinas usadas, solo una, **xpotrf**, es definida en LAPACK, y las restantes tres en BLAS.

Para representar el algoritmo con una CPN, los lineamientos presentados anteriormente dicen que es necesario definir una Transición por cada tarea diferente. En nuestro ejemplo son necesarias solo cuatro Transiciones. También es necesario definir las Plazas de entrada de cada Transición, respetando el número de parámetros de cada tarea. Así, la rutina **xpotrf** necesita solo un parámetro, **xtrsm** y **xsyrk** necesitan dos parámetros cada una, y **xgemm** necesita tres parámetros, haciendo un total de ocho Plazas. De esta forma, se ha definido una parte de la CPN, expuesta en la Fig. 2.5, faltando completar las Plazas de salida, las guardas y los dominios de cada Plaza.

Para completar la red, se continúa con la definición de los dominios de cada Plaza. Antes de nada, se presentan los supuestos sobre los que se basa la definición. La matriz es dividida en $n \times n$ bloques cuadrados. Estos bloques

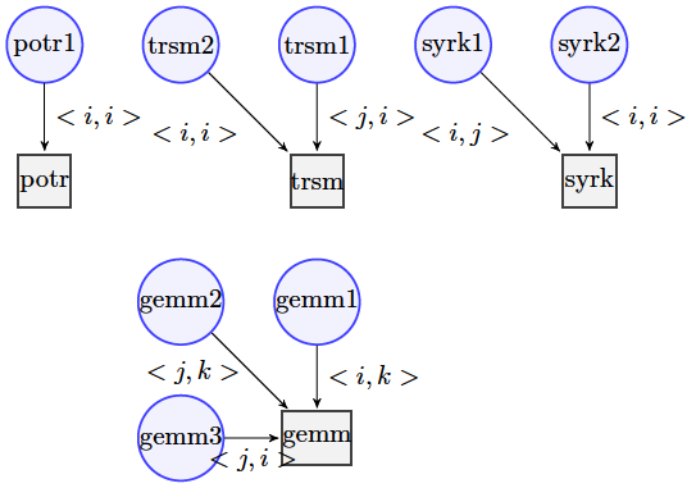


Figure 2.5: Primer paso en el desarrollo del una Red de Petri Coloreada que representa el algoritmo de factorización de Cholesky: definición de Plazas, Transiciones y enlaces de entrada.

son rotulados por medio del par $< i, j >$, donde i representa la fila, j la columna, y ambos tienen rangos $i, j = 1 \ldots n$.

La rutina **xpotrf** usa solo un bloque de datos como parámetro, aquellos ubicados en la diagonal principal de la matriz, por lo que, la Transición respectiva solo necesita una Plaza de entrada, cuyo dominio puede ser denotado como $< i, i >$.

La rutina **xtrsm** usa dos bloques de datos como parámetros, el primero tomado de los bloques de la diagonal principal de la matriz y el segundo puede ser cualquier bloque ubicado en la misma columna del primer bloque elegido, pero debajo de la diagonal principal. De esta forma, los dominios para las Plazas de **xtrsm** son definidos, siguiendo el orden, como $< i, i >$ y $< j, i >$, para $j = i + 1 \ldots n$.

La rutina **xsyrk** usa también dos parámetros, el primero ubicado en la diagonal principal de la matriz y el otro es cualquier bloque en la misma fila, ubicado en la parte triangular inferior. Así, los dominios de las Plazas de entrada de **xsyrk** se definen como $< i, i >$ y $< i, j >$, para $j = 1 \ldots i - 1$.

Finalmente, la rutina **xgemm** usa tres parámetros. Si se considera a la multiplicación de matrices como $C = A \times B$, y se sigue el orden alfabético para la designación de los parámetros, los bloques de A y B deben pertenecer a dos filas diferentes de la parte triangular inferior de la matriz. Como los bloques involucrados provienen de la parte triangular inferior, las dos filas deben tener

números diferentes, de $2 \ldots n$. Será considerado que la fila superior de estas dos, provee los bloques para el parámetro de A y la inferior, los bloques para el parámetro B en la fórmula de anterior. El bloque de la matriz C tiene la misma fila que el bloque de B, y la columna es igual a la fila del bloque de A. Por lo tanto, los dominios de las tres Plazas son:

$$\begin{aligned} A: &\ < i, k > \\ B: &\ < j, k > \\ C: &\ < j, i > \end{aligned}$$

donde $i = 2 \ldots n$, $j = 3 \ldots n$, $i < j$ y $k = 1 \ldots i - 1$. La Fig. 2.5 presenta la gráfica a este punto de avance en el desarrollo de la CPN. Las Plazas son etiquetadas como sus Transiciones y un número que indica el orden del parámetro de entrada en la rutina representada.

A los fines de completar la red, deben agregarse los enlaces que definen las Plazas de salida y las funciones de guarda. Recordar que las Plazas de salida representan la dependencia de datos entre tareas. La salida de la rutina `xpotrf` es un bloque en la diagonal principal usado luego por `xtrsm`, por lo que el bloque $< i, i >$ de `xtrsm` es proveído por `xpotrf`. La rutina `xtrsm` produce bloques utilizados por `xsyrk` y `xgemm`. Como el modelo de CPN no impone restricciones sobre el número de *tokens* inyectados por el disparo de una Transición, los bloques de salida de la ejecución de `xtrsm` son replicados en las Plazas que los usan y en el número de veces que sea necesario para cada Plaza. Notar la similitud de este hecho con el uso de bloques de datos en modo "*read only*" usual en muchas tareas.

La salida de la rutina `xsyrk` es usada por ella misma hasta que todas las etapas del procesamiento sean completadas, en cuyo caso, la rutina `xpotrf` usa esta salida. Finalmente, la salida de la rutina `xgemm` es usada también por sí misma hasta que todos los cómputos que producen el cálculo final del bloque C son completados, en cuyo caso, la salida es utilizada por la rutina `xtrsm`.

Los próximos elementos a ser agregados en la construcción de la CPN son las funciones de guarda. La rutina `xpotrf` tiene sola una Plaza de entrada que no tiene límites, por lo que no requiere una guarda. La rutina `xtrsm` tiene dos Plazas de entrada, con dominios $< i, i >$ y $< j, i >$ respectivamente, lo que define la guarda implícita de que la columna de la segunda Plaza debe ser la misma que la fila de la primea. Esta guarda implícita es innecesario de ser explicitada en la red.

Los dominios de las dos Plazas de entrada de la rutina `xsyrk` son $< i, i >$ y $< i, j >$, un caso similar al anterior en el sentido de la guarda implícita. La rutina `xgemm` presenta una guarda implícita también, pero con tres Plazas de entrada.

Otro tema necesario de analizar es el de la multiplicidad de los *tokens* en las Plazas, cuantos son inyectados por el disparo de un Transición, y cómo queda expresado en términos de guardas de salida. Como cada disparo de Transición absorbe *tokens*, los *tokens* que representan bloques de datos usados en modo solo lectura en la ejecución de la tarea, deben ser replicados en cantidad necesaria para disponer de ellos en cada caso que participen. Es el caso de la rutina `xtrsm`. Una vez que la rutina `xpotrf` ha concluido, su bloque resultante, que se ubica en la diagonal principal de la matriz, es utilizado para computar todos los bloques de la misma columna por debajo de la diagonal. Este hecho impone la necesidad de tener repetida la salida de la rutina `xpotrf` $(i - 1)$ veces en la Plaza número uno de `xtrsm`, siendo i el número de fila computada por `xpotrf`. Esta duplicación es representada en el grafo por medio de la notación $\{expression\_count\}$ en el enlace hacia la Plaza de salida.

Con la red casi completamente definida, el estado actual de la CPN desarrollada es mostrada en la Fig. 2.6 . Algunos nombres de variables (caracteres) en las etiquetas de los dominios han sido cambiados a fines de mantener la consistencia entre las entradas y salidas de una Transición.

A este punto de avance, el algoritmo es completamente representado por la CPN. Es destacable la ausencia de guardas explícitas en su representación. A pesar de ser una situación propia del algoritmo considerado, la correcta definición de los dominios en cada Plaza genera implícitamente las guardas necesarias, porque los límites y condiciones son transferidos desde el algoritmo hacia el dominio de los datos involucrados.

Un importante corolario puede ser deducido de este último hecho: definir el dominio propio de cada parámetro de la tarea simplifica el algoritmo. Un algoritmo simple es fácil de analizar y de paralelizar. A pesar de que este hecho no está en el foco de la tesis, es una consecuencia importante derivada de dos funcionalidades propias de una CPN, la primera, que cada Plaza puede tener el dominio pertinente a ella, y la segunda, que el disparo de una Transición permite inyectar *tokens* en las Plazas de salida de distintos dominios a los de entrada. Esto facilita facilita el traslado de la mayoría de las guardas a la definición de los dominios intervinientes en la tarea.

Un elemento adicional debe ser agregado en la red orientado a la ejecu-

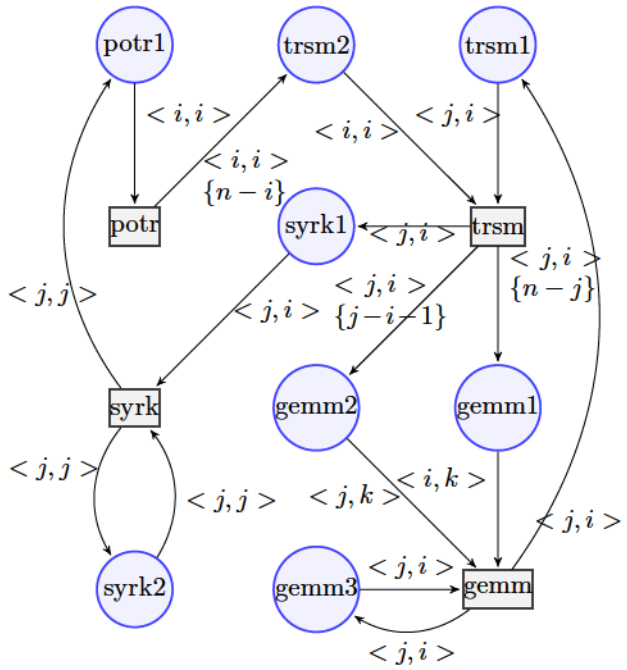


Figure 2.6: Segundo paso en el desarrollo de la Red de Petri Coloreada que representa el algoritmo de factorización de Cholesky: definición de los enlaces de salida y multiplicidad.

ción paralela del algoritmo. Los bloques de datos cuadrados que componen la división de datos de la matriz tienen un impacto sobre la forma de ejecutar las rutinas `xsyrk` y `xgemm`. Estas rutinas han sido planeadas originalmente para tomar bloques rectangulares de datos en su ejecución, y no para bloques cuadrados. Por ejemplo, `xsyrk` produce un bloque cuadrado sobre la diagonal principal utilizando todos los datos de la fila, generalmente, un bloque rectangular, no un bloque cuadrado. De esta forma, la división de datos produce una división de la tarea original en varias subtareas, cada una de las cuales usa una parte de los datos participantes en la tarea completa, y por lo tanto, produce un resultado parcial. Para obtener el resultado completo, cada una de las subtareas debe ser hecha secuencialmente, ya que el resultado parcial de una subtareas debe ser acumulado con el resultado de las siguientes subtareas.

Dada la necesidad de preservar el orden en los cómputos, el dominio de una Plaza en la rutina `syrk` y de otra en `gemm` han sido extendidos con una coordenada adicional que representa el orden secuencial de ejecución. Así, el dominio $< j, j >$ de la Plaza de entrada de `syrk`, es reemplazado por

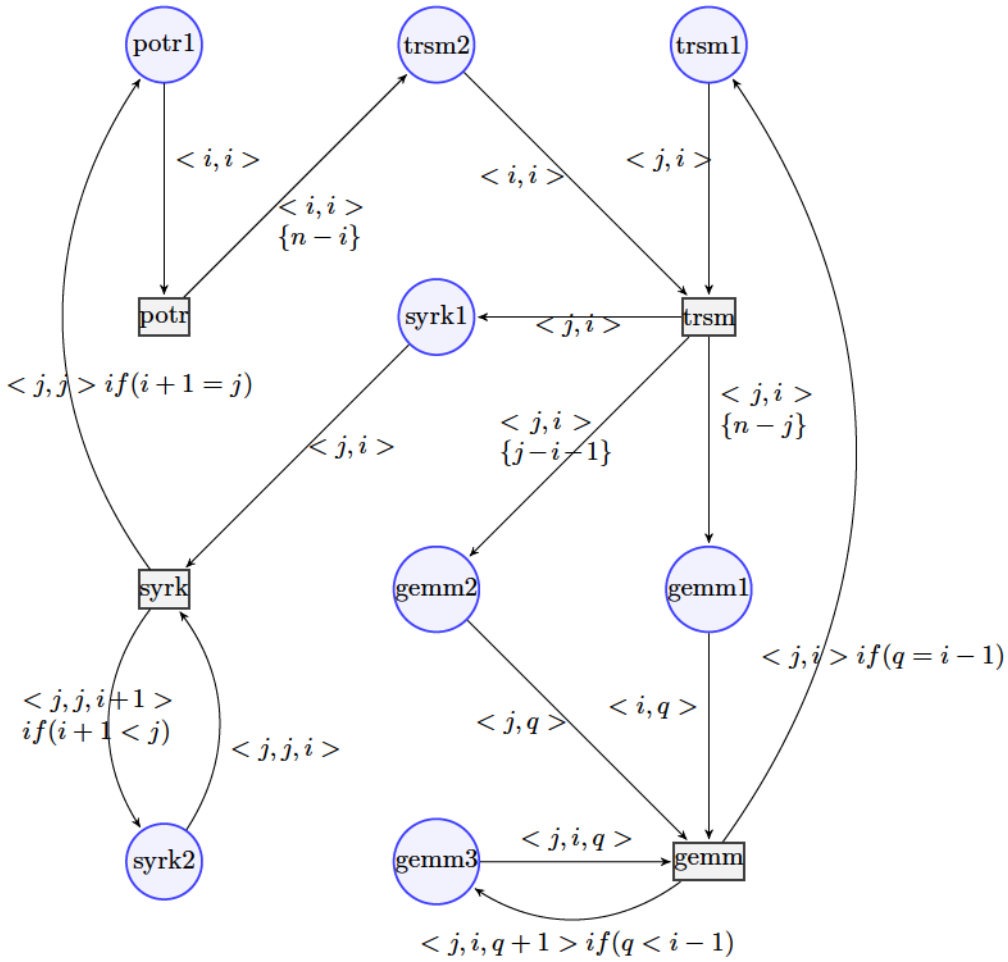


Figure 2.7: Red de Petri coloreada que representa al algoritmo de factorización de Cholesky, decorado para su ejecución paralela.

$< j, j, i >, i = 1 \ldots j - 1$. El orden secuencial es representado por la variable i. Luego, como completar la tarea `syrk` para la fila j debe ser hecho por medio de $j - 1$ subtareas, el rango de i es $1 \ldots j - 1$. Cada vez que una subtarea es hecha, `syrk` genera un *token* de salida a la misma Plaza de entrada, sumando uno a la coordenada de orden, hasta que todas las subtareas son hechas, en cuyo caso, el *token* de salida es dirigido hacia la respectiva Plaza de entrada de `potr`. En la Fig. 2.7, esta condición es representada por medio de dos guardas agregadas a los arcos de salida de la Transición `syrk`. De manera similar, es igualmente necesario extender el dominio de la tercera Plaza de entrada de la rutina `gemm`, quedando como $< j, i, q >, q = 1 \ldots i - 1$, para q que representa el orden de ejecución, y el mismo tipo de guardas son adicionadas a los arcos

de salida de la Transición `gemm`.

Habiendo completado los últimos cambios en la red, la CPN que representa al algoritmo queda completa. Su representación gráfica es expuesta en la Fig. 2.7.

En su libro, Diaz presenta las condiciones que debe satisfacer una CPN para conformar una Red de Petri "bien formada" [Dia09]. Las condiciones son:

- Los colores de los dominios pueden ser unicamente el producto cartesiano de colores básicos, aquellos que no dependen de otros colores.
- Las funciones sobre colores pueden ser hechas solamente con tres funciones elementales sobre los colores de los dominios: identidad, sucesor y *broadcast*.

La CPN construida en esta sección cumple con las condiciones: sus dominios son subconjuntos del producto cartesiano de números naturales y las funciones de guarda solo usan la identidad y el sucesor sobre dichos dominios. El principal beneficio de esta forma de CPN es que puede ser convertida (desplegada) en una TPN de una manera simple. La próxima sección explica cómo hacerlo y el impacto que esto tiene sobre la ejecución del algoritmo.

2.4 Desplegado de Redes de Petri Coloreadas

Dado que una CPN puede ser vista como una representación de alto nivel de una TPN, el proceso de desplegado que transforma una CPN en una TPN, es un proceso que preserva la semántica de la CPN [Dia09], y es lo opuesto al proceso de generalización que representa una CPN. El desplegado es hecho reemplazando los colores de los dominios por Plazas y Transiciones que producen una TPN que mantiene la semántica de la original.

La CPN desplegada es obtenida mediante:

- Cada Plaza en la CPN es reemplazada por un conjunto de Plazas en la TPN, una por cada valor o color en el dominio de la Plaza original, preservando el número de repeticiones de cada color, esto último debido a que se representa con el modelo de *multiset*, por lo que cada Plaza en la TPN obtenida representa a un *token* en la CPN.

- Cada Transición en la CPN es reemplazada por un conjunto de Transiciones en la TPN, una por cada combinación de colores obtenida en el producto cartesiano de las Plazas de entrada en la CPN, restringido a los casos en que sus posibles guardas sean evaluadas como verdadero. Las Plazas de entrada de una Transición desplegada son las respectivas Plazas generadas en el punto anterior.

La última condición es la clave en el proceso de desplegado: dado el producto cartesiano de los colores de m Plazas de entrada de una Transición y para cada evaluación de su función de guarda como verdadero, cada combinación de estas define una Transición en la TPN y m Plazas de entrada son enlazadas a la Transición desplegada. La semántica es preservada ya que cada combinación en la CPN es reemplazada por un par único de (Plazas de entrada, Transición) en la TPN. De esta forma, en la TPN, una Transición representa a un único conjunto de valores que evaluaron verdadero en la CPN, y cada Plaza representa un valor en el respectivo dominio en la CPN.

El siguiente ejemplo ilustra el proceso de desplegado. Por simplicidad, solo será considerado el desplegado de la Transición **syrk** y se muestra su gráfica en la Fig. 2.8. En dicha gráfica se ve que la Transición referida tiene dos Plazas de entrada, **syrk1** y **syrk2**, con dominios $< j, i >$ y $< j, j, i+1 >$ respectivamente, con rangos $j = 1 \ldots n$ y $i = 1 \ldots j - 1$ para ambos casos. Las Plazas de salida son **syrk2** y **potr1**, restringidas por las guardas que evalúan si $i + 1 < j$ o no.

Bajo el supuesto de que la matriz es dividida en $n \times n$ *tiles* cuadrados, con $n = 3$, el producto cartesiano de los dominios de **syrk1** y **syrk2** es compuesto por solo tres valores para cada Plazas: $< 2, 1 >$, $< 3, 1 >$ y $< 3, 2 >$ para **syrk1** y $< 2, 2, 1 >$, $< 3, 3, 1 >$ y $< 3, 3, 2 >$ para **syrk2**, definiendo entonces seis Plazas y tres Transiciones en la red desplegada. La Fig. 2.8 muestra la representación gráfica de este caso. La notación de las Plazas y Transiciones representa el número de Plaza / Transición y número de color correspondientes. Por ejemplo, **sy2331** representa el segundo parámetro con valor $< 3, 3, 1 >$. Las condiciones de guardas determinan que la salida de **syrk31** es **sy2332**, pero la salida de **syrk32** es **potr13**.

En este momento es necesario destacar la importancia de la representación de la TPN, ya que la TPN desplegada, preserva la semántica de la CPN, y también representa al algoritmo. Cada Transición en la TPN representa a cada tarea individual y cada Plaza a un bloque de datos en particular.

Además de lo señalado en el párrafo anterior, la representación matricial

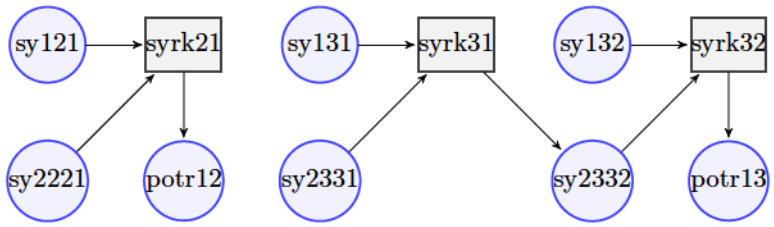


Figure 2.8: TPN desplegada de la CPN que representa al algoritmo de factorización de Cholesky, para la tarea **syrk** y $n = 3$.

de la red desplegada puede ser hecha en programáticamente. Un programa puede construir ambas Matrices de Incidencia, negativa y positiva, y el Vector de Marcado en su estado inicial, representando internamente cada color y las condiciones de las guardas. El número de Plazas y Transiciones desplegadas depende del producto cartesiano de los colores en los dominios, los que a su vez, representan al número de divisiones de datos.

También se destaca que la red desplegada puede tener un gran número de Plazas y Transiciones. Mayor sea la cardinalidad de los dominios involucrados, mayor será el número de Plazas y Transiciones desplegados. El número de Plazas de entrada de una Transición también impacta en el número de componentes en la red desplegada, ya que cada combinación en el producto cartesiano evaluada verdadero debe ser incluida.

El proceso de desplegado produce dos matrices de rango $t_u \times p_u$ de valores 0 y 1, siendo t_u el número de Transiciones desplegadas y p_u, el número de Plazas desplegadas. En la Matriz de Incidencia negativa (positiva) resultante, la posición (i, j) tendrá un valor de 1 si la Plaza j es Plaza de entrada (salida) de la Transición i, o 0 de lo contrario. Un valor 1 representa que el bloque de datos relacionado es usado (producido) por la tarea respectiva.

La TPN desplegada tiene las siguientes propiedades: es 1-*bounded*, pseudo-viva porque siempre hay una Transición habilitada para ejecutar, y el grafo de alcanzabilidad incluye al estado final del algoritmo, lo que dicho de otra forma, el algoritmo ejecuta todas las tareas y finaliza su procesamiento.

A pesar de que una red desplegada es virtualmente inmanejable manualmente por su gran tamaño, su importancia resalta por dos factores. Primero, representa la ejecución del algoritmo. En efecto, cada Transición representa una tarea que debe ser realizada, y la dependencia de datos puede ser observada por el doble rol de las Plazas, al trabajar como entrada y salida de dos, posiblemente diferentes, Transiciones. La ejecución paralela de las tareas y un camino crítico de ejecución puede ser analizado a partir de

esta red.

El segundo factor del desplegado que sobresale es que la TPN puede ser representada por medio de dos Matrices de Incidencia y un Vector de Marcado. Como cada Transición en la desplegada es una combinación particular del par (tarea, parámetros), puede ser usada para relacionar unívocamente cada fila y columna de la Matriz de Incidencia con una rutina y los bloques de datos usados por esta.

Como las Redes de Petri tienen el concepto de su ejecución, la matrices desplegadas junto con la relación entre tareas y datos con las Plazas y Transiciones, definen un orden de ejecución que representa al algoritmo, lo que, en otras palabras, define un programa. En concordancia, disparar una red desplegada como las construida aquí, es equivalente a ejecutar un programa.

La ejecución del programa es guiada ahora por operaciones matriciales simples que tienen una sobrecarga de ejecución mínima. Son obviamente necesarios los procesadores que ejecuten las tareas representadas por las Transiciones. Si se dispone de más de un procesador que corran en paralelo, se obtiene una ejecución paralela del algoritmo. La ejecución paralela tradicional, que tiene que resolver problemas como el control de la secuencia de ejecución, las barreras de ejecución y la sincronización de tareas, es reemplazada por simples operaciones de sumas y diferencias matriciales y comparaciones, generando así un modelo de ejecución paralelo diferente. La ejecución secuencial o paralela dependerá del número de procesadores usados en la ejecución.

En este capítulo ha sido dedicado principalmente a la representación del algoritmo en una Red de Petri que tiene facilidades para ser ejecutada. El siguiente capítulo presenta el desarrollo y funcionamiento del modelo de ejecución basado en lo producido hasta aquí.

Capítulo 3

Modelo de Ejecución Paralela

En el capítulo anterior se muestra cómo modelar un algoritmo con Redes de Petri coloreadas (CPN). El despliegue de una CPN a una Red de Petri simple (TPN) transforma una red compacta en una más grande pero más simple de ejecutar. En este capítulo se describe el modelo de ejecución que permite correr en paralelo el algoritmo representado por la TPN desplegada. Se incluyen secciones con el diseño y la ejecución del modelo, y finalmente el capítulo concluye con una comparación con los trabajos relacionados.

3.1 Definición del Modelo de Ejecución Paralela

En el capítulo anterior se introdujo una manera de utilizar las Redes de Petri para modelar un algoritmo, evolucionando desde un alto nivel de representación con las CPN a una representación detallada y pero extensa con las TPN. Se dijo que la representación matricial es preferida debido que puede ser utilizada para ejecutar el algoritmo. En esta sección se presenta el modelo necesario para ejecutar la TPN desplegada y que cierra la brecha entre el modelo y la ejecución.

El Modelo de Ejecución Paralela (PEM), como su nombre lo indica, es un modelo para ejecutar programas en paralelo, basado en la definición del algoritmo a ejecutar por medio de una TPN. De hecho, en lugar de utilizar las instrucciones de bajo nivel o directivas del compilador para indicar las secciones paralelas de un programa, las matrices definidas anteriormente en la etapa del modelado del algoritmo, son utilizados por el PEM como parámetro

de la ejecución del programa paralelo. Además de las matrices, es necesario definir otros elementos a fines de completar el modelo de ejecución, lo cual se explica a continuación.

El modelo PEM usa varios de los elementos definidos en el capítulo anterior, y otros adicionales que se presenta en esta sección. Para refrescar los conceptos de una TPN, expresados en la sección 2.1, se tenía que:

- P es un conjunto finito de Plazas P_i, con cardinalidad $|P| = p$, $i = 1 \dots p$.
- T es un conjunto finito de Transiciones T_j, con cardinalidad $|T| = t$, $j = 1 \dots t$.
- I^- e I^+ son las Matrices de Incidencia negativa y positiva de la TPN, de dimensiones $p \times t$ (I^- e $I^+ \in \mathbb{N}^{p \times t}$).
- M, es el Vector de Marcado para las Plazas, $p \times 1$ ($M \in \mathbb{N}^p$).

con la única diferencia respecto a la citada sección que, originalmente las Matrices de Incidencia están denotadas como D^- y D^+ como es usual en la bibliografía, pero aquí, por razones prácticas, han sido redenominadas como I^- e I^+.

Los nuevos conceptos introducidos en este capítulo son:

- M_f, es el Vector de Marcado Final, $p \times 1$ ($M_f \in \mathbb{N}^p$) que representa el estado del Vector al final de los cálculos. Se necesita este vector con el fin de establecer el final de la transformación mediante su comparación con el Vector de Marcado M.
- S es el conjunto de tareas que el algoritmo debe ejecutar, en otras palabras, es el conjunto de cómputos individuales que procesan los datos, tomados como una tarea indivisible. Se obtienen a partir del análisis hecho sobre el algoritmo con el fin de que sea ejecutado en paralelo.
- τ es una función que va del conjunto de Transiciones T en el conjunto de tareas S, $\tau : T \to S$. Esta función asocia cada Transición con una tarea.
- δ es una función que va del conjunto de Plazas P en el conjunto de divisiones de datos, y asocia cada Plaza con uno de los bloques en que fueron divididos los datos.

- Π es un conjunto finito de Procesadores Π_i, con cardinalidad $|\Pi| = \pi, i = 1 \ldots \pi$. Un procesador Π_i es un objeto capaz de ejecutar las tareas asociadas a cada Transición por medio de una llamada a una rutina (*kernel*) de un programa. Cada procesador Π_i tiene una variable lógica e ($\Pi_i.e$), que tiene valores verdadero o falso según el procesador este ejecutando o inactivo.

- γ_i es una función que asocia para cada procesador Π_i, una tarea $s \in S$ con un *kernel* que el procesador ejecuta, a los efectos de realizar la tarea. Define el *kernel* a ser ejecutado cuando la Transición es disparada, y representa el nexo entre el modelo teórico de las TPN con la ejecución del algoritmo. Cada procesador Π_i tiene su propia función γ_i, lo que permite que procesadores diferentes resuelvan la ejecución de la tarea s a su manera. Por ejemplo, si la Transición representa una tarea de multiplicación de matrices, un procesador puede ejecutar la tarea por medio de una llamada a la rutina *xgemm* de la biblioteca BLAS implementada sobre CPU, y otro procesador a la misma rutina, pero implementada sobre GPU.

- Γ es el conjunto de funciones γ_i.

- χ es una variable lógica que representa un mecanismo de exclusión mutua sobre M y que permite que cada Π_i actualice el Vector de Marcado M en forma segura.

El Modelo de Ejecución Paralela (PEM) es definido como la n-upla:

$$PEM = (P, T, I^-, I^+, M, M_f, S, \tau, \delta, \Pi, \Gamma, \chi) \tag{3.1}$$

para todas las componentes como fueron previamente definidas.

El estado inicial del PEM es:

- $M = M_0$, el Marcado inicial en la TPN.

- $\chi = true$, la exclusión esta abierta.

- $\Pi_i.e = true$, $\forall i = 1 \ldots \pi$, dado que todos los procesadores están inactivos.

El modelo PEM es similar al de las Redes de Petri temporizadas (*Timed Petri Nets*) [Wan98]. Ambos comparten el concepto de que el disparo de una

Transición no es instantáneo ya que hay un tiempo que transcurre entre el inicio y el final del disparo. Al igual que en PEM, la acción del disparo representa la ejecución de una tarea, pero la diferencia es que en PEM el disparo no se hace de manera autónoma una vez que la Transición está habilitada como lo es en las temporizadas. Un procesador inactivo es el responsable de disparar la Transición seleccionada entre todas las habilitadas. Por lo tanto, en el modelo presentado, disparar una Transición es equivalente a ejecutar una tarea.

El modelo PEM se completa con el hecho de que cada procesador corre una tarea independientemente del resto. Una vez seleccionada una tarea a ejecutar, la dependencia de datos es garantizada por la Red de Petri. Si dos tareas tienen dependencia entre sí, el modelo de Petri no permitirá ejecutar ambas en paralelo, por lo que cuando más de una tarea está habilitada para ejecutarse, se debe a que es independiente de las restantes. Esta propiedad produce un efecto fuerte en el PEM, ya que elimina completamente la necesidad de introducir sincronizaciones en la ejecución paralela.

El número de Transiciones habilitadas puede ser menor o mayor al número de procesadores. Como resultado de ellos, pueden quedar procesadores inactivos sin Transiciones para disparar, o Transiciones habilitadas esperando por un procesador libre, dependiendo del número de Transiciones habilitadas en relación al número de procesadores. En el primer caso, el *speedup* [RR10] del programa será pobre, y esta situación debe ser evitada. En el segundo caso, el procesador debe seleccionar la Transición mas apropiada para disparar. Por lo tanto, es necesario definir a cada procesador un criterio que permita seleccionar la Transición a disparar cuando hay más de una habilitada.

Debido a la existencia de varios procesadores corriendo en paralelo, es posible que dos o más de ellos intenten leer o escribir en forma simultánea la estructura de datos que refleja el estado del procesamiento, el Vector de Marcado M. Escrituras concurrentes sobre dicho vector pueden producir un estado incorrecto de la ejecución del algoritmo. Para evitar esta situación, el modelo de ejecución se ayuda de un mecanismo de Exclusión Mutua (*mutex*) sobre el Vector. De esta forma, los procesadores no se interfieren al momento del uso de dicho Vector, actuando en forma secuencial sobre este, especialmente en oportunidad de seleccionar la Transición a disparar, o de actualizar por finalización de la misma, esperando que el mecanismo les de paso.

El pseudo-código del algoritmo de ejecución del PEM para cada procesador es expuesto en la Fig. 3.1. Cada procesador Π_i corre en paralelo este mismo algoritmo con el resto de los procesadores. El algoritmo finaliza cuando

```
1  While main algorithm not finished
2    If it can hold the mutual exclusion
3       Compute h function
4       Select one task to execute (Tk)
5       Update M by absorbing tokens
6       Free the exclusion
7       Task execution
8       Inject tokens in M
9    Else
10      Delay
11   Endif
12 End
```

Figure 3.1: Pseudo-código del algoritmo de selección de tareas, que cada procesador Π_i corre en paralelo.

el Vector de Marcado M alcanza los mismos valores para todas sus posiciones que el vector es estado final, M_f ($M = M_f$ en la línea 1).

Siguiendo un criterio Round-Robin, cada procesador Π_i con la bandera de disponibilidad en verdadero, intenta apoderarse del mecanismo de exclusión mutua. Si lo logra, busca una tarea para ejecutar basado en el modelo del algoritmo representado por la TPN. Para determinar que Transiciones están habilitadas, solo se utilizan operaciones simples del álgebra lineal. En efecto, si llamamos I_j^- y I_j^+ a la j-ava columna (Transición) en I^- y I^+ respectivamente, la j-ava Transición está habilitada si la diferencia vectorial $M - I_j^-$ no arroja ningún valor negativo entre las posiciones del vector resultante. Esta situación implica que todas las Plazas de entrada de la Transición j tienen *tokens*. La función h a continuación determina esta evaluación. Tiene aridad $h : \mathbb{N}_{1\ldots t}, \mathbb{N}^{p\times t}, \mathbb{N}^{p\times 1} \to \{0, 1\}$, con parámetros j, M y I^-, y sus valores resultantes son:

$$h(j, I^-, M) = \begin{cases} 1 & \forall k = 1 \ldots p : (M - I_j^-)_k \geq 0 \\ 0 & else \end{cases} \quad j = 1 \ldots t$$

por lo que computando h para todas las columnas, todas las Transiciones habilitadas y listas para ser disparadas, quedan determinadas.

Para seleccionar la tarea para ser ejecutada (paso 4), fue desarrollado un selector en tiempo de ejecución. Cuando cada procesador está corriendo, usa una función de valuación que es aplicada al conjunto de Transiciones habilitadas, seleccionando aquella con mayor valoración, T_k. Esta función de

valuación es la clave para obtener un mejor rendimiento en la ejecución paralela, debido a que puede ser particular a cada tipo de procesador y algoritmo, en vista a seleccionar la más conveniente para obtener los rendimientos más altos.

Un ejemplo puede ayudar a comprender el párrafo anterior. Si se desea que la ejecución paralela concluya lo antes posible en un sistema de procesadores homogéneos, lo cual es conocido como el problema de minimización *makespan* [RV09], un posible criterio para seleccionar la tarea, puede ser que la función h evalúe cada tarea asignando mayores valores en relación al número de tareas paralelas que habilite al concluir. Por otro lado, en un sistema heterogéneo, los procesadores más veloces deben seleccionar la tareas con el mismo criterio del caso anterior, para evitar "cuellos de botella" en la ejecución, pero los procesadores más lentos deben seleccionar tareas que no sean prioritarias, realizando tareas que no estén en el "camino crítico" del algoritmo, pero colaborando para finalizar más rápido. Más aún, para estos procesadores lentos, la función de valuación puede no elegir a ninguna tarea a realizar, para evitar que el tiempo global se vea afectado por la lentitud de estos procesadores. Todo depende del caso corriendo. En el capítulo 4 se verá un experimento de multiplicación de matrices con un sistema heterogéneo de CPU y GPU que utiliza esta facilidad.

Los pasos 5 y 8 del pseudo-código representan la evaluación en la ejecución. Similar a las Redes de Petri temporizadas, los *tokens* son absorbidos e inyectados en dos tiempos. En el paso 5 los *tokens* que vienen de las Plazas de entrada de T_k son absorbidos cuando la Transición es disparada, y en el paso 8, son inyectados en sus Plazas de salida. Ambos pasos son hechos con la ayuda de operaciones de álgebra lineal:

$$M' = M - I_k^- \qquad \text{en 5 a tiempo } t_0 \tag{3.2a}$$

$$M''' = M'' + I_k^+ \qquad \text{en 8 a tiempo } t_0 + \Delta_k \tag{3.2b}$$

y luego del paso 8, potencialmente nuevas Transiciones quedan habilitadas. Para inyectar los *tokens*, es decir obtener M''', la exclusión no es necesaria dado que cada Plaza actúa como Plaza de salida de solo una Transición por diseño, recordar la propiedad 1-*bounded* antes expuesta, por lo que escrituras concurrentes no son posibles.

El Vector de Marcado M y M'' son los marcados en tiempo t_0 y $t_0 + \Delta_k$, donde Δ_k es el tiempo que la ejecución de la tarea T_k insume. El ciclo es repetido hasta que el final del algoritmo es alcanzado, cuando $M = M_f$.

Una vez que los *tokens* son absorbidos en el paso 5 y la exclusión es liberada, la ejecución de la tarea es realizada en el paso 7 usando la función γ_i, que relaciona la Transición seleccionada con el *kernel* a ejecutar. Además, los bloques de datos necesarios son seleccionados por la función δ que usa las Plazas de entrada de la Transición seleccionada para obtener los bloques apropiados.

Las relaciones entre las Transiciones y las Plazas con las rutinas y los bloques de datos puede no ser única para todos los procesadores, y es lo que permite adaptar el PEM a un sistema heterogéneo. Por ejemplo, en un sistema *multicore* - *multigpu*, cada tipo de procesador tiene sus propios mapeos, tanto para rutinas como para datos de entrada. Esto permite, además de flexibilizar el *kernel* relacionado con cada tarea, configurar diferente granularidad de datos para cada tipo de procesador, de ser necesario.

Finalmente, resta considerar la sobrecarga introducida por el modelo de ejecución paralelo, Tres factores la determinan. Primero, el mecanismo de exclusión mutua sobre el Vector de Marcado, el cual, con una implementación eficiente, usa solo unos pocos ciclos del reloj, por lo que su impacto sobre el tiempo total de ejecución es descartable. Segundo, las operaciones básicas de álgebra lineal sobre las Matrices de Incidencia y el Vector de Marcado, necesarias para determinar las tareas disponibles de realizar. En este caso, desarrolladores de la biblioteca BLAS las implementan optimizadas y no suelen tomar más de millisegundos con los procesadores actuales. En tercer lugar, la política de selección de Transiciones entre las habilitadas, que debe ser conducida por un criterio de balance entre la mejora en el rendimiento general del algoritmo paralelo y el tiempo que insume la selección. En realidad, la suma de los tiempos de ejecución de estos tres factores es del orden de los milisegundos, por lo que los *kernels* de ejecución deben ser de uno o dos órdenes de magnitud mayores, de forma tal que sea mínimo el impacto de la sobrecarga generada. Los experimentos presentados en el Capítulo 4, cuyos *kernels* son del orden como los recién planteados, demuestran un mínimo impacto.

3.2 Diseño del Modelo

Con la finalidad de realizar pruebas sobre el modelo definido en la sección anterior, fue desarrollado un *framework* que permite ejecutar en paralelo los algoritmos modelados con las definiciones del PEM. A continuación se presenta una breve introducción a los *frameworks* de aplicaciones.

Un *framework* de aplicaciones es un artefacto de *software* reusable, semi-completo, extensible y especializado que produce aplicaciones particularizadas. Está compuesto por un conjunto de objetos que interaccionan entre sí y que representan un diseño de *software*. Alguno de los objetos que componen el *framework* son parcialmente implementados para facilitar su extensibilidad. El programador que usa un *framework* debe proveer piezas de código que completen el diseño del *framework* con las particularidades del problema que se intenta resolver, resultando como producto final una aplicación especializada [FSJ99, Lar04].

Una de las características sobresalientes de un *framework* es la inversión del control, lo que significa que el *framework* tiene el control de la ejecución y el desarrollador solo provee el código que es llamado por el *framework*. Resumiendo, un *framework* representa un diseño que implementa una solución a un problema particular, dejando al programador una participación destinada a particularizar los puntos no definidos en el diseño.

Siguiendo estos conceptos, el diseño del *framework* usado en este trabajo se basa en la definición de siete clases de objetos, los cuales son graficados en el Diagrama de Clases de la Fig. 3.2 [Lar04]:

1. El *Paralell Program*: representa la ejecución paralela en su totalidad. Es el responsable de lanzar la ejecución paralela del algoritmo representado por la TPN y de la asociación de los restantes objetos que componen el *framework*. Es instanciado solo una vez.

2. La *Petri Net*: representa a la Red de Petri simple (TPN) y contiene todos los datos relativos a la red, como las Matrices de Incidencia, el Vector de Marcado y el mecanismo de exclusión mutua; también contiene el Vector de Marcado Final y las implementaciones de las funciones τ y δ. Utiliza dos estructuras de diccionarios, las cuales son usadas para contener la relación entre un número y el nombre de las Transiciones y las Plazas. Es la responsable de determinar el final de la ejecución. Existe solo una vez en el *framework*.

3. El *Processor*: representa un procesador lógico. Es instanciado π veces, según el número de procesadores que se definan para correr en paralelo. Contiene varios objetos y estructuras de datos, como el mapeo de cada Transición con el *kernel* a ejecutar (la implementación de la función γ_i), las instancias del *Evaluator* y del *ThreadPool* (desarrollados a continuación) y el enlace con el objeto *Bank*. Clases diferentes de procesadores heredan de esta clase, particularizando la configuración y los mapeos.

4. El *ThreadPool*: representa al conjunto de hilos (*threads*) que ejecutan los *kernels*. Cada *Processor* está compuesto por un conjunto de hilos que posibilitan ejecutar en paralelo internamente. Esta arquitectura representa un modelo de paralelismo anidado ya que, en el primer nivel de paralelismo es definido por el conjunto de *Processors*, y el segundo nivel es definido en cada *Processor* por su conjunto de hilos. Está motivado en el hecho que muchas implementaciones de BLAS son optimizadas para correr en paralelo eficientemente. Contiene además información sobre la afinidad de los *cores* físicos donde los hilos son ejecutados, implementando localidad espacial [RR10]. Es instanciado uno para cada *Processor* Π_i.

5. El *DataPool*: representa los datos utilizados en los cómputos. Contiene referencias a cada bloque de datos en que es dividido la totalidad de los mismos, y es el responsable de suministrar los datos que son utilizados por los *kernels*, y capturar de estos el resultado de la computación. Es instanciado solo una vez para todo el sistema.

6. El *Evaluator*: representa la función que determina la valoración de las Transiciones habilitadas en la TPN y define la mejor. Puede utilizar el estado de la red para evaluar acorde sea este. Cada *Processor* tiene una instancia de este objeto, pero diferentes *Processors* pueden tener instancias diferentes de *Evaluators*, lo que permite implementar esquemas de prioridades propios a cada tipo de *Processor*. Por ejemplo, permite particionar al conjunto de *kernels* utilizados en la ejecución del algoritmo según el tipo de procesadores tenga la máquina usada, donde algunos *kernels* sean ejecutados exclusivamente por un tipo de procesadores y otro tipo de *kernels* por otros procesadores. Para este caso, el *Evaluator* de cada clase de *Processor* debe considerar solo los *kernels* que puede ejecutar, descartando los restantes.

7. El *Bank*: representa al conjunto de bloques últimamente utilizados, permitiendo llevar cuenta de la localidad temporal de bloques [RR10] en la selección de la tarea a ejecutar. Es instanciado uno por cada *Processor*. Contiene la referencia de los últimos bloques de datos utilizados por el *Processor*, lo cual puede ser usado por el *Evaluator* para definir el orden entre las Transiciones habilitadas. Hay tantas instancias como unidades de localidad temporal sean deseadas en el sistema.

Figure 3.2: Diagrama de Clases del Modelo de Ejecución Paralelo (PEM).

Como puede observarse, el diseño propuesto para el PEM tiene componentes que se corresponden con la estructura dada para la definición del mismo en la sección previa, además de otros componentes orientados a una ejecución eficiente. Las clases *Petri Net, Processors, DataPool* y *Evaluator* provienen de la definición y comprende todos los elementos allí definidos. Estos son las Plazas P, las Transiciones T, las Matrices I^- y I^+, el Vector de Marcado M y M_f, el mecanismo de exclusión mutua χ, las tareas S, los *Processors* Π, y las funciones τ, δ y Γ.

Las clases agregadas a la definición del PEM son la *Paralell Programm*, destinada al control de la operación conjunta de todos los *Processors*, la clase *ThreadPool*, que provee paralelismo interno y afinidad a los *Processors*, y la clase *Bank*, orientada a la localidad de datos. Todas estas clases se originan por la ejecutabilidad del modelo.

La dinámica de la operación del sistema conformado por todos los componentes es expuesta en los siguientes dos Diagramas de Secuencia.

La inicialización del sistema es graficada en el Diagrama de Secuencia de la Fig. 3.3 [Lar04]. El objeto *Parallel Program* lee la configuración del sistema de archivos que tienen los parámetros de ejecución del sistema. Posteriormente crea las instancias de objetos del sistema: la *Petri Net*, el *DataPool*, y todas las instancias de *Bank*, *Processors* y *Evaluators*. Luego, y por razones de diseño y eficiencia, el *Parallel Program* vincula cada *Evaluator* a la *Petri Net*, cada instancia de *Processor* con las respectivas *ThreadPool*, *Evaluator* y *Bank*, y cada *ThreadPool* con su *Processor* y la *Petri Net*. Si bien el Diagrama de Clases no expone un enlace entre cada *Evaluator* y la *Petri Net*, o entre cada *ThreadPool* y la *Petri Net*, estos enlaces son creados para evitar llamadas en cascada y poder tener un acceso directo a la *Petri Net*.

Una vez creadas todas las instancias e inicializadas acorde a los archivos de configuración, las instancias de *ThreadPool* comienzan a corren en paralelo ejecutando los *kernels* que componen las tareas del algoritmo. Cada *ThreadPool* entra en el ciclo de ejecución mientras la TPN no determine el final del algoritmo, intentando ejecutar algún *kernel*, siguiendo al algoritmo de Fig. 3.1.

El Diagrama de Secuencia de la Fig. 3.4 muestra el ciclo de ejecución de cada objeto *ThreadPool*. Si este captura la exclusión mutua, se solicita a *Petri Net* el conjunto de tareas habilitadas. El objeto *Evaluator* usa este resultado y selecciona la más apropiada a ejecutar para el *ThreadPool* en cuestión, o ninguna, si el criterio descarta todas las habilitadas. La selección es hecha a nivel de referencias en la TPN, por lo que, el nombre y parámetros del *kernel*

a ejecutar, necesitan ser decodificados, lo que implica llamar a las funciones τ para obtener la tarea que representa la Transición elegida, γ_i para obtener el *kernel* especifico del procesador que ejecuta la tarea, y δ para obtener los bloques de datos usados por el *kernel*.

Luego que la tarea seleccionada es decodificada y todos los elementos para correr el *kernel* respectivo están disponibles, el Vector de Marcado M es actualizado absorbiendo los *tokens* de las Plazas de entrada de la tarea, y la exclusión se libera. El *kernel* es ejecutado por el *ThreadPool*, quien tiene la responsabilidad del segundo nivel de paralelismo anidado. Luego de finalizada la ejecución, su resultado permite actualizar los datos obtenidos, y los *tokens* son inyectados en la *Petri Net* por medio de la renovación del Vector de Marcado M en las posiciones que representan las Plazas de Salida de la tarea realizada, finalizando el ciclo. En el caso en que el objeto *Evaluator* no seleccione ninguna tarea, o la exclusión mutua no sea capturada, el *Processor* entra en un tiempo de espera de inactividad, y vuelve al principio de la ejecución.

3.3 Implementación del Modelo

Para implementar el diseño, fue desarrollado un *framework* con la premisa de ser fácilmente configurable para un amplio número de casos, no solo por la forma y el número de divisiones de datos, sino también por el *hardware* paralelo a usar para ejecutar el algoritmo.

El desarrollo del *framework* del PEM fue hecho usando el lenguaje de programación Fortran 2003. La selección del lenguaje se basó en dos factores. Primero, como los algoritmos con los cuales se realizaron los pruebas del modelo se descomponen en subtareas que se resuelven con llamadas a rutinas de la biblioteca BLAS, C/C++ o Fortran son los lenguajes indicados para su programación. La elección de Fortran se debió al segundo factor: este lenguaje tiene un manejo de datos matriciales más simple que el de C++. Además, la elección por la versión 2003 se debe a la cobertura de funcionalidades de orientación a objetos que esta versión trae en comparación con versiones anteriores. Estas extensiones al Fortran son de gran ayuda a la hora de desarrollar la implementación de la clases definidas en la etapa de diseño. Como nota de la implementación, todas las clases codificadas tienen una relación 1 a 1 con las respectivas del diseño.

El alcance del *framework* se limita a computadoras multiprocesador de

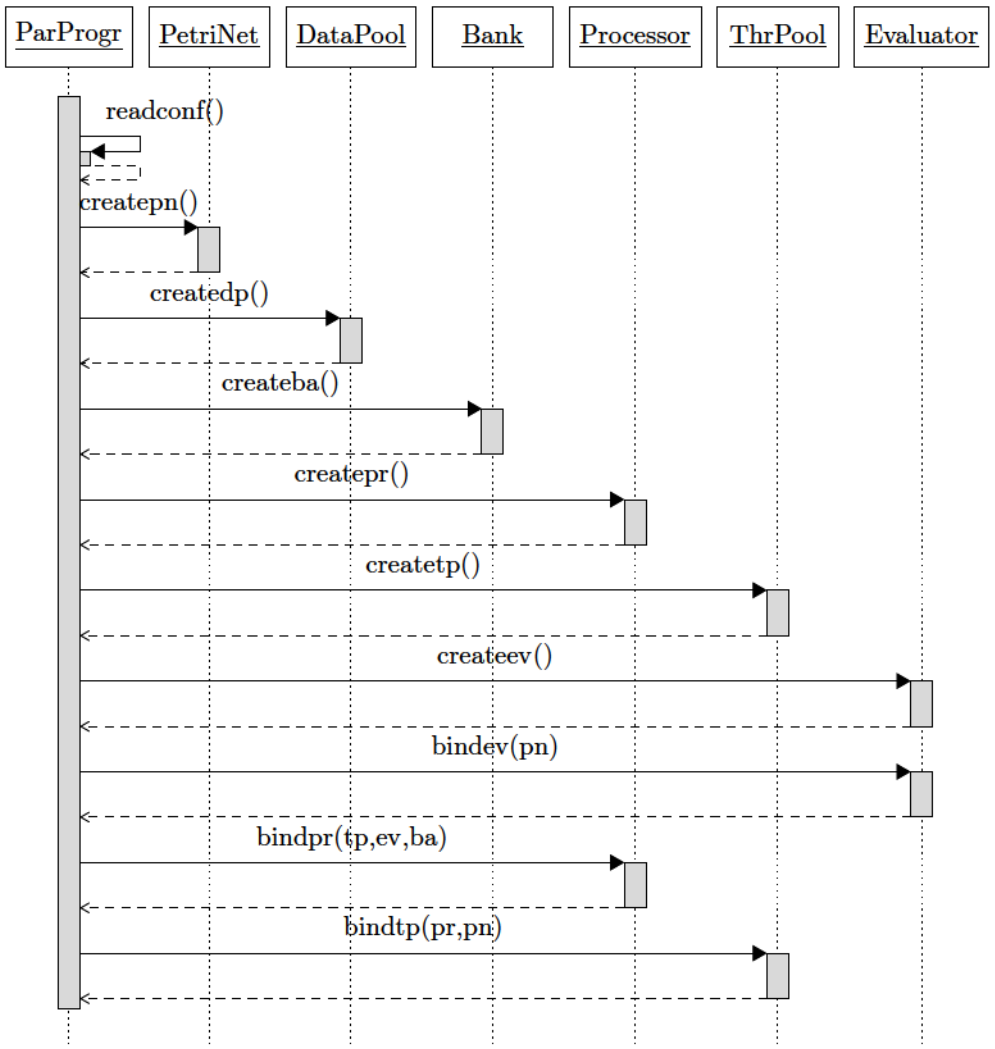


Figure 3.3: Diagrama de Secuencia del proceso de inicialización del hilo principal de ejecución.

memoria compartida. Debido a esto, el paralelismo del conjunto de objetos *Processors* definidos en la fase de diseño, son implementados utilizando la directivas de compilador que implementan el modelo OpenMP [Boa]. El compilador utilizado para generar el programa ejecutable debe implementar la definición de ejecución de hilos anidados OpenMP para poder cumplir con el doble nivel de paralelismo del diseño: el conjunto de *Processors* en un primer nivel y el de hilos internos de cada *Processor*, a un segundo nivel.

Además, el compilador debe satisfacer la funcionalidad de afinidad de los procesadores *multicore*, para poder aceptar la definición de afinidad del objeto

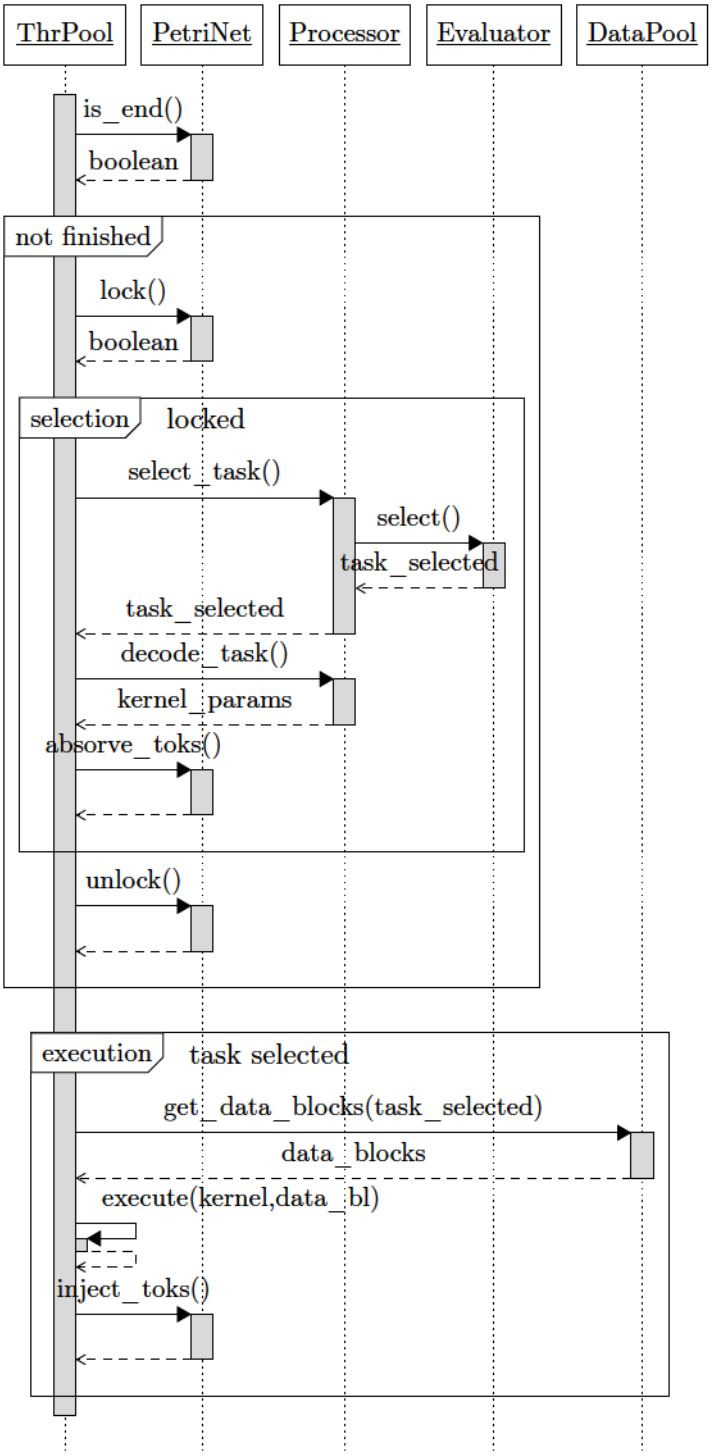


Figure 3.4: Diagrama de Secuencia de un hilo en ejecución.

ThreadPool, lo que permite implementar localidad espacial en la ejecución de los *kernels*.

La asociación de hilos en dos niveles permite ejecutar los programas paralelos en arquitecturas de procesadores heterogéneos, como lo es, por ejemplo, una máquina heterogénea compuesta por varios núcleos en un equipo multiprocesador simétrico (SMP) y por placas coprocesadoras, como las GPGPU, las Xeon Phi, u otras similares. En efecto, como cada objeto *Processor* puede ser configurado individualmente, se puede definir para cada uno de estos, sus parámetros de ejecución. Siguiendo el ejemplo, basta con incluir (*link*) en el ejecutable, los *kernels* a ejecutar en el coprocesador, y las bibliotecas necesarias. Dicho en términos del PEM, enlazar el valor pertinente de la función γ_i. Así, es mínimo el esfuerzo para utilizar máquinas con procesadores heterogéneos. El uso de estos coprocesadores puede ser conjunto a los núcleos del SMP, o como único tipo de procesador a utilizar.

Por otro lado, el *framework* usa un conjunto de archivos con marcas tipo XML para configurar los parámetros. Estas marcas indican los valores de cada parámetro. Por ejemplo, la definición del valor de la función γ_i de cada *Processor*, es decir, el nombre de la rutina a correr para cada tarea, es determinada en el archivo que contiene las marcas y los valores de esta asociación. El nombre del archivo de configuración es pasado como argumento de ejecución al *framework*, por lo que queda configurado en forma simple, según sea la máquina paralela a utilizar. Los detalles sobre el contenido de los archivos de configuración se presentan en la sección próxima.

3.4 El uso del modelo PEM para el programador

El modelo PEM otorga un modelo de ejecución de programas paralelos que ayuda al programador en muchos puntos de la tarea de codificación paralela. Esta sección es destinada a describir la serie de pasos que el programador debe completar, a fines de ejecutar un programa paralelo con PEM, y resaltar sus ventajas.

El modelo se basa en la representación del algoritmo en una CPN. Entonces, el primer paso a realizar es analizar el algoritmo a ejecutar, y realizar las definiciones de la división de datos y tareas, como en cualquier programa paralelo. Esto es crucial en el desarrollo de un programa paralelo, pero en este caso, aún más. En PEM, la representación del algoritmo en una Red

de Petri se basa en las relaciones entre las tareas con las Transiciones, y los bloques de datos con las Plazas, por lo que, la definición de las tareas y la división de datos tienen un impacto muy fuerte en el diseño de la CPN.

En relación a la división de datos, la forma que esta tome impacta en el diseño de la CPN, pero no el número de sus divisiones. La forma impacta en el orden de ejecución de las tareas, y por lo tanto, en la dependencia entre ellas. Por otro lado, el número de divisiones en que es dividido el conjunto de datos, para una forma dada, es introducido como un parámetro de dominio en la CPN. Por ejemplo, en el algoritmo de multiplicación de matrices, la división de las matrices en bloques cuadrados produce una secuencia diferente de tareas que la división por bandas horizontales o verticales. La forma y el número de divisiones determina el dominio de cada Plaza en la CPN final.

En relación con la división de tareas, el número de tareas define el número de Transiciones de la CPN, pero el número de repeticiones en que cada tarea es ejecutada, depende de la división de datos. Una vez que ambas divisiones son definidas, de datos y de tareas, la Red de Petri puede ser construida y probada para garantizar la correctitud de su representación. Las pruebas pueden ser hechas con cualquier herramienta que permita modelar una CPN, como ser el paquete SNAKES [Pom].

El segundo paso es, una vez definida la CPN, desplegarla en una TPN. El producto de este paso debe ser un archivo que contiene la TPN desplegada en notación matricial. El archivo obtenido será usado como parámetro en la ejecución del *framework*. Se aconseja escribir un pequeño *script* que genere este archivo tomando en cuenta el número de divisiones como parámetro de ejecución. El *script* debe respetar los dominios definidos para cada Plaza en la CPN y generar todas las combinaciones de colores que habilitan la respectiva Transición, siguiendo el proceso de desplegado definido previamente en la sección 2.4.

El archivo que contiene la TPN también debe tener referencias a los bloques de datos que representa cada Plaza de la TPN y el nombre de la tarea que representa cada Transición. Esta información es usada en tiempo de ejecución para correr el *kernel* apropiado. También debe contener el Vector de Marcado en su estado Inicial y el Vector de Marcado final.

Luego de que la red desplegada es incluida en el archivo, se deben definir otros parámetros de ejecución. Deben definirse cada uno de los objetos *Processor*, indicando la relación entre las tareas del algoritmo y los *kernels* a ejecutar por cada *Processor*, y su asociación con los respectivos objetos *Evaluator, Bank* y opcionalmente la afinidad con los núcleos del SMP, de ser

necesario. También debe informarse el número de matrices usadas en los cómputos y el número de sus divisiones en filas y columnas.

Solo resta una tarea necesaria para hacer. Es generar el programa ejecutable enlazando (*linking*) el *framework* con los archivos que contienen los *kernels* necesarios para correr. En resumen, esta es la única parte que necesita ser codificada por parte del programador.

Como fue dicho anteriormente, el modelo PEM facilita el proceso de generación de programas paralelos. Con solo hacer el modelo del algoritmo en la CPN, muchos de los problemas de la programación paralela son resueltos: el control de la dependencia de datos, la barreras, los ciclos paralelos, la comunicación entre procesos, y otros elementos propios de la programación paralela, son eliminados desde el punto de vista del programador. Además, el ejecutable es asíncrono, lo cual es un paso importante hacia la obtención de altos rendimientos de ejecución.

Otra facilidad importante del modelo PEM es la adaptabilidad a una múltiple granularidad en la partición de datos. La granularidad de la división no necesita ser única, pudiendo existir varias divisiones simultáneas de los datos. Cada *Processor* elige el bloque de datos más pertinente al *kernel* a ejecutar. El programador debe ser cuidadoso con la representación de este hecho en los dominios de la CPN, para que quede garantizada la completitud y la exactitud del algoritmo.

En resumen, el diseño del PEM permite ser adaptable gracias al uso de archivos de configuración. Estos contienen las Matrices de Incidencia que representan al algoritmo, los Vectores de Marcado, el número y tipo de procesadores, la relación entre Transiciones y tareas y entre Plazas y bloques de datos, además de la referencia a los *kernels* a ejecutar. La adaptabilidad permite hacer cambios:

- en los algoritmos por medio de las Matrices de Incidencia.
- en la partición de datos por medio de la relación entre Plazas y bloques de datos.
- en el número de procesadores a correr en paralelo por medio de sus respectivas definiciones.
- en la arquitectura interna de cada *Processor* por medio de los parámetros del *Threadpool*.
- en el tipo de procesador por medio de los *kernels* relacionados.

- en la obtención de mejores rendimientos por medio de ajuste de la función de evaluación del objeto *Evaluator*.

Se resalta que el modelo propuesto cubre un amplio espectro en el desarrollo de un programa paralelo, que va desde el diseño del algoritmo paralelo hasta su ejecución, no solo para un algoritmo o máquina particular, sino para un gran número de alternativas en la dos coordenadas que componen una ejecución paralela: algoritmo y máquina. Modelando el algoritmo con una CPN como fue descripto en el capítulo anterior, su ejecución paralela requiere poco esfuerzo de programación y adaptación a la máquina donde se ejecute el programa.

3.5 Trabajos relacionados

En capítulo anterior y en las secciones previas fue descripto el modelo PEM presentado en este trabajo. A continuación se presentan los trabajos relacionados al desarrollado aquí.

Con el surgimiento de los Multiprocesadores Simétricos (SMP) o procesadores *multicore* en la última década, se popularizó el concepto de paralelismo a nivel de hilo (*Thread Level Parallelism* - TLP), que se enfoca en el problema de la obtención de rendimientos aceptables en estos procesadores. Las *Lapack Working Notes* (LAWN) son una colección de documentos y tutoriales sobre las rutinas más usuales del álgebra lineal y su implementación computacional eficiente. En la LAWN 191 [BLKD07], se sugiere que para solucionar el problema de la escalabilidad en un gran número de hilos (*threads*) en un SMP, los algoritmos deben poseer dos propiedades principales: granularidad fina y ser asíncronos. Este trabajo se basa en el último concepto, dada la gran pérdida de rendimiento que el sincronismo impone al TLP cuando trabaja sobre un número importante de hilos.

Como un programa paralelo suele tener más tareas habilitadas que procesadores disponibles, es posible hacer diferentes combinaciones de tareas para definir la planificación de la ejecución paralela. Los planificadores estáticos son aquellos que definen la planificación antes de ejecutar el algoritmo. Ejemplos de estos son el *left looking* (LL) y el *right looking* (RL) de los algoritmos de factorización de matrices, que difieren en la prioridad de la actualización de los paneles de la izquierda o de la derecha [KD06, HLYD11]. Ambas estrategias de planificación son expuestas en las Fig. 3.5 y 3.6 para el caso del

```
1  do step=1:bl_nu
2    do i=1:step-1
3      syrk step,i
4    end
5    potr step
6    do j=step+1:bl_nu
7      do k=1:step-1
8        gemm step,k,j,k
9      end
10     trsm j,step
11   end
12 end
```

Figure 3.5: Algoritmo izquierdo (LL) para Cholesky

```
1  do step=1:bl_nu
2    potr step
3    do i=step+1:bl_nu
4      trsm i,step
5      syrk i,step
6    end
7    do j=step+1:bl_nu-1
8      do k=j+1:bl_nu
9        gemm j,step,k,step
10     end
11   end
12 end
```

Figure 3.6: Algoritmo derecho (RL) para Cholesky

algoritmo de factorización de Cholesky, y tienen en común la sincronización basada en el modelo *fork-join*.

Otra técnica conocida de planificación estática es la de *look ahead*. Similar a LL y RL, se basa en que un hilo realiza la tarea de factorización de un bloque mientras en los restantes hilos se realiza la actualización de los bloques intervinientes en las etapas previas de procesamiento. Fue observado que LL y RL son puntos extremos en el espectro de posibilidades de selección de tareas habilitadas, siendo ambos versiones parametrizadas del *look ahead* en el camino de ir de una punta a la otra del espectro (de LL hacia RL)[KD06]. Todas las alternativas generan puntos de inactividad en la ejecución paralela dada la naturaleza estática del planificador.

La granularidad fina impacta en el número de tareas a ejecutar, y a mayor número, más complejo es su administración. En la misma LAWN 191 referida anteriormente [BLKD07] se sugiere el uso de Grafo Dirigido Acíclicos (*Directed Acyclic Graphs* - DAG) para modelar algoritmos, donde los vértices representan las tareas y las aristas representan la dependencias entre ellas. El grafo es conocido también como Grafo de Dependencias. La ejecución asíncrona del algoritmo paralelo es ayudada por el uso de los DAG para controlar la dependencia de tareas. Una vez definido el DAG, es utilizado por el planificador del algoritmo para seleccionar la próxima tarea a ejecutar.

La LAWN 191 también describe el concepto de "Camino Crítico" en el Grafo de Dependencias como el camino que conecta los nodos que tienen el mayor número de aristas salientes. El planificador puede dar prioridad a las tareas que están en este camino para mejorar el rendimiento. Esta

misma estrategia es implementada por un objeto *Evaluator* usado en los experimentos del próximo capítulo.

Los algoritmos basados en *tiles* emergen como una solución al problema del balance de carga para algoritmos de álgebra lineal densa en equipos SMP [BLKD07]. Esta clase de algoritmos han evolucionado desde los basados en bloques columna o fila hacia los bloques cuadrados (*tiles*) de datos. Los algoritmos basados en *tiles* presentan, como también muchos otros algoritmos de la biblioteca LAPACK, dos pasos fundamentales: la factorización de un bloque y la actualización de los bloques que forman la submatriz interviniente en la operación [BLKD07].

Se destaca que la LAWN 191 ha sido una fuente importante de ideas para esta tesis. Sin embargo, carece de explicaciones sobre cómo construir el Grafo de Dependencia, y de como usarlo en tiempo de ejecución, siendo ambos temas de resolución compleja.

Los planificadores dinámicos son introducidos como mejora a los estáticos, ya que seleccionan las tareas en tiempo de ejecución según la disponibilidad de procesadores libres y de tareas habilitadas. Este tipo de planificadores se orienta a prevenir el surgimiento de puntos de inactividad comunes en planificadores estáticos. Sin embargo, son complejos y provocan una sobrecarga en la ejecución del algoritmo [HLYD11].

La investigación de Hogg no demuestra ventajas significativas en el uso de planificadores dinámicos en lugar de estáticos [Hog08]. El se basa en el modelado con DAG de los algoritmos y en el uso de estos grafos para el control de la ejecución. El control de la concurrencia y la implementación del DAG generan una sobrecarga que parecen absorber las mejoras en el rendimiento que el planificador dinámico genera. Los experimentos en el capítulo siguiente demuestran que esto no es necesariamente cierto.

En la misma línea de los planificadores dinámicos, la LAWN 243 [HLYD11] presenta el uso de un parámetro de localidad para ayudar al planificador a seleccionar dinámicamente la tarea a asignar al procesador, acorde a los datos previamente utilizados. Las mejorías en la ejecución paralela debido a la localidad de datos depende del tipo de algoritmo paralelo, si, por ejemplo, son del tipo LL o RL visto antes [KD06, HLYD11]. También dependen del tamaño de la ventana del DAG residente en memoria y del número de *tiles* en que la matriz sea dividida.

El proyecto PLASMA es un proyecto desarrollado por la Universidad de Tennessee [tUoTb, ADD+09] que tiene el objeto de optimizar rutinas del

álgebra lineal densa ejecutadas sobre arquitecturas *multicore*. Se basa en dos conceptos centrales, los bloques de tipo *tile* y el planificador dinámico, en la misma senda de las ediciones anteriores de LAWN. El proyecto se orienta a mejorar el uso del CPU, y a mejorar el paralelismo de las versiones de BLAS y LAPACK preexistentes.

Por medio del uso de los bloques *tile*, PLASMA evoluciona de las implementaciones previas basadas en divisiones de bloques columna o fila, a una implementación basada en pequeños bloques cuadrados que son manejados más eficientemente por la memoria *cache*, y define una granularidad fina de tareas, lo que determina la existencia de muchas tareas que deben provocar la mejora en el paralelismo. Más tareas suponen mayor disponibilidad de procesamiento en paralelo de estas, y así mantener más procesadores corriendo en paralelo. Un número grande de tareas debe ser organizado por el planificador, el que determina la ejecución de tareas en forma dinámica, opuesto al modelo de planificación *fork-join*.

El planificador de PLASMA se basa en un Grafo de Dependencias como el antes descripto, el cual es usado para definir un flujo de ejecución "fuera de orden" (fuera del orden que define un esquema como el LL o el RL). Sin embargo, el criterio para la selección de las tareas a lanzar no es explicado, y sería bueno saber, entre otras cosas, cómo resuelve cuando hay muchas tareas fuera del camino crítico del grafo. [DFLL11].

El proyecto MAGMA es otro proyecto desarrollado en la Universidad de Tennessee [tUoTa], similar a PLASMA, orientado a las computadoras heterogéneas *multicore* con GPU. Implementa las funcionalidades de la biblioteca LAPACK en arquitecturas como la descripta. La estrategia principal de trabajo es procesar la matriz por bloques *tiles* y usar un planificador de tareas dinámico y conducido por las dependencias, llamando a los *kernels* apropiados a cada tipo de procesador.

La solución de MAGMA al problema de la mezcla de procesadores es dividir las tareas a ser computadas por cada tipo de procesador, haciendo que los núcleos de la CPU trabajen sobre tareas complejas de grano fino existentes en el camino crítico del algoritmo y las GPU trabajando en tareas de grano grueso fuera del camino crítico. En términos generales, la CPU usarla para tareas resolubles con rutinas de nivel 2 BLAS, como la factorización de bloques, y la GPU para tareas con rutinas de nivel 3, como la actualización de bloques [TDB10, KLFD13].

El planificador de tareas de MAGMA es declarado como dinámico ya que es basado en el proyecto StarPU (descripto más adelante) para los *kernels*

de GPU y basado en PLASMA para los núcleos de CPU. El dinamismo manifestado es parcial, ya que existe una división estática de tareas entre las CPU y las GPU.

En lo mejor de nuestro conocimiento, todos los intentos de un planificador dinámico se basan en el uso de DAG, los cuales son buenos para representar la estructura del algoritmo, pero no cubren la brecha existente entre el grafo y la definición de la ejecución por parte del planificador. Ambos, ejecución y planificador, son implementados en forma *ad-hoc* de forma aparentemente sofisticada, sin un modelo explícito de ejecución paralela.

Otra fuente de trabajos relacionados son las Redes de Petri. Las Redes de Petri con tiempo y temporizadas (*Time and Timed Petri Nets*) son una evolución de la Red de Petri original que incluyen el tiempo en el proceso de disparo de la Transición. Las redes con tiempo incluyen la noción de un intervalo de tiempo $[a, b], a \leq 0, a \leq b < \infty$, asociado a una Transición, que define el momento en que está habilitada para ser disparada. Si el último tiempo en que una Transición t estuvo habilitada fue c, entonces t vuelve a quedar habilitada en el intervalo $[c + a, c + b]$, y si no es disparada una vez pasado $c + b$, se inhabilita. El disparo en sí es instantáneo [PZ91]. Este tipo de redes no son utilizadas a lo largo de esta tesis.

Las Redes de Petri temporizadas tienen un tiempo asociado a la Transición que actúa como el tiempo del disparo, siendo este, el tiempo, o intervalo de tiempo, que debe transcurrir antes de que los *tokens* sean inyectados a las Plazas de salida. Las Transiciones quedan habilitadas inmediatamente a que las Plazas de entrada satisfagan las restricciones. Las redes temporizadas fueron originalmente desarrolladas para evaluar rendimientos, mientras que las redes con tiempo, para modelar protocolos de comunicación [PZ91, Wan98].

Como fue dicho en la sección previa, las Redes de Petri temporizadas son las más próximas al modelo PEM desarrollado en este documento. La diferencia principal es la semántica del disparo. Mientras que para las redes temporizadas las Transiciones son disparadas al quedar habilitadas, en el modelo PEM, el responsable del disparo es un elemento adicional, un objeto *Processor* inactivo.

Los Sistemas de Eventos Discretos (*Discrete Event System* - DES) son sistemas conducidos por eventos, es decir, sistemas cuyo estado de evolución depende completamente de la ocurrencia de un evento discreto asíncrono en el tiempo [CL08]. El tiempo es tomado como discreto y puede causar cambios en el sistema en ciertos puntos. Los eventos son normalmente considerados

como instantáneos y pueden generar un cambio en el estado del sistema.

Un sistema de colas es un sistema donde hay ciertos recursos que brindan servicios y son utilizados por entidades que esperan por su uso [CL08]. Estos tienen tres características básicas: las entidades o clientes que esperan por un recurso, los recursos por los que se genera la espera, y la cola, que representa la espera. Dado que los recursos son limitados, se debe estudiar el uso eficiente de estos, y la satisfacción de los clientes. Un sistema de colas con la llegada asíncrona de clientes es un caso particular de DES.

Un algoritmo paralelo puede ser visto como un DES, donde los eventos son la disponibilidad, el inicio y la finalización de la tarea, que cambian el estado del procesamiento. También puede ser visto como un sistema de colas donde las tareas esperan por un procesador que las ejecute, derivando en el estudio de la planificación de recursos.

Las álgebras de CPN son presentadas por F. Pommereau en su tesis doctoral [Pom09]. El comienza su estudio con la propiedad de composición de las CPN y el flujo de control de la red, lo que permite definir Redes de Petri con excepciones e hilos de ejecución. Las álgebras están orientadas a modelar la ejecución de la red, lo que motiva la inclusión del flujo de control en el modelo. Este tipo de Redes de Petri tiene como objetivo la verificación del modelo más que su uso para la ejecución real de un problema.

Una diferencia clave con el modelo PEM, es que en este trabajo el control del algoritmo no es incluido en el modelo de la Red de Petri. Los procesadores, o máquinas en términos de la teoría general de la planificación (*scheduling theory*) [RV09], son una parte separada e independiente de la Red de Petri y no se los incluye para nada en la CPN generada. Las Redes de Petri son usadas exclusivamente para modelar el algoritmo independientemente de su ejecución, y nada referido al control de la ejecución es incluido en el modelo del algoritmo.

Un fundamento para no incluir los procesadores en el modelo de la Red de Petri es que hay un número muy grande de procesadores y de posibles máquinas paralelas que puedan correr un algoritmo. Entonces cada máquina necesitaría de un modelo diferente para representar su *hardware*, lo que es un inconveniente grave para un modelo general. Teniendo a los procesadores como un parámetro del *framework* que implementa PEM, es claramente más simple configurar solo este punto de cambio y no todo el modelo.

No incluir el control dentro de la CPN es un criterio que se basa en distinguir el modelo de su implementación. De lo contrario, según cambie el

número u homogeneidad de los procesadores que ejecutan el algoritmo, el modelo cambiará. Si la computadora paralela dispone de n procesadores homogéneos, y el control de la ejecución es incluido en el modelo TPN, un cambio en el número de procesadores a usar, por cualquier motivo que sea, como puede ser una escalabilidad pobre, producirá un modelo TPN diferente, lo que es inadecuado para el programador. Este, por practicidad y para evitar la regeneración del modelo ante cada cambio que pueda surgir, debería generar una serie de modelos TPN alternativos con diferente número de procesadores y elegir el modelo que coincida con los procesadores a utilizar.

En otro caso, y aún conservando fijo el número de procesadores a utilizar en n, pero siendo estos heterogéneos, el control de la ejecución puede ser tan complejo, que demande al programador un gran esfuerzo para definir una política de selección acertada para la TPN desplegada. Además, el nivel de diferencia de capacidad entre los procesadores también influye sobre el control. El modelo definido en esta tesis simplifica estas complejidades y practica una separación de responsabilidades entre la estructura del algoritmo paralelo y la ejecución sobre una máquina paralela en particular.

La facilidad para modelar el algoritmo con independencia de su control es una de las llaves de la flexibilidad del modelo PEM. Así, el algoritmo es exclusivamente modelado con la CPN, y cambios en el algoritmo, como una división de datos diferente, son reflejados en la red. El control de la ejecución es configurado exclusivamente con elementos incluidos en el *framework* del PEM, y cambios en aquella no afectan a la CPN previamente definida. En términos de Ingeniería de Software, se hace uso de las ventajas de un sistema débilmente acoplado [Fow01].

La planificación de tareas (*scheduling*) es otra fuente de trabajos relacionados. El proyecto Quark es un sistema de colas para un planificador dinámico de ejecución de aplicaciones para equipos *multicore* de memoria compartida desarrollado por la Universidad de Tennessee [YKD11]. Es principalmente utilizado en el proyecto PLASMA de los mismos autores [tUoTb], pero es abierto a otras aplicaciones paralelas. La planificación se basa en la dependencia de datos entre tareas, representado por un grafo.

La implementación de Quark infiere la dependencia de datos en tiempo de ejecución con la ayuda de ciertos "*hazards*" incluidos en las rutinas a ser ejecutadas. El código secuencial (*kernel*) debe ser provisto con ciertas anotaciones que son utilizadas para armar el grafo necesario para computar las dependencias. Los *kernels* son ejecutados en forma asíncrona. Estos son encolados y ejecutados siguiendo una política FIFO, y la localidad de datos

es analizada para dar prioridad a tareas que usan como entrada, salidas producidas por el mismo hilo de ejecución. Además, el programador debe proveer los niveles de prioridad que cada *kernel* debe tener a la hora de su ejecución.

No se explica en la documentación del sistema cómo es la estructura interna del sistema de colas y cómo son computadas las dependencias. Algunos datos sobre estos puntos pueden ser tomados de la tesis doctoral de uno de sus autores [Yar12]. Los argumentos de las rutinas encoladas son utilizados para definir la dependencia por medio del ordenamiento de lectura / escritura. El sistema de planificación encola tareas que no esperan dependencias, luego ejecuta la rutina, y una vez concluida, actualiza las tareas que esperan por los datos recién calculados.

El proyecto Quark es destinado a computadoras del tipo SMP, donde todos los procesadores son homogéneos. No hay posibilidades de definir diferentes prioridades a hilos de ejecución, y es muy complicado de ser implementado en sistemas heterogéneos.

A pesar de que muchos de los conceptos de este proyecto son similares a los usados en el modelo PEM, la mayor diferencia se da en que el programador debe aportar el código secuencial y marcar en el las definiciones de los *kernels* que Quark utilizará para la planificación. Es responsabilidad del programador definir en el código fuente, las partes del mismo a ejecutar en paralelo. En este sentido, puede ser visto como una evolución de OpenMP, por la definición de prioridades y la ejecución asíncrona. No obstante, las anotaciones del sistema de ejecución son fijas, y si una modificación en los parámetros es introducida, se debe recompilar el código.

El proyecto StarPU desarrollado en el instituto INRIA es un sistema de tiempo de ejecución basado en un conjunto de tareas dadas como *kernels* para ejecutar, destinado a sistemas heterogéneos de computación [ATN10]. El programado provee *codelets* que son piezas de código para ser ejecutadas en el *hardware*, que incluyen información sobre la dependencia de datos. La misma tarea debe ser codificada en diferentes *codelets* si debe ser ejecutadas en diferente *hardware*.

El sistema de ejecución StarPU también incluye un sistema de planificación. El modelo se basa en "*workers*" que tienen o comparten una cola donde las tareas son informadas. Los datos son manejados y accedidos en StarPu por medio de un mecanismo que abstrae las posiciones de memoria para inferir dependencias. El programador provee información de prioridad de ejecución para cada tarea, y también provee políticas de planificación de

tareas, como lo son FIFO, pila y cola de prioridades, por ejemplo. El historial de rendimiento es utilizado por el planificador para seleccionar la estrategia más eficiente [AAD+10].

En la misma línea de anotaciones en el código fuente también está el proyecto XKaapi [GFLMR13]. Este trabaja usando directivas de compilación incluidas en el código fuente que determina las tareas a ejecutar en paralelo, similar a como se usa OpenMP. El planificador es dinámico y sigue un orden FIFO sin considerar algún otro factor de optimización. Las dependencias son computadas por el sistema a medida que una nueva tarea es necesitada.

El sistema XKaapi soporta sistemas heterogéneos al permitir la definición de varias versiones de código que implementen una misma tarea. El *kernel* correcto es ejecutado con la ayuda de meta-datos provistos por el programador. La ejecución de las tareas es asíncrona y también se utiliza localidad de datos en el proceso de selección de la tarea a ejecutar [FLBGR13].

Tanto StarPU como XKaapi son sistemas de definición de tareas que determinan automáticamente la dependencia de datos y con ejecución asíncrona, basados en anotaciones incluidas en el código fuente original. Sin embargo, no proveen ninguna herramienta que permita hacer el modelado del algoritmo ni un análisis preliminar del mismo. Dada la similitud de ambos modelos con OpenMP, y que la experiencia en el análisis del algoritmo paralelo a partir del código decorado para OpenMP es muy compleja, donde muchos de los factores que impactan en el rendimiento quedan ocultos en el código, la optimización de este tipo de código es compleja y de estilo "prueba y error".

El *Heterogeneous Earliest Finish Time* (HEFT) es una heurística de planificación cuyo objetivo es minimizar el tiempo de finalización ("*makespan*") en plataformas heterogéneas [THW99]. Modela la aplicación paralela con un DAG en múltiples tareas con dependencias de datos, y el problema del planificador es determinar una asignación de tareas a procesadores para que sean completadas en un mínimo tiempo.

La heurística HEFT es un ejemplo de las heurísticas basadas en listas [RV09]. Disponiendo del tiempo promedio para completar cada tarea, el algoritmo ordena las mismas en forma decreciente según el tiempo calculado para llegar a la tarea final, y selecciona para cada procesador, la tarea con mayor valoración. Esta heurística es utilizada en el trabajo de Agullo et.al. [AAD+10] obteniendo los mejores rendimientos.

Una variante de HEFT, que utiliza *lookahead*, fue presentada por Bit-

tencourt et.al. [BSM10]. La variante incluye una graduación de tareas, y la disponibilidad de tareas posteriormente habilitadas al finalizar la ejecución de cada una. Se trata de evitar "cuellos de botella" al valorar mejor aquellas tareas que habiliten más tareas a su finalización.

En términos generales, los trabajos relacionados pueden ser divididos en tres categorías: el modelo, la ejecución y la planificación. La primera incluye a los diversos modelos de Redes de Petri, la segunda incluye a los proyectos Quark, StarPU, XKaapi y sistemas similares, y la última categoría incluye las optimizaciones de los planificadores. Ninguno de ellos cubre las tres categorías como el modelo PEM lo hace: el modelado del algoritmo en alto nivel, luego su conversión a un conjunto de parámetros que generan un programa ejecutable con la ayuda del *framework*, y finalmente, la sintonía fina en el rendimiento con el uso de los objetos de clase *Evaluator*. En lo mejor de nuestro conocimiento, ningún modelo presenta en forma conjunta los tres aspectos de la programación paralela.

En el próximo capítulo son presentados los experimentos que convalidan el modelo presentado, usando como banco de pruebas, a un par de algoritmos del álgebra lineal frecuentemente utilizados.

Capítulo 4

Experimentación

En los capítulos anteriores fueron presentados los antecedentes y el modelo PEM basado en Redes de Petri y los elementos de su diseño, además del *framework* de ejecución del modelo. En el presente capítulo se exponen los experimentos realizados para validar el modelo, sus antecedentes, decisiones de implementación y los resultados de los mismos.

El cápitulo comprende los experimentos realizados con los algoritmos de factorización de Cholesky y de Multiplicación de Matrices, bajo distintos tipos de máquinas paralelas, como son los *multicore* homogéneos y heterogéneos con placas coprocesadoras del tipo GPGPU.

4.1 Algoritmo de Factorización de Cholesky

El primero de los algoritmos que se utilizó para realizar las pruebas con el modelo PEM es el algoritmo de factorización de Cholesky. El mismo ya fue introducido en la sección 2.3 y se repiten aquí las partes esenciales del mismo a los fines de tener a mano el algoritmo y el modelo desarrollado para él.

El algoritmo de factorización de Cholesky es uno de los más usuales en el ámbito del álgebra lineal. Es aplicado para la resolución de sistemas de ecuaciones con ciertas propiedades, que permiten sea más eficiente que un algoritmo general. Dada una matriz A cuadrada, simétrica y definida positiva, de rango r, se la factoriza como $A = L * L^T$, donde L es una matriz triangular.

Los valores de L se calculan con las siguientes fórmulas:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad 1 \leq j < i \leq r \tag{4.1}$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad 1 \leq i \leq r \tag{4.2}$$

Se destaca que ambas ecuaciones imponen una secuencia de pasos a seguir, ya que para computar los valores de la fila i, por ejemplo l_{ii}, por medio de la ec. 4.2, previamente debe haberse computado los valores de l_{ik} en la misma fila, $k < i$, utilizando la ec. 4.1, y una vez completados estos cálculos, recién están disponibles todos los valores necesarios para calcular el elemento de la diagonal principal l_{ii}.

A su vez, para computar cada valor fuera de la diagonal principal, l_{ij}, es necesario calcular primero todos los valores de las filas i y j, hasta la columna $j-1$. Esta secuencia de cálculos define una fuerte dependencia de datos, con un impacto importante en las restricciones a ser respetadas por una ejecución paralela del algoritmo.

El algoritmo puede ser calculado por bloques en lugar de valores individuales, como es presentado en la " *LAPACK Working Notes* 191" (LAWN 191) [BLKD07], y por lo tanto, utilizar las rutinas definidas para bloques de datos en la colección BLAS [BLA01] y LAPACK [ABB+99]. En dicha LAWN se aconseja, y se sigue en este trabajo, una división de datos en forma de bloques cuadros ("*tiles*"). Por lo tanto, una matriz de rango q se la divide en $n \times n$ bloques cuadrados, cada uno de ellos de rango $r = q/n$. Se toma la convención de calcular la parte triangular inferior de la matriz, dado el carácter simétrico que presenta.

El procesamiento por bloques de la matriz utiliza cuatro rutinas, `xpotrf` de LAPACK y `xsyrk`, `xgemm` y `xtsrm` de BLAS. La "x" que encabeza los nombres debe reemplazarse por "s" o"d" según la precisión sea simple o doble. Para el cálculo de los bloques en la diagonal principal se utilizan las dos primeras y para los bloques fuera de la diagonal, los dos restantes, operaciones conocidas como "*panel factorization*" y "*panel update*" respectivamente (LAWN 191 [BLKD07]). Se remite a la figura de página 25, donde se grafican los pasos del algoritmo por bloques.

El gráfico de la Fig. 4.1 muestra una secuencia parcial de pasos en la ejecución del algoritmo, con una división de datos en "*tiles*" de 5×5. Esta

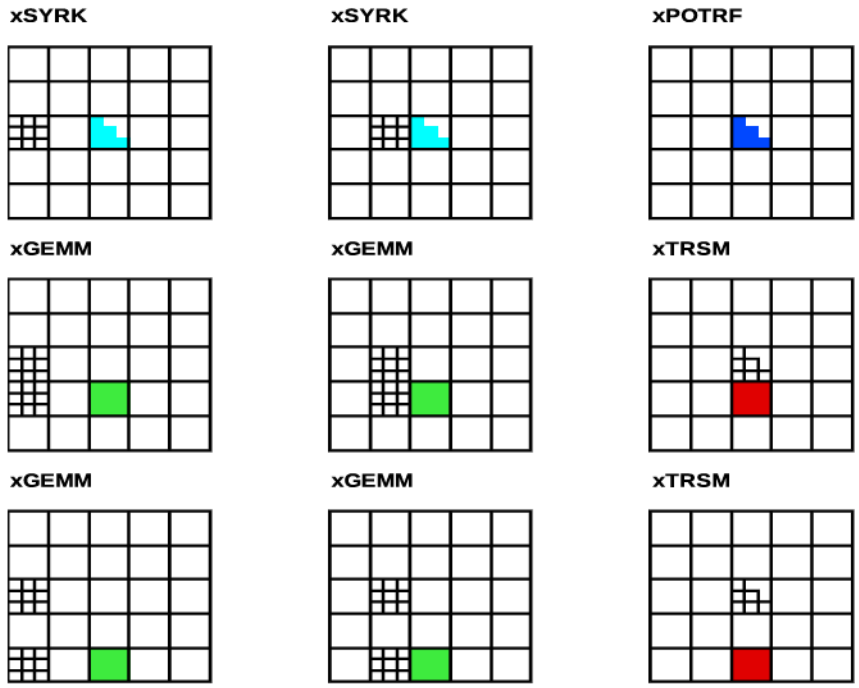


Figure 4.1: Secuencia parcial de pasos en la ejecución del algoritmo de Cholesky, división de datos en "*tiles*", $n = 5$, *"left looking"*.

secuencia se corresponde con una ejecución siguiendo un esquema "*left looking*" (LAWN 223 [LTN+09]). en la primera fila de cuadros se exponen las tareas del *"panel factorization"*, donde se actualiza el bloque de la diagonal principal por medio de la rutina `xsyrk` y luego se aplica la rutina `xpotrf`, obteniendo el resultado final para ese bloque. Luego, las tareas del *"panel update"*, por medio de la multiplicación de matrices (*xgemm*) y la rutina *xtrsm*, con lo cual acaba el ciclo.

El análisis del algoritmo facilita el modelado con la CPN. Como puede apreciarse, son cuatro las rutinas utilizadas a lo largo del algoritmo, lo cual permite definir la misma cantidad de Transiciones en el modelo de la CPN. Además, el análisis de los bloques utilizados en cada una de ellas permite determinar los dominios que corresponden a cada Plaza de la CPN. De esta forma se construye la CPN que modela el algoritmo. La explicación detallada de este proceso fue presentado anteriormente en la sección 2.3, por cuanto se remite a ella. El modelo final del algoritmo es vuelto a presentar en la Fig.4.2 para su uso en esta sección.

A los fines de una presentación formal del modelo del algoritmo siguiendo la definición de CPN de pag. 17, donde se planteó que una CPN es:

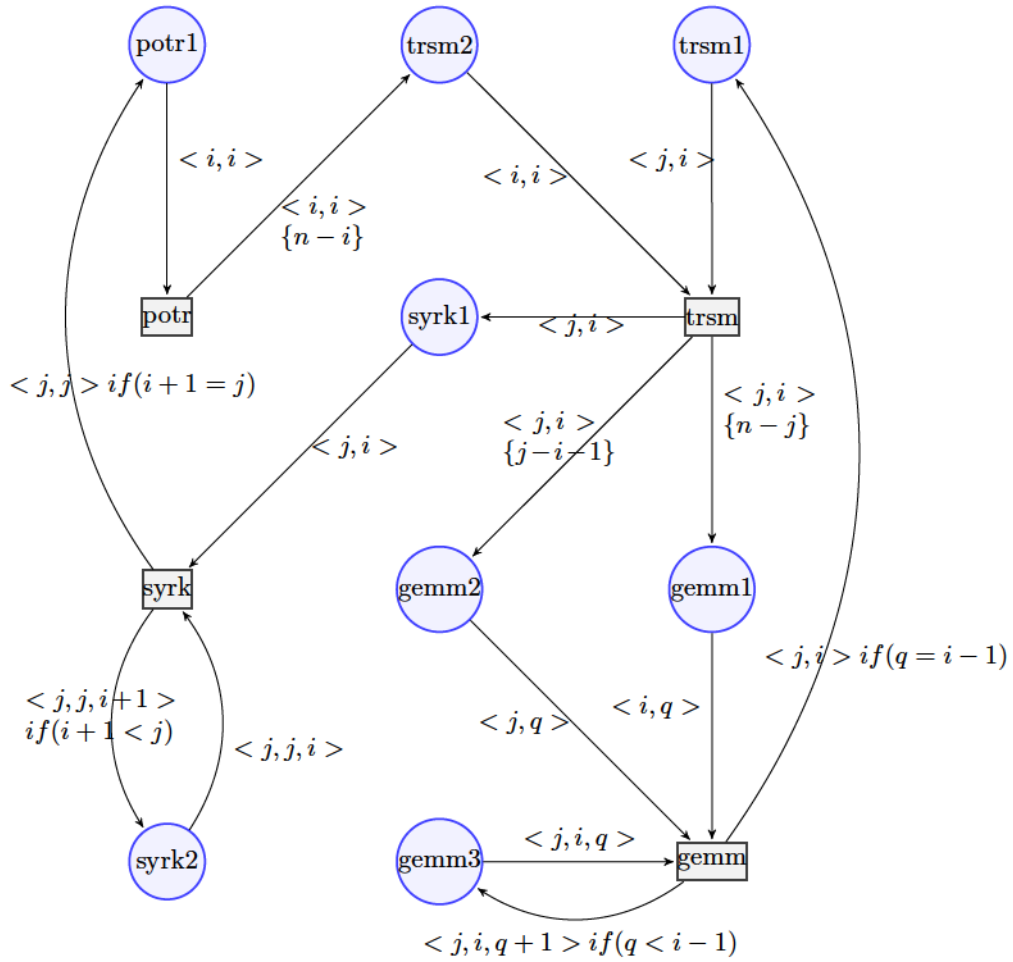


Figure 4.2: Red de Petri coloreada que representa al algoritmo de factorización de Cholesky, decorado para su ejecución paralela.

$$CPN = (P, T, A, \Sigma, V, C, G, E) \tag{4.3}$$

se describirán cada uno de los conjuntos que conforman la definición.

- El conjunto de Transiciones T es:

  $T = \{potr, trsm, syrk, gemm\}$

- El conjunto de Plazas P es:

  $P = \{potr1,\ trsm1,\ trsm2,\ syrk1,\ syrk2,\ gemm1,\ gemm2,\ gemm3\}$.

El número de Plazas de entrada de cada Transición se corresponde con el número de argumentos que la respectiva rutina tiene.

- El conjunto de arcos dirigidos A es:

  $A = \{(potr1, potr),\ (trsm1, trsm),\ (trsm2, trsm),\ (syrk1, syrk),$
  $(syrk2, syrk),\ (gemm1, gemm),\ (gemm2, gemm),(gemm3, gemm),$
  $(potr, trsm2),(trsm, syrk1),(trsm, gemm1),(trsm, gemm2),$
  $(syrk, syrk2),(syrk, potr1),(gemm, gemm3),(gemm, trsm1)\}.$

- El conjunto de dominios Σ es:

  $\Sigma = \{D1, D2, D3, D4, D5\}$
  $D1, D2, D3 \subset \{\mathbb{N} \times \mathbb{N}\}$
  $D4, D5 \subset \{\mathbb{N} \times \mathbb{N} \times \mathbb{N}\}$

- El conjunto de variables V es:

  $V = \{i, j, q\}$

- La función de asignación de dominios a las Plazas C es:

$$C(x) = \begin{cases} D1 & \text{if } x = potr1, trsm2 \\ D2 & \text{if } x = trsm1, syrk1, gemm1 \\ D3 & \text{if } x = gemm2 \\ D4 & \text{if } x = syrk2 \\ D5 & \text{if } x = gemm3 \end{cases}$$

- La función de guardas en las Transiciones G es:

  $G(x) = true \quad \forall x \in \{potr, trsm, syrk, gemm\}$

  Las Transiciones tienen implícita la condición de igualdad de las variables de los *tokens* de entrada.

- La función de guardas en los arcos E es:

$$E(x) = \begin{cases} (q = i - 1) & \text{if } x = (gemm, trsm1) \\ (q < i - 1) & \text{if } x = (gemm, gemm3) \\ (i + 1 = j) & \text{if } x = (syrk, potr1) \\ (i + 1 < j) & \text{if } x = (syrk, syrk2) \end{cases}$$

Estas guardas se corresponden con las condiciones de acumulación de resultados dada la división en *"tiles"*.

La tabla en Fig.4.3 contiene la definición precisa de los dominios de cada una de las Plazas de la CPN de Fig.4.2. El dominio D1 corresponde a los bloques en la diagonal principal de la matriz $< i, i >$. El dominio D2 corresponde a los bloques en la parte triangular inferior que son argumento de `xsyrk` y `xtrsm`, y el primer argumento de `xgemm`. El dominio D3 es similar, pero para el segundo argumento de `xgemm`, que es de una fila mayor a la del primero. D4 es el dominio de la diagonal principal de `xsyrk`, pero que lleva una dimensión adicional para controlar la secuencia. Igual sucede con D5 para el tercer argumento, el resultado, de `xgemm`.

Los valores que tiene el Vector de Marcado, M, antes del inicio de la ejecución, se corresponden con los bloques de la matriz en su estado inicial, es decir, los bloques $< 1, 1 >, < 1, 2 >, \ldots, < 1, n >, < 2, 1 >, \ldots, < n, n >$ están repartidos por las Plazas de la CPN, de la siguiente manera:

Plaza en CPN	Valores
potr1	$< 1, 1 >$
trsm1	$< j, 1 >, j = 2 \ldots n$
syrk2	$< i, i, 1 >, i = 2 \ldots n$
gemm3	$< j, i, 1 >, j = 3 \ldots n, i = 1 \ldots j - 2, j > i$

lo cual representa que la única tarea habilitada es `xpotr` con $< 1, 1 >$. Los restantes bloques están repartidos, los de la columna uno bajo la diagonal principal, como argumentos de `xtrsm`, los de la diagonal principal, como argumentos de `xsyrk`, y los restantes, las columnas $2 \ldots n-1$ bajo la diagonal, como argumentos de `xgemm`.

El Vector de Marcado Final, M_f, contiene cero en todas sus posiciones, es decir, que los *tokens* fueron absorbidos por las Transiciones y no se volvieron a generar. La última tarea, `xpotr` aplicada al último bloque bloque de la diagonal principal, el bloque $< n, n >$, no genera ningún *token*, ya que la salida de dicha Plaza tiene como indicador de repetición, a la expresión $n - i$, por lo que cuando $i = n$, se generan cero *tokens*.

Nombre	Plaza en CPN	Dominio
D1	potr1 trsm2	$< i, i >, i = 1 \ldots n$
D2	trsm1 syrk1 gemm1	$< j, i > j = 2 \ldots n, i = 1 \ldots j-1, j > i$
D3	gemm2	$< j, i >, j = 3 \ldots n, i = 1 \ldots j-2, j > i$
D4	syrk2	$< j, j, i >, j = 2 \ldots n \wedge i = 1 \ldots j-1 \wedge j > i$
D5	gemm3	$< j, i, q >, j = 3 \ldots n, i = 2 \ldots n-1,$ $q = 1 \ldots i-1 \wedge j > i \wedge i > q$

Figure 4.3: Dominios de las Plazas de la Red de Petri Coloreada de la Fig.4.2.

Puede verse la correspondencia entre el gráfico de secuencia de tareas de Fig.4.1 y el modelo de CPN de Fig. 4.2. En efecto, en el gráfico de secuencia, la primera fila de cuadros corresponde con la realización de las Transiciones (tareas) *syrk*, y una vez completa toda la fila, se realiza *potr*. La segunda y tercera fila de cuadros en el gráfico de secuencias se corresponden con la realización de *gemm* y *trsm*. Previo a la realización de `xtrsm`, todos los bloques bajo la diagonal deben haber sido actualizados con `xgemm`, lo cual se refleja en la CPN por la dependencia que genera el arco de salida de *gemm* hacia la Plaza *trsm*1. La dependencia de `xtrsm` hacia `xgemm` es implícita en el gráfico de secuencia, dado que los bloques actualizados con `xtrsm` en la última columna, servirán de argumento de entrada para el *update* de la siguiente columna.

Para realizar el despliegue de la CPN en TPN se debe realizar el conteo de las repeticiones de cada Transición. Se mantiene la nomenclatura del número de divisiones en *"tiles"* como n, existiendo por lo tanto $n \times n$ bloques cuadrados. El cuadro en la Fig. 4.4 muestra el número de veces que cada tarea se repite. La tarea `xpotr` se repite tantas veces como bloques existan en la diagonal principal. Las tareas `xtrsm` y `xsyrk` tienen un conteo que arroja un valor igual a la sumatoria de 1 hasta $n-1$, lo cual se refleja en las respectivas fórmulas, con un orden de crecimiento cuadrático. El conteo de `xtrsm` va desde $n-1$ hasta 1 por columnas, y el de `xsyrk` va desde 1 hasta $n-1$ por filas.

El conteo para la repetición de la tarea `xgemm` es un más complejo. El número de repeticiones es la sumatoria para q de 1 hasta $n-2$ del número triangular para q, lo que es equivalente al número tetrahédrico para $q-2$ [Wei]. Este valor tiene un orden de crecimiento cúbico.

Tarea	Repeticiones
xpotr	n
xtrsm	$\sum_{i=1}^{n-1} \; i = \frac{n \times (n-1)}{2} = \frac{(n^2-n)}{2}$
xsyrk	$\sum_{i=1}^{n-1} \; i = \frac{n \times (n-1)}{2} = \frac{(n^2-n)}{2}$
xgemm	$\frac{(n-2) \times (n-1) \times n}{6} = \frac{(n-2)^3}{6} + \frac{(n-2)^2}{2} + \frac{(n-2)}{3}$

Figure 4.4: Número de repeticiones de cada tarea en la ejecución del algoritmo de Cholesky, para un número n de divisiones en *"tiles"*.

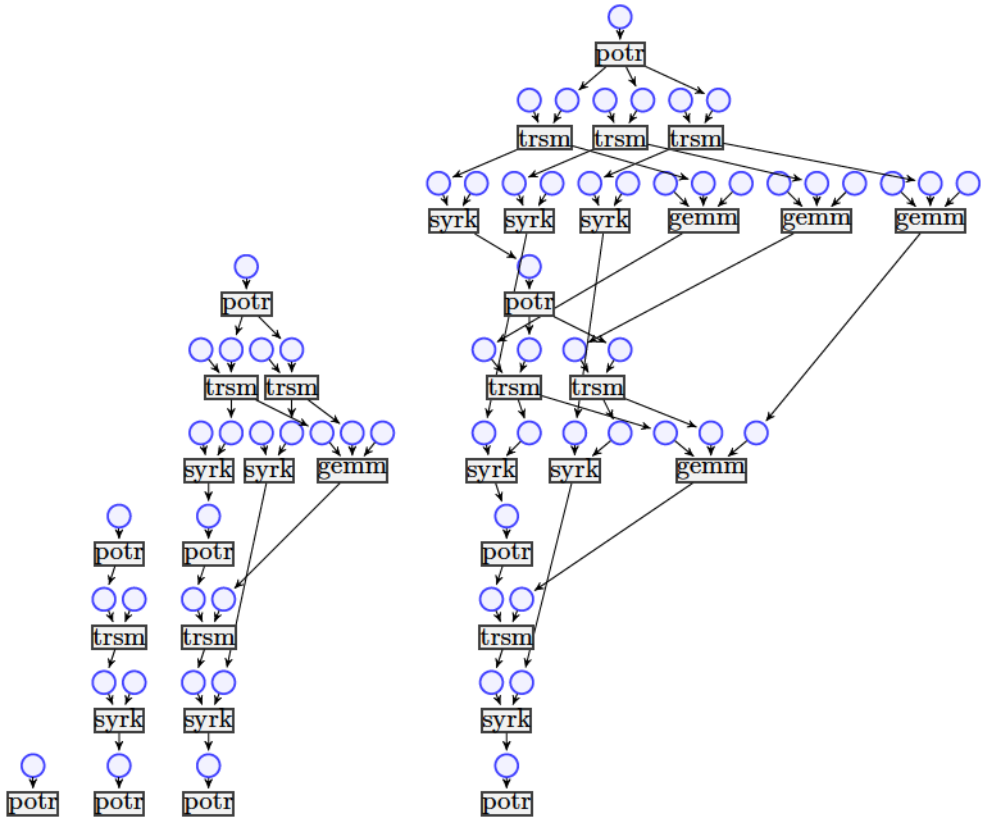


Figure 4.5: TPN desplegada desde la CPN de Fig.4.2 usando diferente número de divisiones en *"tiles"* (n).

op \ n	1	2	3	4	5	6	8	10	12	15	20
potr	1	2	3	4	5	6	8	10	12	15	20
syrk	0	1	3	6	10	15	28	45	66	105	190
trsm	0	1	3	6	10	15	28	45	66	105	190
gemm	0	0	1	4	10	20	56	120	220	455	1140
total	1	4	10	20	35	56	120	220	364	680	1540
tareas secuen.	1	4	7	10	13	16	19	22	25	28	31

Figure 4.6: Número de cada una de las tareas según el número de divisiones en *"tiles"* del algoritmo de Cholesky.

La red TPN permite realizar un interesante análisis del algoritmo para su ejecución paralela. El mismo se halla graficado en la Fig. 4.5, donde se muestran los casos desplegados para una división de *"tiles"* de $n = 1, 2, 3, 4$. El gráfico muestra el número ascendente de tareas que se van generando a medida que aumenta el número de divisiones, manteníendose a un mismo nivel las tareas restantes para finalizar la ejecución. Además, cada nivel tiene las tareas que quedan habilitadas al haberse finalizado las del nivel superior.

Del análisis del gráfico de la Fig. 4.5 se desprende la existencia de un "camino crítico" formado por la secuencia de tareas que genera una dependencia de datos que no puede ser sobrepasada. A cada incremento en el número de divisiones, se genera una secuencia formada por las tareas `xpotr`, `xtrsm` and `xsyrk` que deben ser ejecutadas en forma secuencial.

Otro hecho importante en el análisis de la ejecución paralela es la existencia de puntos de "cuello de botella" generados cada vez que se realiza la tarea `xpotr`, por lo que para evitar la inactividad de los procesadores paralelos, se debe tratar de evitar de completar todas las tareas disponibles a cada nivel (o altura) del algoritmo, avanzando sobre `xpotr` cada vez que esté habilitada.

La tabla de la Fig. 4.6 muestra el número de veces que cada tarea debe realizarse según sea el número de divisiones. La tarea `xpotr` tiene un crecimiento lineal, `xtrsm` y `xsyrk` cuadrático y `xgemm` cúbico, por lo que a los fines de una ejecución de alto rendimiento debe optimizarse la ejecución de ésta última. El último renglón de la tabla presenta el número de tareas secuenciales, acorde a lo expresado en el párrafo anterior.

Del estudio del gráfico de despliegue y de la tabla de cantidad de operaciones, puede desprenderse que es conveniente tener un alto número de divisiones a los fines de aprovechar el potencial de cálculo de una máquina con

más de cuatro procesadores paralelos. En efecto, supongamos que el número de divisiones de "*tiles*" es de ocho $n = 8$. En este caso tendremos un total de 120 tareas a realizar, pero con 19 pasos secuenciales, por lo que haciendo un cálculo redondeado, $120/20 \approx 6$, es el número de procesadores utilizado en promedio por la ejecución paralela. Una subutilización de la capacidad de procesamiento es evidente.

Sin embargo, debe tenerse en cuenta el número de tareas generadas por un alto valor de n, ya que para el caso en que $n = 20$, se generan más de mil quinientas tareas, lo que impone una gravosa carga de procesamiento en la administración de las tareas que debe ser evitada. En este caso, las Matrices de Incidencia podrían ocupar más espacio y requerir más procesamiento que los bloques de datos sobre los que se quiere realizar el cómputo necesario. Por lo que se requiere un equilibrio entre el número de procesadores paralelos y el número de divisiones de la matriz para optimizar rendimiento.

4.1.1 Ejecución Simulada

Con la finalidad de poder realizar una prueba preliminar del modelo, se desarrolló un simulador en lenguaje de alto nivel que permita comprobar el comportamiento del mismo. Para ello se programó en lenguaje Smalltalk un simulador que permite ejecutar hilos, e implementar en un lenguaje con orientación a objetos, el modelo diseñado.

En el trabajo con el simulador del modelo PEM [WDG13] se simuló la ejecución del algoritmo de Cholesky en una máquina paralela compuesta por cuatro placas GPU del tipo NVIDIA GTX 470. Se obtuvieron los tiempos de ejecución de cada rutina y esos tiempos fueron usados en la simulación. Los tiempos se tomaron para diversos tamaños de bloques cuadrados utilizando la biblioteca CUDA 4.0 [Corc], como implementación de BLAS para GPGPU, para las rutinas `xtrsm`, `xsyrk` y `xgemm`. Para la rutina `xpotr` se tomó el tiempo utilizando la implementación de la biblioteca MAGMA [tUoTa]. En todos los casos se incluyó el tiempo de la comunicación de datos desde la memoria principal a la placa GPGPU y viceversa para la devolución del resultado. Los tiempos obtenidos se muestran en la tabla de Fig. 4.7. Se asumió que por cada placa GPGPU se utiliza un *thread* de CPU para su control.

Una de las principales pruebas que fueron realizadas en la simulación es la referidas al *scheduling* dinámico. Para ello se definieron cuatro estrategias de *scheduling* distintas. Dos de ellas fueron las clásicas *"left looking"* (LL) y *"right*

Rutina	Pres. simple 6000	Pres. doble 6000	Pres. simple 8000	Pres. doble 8000
xpotr	0.249	0.882	0.509	1.895
xtrsm	0.568	2.018	1.122	N/A
xsyrk	0.465	1.907	1.001	N/A
xgemm	0.755	3.506	1.678	N/A

Figure 4.7: Tiempos obtenidos para las rutinas ejecutadas en una placa GPGPU NVIDIA GTX 470, en segundos.

looking" (RL) (ver pag. 52). Recordemos que ambas son sincrónicas, y que *"left looking"* actualiza los bloques en la columna actual con los argumentos a la izquierda, mientras que *"right looking"* actualiza bloques a la derecha de la columna actual [KD06].

Dos estrategias dinámicas fueron simuladas, ambas basadas en el DAG de dependencia, con diferencia en la métrica utilizada para la selección entre varias tareas habilitadas. La primera, llamada *"height tree (HT)"*, selecciona la tarea habilitada que está en una posición superior en el grafo de dependencia, es decir, actua como una cola FIFO. La segunda, llamada *"inverse tree (IT)"*, selecciona la tarea que tiene un camino más largo hasta el final de la ejecución, es decir, que tiene mayor número de tareas por delante, sin tener en cuenta el tiempo de ejecución de cada una de ellas. En caso de igualdad, se aplica una selección no determinística.

Las figuras 4.8 y 4.9 muestran ejemplos de los DAG para los *schedulers* dinámicos utilizados en las pruebas. En HT el nivel de la tarea es asignado según la etapa en que la misma quede habilitada. En IT, de acuerdo al número de tareas restantes en su camino más largo hasta el final. Por ejemplo, la tarea *trsm41* tiene nivel 8 en el primer caso, pero nivel 6 en el segundo.

Las diferencias entre ambos *schedulers* se muestran en el siguiente ejemplo. Suponga que tiene una máquina con tres procesadores paralelos homogéneos. La primera tarea a ejecutar es *potr11*, y en la segunda etapa *trsm21, trsm31* y *trsm41*. En la tercera etapa, *scheduler* HT selecciona cualquiera entre las 7 que tiene disponible a esa altura de la ejecución, pero el *scheduler* IT selecciona *syrk21, gemm2131* y *gemm2141*. Puede verse en la Fig. 4.9 que *syrk21* es una tarea prioritaria en el camino al final de la ejecución porque habilita a *potr22*. El *scheduler* HT, puede dilatar su selección a una etapa posterior, impactando en el tiempo total de ejecución.

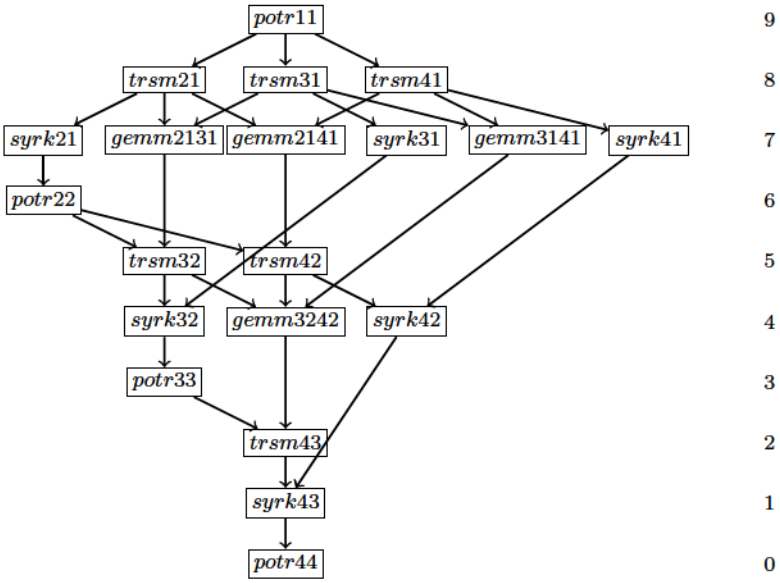


Figure 4.8: Grafo de Dependencia del algoritmo de Cholesky, división 4×4. Los valores de la derecha referencian el número de etapa en que la tarea queda habilitada, mayor valor es anterior.

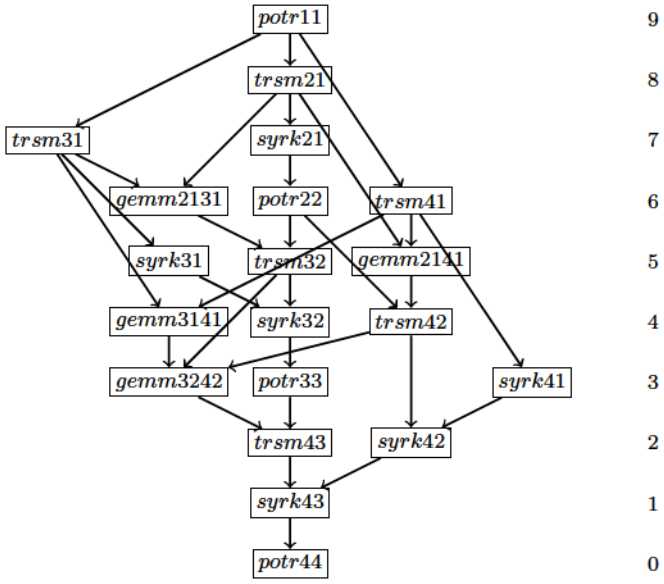


Figure 4.9: Grafo de Dependencia del algoritmo de Cholesky, división 4×4. Los valores de la derecha referencian el número de etapa en que la tarea más tardía en que debe ser realizada, mayor es anterior.

En la tabla de Fig. 4.10 se presentan los resultados de las simulaciones de ejecución para cuatro procesadores con bloques (*tiles*) de rango 6000 y 8000, precisión simple, siendo estos tamaños representativos de la simulación en general. La métrica de rendimiento utilizada es la inactividad de los procesadores, calculada como la diferencia entre el total de la ejecución y el tiempo de efectivo cálculo. Par los *scheduler* dinámicos, el nivel de altura que cada tarea tiene fue calculado por el objeto *Evaluator* con un patrón de diseño *"singleton"* al iniciar la ejecución del programa.

Tam. Bloque	*tiles*	Procs.	Algor.	Tiempo (seg)	% tiempo inact.
6000	6	4	LL	17.50	53.66
6000	6	4	RL	13.26	38.77
6000	6	4	HT	10.16	20.56
6000	6	4	IT	9.51	14.97
6000	8	4	LL	40.59	53.45
6000	8	4	RL	26.33	28.47
6000	8	4	HT	20.98	10.95
6000	8	4	IT	20.69	9.65
8000	6	4	LL	36.89	53.35
8000	6	4	RL	27.79	38.08
8000	6	4	HT	21.37	19.64
8000	6	4	IT	19.95	13.96
8000	8	4	LL	85.34	53.16
8000	8	4	RL	55.05	25.37
8000	8	4	HT	44.30	10.21
8000	8	4	IT	43.71	8.97

Figure 4.10: Resultado de la simulación para división en *"tiles"* de 6 y 8.

Para el algoritmo de factorización de Cholesky, el *scheduler* RL reporta los mejores resultados para un *scheduler* estático, lo que es consistente con los resultados de la LAWN 223 [LTN+09]. Para un *scheduler* dinámico, algoritmo IT da resultados próximos al óptimo. Las figuras 4.11 and 4.12 presentan un gráfico de línea de tiempo para los *scheduler* RL e IT.

Figure 4.11: Línea de tiempo de la simulación para el *scheduler* RL, *tiles* con $n = 8$, rango del bloque de 8000 y 4 procesadores

Figure 4.12: Línea de tiempo de la simulación para el *scheduler* IT, *tiles* con $n = 8$, rango del bloque de 8000 y 4 procesadores

En los gráficos de las figuras anteriores se destacan dos cosas: primero, el tiempo inactivo de los procesadores en los puntos de sincronización para RL, lo que sucede al acercarse a la tarea *potr* de un nuevo ciclo, segundo, y la prácticamente ausencia de tiempos inactivos para IT. Como se expuso en la sección anterior, la existencia de una secuencia de tareas secuenciales no puede ser evitada, ni al comienzo de la ejecución, ni al final. Fuera de estos tramos, no existen tiempos inactivos para este algoritmo, lo que se refleja en los resultados expuestos en la tabla respectiva.

4.1.2 Ejecución en el *framework*

Con resultados tan promisorios en el simulador, la próxima etapa consistió en codificar el *framework* y realizar ejecuciones con datos y comunicaciones reales. Para ello fue implementado el diseño en lenguaje FORTRAN, versión 2003, utilizando las funcionalidades de orientación a objetos que la misma tiene. Además se respetó la arquitectura de anidamiento de *threads* en dos niveles para poder tener una configuración flexible de los procesadores paralelos y probar combinaciones para lograr mejores rendimientos.

Los experimentos se realizaron sobre dos máquinas SMP distintas, una con procesadores AMD y otra con procesadores Intel, de forma tal que sea necesario realizar ajustes en el configuración de los procesadores en cada máquina con la finalidad de optimizar rendimientos y que sean diferentes en cada equipo [WDG14b].

La máquina AMD consta de cuatro *chips* AMD 6344 de 12 *cores* cada uno, lo que hace un total de 48 procesadores para memoria compartida a 2600 Mhz. La otra máquina tiene dos *Intel Xeon E5-2680* de 8 *cores* cada una con *hypethreading*, lo que hace un total de 16 *cores* y 32 hilos de ejecución a 2700 Mhz.

4.1.3 La máquina con procesadores AMD

En la máquina AMD se utilizó el compilador gfortran 4.7.2 y la biblioteca ACML 5.3.0, los cuales son los más apropiados para su arquitectura, ya que el compilador dispone de anidamiento de hilos y afinidad de *cores* para la arquitectura AMD, y la biblioteca de rutinas matamáticas ACML, es la desarrollada por la misma empresa AMD.

Para la configuración de los experimentos con el *framework* se mantuvo

la misma definición del algoritmo para la CPN y su red desplegada, dadas en la sección anterior. Solo fue necesario configurar el tipo y número de objetos *Processor* para cada máquina paralela, el número de cores que conforman a cada uno de ellos, la afinidad de estos para aprovechar la localidad espacial, y el soporte informático de la función γ_i, que asigna el *kernel* a ejecutar para cada tarea. Dado que se utilizaron dos diferentes bibliotecas como implementación de BLAS, en cada caso hubo que ajustar γ_i. Cada uno de estos parámetros fue configurado en archivos de configuración de formato XML.

La arquitectura de la máquina AMD tiene cuatro *chips* de 12 *cores* cada uno. Cada *chip* tiene dos bloques de memoria *cache* L3 asociado a seis *cores*, por cuanto la localidad espacial es mejor de a seis *cores* vecinos que comparten dicha memoria L3. Para operaciones de punto flotante, el microprocesador comparte cada dos *cores* una unidad de multiplicación y suma fusionada (FMA) que puede realizar simultáneamente ambas operaciones con registros de 256 bits, lo cual mejora el rendimiento de este tipo de operaciones sobre el procesador básico. Como la biblioteca ACML utilizada hace uso de estas unidades, se limitó a 24 el número máximo de objetos *Processor* que se utilizarán para esta máquina.

La arquitectura de la máquina AMD, utilizando las unidades FMA, permite configurar un rango de procesadores lógicos que abarca desde combinaciones de 24×1, es decir 24 de una sola unidad FMA, hasta 4 × de 6 FMA's cada uno. El rendimiento en el procesamiento que se logre alcanzar depende de lo eficientemente que la biblioteca ACML esté programada para utilizar estas arquitecturas.

La disposición lógica del *hardware* finalmente utilizada en las pruebas ha sido de n×1 procesadores lógico debido a dos causas. La primera, aunque de menor peso, es que los experimentos están destinadas a probar la bondad del modelo PEM, por lo que un mayor número de procesadores implica una mayor carga de administración del procesamiento paralelo, lo cual permite comprobar las ventajas o los límites del mismo. La segunda causa, es que la versión de ACML utilizada, la más avanzada en su momento, tiene un pobre *speedup* al usar más de una unidad FMA. En efecto, la versión que implementa ejecución paralela con FMA, escala pobremente para las rutinas *ssyrk*, *strsm* y *spotrf* cuando se configura para varios hilos (6,8,etc.).

En consecuencia, cada objeto *Processor* ejecuta el *kernel* de la versión secuencial de ACML con uso de FMA. Sin embargo, y a los fines de evitar solapamiento, la afinidad con los *cores* físicos de cada procesador lógico es importante, para que cada uno de estos utilice en forma excluyente una

procs		8		16		24	
rango	*tiles*	segs	flops	segs	flops	segs	flops
12000	8	3.14	183	2.89	199	3.12	185
	12	2.91	198	2.21	260	2.38	242
	15	4.05	142	4.07	141	4.42	130
24000	8	22.11	208	18.98	243	20.33	227
	12	19.28	239	13.49	342	14.98	308
	15	19.21	240	12.06	382	13.73	336
30000	8	43.05	209	36.99	243	40.90	220
	12	36.02	250	25.11	358	25.74	350
	15	35.50	254	23.11	389	23.42	384

Figure 4.13: Tiempo en segundos y rendimiento en GFlops para pruebas con rango de matriz de 12000, 24000 y 36000; 8, 16 y 24 procesadores lógicos; división en *tiles* para $n = 8, 12, 15$, en la máquina AMD.

tiempo	xpotr	xsyrk	xtrsm	xgemm
ejecución	0.1010	0.2098	0.2042	0.2853
preparac	0.0039	0.0066	0.0049	0.0045

Figure 4.14: Tiempo promedio en segundos para cada rutina utilizada en el algoritmo para el caso de 16 procesadores, división de *tiles* $n = 15$ y rango de matriz 24000, en la máquina AMD.

unidad FMA sin tener que compartirla con otro procesador lógico. Para ello se definió que los 48 *cores* sean divididos en 24 bloques de 2 *cores* consecutivos cada uno, de forma tal de que cada unidad FMA es solo utilizada por un único procesador lógico.

Los resultados de algunas pruebas con la disposición recién expuesta son presentados en la tabla de Fig. 4.13. El cuerpo de la misma contiene los tiempos en segundos y el rendimiento en *flops* obtenidos para diversas combinaciones de división de *tiles* ($n = 8, 12, 15$), rango de matriz a factorizar y número de procesadores lógicos utilizados. Todos los resultados corresponden a pruebas realizadas siguiendo el *scheduler* IT de la sección anterior. Por brevedad se presentan los resultados más representativos. Las pruebas con una división de *tiles* menor que 8 y mayor a quince dan resultados cuyo rendimiento es peor al expuesto y dejan de ser significativos.

La tabla de Fig. 4.14 contiene los tiempos de ejecución promedio de cada rutina utilizada en el algoritmo, implementación ACML 5.3.0, utilizando la unidad FMA, un único *core*, obtenidos de la ejecución del algoritmo para un rango de 24000, división de *tiles* $n = 15$ y 16 procesadores lógicos, es decir, para un bloque cuadrado de rango 1600. Está expuesta como referencia de

los tiempos unitarios insumidos por cada rutina. Adicionalmente, expone los tiempos de *"overload"* promedio, que incluye el tiempo de selección de la tarea, búsqueda de los datos respectivos y la inserción de *tokens* al finalizar la ejecución, es decir, toda la sobrecarga debida al modelo y al *scheduler* utilizado.

El análisis de los resultados de las pruebas sobre las máquina AMD expuesto en las dos tablas anteriores permite concluir que:

- Un número pequeño de división en *tiles*, menor a 8, genera insuficientes tareas para un número de 8 o más procesadores. Por el contrario, un alto número de divisiones genera una sobrecarga de procesamiento, por lo que se evidencia la necesidad del balance entre la cantidad de tareas paralelas y el número de procesadores paralelos utilizados.

- La sobrecarga de cada tarea individual es del orden de los milésimos de segundo, por lo que no conviene para tareas que se ejecuten en el orden de las centésimas, al menos deben ser del orden de las décimas de segundo, es decir dos órdenes superiores. En particular, en las pruebas realizadas, la sobrecarga promedió el 2% del tiempo de ejecución de cada tarea, lo cual es bajo impacto sobre el total de la ejecución.

- La división de *tiles* $n = 15$ genera 680 tareas con 1800 parámetros, el cual el tamaño de cada Matriz de Incidencia. Si los bloques de datos utilizados para el cálculo propiamente dicho son de rango 1600, hay una similitud de magnitudes entre los datos a procesar y los datos necesarios para procesar en paralelo. En el ítem anterior se expuso que la sobrecarga real verificada fue pequeña, pero crece exponencialmente por el impacto de la tarea `xgemm` en el algoritmo, como se expuso en la Fig. 4.6.

- El mejor resultado se obtuvo para una matriz de rango 30000, 16 procesadores lógicos y la división de *tiles* $n = 15$, alcanzando 389 Gflops. Dado que la máquina AMD tiene un pico teórico de procesamiento de 998 Gflops[1], se logró una utilización de casi el 40% de su capacidad máxima. Si además se tiene en cuenta que fue logrado con solo 16 de los 24 unidades FMA disponibles, el porcentaje de procesamiento crece al 58%. Estos valores son muy buenos considerando la parte secuencial del algoritmo y las fallas de cache necesariamente ocasionadas por el alto volumen de tráfico de datos entre la memoria principal y la cache L3.

[1] 998 Gflops = 2.6 Ghz x 24 fma x 2 ops x 8 valores de precisión simple

- El uso de 24 procesadores lógicos no genera rendimientos superiores al uso de 16. La configuración física de la memoria en el equipo, donde ocupa un solo banco de memoria y por lo tanto utiliza un único canal, genera una saturación en el uso de dicho canal para el caso de 24 procesadores. Este resultado se concluye debido a que los tiempos unitarios de ejecución de las rutinas presentan una alta variabilidad para este caso, con casos extremos del doble de tiempo. Para 16 o menos procesadores, las diferencias entre los tiempos unitarios para una misma tarea apenas llegan al orden de las centésimas, siendo muy estables.

4.1.4 La máquina con procesadores Intel

La máquina basada en procesadores Intel tiene dos microprocesadores del tipo Xeon E5-2680, con 8 *cores* cada uno, lo que hace un total de 16 procesadores físicos. Además, cada *core* dispone de una unidad AVX (*Advanced Vector Extensions*) que utiliza registros de 256 bits que permite realizar en forma simultánea adición y multiplicación sobre dichos registros, similar a la unidad FMA de AMD. Disponiendo de 16 unidades AVX, la capacidad teórica de procesamiento es similar a la máquina AMD, alcanzando los 691.2 Gflops.

Para este equipo se utilizó el compilador Ifortran y la biblioteca de rutinas MKL, versión Intel Composer 2013 suite, a priori el más apropiado para la arquitectura en esta máquina al momento de realizar las pruebas. Al igual que para el caso de la máquina AMD, en todas las pruebas, se utilizó precisión numérica simple, para mantener consistencia.

La tabla de de la Fig. 4.15 contiene los resultados más significativos de las pruebas realizadas sobre esta máquina. De forma similar a los experimentos realizados con la máquina AMD, la división de procesadores lógicos se hizo tomando una sola unidad AVX para cada uno. En esta oportunidad, al usar toda la capacidad de procesamiento se obtiene mejores resultados que al utilizar solo una parte. En la maquina Intel la memoria está distribuida en dos canales, por lo que la saturación del *memory channel* observada anteriormente no se repite.

A diferencia de la biblioteca ACML, su par MKL está mejor preparada para la ejecución de sus *kernels* en paralelo, por lo que se presentan resultados de pruebas adicionales con esta librería con varias combinaciones del número y composición interna de los objetos procesador para los 16 *threads* (o unidades AVX) disponibles. Los resultados más significativos se presen-

procs		8x1		16x1	
rango	*tiles* n	segs	flops	segs	flops
24000	8	17.08	270	13.25	348
	12	14.35	321	9.52	484
48000	8	140.59	262	101.73	342
	12	109.24	337	70.99	519

Figure 4.15: Tiempo en segundos y flops en GFlops, con matrices de rango 24000 y 48000, 8 y 16 procesadores, división en *tiles* $n = 8, 12$, en la máquina Intel.

procs		1x16		2x8		4x4		8x2	
rango	dvs	segs	flps	segs	flps	segs	flps	segs	flps
24000	12	11.69	392	10.17	453	8.98	513	8.40	549
48000	12	69.73	529	62.72	588	59.81	616	60.33	611

Figure 4.16: Tiempo en segundos y flops en GFlops para pruebas de rango 24000 y 48000, con 16 *threads* divididos lógicamente en doble nivel (procesador x *threads* internos) en combinacion de 1x16,2x8,4x4,8x2, y división de datos en *tiles* $n = 12$, en la máquina Intel.

tan en la tabla de Fig. 4.16, los cuales fueron obtenidos tomando el mejor caso observado de la división de (16x1), el cual, según la tabla precedente, es para el caso de división de *tiles* $n = 12$, fijando este número de divisiones y realizando las pruebas para otras la combinaciones en la división lógica de procesadores, a saber 1x16, 2x8, 4x4 y 8x2.

El análisis de los resultados nos permite conluir que:

- En forma similar a los resultados obtenidos por la máquina AMD, el número de divisiones de *tiles* debe ser lo suficientemente alto para disponer de un tareas habilitadas y no trabar la ejecución paralela. En este caso ha sido el óptimo para $n = 12$.

- Aumentar el número de procesadores lógicos de 8 a 16 no escala propiamente, pero genera una mejora en el rendimiento del 50%, y no se mantiene estable como en el caso de la máquina AMD.

- El mejor resultado para los objetos *Processor* compuestos de un único *thread* se obtuvo para un rango de 48000 y división *tiles* $n = 12$ arrojando un resultado de 519 Gflop. Considerando el pico de capacidad de procesamiento de los 16 procesadores, la tasa de uso de estos alcanza cerca de un 75% de su capacidad teórica, lo que evidencia un buen

manejo de las pérdidas de cache y la administración de los procesos paralelos.

- Las pruebas para procesadores lógicos con doble nivel de división de hilos muestran el mejor resultados para cuatro objetos *Processor* con 4 hilos (AVX) cada uno, alcanzando un pico de rendimiento de 616 gflops, muy cercano al máximo teórico de 691.2 gflops.

- Estos resultados dan evidencia de dos factores: el buen manejo de la biblioteca MKL para la ejecución con múltiples *threads*, y por otro lado, que el uso de todos los recursos de procesamiento del equipo físico es mejor aprovechado por el modelo bajo prueba que por la misma biblioteca. En efecto, si consideramos la ejecución de la biblioteca con los 16 hilos en un único proceso (1x16), arroja un rendimiento de 392 y 529 gflops para el caso de matrices de rango 24000 y 48000 respectivamente. Sin embargo, si para la ejecución se utiliza el modelo PEM con una configuración de 4 procesadores con 4 hilos cada uno (4x4), o de 8x2 procesadores, los resultados son mejores que para la propia biblioteca, dando prueba de lo conveniente del modelo al poder configurar la ejecución paralela en forma ágil e independiente del modelo del algoritmo, y lograr una ejecución real con el 89% del uso máximo de la capacidad de procesamiento.

Finalmente, y a modo de ilustración, la Fig. 4.17 muestra la linea de tiempo de la ejecución de uno de las pruebas para la máquina Intel, con rango de matriz 24000, división *tiles* $n = 12$ y 16 procesadores. Las ejecuciones con rango de matriz mayor generan procesamientos más largos que son difícil de incluir en un gráfico de una página. Asímismo, los nombres de las tareas se han recortado a los fines de que quepen.

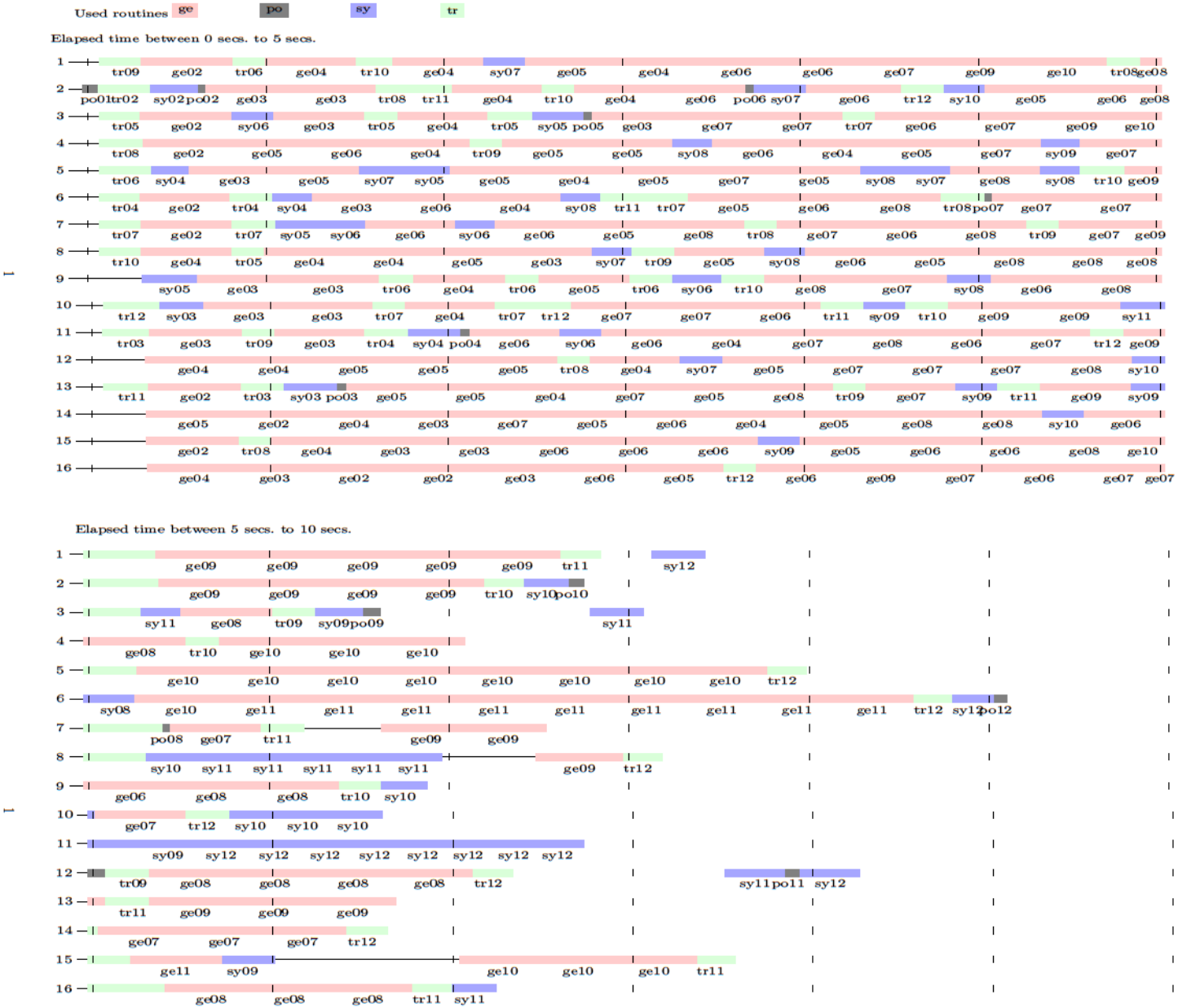


Figure 4.17: Línea de tiempo de la ejecución para la máquina Intel, rango 24000, *tiles* $n = 12$, 16 procesadores

La inactividad de los procesadores se puede observar al inicio y en las etapas finales de la ejecución del algoritmo, consistente con lo expuesto anteriormente y graficado en pag 68. Sobresale la inactividad al final del procesamiento. Tener en cuenta que luego de la tarea *potr9* solo quedan cuatro etapas del algoritmo, con 20 tareas en total por realizar, de las cuales 10 son secuenciales, por lo que la posibilidad de realizar tareas en paralelo disminuye drásticamente. En la figura puede verse que luego de dicha tarea se realizan más de 20, lo cual es por la priorización que el *scheduler* IT hace sobre las tareas que más pasos tienen que hacer para llegar al final, generando una mayor utilización de los procesadores en estas últimas etapas.

Se destaca la ausencia de tiempos de inactividad a lo largo del resto de las etapas de procesamiento. La baja sobrecarga tambien se evidencia en la ausencia de espacios inactivos entre el final de una tarea y el inicio de la siguiente.

4.2 Multiplicación de Matrices

Los buenos resultados obtenidos en los experimentos de la sección anterior, no son suficientes para satisfacer las expectativas de la bondad del modelo. Con los experimentos sobre el algoritmo de Cholesky se comprobó que la tarea de modelado del mismo bajo las premisas del modelo PEM, son relativamente simple y flexibles a la hora configurar el entorno de ejecución paralelo, por cuanto una vez modelado el algoritmo, no fue necesario modificarlo para su ejecución en diversos entornos de máquinas SMP.

Otro tipo de pruebas fueron realizadas y consistieron en utilizar el modelo para un entorno de máquinas de memoria compartida, pero con procesadores heterogéneos, donde resalte la bondad del diseño del modelo, sobre todo, en lo referente a la flexibilidad que brindan los objetos *Evaluator*, los responsables para cada objeto *Processor* en la selección de la próxima tarea a realizar. Dentro de un entorno heterogéneo, cada tipo de procesador debe tener un criterio de selección de tareas distinto, el cual permita que los procesadores más lentos ayuden en el procesamiento global con tareas que no generen demoras o cuellos de botella a la ejecución total.

El algoritmo seleccionado es el clásico de multiplicación de matrices (MM). Se consideran matrices cuadradas de números de precisión simple. Dadas tres matrices cuadradas, A, B, C, de igual rango q, se toma como producto matricial a $C = A \times B$, cuya fórmula general es:

$$C_{ij} = \sum_{k=1}^{q} A_{ik} \times B_{kj} \tag{4.4}$$

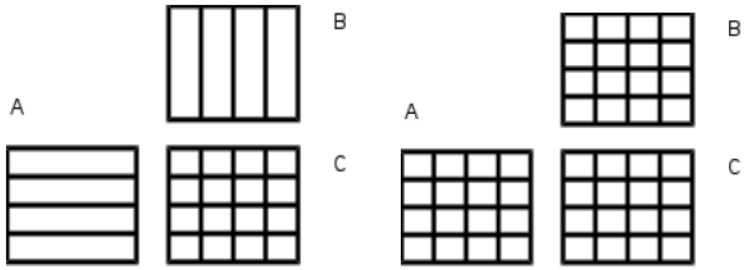
donde X_{ij} representa el valor en la matriz ubicado en la fila i, columna j, y C_{ij} se obtiene por medio del producto vectorial entre el vector fila i en la matriz A y el vector columna j de la matriz B.

Este algoritmo es fácilmente paralelizable ya que no existen dependencias de datos entre los resultados en cada posición, por lo que en una máquina paralela con memoria compartida, se puede obtener el resultado en con $q \times q$ procesos paralelos potencialmente, cada uno de los cuales realiza un producto vectorial, cuyo resultado determina cada valor en la matriz C. Sin embargo, esto último es prácticamente inviable ya que la ejecución paralela de este algoritmo tiene sentido en la medida de ser ejecutada con matrices cuyo rango excede largamente el número de procesadores disponibles en una máquina paralela.

La ejecución paralela de este algoritmo necesita entonces de la división de datos, para que cada procesador compute una parte del resultado total. Hay dos formas frecuentemente utilizadas para esta división: división por bandas y división por bloques cuadrados (*tile*) [BLKD07]. La división por bandas divide a la matriz A en n bandas horizontales, a la matriz B en n bandas verticales, por lo que la matriz resultante C presenta una división de $n \times n$ bloques cuadrados, de forma tal que $C_{i,j} = A_i \times B_j$, donde los subíndices representan los bloques de cada matriz. La división por *tile* divide todas las matrices en $n \times n$ bloques cuadrados (*tiles*), $C_{i,j} = \Sigma_k A_{i,k} \times B_{k,j}$, donde los subíndices representan los bloques cuadrados. En la Fig. 4.18 se muestran los gráficos ambas divisiones de datos.

Desde una óptica de la ejecución paralela, existen ciertos problemas con ambas formas de dividir los datos. En la división por bandas existe un problema de balance de carga en el procesamiento. Si se dividen las matrices A y B en b bandas, se genera como resultado b^2 bloques cuadrados en C, cada uno de los cuales representa una tarea indivisible. Todas las tareas tienen la misma carga de cómputo. El problema es que el número de tareas debe ser múltiplo del número de procesadores para que exista balance de carga, de lo contrario, surge un desbalance.

Para ejemplificar el problema del balance de carga en la división por bandas, si se dispone de 10 procesadores, y se divide en 4 bandas, se generan 16 tareas de igual carga, pero que al ejecutarse en paralelo, la última etapa



(a) División por bandas (b) División por *tile*

Figure 4.18: Ejemplos de división por bandas y por "*tiles*".

tendrá seis procesadores calculando y cuatro inactivos.

En el caso de la división por *tiles*, si se divide A y B en 4×4 bloques, en C se generan la misma cantidad de bloques y además el procesamiento total está compuesto por 64 tareas (4^3) para resolver los 16 bloques de C. El problema del desbalance sigue existiendo en la última etapa del procesamiento, pero atenuado, ya que la carga del procesamiento de cada tarea de *tile* es menor, lo cual genera un lapso de inactividad menor en los procesadores no utilizados al finalizar.

La división por *tiles* genera una problema de dependencia de datos, por cuanto el resultado final $C_{i,j}$ es la suma de n productos parciales, los cuales deben ser preservados para lograr el resultado final. Esta dependencia no hace referencia a respetar un orden de ejecución, pero sí en cuanto a que los resultados parciales que deben ser acarreados. Dicho de otra forma, si cada producto matricial parcial es realizado por distintos procesadores, o bien debe hacerse uno a continuación de otro para ir acarreando el resultado, o bien realizar el proceso de sumatoria de productos parciales en dos etapas, una que realice los productos matriciales en donde cada procesador es independiente y la segunda que acumule los resultados parciales de la primera. Esta última opción implica una sincronización entre los procesos al momento de hacer la acumulación (operación de tipo *reduce*).

La rutina de BLAS elegida para el cómputo de los productos es `xgemm` y dicha rutina resuelve en forma eficiente:

$$C = \beta C + \alpha A \times B \tag{4.5}$$

donde A, B y C son las matrices y α y β son escalares. Por otro lado, algunos

microprocesadores tienen instrucciones primitivas que resuelven en forma simultánea la adición y la multiplicación (*Fused Multiplication Adition*), por lo que separar ambas en dos tareas distintas, es inapropiado desde el punto de vista del rendimiento. Sin embargo, como se plateó anteriormente, realizar la multiplicación y la adición en forma simultánea tiene la desventaja de generar una secuencia de tareas que no se pueden realizar en paralelo por el acarreo del resultado.

Un fuerte impacto por utilizar rutinas que fusionen la multiplicación con la adición, es que desde el punto de vista de su ejecución paralela, no pueden realizarse en paralelo cálculos sobre un mismo bloque de la matriz C, para que las acumulaciones se realicen correctamente. Para evitar el desbalance de los últimos bloques, debe evitarse realizar todo el cálculo sobre un bloque C antes de continuar con otro bloque, por que de esta forma se está secuencializando el procesamiento. Para generar un mayor número de tareas disponibles a ejecutarse en paralelo y minimizar el desbalance final, se debe realizar un procesamiento que deje parcialmente calculados la mayor cantidad posible de bloques de C, de forma tal de que los procesadores siempre tengan bloques para trabajar, y evitar la secuencialización de los últimos bloques. Tener en cuenta este criterio puede generar una sobrecarga en la gestión de la ejecución paralela.

El equipo con procesadores heterogéneos sobre el cual se realizaron las pruebas es un equipo SMP con dos placas GPGPU. Las características generales de estos equipos es que las GPGPU tiene un rendimiento superior a los *cores* de los microprocesadores, pero que estas no actúan independientemente del equipo principal, sino que más bien son "coprocesadores" auxiliares del principal. Las placas GPGPU disponen de una memoria local donde realizan sus cálculos, por lo que los datos deben ser trasladados desde la memoria principal a la memoria de la placa GPGPU, luego calcular y devolver el resultado a la memoria principal.

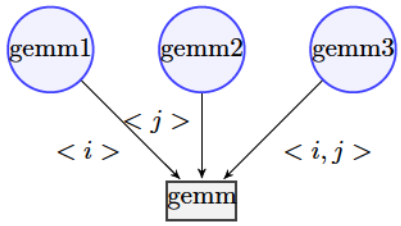
La heterogeneidad en esta clase de equipos es muy fuerte, ya que para algunas rutinas de BLAS las GPGPU alcanzan un rendimiento muy superior a las CPU (x10 o superior). La contrapartida en la GPGPU es el limitante de la memoria interna, muy inferior cantidad a la principal del equipo. El desafío para matrices de gran tamaño, cuyo requerimiento de memoria exceda al disponible en una GPGPU, es particionarlas y distribuir el procesamiento entre las GPGPUs y los *cores* del equipo, de forma tal que el rendimiento final sea obtenido por la potencia de las GPGPU combinada con la potencia de los *cores*.

La arquitectura de la máquina paralela heterogénea debe ser tenida en cuenta a los fines de desarrollar el modelo del algoritmo con PEM. En particular, la ejecución de MM se realiza sobre el mismo equipo AMD de sección 4.1.2. Se recuerda que consta de cuatro microprocesadores AMD opteron 6344 de 12 *cores* cada uno, lo que hacen un total de 48 *cores*, 48 Gigabyte RAM montados sobre un solo *memory channel*. Cada microprocesador tiene dos bloques de memoria cache L3 cada uno de 8 MBytes y asociado a seis *cores*. Para operaciones de punto flotante, se dispone de una unidad *Fused Multiplication Addition* (FMA) compartida cada dos *cores*. Cada FMA puede realizar concurrentemente una adición y una multiplicación sobre 256 bits, los cuales pueden dividirse en registros de 32 o 64 bits. Al igual que para el caso de la factorización de Cholesky, serán utilizadas estas unidades para el cómputo sobre CPU, lo cual limita a 24 el número de procesadores simultáneos.

Para el procesamiento con GPGPU se montaron sobre el canal PCI de dicho equipo, dos placas Nvidia GTX 680, cada una de las cuales consta de 2 GBytes de memoria y 1536 hilos de ejecución paralela a una velocidad de procesamiento de 1006 MHz. El fabricante de las placas provee para uso libre, una implementación de BLAS para ejecutarse sobre las mismas, cuBLAS [Corb], por lo que el algoritmo paralelo se unifica en cuanto al uso de la rutina `xgemm`, con implementaciones acorde a cada tipo de procesador utilizado.

Para lograr el mejor rendimiento conjunto, cada tipo de procesador debe trabajar sobre el tamaño de bloque que mejor rendimiento provea. En un trabajo previo [SOW13] se concluyó que el tamaño óptimo para la rutina `xgemm` utilizando ACML sobre CPU es el mayor posible que quepe en la memoria cache L3. Por otro lado, para el caso de las GPGPUs, el mejor rendimiento para dicha rutina se logra con el mismo criterio que para la CPU, pero sobre el límite de la memoria en la placa [WDG14a]. Por lo que el diseño del algoritmo paralelo debe tener en cuenta una doble división de datos, una de grano grueso para las placas GPGPUs, y otra de grano fino para las CPU.

Una decisión de diseño que fue tomada al desarrollar las pruebas es que para facilitar la administración de los bloques de datos de dos tamaños, la división de grano fino sea una fracción de la de grano grueso, de forma tal que una tarea definida para grano grueso pueda ser realizada equivalentemente por un conjunto de tareas de grano fino. En nuestro caso, como los bloques menores se asignan a las CPU's, cada CPU's realiza una fracción de la tarea que realiza una GPGPU, y pueda ser completada en forma equivalente, de forma tal que sea indistinto lanzar una tarea sobre un procesador GPGPU o en un conjunto de procesadores CPU. Esta decisión de división de datos

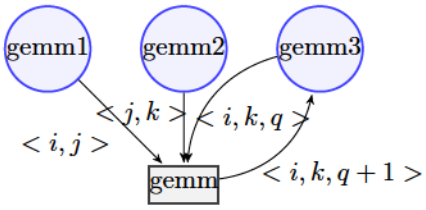


(a) Red de Petri Coloreada (CPN)

Plaza	Dominio
gemm1	$< i >, i = 1 \dots n\{n\}$
gemm2	$< j >, j = 1 \dots n\{n\}$
gemm3	$< i, j >, i, j = 1 \dots n$

(b) Dominios de la CPN

Figure 4.19: División por bandas para la multiplicación de matrices.



(a) Red de Petri Coloreada (CPN)

Plaza	Dominio
gemm1	$< i, j >, i, j = 1 \dots n\{n\}$
gemm2	$< j, k >, j, k = 1 \dots n\{n\}$
gemm3	$< i, k, q >, i, j, q = 1 \dots n$

(b) Dominios de la CPN

Figure 4.20: División por *tile* para la multiplicación de matrices.

facilita el modelado del algoritmo, como se verá a continuación.

A los fines de de determinar el formato más conveniente para ambos tipos de divisiones, los mismos se analizan a continuación. En la Fig. 4.19 se presenta el gráfico de la CPN que define el algoritmo para la división por bandas y los respectivos dominios. Puede verse que la CPN es muy simple, ya que tan solo existe como tarea `xgemm` y sus tres Plazas, que representan los bloques de las matrices A, B y C.

Los dominios de cada Plaza ayudan a comprender mejor la CPN. Las dos primeras Plazas representan las bandas bloque de A y B respectivamente, para las cuales necesita una sola dimensión en los subíndices, mientras que la tercer Plaza representa la matriz resultante C, donde es necesario disponer de dos dimensiones para los subíndices, según lo expuesto anteriormente. Puede verse en la tabla de dominios que las dos primeras Plazas tienen sus *tokens* repetidos n veces, siendo n el número de bloques en la división, lo cual es necesario ya que cada banda interviene n veces en los cálculos.

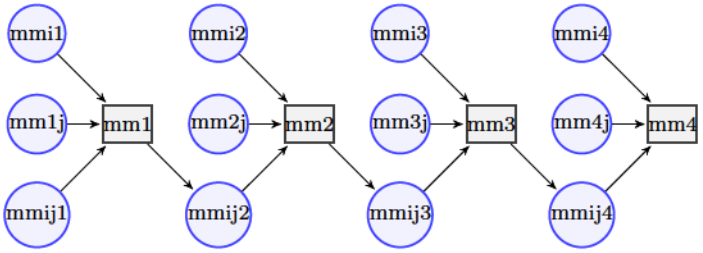


Figure 4.21: Despliegue de la CPN de para la división por *tile*, para un solo bloque genérico $C_{i,j}$, y bajo el supuesto de que la cantidad de divisiones $n = 4$

En la Fig. 4.20 se presenta el gráfico de la CPN que define el algoritmo para la división por *tile* y sus dominios. Aquí la CPN también es sencilla, solo que cambian los dominios de las dos primeras Plazas, ya que en este caso es necesario contar con dos dimensiones para los subíndices. La tercera Plaza necesita de tres dimensiones, ya que ahora es necesario agregar una dimensión que represente la secuencia de la ejecución para permitir la acumulación de los resultados. Al igual que en la división por bandas, los dos primeros dominios tienen sus *tokens* repetidos n veces, por el mismo motivo del caso anterior.

En el caso de la división por *tile* resalta la conexión desde la Transición a la tercera Plaza, actúando esta entonces, tanto como Plaza de entrada como de salida. Cuando el *token* regresa, lo hace con la tercera dimensión incrementada, lo que permite avanzar a la próxima etapa, repitiéndose hasta que se alcanza la última, cuando vale n, en cuyo caso concluye el cálculo para el bloque en cuestión.

En la Fig. 4.21 se presenta con fines ilustrativos el gráfico de la red desplegada, solo la parte correspondiente a un bloque genérico $C_{i,j}$, bajo el supuesto de que en la división por *tile* hay cuatro divisiones, $n = 4$.

Con la finalidad de definir el tipo división que tendrá cada nivel de granularidad, es necesario realizar algunas cuentas para determinar, *a priori*, cual es la más conveniente para cada una. En primer lugar, el formato para la división de grano grueso, que corresponde a las GPGPU's, que a su vez tienen un límite de memoria de 2 GBytes. Un problema de la división por bandas respecto del uso de memoria, es que al aumentar el número de divisiones, no disminuye proporcionalmente la cantidad de memoria necesaria para cada operación de bandas, ya que la división es en una sola dimensión, y no en las dos como en el caso de la división por *tile*.

Por otro lado, el tamaño de las matrices para las cuales se hace necesario

una división en bloques para su procesamiento es tal que debe superar al menos un orden de magnitud la memoria de la GPGPU. Suponiendo el caso de una matriz de rango de 32.000, esta ocupa 4GBytes de RAM para precisión simple. Es decir, la operación requiere de 12 GBytes. El menor número de bandas cuya ocupación total de memoria quepe en los 2 GBytes de RAM de la GPGPU, es de 8. En este caso, la memoria requerida es de algo más de 1GByte. Recordar que 6 y 7 no son divisores de 32000. Cada bloque banda es de dimensión 4000 x 32000. El total de bloques cuadrados en la matriz C es de $8^2 = 64$, cada uno de los cuales es de rango 4000.

El problema a resolver es la división de grano fino. Parece natural dividir cada banda en bloques cuadrados de rango 4000, por lo que para el grano fino se aplicaría un división por *tile*. Debido a que, como se expuso anteriormente, cada cómputo de la división por *tile* debe hacerse secuencialmente para un bloque dado de la matriz resultante, el mínimo tiempo para que las CPU's terminen con dicho cálculo es ocho veces el tiempo de cálculo de un bloque de grano fino, lo cual, a priori, puede ser una causa de desbalance en el algoritmo dada la diferencia de rendimiento entre ambos tipos de procesadores. Para rangos de matrices superior a los 32000, el problema se agrava.

Estos últimos cálculos justifican buscar otra combinación de divisiones. Para el grano grueso, si aplicamos *tile* con $n = 4$, manteniendo el mismo rango de matrices a multiplicar, estas se dividen en $4 \times 4 = 16$ bloques cuadrados, generando el mismo número de operaciones que en el caso de bandas, 64 operaciones, y cada bloque queda de rango 8000, por lo que una operación de multiplicación de estos bloques requiere de 768MBytes de memoria, lo cual es holgado para cada GPGPU. Por cierto que los bloques cuadrados son utilizados por las implementaciones de BLAS para esta rutina, ya que tienen una mejor localidad espacial y generan menor número de fallas de *cache*.

Parece razonable que la división de grano fino, siguiendo con el ejemplo citado, siga la misma idea, una división por *tile* con $n = 4$. Como cada bloque de grano grueso es de rango 8000, una subdivisión por 4 en cada dimensión, genera 16 bloques de rango 2000. Dado que estos bloques que serán utilizados por las rutinas en CPU, las 16 FMA's pueden realizar cada una de ellas, una operación a nivel de grano fino, con un requerimiento de memoria de 48 Mbytes. Este volumen de memoria es un poco superior al que disponen las memorias físicas *cache* L3, pero es un tamaño de bloque, que en general, las bibliotecas BLAS manejan con eficiencia. Parecería conveniente hacer una división mayor que de 4×4 para el grano fino, de forma tal que la cantidad de memoria requerida sea menor, atenuando el problema de las fallas de *cache*.

Sin embargo, la doble división por *tile*, tiene un inconveniente serio, y es el número de tareas generado al realizar el despliegue del modelo CPN. En efecto, para el caso de división con $n = 4$, el dominio de cada Plaza de la operación `xgemm` en la CPN tiene 64 *tokens*, que provienen de 16 *tokens* diferentes, repetidos 4 veces cada uno. Pero si a nivel de grano fino, cada uno de estas 64 operaciones va a ser resuelta a suz por 64 suboperaciones, la red desplegada tiene $4^3 \times 4^3 = 4096$ operaciones con 12.288 Plazas, a razón de tres Plazas por operación. Es decir, se necesitaran dos Matrices de Incidencia de tamaño 4096×12288, lo cual es absurdo para realizar operaciones con matrices de tamaño 2000.

Una doble división por *tile* tiene el problema de que el número de operaciones generadas es exponencial a la sexta con el número de bloques n en que se particione la matriz, lo cual, además del problema de la memoria requerida por las Matrices de Incidencia, presenta la complejidad en la gestión de miles de operaciones en paralelo lo que provoca una sobrecarga de procesamiento que debería ser eludida.

Por los motivos del párrafo anterior, se descarta la división de grano fino por *tile* y se considera por bandas. En este caso, si bien la utilización de memoria es mayor, el número de operaciones generado es de orden cuadrático con el número de bandas, es decir, se disminuye un orden el número de operaciones de grano fino. Siguiendo con el ejemplo anterior, una combinación de $n = 4$ para *tile* y de 5 bandas para el grano fino, $r = 5$, define $4^3 \times 5^2 = 1600$ operaciones y 4800 Plazas, lo cual sigue siendo voluminoso, pero cerca de un tercio del caso anterior.

Dado que los números de operaciones para la división de grano grueso en forma de *tile* y la de grano fino en forma de bandas, son más razonables, y que la memoria necesaria para esta combinación de operaciones es próxima al óptimo, se decidió realizar pruebas para este formato de división. En el gráfico de la Fig. 4.22 se presenta la CPN con este formato y en la tabla de Fig 4.23 se exponen los dominios de sus Plazas.

La CPN de doble granularidad para la multiplicación de matrices presenta tres tareas adicionales que no están descriptas en el algoritmo, pero que son necesarias a los fines de la ejecución paralela. Ellas son la partición, la multiplicación en grano fino y la unificación de bloques. La primera es la encargada de particionar en forma lógica el bloque de datos de la división de grano grueso en los sub-bloques banda de grano fino. Si bien no es una tarea de cómputo propiamentedicha, es una tarea necesaria para definir el tipo de procesador que va a realizar el cálculo sobre ese bloque en particular, en este

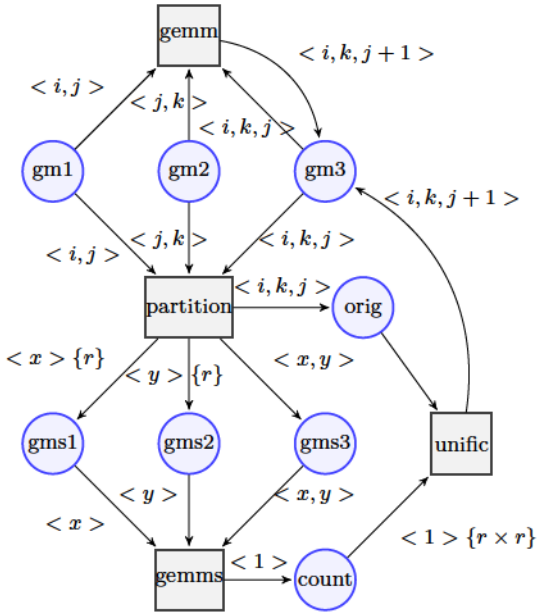


Figure 4.22: CPN de para la división por *tile* en grano grueso y por bandas en grano fino

Plaza	Dominio
gm1	$< i, j >, i, j = 1 \ldots n\{n\}$
gm2	$< j, k > j, k = 1 \ldots n\{n\}$
gm3	$< i, k, j >, i, k, j = 1 \ldots n$
gms1	$< x >, x = 1 \ldots r\{r\}$
gms2	$< y > y = 1 \ldots r\{r\}$
gms3	$< x, y >, x, y = 1 \ldots r$
orig	$< i, k, j >, i, k, j = 1 \ldots n$
count	$< 1 >$

Figure 4.23: Dominios de las Plazas de la CPN de Fig 4.22

caso, los CPU's. Por ello, se definen tres Plazas de salida de esta Transición, cada una de las cuales contiene los tres bloques banda de la partición de grano fino, con un número r de repeticiones para cada una de ellas.

La segunda tarea agregada es la encargada de realizar la multiplicación para los bloques de la división fina, que se denominó como `gemms`, la última 's' por subdivisión. Es claro que no es la misma que para los bloques de grano grueso, ya que no solo es ejecutada por otro tipo de procesador, sino que el

número de *tokens* que componen sus Plazas de entrada es paramétricamente diferente al caso de grano grueso. Además esta tarea tiene como salida a la Plaza *count* con un dominio cuyos *tokens* son una constante. El objetivo de esta es que sirva de entrada a la tarea `unific` como conteo de la cantidad de bandas calculadas.

La tercera tarea referida anteriormente es la que realiza la unificación de las partes de grano fino para determinar que su equivalente en grano grueso ha sido completado, y es la denominada `unific`. Al igual que `partition`, no realiza ningún cómputo, pero se encarga de notificar que el bloque $< i, k, j >$ ha sido calculado y se puede continuar con la ejecución de $< i, k, j+1 >$. Esta tarea tiene como Plazas de entrada al identificador del bloque, es decir el trio $< i, k, j >$. Este dato es necesario, ya que al realizar la partición, se pierde la información del grano grueso de origen. Como segunda Plaza de entrada está la Plaza *count*, que tiene la finalidad de contar la cantidad productos de bloques bandas realizados. Al realizar la partición del bloque y determinar que su cálculo será realizado por $r \times r$ subtareas, es necesario determinar cuando se hicieron todas estas, y poder continuar con el siguiente bloque de grano grueso.

En resumen, el algoritmo, a nivel de grano grueso, es igual al diagramado en la Fig. 4.20 y a nivel de grano fino es como el diagramado en 4.19. Como la definición de cuales son los bloques que se computarán con grano fino no es predeterminada, y es hecha dinámicamente, es necesario introducir las tareas de partición de bloque grueso en bloque fino y su posterior unificación en bloque grueso nuevamente, una vez concluidas todas las subtareas.

Como puede deducirse del modelo en CPN, el control de la ejecución del algoritmo no esta incluido en el modelo. Como se explicó anteriormente al definir las políticas del modelado, el control corresponde a la etapa de ejecución y no de modelado. Un ejemplo permite aclarar el concepto. En el modelo CPN del algoritmo como ha sido definido, presenta una inconsistencia. Puede suceder que dos o más bloques de grano grueso entren en la tarea de partición y generen los respectivos bloques bandas, con lo cual, al momento de realizar el producto de las bandas, este podría ser hecho con bandas que corresponden a dos bloques de grano grueso diferentes. Para evitar este problema se puede encontrar al menos tres soluciones:

1. Agregar en la identificación de los bloques de grano fino tres dimensiones adicionales, que permitan definir el origen del bloque banda. Es decir, identificar por medio de una tupla del tipo $< i, k, j, x, z, y >$, donde i, k, j son los identificadores del bloque de grano grueso, y x, y, z

son los identificadores de grano fino.

2. Agregar una Plaza que actúe de elemento de control dentro del modelo CPN, deshabilitando la entrada a `partition` cuando haya cálculos aún pendientes de realizar sobre el producto de grano fino (`gemms`), evitando de esta forma la existencia de bandas originarias de bloques gruesos diferentes.

3. Derivar al objeto *Evaluator* del modelo PEM la selección del bloque grueso que entra en la partición y que este se bloquee asimismo para particionar un nuevo bloque en la medida que queden pendientes tareas de `gemms` por realizar. Esta opción es factible ya que, cuando dicho objeto realiza el proceso de selección de tareas a ejecutar, debe determinar todas las tareas habilitadas y elegir entre ellas la más conveniente. Como tiene que elegir entre todas las habilitadas, si entre ellas hay al menos una tarea `gemms` habilitada, esta toma prioridad sobre una tarea `partition` y se evita la inconsistencia.

A los fines de esta tesis, se elige la opción de derivar el control sobre el objeto *Evaluator*, que es su razón de ser. En el ejemplo se puede ver que es la opción más genérica y que da más flexibilidad a la ejecución del modelo. La opción de extender el identificador es viable, pero es poco flexible, ya que se desconocen todos los posibles casos de identificación. La opción de incluir el control en el modelo CPN es rígida, ya que no solo has más complejo al modelo, sino que cuando sea necesario agregar nuevos elementos de control, el modelo debe ser rearmado, incluyendo nuevas Plazas como mínimo, mezclando el algoritmo con el control del mismo.

Derivar el control sobre el objeto *Evaluator* permite, por un lado, definir diferentes criterios de control según el procesador que lo utilice. En el presente caso, la instancia de *Evaluator* asociada al procesador de tipo GPGPU, literalmente ignora las tareas `partition` habilitadas, y viceversa, el *Evaluator* asociado a los procesadores de tipo CPU ignora las tareas `gemm`. Por otro lado, ante un nuevo elemento o criterio de control, se modifica en este objeto el algoritmo de priorización de tareas y se continúa con la ejecución paralela. En términos de Ingeniería de Software, es un ejemplo del concepto de *"separation of concerns"* [Dij82].

Si bien existe bibliografía donde el modelo de la ejecución y el control están reflejados en Redes de Petri [IA06, SB09], estos casos están orientados a los circuitos electrónicos o computadoras embebidas, donde la ejecución del programa es hecha por un *hardware* específico, y no por software. Se

tile n		$n=2$		$n=3$		$n=4$		$n=5$	
rango	gpgpu's	segs	gflps	segs	gflps	segs	gflps	segs	gflps
24000	1	23.0	1202	26.7	1035				
24000	2	13.7	2018	15.5	1784	18.1	1527		
36000	1			76.3	1223				
36000	2			41.9	2227	44.8	2083	52.8	1767
48000	2					96.2	2299	114.8	1927

Figure 4.24: Tiempo en segundos y rendimiento en gflops para pruebas de MM de varios rangos y número de divisiones, ejecutados sobre GPGPU exclusivamente.

bandas r	$r=1$		$r=2$		$r=3$	
rango	segs	gflops	segs	gflops	segs	gflops
6000	1.51	286.1	2.20	196.3	2.29	188.6
12000	8.6	401.8	9.9	349.1	11.9	290.4
24000	77.7	355.6	64.8	426.7	88.0	312.2
36000			430.9	216.5	515.9	180.9

Figure 4.25: Tiempo en segundos y rendimiento en gflops para pruebas de MM de varios rangos y número de divisiones, ejecutados sobre 16 unidades FMA de CPU exclusivamente.

considera que, como la presente tesis es de software, es más apropiado que el modelo del algoritmo paralelo sea hecho con Redes de Petri, y que una pieza de software independiente lo controle.

Definidos los parámetros del modelo de ejecución para la arquitectura heterogénea, con una doble granularidad, de grano grueso en forma de *tile* para las GPGPU's y de grano fino en forma de bandas para las CPU, se realizaron las pruebas de ejecución del algoritmo. Se utilizó el compilador gcc 4.7.2 y como implementaciones de BLAS a las bibliotecas ACML 5.0 para CPU [Cora] y cuBLAS 6.0 para las GPGPU [Corb]. La versión de ACML utilizada es la optimizada para el uso de las unidades FMA y con ejecución secuencial, por los motivos anteriormente expuestos. Las pruebas se realizaron con precisión numérica simple y con rangos de matrices de 24000, 36000 and 48000, ya que se pueden particionar sin fracciones por 2, 3, 4, 5, 6, etc. Se descartó el uso del rango 32000 por no ser múltiplo de 3 y no poder hacer pruebas con ese número de divisiones.

La tabla de Fig.4.24 muestra el resultado de las pruebas más significativas usando exclusivamente una o dos GPGPU's, sin intervención de los *cores*

de CPU para el cálculo. Se alcanza un rendimiento superior cercano a los 2.3 Tflops con ambas placas, cuando el número de divisiones en *tile* es el menor posible que permite almacenar todos los datos de las matrices en la placa GPGPU. Los tiempos en estas pruebas incluyen las tareas de traslado de datos de memoria principal a la placa procesadora y de devolución del resultado a la memoria principal, sin solapamiento de comunicaciones con cómputo. Esto motiva que el rendimiento observado diste del máximo teórico para estas placas, el cual es de 3.09 Tflops [gtx12] [2] para cada una de ellas.

De igual forma, la tabla en Fig. 4.25 muestra los resultados más significativos de pruebas para ejecuciones sobre las CPU exclusivamente, 16 procesadores lógicos usando las unidades FMA, con rendimientos superiores a los 420 Gflops. Si se tiene en cuenta que el rendimiento teórico máximo para las 16 unidades FMA es de 665.6 Gflops, se alcanzó un rendimiento observado cercano a los dos tercios del teórico, ejecutando la versión secuencial de las rutinas y gestionando el paralelismo de 16 de estos procesos con el modelo PEM.

En la tabla de la Fig. 4.26 se exponen los resultados más significativos de las pruebas del uso conjunto de los 16 procesadores lógicos en CPU con las dos placas GPGPU. Los resultados no presentan una mejora en el uso combinado de ambos tipos de procesadores respecto del uso de las dos placas GPGPU exclusivamente. El mejor resultado combinado logra un rendimiento de 2.0 TGflops.

Cabe destacar que en la tabla antes referida se presentan resultados con rendimientos superiores a este, pero no son ejecuciones de uso conjunto de ambos tipos de procesadores, y se explica sus causas. En el caso de división por *tile* con $n = 2$, el número total de tareas es de 8. Completar cada tarea para una GPU toma menos de la octava parte que para un conjunto de CPU's, por lo que dos placas GPGPU hacen las 8 tareas en menos tiempo que le toma a las 16 FMA de CPU en completar una sola. Es por ello que el objeto *Evaluator* de las CPU's tienen una condición para seleccionar una tarea, y es que la lista de habilitadas sea mayor a 8, para que no existan esperas al procesador más lento. De esta forma, cuando $n = 2$, las CPU no computan nada.

Se atribuye la caída en el rendimiento conjunto a la saturación del uso del canal de memoria. Al analizar los tiempos de ejecución de cada tarea, se puede comprobar que a igual tarea en igual procesador, hay diferencias significativas de tiempo en su ejecución. Por ejemplo, para el caso de un

[2] 1536 hilos $\times$ 1006 Ghz $\times$ 2 ops FMA = 3090.4 Gflops

		tiles/bnds		$r=3$		$r=4$		$r=5$		$r=6$	
range	gpus	cpus	n	secs	gflops	secs	gflops	secs	gflops	secs	gflops
24000	2	16	2*	13.8	2003.4	13.6	2032.9	13.2	2094.5	13.9	1989.0
24000	2	16	3			18.6	1486.4	18.3	1510.8		
24000	2	16	4	18.9	1462.8	21.3	1298.0	20.6	1342.1		
36000	2	16	2*	38.8	2404.9	38.7	2411.1	38.1	2449.1	38.7	2411.1
36000	2	16	3	46.5	2006.6	48.2	1935.9	50.8	1836.8	48.8	1912.1
36000	2	16	4	53.6	1740.8	48.9	1908.2	50.7	1837.1	54.2	1721.6
48000	2	16	3			139.1	1590.1	188.6	1170.9		
48000	2	16	4	139.7	1583.3	195.1	1133.7	165.1	1339.7	306.3	721.4
48000	2	16	5			213.6	1035.5	269.6	820.4		

Figure 4.26: Tiempo en segundos y rendimiento en gflops para pruebas de rango 24000, 36000 y 48000; combiando divisiones de *tile* y bandas distintas, con dos GPGPU's NVIDIA GTX 680 GPU's y 16 procesadores logicos de unidad FMA4. (*) Estos casos utilizan solamente las GPGPU's.

rango de 24000, $n = 3$ y $r = 5$, es decir para bloques cuadrados de tamaño 8000 y bandas de 2000 x 8000, la ejecución en la placa GPGPU número 1 de la rutina `xgemm`, incluyendo la copia de datos desde y hacia la memoria principal, toma entre 1.15 a 1.63 segundos, y para cada banda, entre 1.46 y 3.25 segundos. En este caso, el tiempo de la ejecución total del algoritmo es de 18.3 segundos, mientras que ejecutado solamente con las dos placas GPGPU es de 13.2 segundos. Una amplitud tan grande solamente puede justificarse por demoras en la lectura o escritura de los datos, los cuales, como son procesos internos de las rutinas `xgemm` utilizadas, no han sido posible de cuantificar.

Para comprobar dicho efecto, se realizaron dos experimentos adicionales, fijando el rango de las matrices en 24000 y 36000, fijando la división en $n = 4$ y $r = 4$, y variando el número de procesadores CPU utilizados, desde cero, hasta 16, es decir, solamente se usa GPGPU en un extremo y combinado con hasta 16 unidades FMA de CPU en el otro. Los resultados se presentan en la tabla de la Fig. de 4.27.

Para el caso de utilizar ambas placas GPGPU, el mejor resultado se obtiene sin procesadores en CPU adicionales. En efecto con solo agregar 2 unidades FMA, los tiempos de ejecución suben. Dicho de otra forma, compartir el canal de acceso a memoria entre dos placas GPGPU y 2 o más procesadores genera una saturación en el uso del canal tal, que el tiempo individual de ejecución de las placas crece, lo que genera una pérdida general de rendimiento. En ningún caso, la ganancia potencial de dividir el procesamiento entre CPU y GPGPU es tal que compensa la caída del rendimiento provocada por la saturación del canal.

Es más evidente el efecto de saturación al realizar el mismo experimento,

rango		24000		36000	
gpgpu's	cpu's	segs	gflps	segs	gflps
2	0	17.0	1626	44.7	2087
2	2	20.3	1362	46.3	2015
2	4	18.4	1478	45.0	2074
2	8	17.4	1589	49.5	1885
2	16	21.3	1298	48.9	1908

Figure 4.27: Tiempo en segundos y rendimiento en gflops para pruebas de MM para rangos de 24000 y 36000, doble granularidad $n = 4$ y $r = 4$, con distinto número de procesadores lógicos en CPU y dos GPGPU's.

rango		24000		36000	
gpgpu's	cpu's	segs	gflps	segs	gflps
1	0	30.5	906	86.4	1080
1	2	28.4	974	83.8	1114
1	4	27.9	991	82.9	1126
1	8	29.4	940	94.0	993
1	16	36.9	749	123.1	758

Figure 4.28: Tiempo en segundos y rendimiento en gflops para pruebas de MM para rangos de 24000 y 36000, doble granularidad $n = 4$ y $r = 4$, con distinto número de procesadores lógicos en CPU y un solo GPGPU.

pero con una sola placa GPGPU, cuyos datos se presentan en la tabla de la Fig. 4.28. Para el caso de rango 24000, como la matriz es más chica, y con una sola placa GPGPU, agregar procesadores CPU mejora el rendimiento hasta el caso de 8 procesadores, ya que para 16, el rendimiento decae. Sin embargo, para matrices de rango 36000, al agregar 2 o 4 procesadores, el rendimiento mejora marginalmente, y para más procesadores, declina, lo que representa una evidente saturación en el uso del canal.

Adicionalmente en la Fig. 4.29 se presenta un gráfico de línea de tiempo de la ejecución para el caso de rango 24000, $n = 3$ y $r = 4$. Los procesadores 1 y 2 corresponden a las GPGPU's y el resto, a los procesadores en CPU. Puede verse que las GPGPU's no tienen tiempos inactivos, mientras que las CPU si los presentan, aunque poco siginificativos. Este hecho se debe a que las CPU's no comienzan con una nueva división en bloques hasta que no es completado el cómputo de todo el bloque de grano grueso sobre el cual trabajan. El efecto negativo de esta sincronización no es tan grave como el de la saturación del canal de memoria para el acceso a datos de la GPGPU, el cual es el factor que más condiciona la ejecución.

Figure 4.29: Línea de tiempo de la ejecución para el caso de rango 24000, 16 procesadores,$n = 3$ y $r = 4$

Como conclusiones del caso de prueba planteado en esta sección, se puede destacar que modelar la MM con CPN para un equipo heterogéneo como el utilizado y su ejecución con el *framework* desarrollado, permitió:

- Realizar un análisis del algoritmo paralelo para poder determinar la mejor forma de realizar la doble granularidad que el caso impone. El número de tareas a realizar es un determinante para realizar una división de datos adecuada.

- Plantear una ejecución con doble granularidad y *scheduler* dinámico de tareas. La doble granularidad es compleja de realizar, y es usual definirla previamente a la ejecución, es decir asignar estáticamente cuales datos serán computados por un tipo de procesador y cuales parte por otro. Realizar la asignación en forma dinámica permite optimizar los resultados, ya que no se está sujeto a valores estáticos de parámetros que generen resultados inadecuados. Un *scheduler* dinámico solo puede

ser planteado a partir de un modelo como el introducido en esta tesis.

- Alcanzar el límite de rendimiento que la máquina tiene, con sus capacidades y sus limitaciones. Si bien lo esperado de este caso era sumar rendimientos de ambos tipos de procesadores, ello no fue posible por limitaciones del *hardware*, pero los experimentos que permiten concluir que el límite fue alcanzado, fueron fácilmente realizados gracias a la parametrización flexible del *framework*.

Capítulo 5

Conclusiones, impacto esperado y líneas abiertas de investigación

A lo largo de este documento se han presentado los antecedentes del trabajo de investigación, el marco teórico de las Redes de Petri, el modelo de ejecución con su *framework* y los experimentos realizados siguiendo la técnica de modelado planteada y utilizando las herramientas desarrolladas.

El presente capítulo expone un resumen de lo realizado, las metas alcanzadas y el futuro esperable del trabajo.

5.1 Conclusiones e impacto esperado

Se afirma que los objetivos planteados en el capítulo inicial fueron alcanzados, ya que se desarrolló un modelo de ejecución paralela asíncrono, que permite pasar de un modelo del algoritmo con alto nivel de abstracción a un modelo de ejecución cubriendo un *gap* tradicionalmente existente en el dominio de las Redes de Petri respecto a la ejecución del modelo teórico. Adicionalmente, se posibilita realizar ajustes en el modelo de ejecución para lograr alto rendimiento de forma simple que permite realizar múltiples pruebas rápidamente.

Se debe tener presente que esta tesis ha tenido un énfasis fundamentalmente práctico. El objetivo principal, siempre fue ejecutar en paralelo con alto rendimiento, y con flexibilidad en cuanto al código y al *hardware* utilizado. Numerosas desarrollos han sido propuestos por investigadores para la

optimización de un algoritmo dado en un *hardware* determinado. El presente trabajo no se presume de lograr un óptimo para casos particulares, pero si de alcanzar rendimientos altos para una amplia gama de algoritmos y de máquinas paralelas.

Haciendo un repaso del trabajo realizado, a lo largo de este se desarrolló un modelo de ejecución que permite:

- Modelar un algoritmo paralelo con alto nivel de abstracción y generalidad, y que puede ser verificado, gracias a las propiedades analíticas de las Redes de Petri.

- Analizar el algoritmo, sus dependencias y opciones para la división de datos y tareas en vista a su ejecución paralela. El análisis de las dependencias, el conteo de tareas y las limitaciones en el paralelismo que puede alcanzar el algoritmo, son frutos secundarios, pero de no menor importancia, del modelo desarrollado.

- Se estipuló una estrategia de modelado que permite su conversión en una Red de Petri detallada, la cual es tomada por el *framework* de ejecución que finalmente corre en paralelo el programa modelado en alto nivel. Dicha conversión puede ser hecha en forma automática basada en los dominios que cada Plaza tenga asignada.

- El modelado en alto nivel que cumpla con los requerimientos del programa, al ser convertido al modelo desplegado, garantiza la correctitud en la ejecución, debido a la generación automática del segundo modelo a partir del primero.

- Permite una configuración adaptable a la máquina de ejecución. Cambiar de programa es cambiar de Matrices de Incidencia, y asignar las rutinas que corresponden a cada Transición.

- Se obtuvieron resultados de rendimiento cercanos al óptimo teórico. Para la multiplicación de matrices se logro 2.4 Tflops con dos placas GPGPU's, mientras que para Cholesky, más de 600 Gflops en una máquina con un máximo teórico de de 691.2 Gflops, siempre en precisión simple.

- Se hicieron pruebas sobre máquinas heterogéneas en las cuales se ejecutó en cada tipo de procesador, las rutinas que mejor rendimiento presentan en el procesador respectivo.

- Se modeló la ejecución paralela con una doble granularidad que permite a cada tipo de procesador utilizar el tamaño de bloque más conveniente para su mejor rendimiento.

Se desarrolló un *framework* que es una máquina de ejecución paralela cuyo "programa" está compuesto solamente por las Matrices de Incidencia, los Vectores de Marcado y el mapeo entre Plazas y Transiciones a bloques de datos y tareas. Esto posibilita un ágil desarrollo de nuevos programas en entornos reales, con facilidades para realizar ajustes de optimización y por sobre todo, garantizando la correctitud del programa.

Se alcanzó el objetivo inicialmente planteado, en un alto porcentaje: se creo una metodología de desarrollo que automatiza en gran medida el proceso, ya que se manejan tareas y no mecanismos básicos de programación paralela.

Dos conceptos importantes surgieron de las investigaciones, que no habían sido planteados como objetivos, pero que son de gran importancia:

- El análisis de los dominios de datos en alto nivel. Al realizar el análisis y la definición de los dominios de cada Plaza en oportunidad de la construcción de la CPN, y definir las restricciones propias de cada Transición en los dominios de sus Plazas de entrada, se minimiza la necesidad de imponer guardas a la tarea. Dicho de otra forma, definir subdominios particulares a cada tarea elimina la necesidad de las tareas con guardas. Volcado el concepto a la programación orientada a objetos, se simplifica la codificación al eliminar guardas necesarias si los objetos son de carácter general.

  El ejemplo de las Plazas en el modelo CPN para Cholesky permite aclarar el concepto. En este caso hay varios dominios similares con solo dos dimensiones pero diferentes entre sí, por las condiciones que los índices de los bloques de datos deben cumplir en cada dominio. Si en lugar de estas particularizaciones de los dominios, se hubiera optado para todas las Plazas por un dominio genérico del tipo $< x, y >$, sin ninguna restricción adicional, las condiciones deberían agregarse como guardas en cada Transición, lo cual genera una red compleja y por lo tanto, un programa "sucio" de difícil interpretación.

- Máquina de ejecución paralela. Quizás este sea el aporte más significativo de la tesis. La máquina de Turing se plantea como una máquina que ejecuta una secuencia de instrucciones con un área de memoria para datos y otra para instrucciones. El modelo PRAM es una extensión

de la misma para ejecución del modelo SIMD, donde la instrucción se comparte y ejecuta por varios "procesadores" que trabajan sobre posibles diferentes direcciones de memoria [CLR09]. En ambos casos, el programa es una secuencia de instrucciones que la máquina sabe como ejecutar.

El modelo PEM aquí planteado puede verse como una máquina que ejecuta instrucciones (las tareas asignadas a cada Transición) y el programa es definido por las Matrices de Incidencia que determinan junto al Vector de Marcado, las tareas habilitadas para poder ser realizadas. El modelo teórico puede tener un ilimitado número de procesadores que ejecuten siempre todas las tareas habilitadas, lo que es propio del modelo de Redes de Petri. La memoria es alcanzable por todos los procesadores.

Lo destacable del modelo PEM, es que ejecutar otro programa es solo cambiar las Matrices de Incidencia y asignar la tarea pertinente a cada Transición. No es necesario programar más que los *kernels* de cada Transición, ni agregar sincronización de tareas: las Matrices de Incidencia definen el programa.

El experto notará la ausencia a lo largo de todo el documento de un concepto habitual en programación paralela: la escalabilidad. En ningún momento se la utilizó como métrica de bondad del paralelismo, y esto fue realizado adrede. En todos los experimentos, la métrica utilizada fue la del rendimiento absoluto en términos de flops y comparado con el pico teórico que cada equipo posee.

La métrica utilizada es más dura que la escalabilidad, pero a su vez, contundente. Si bien se utilizaron rutinas de bibliotecas desarrolladas para altos rendimientos, y no se tuvo acceso ni posibilidad de realizar ajustes sobre el código, las pruebas estuvieron siempre planteadas para un gran volumen de datos, lo cual requiere de ajustes en la partición y la sincronización de la ejecución para lograr los rendimientos alcanzados.

Escalar una rutina que tenga un bajo rendimiento, por ejemplo de un 10%, puede ser más simple que alcanzar rendimientos del 66% del pico teórico, como el alcanzado para el algoritmo de Cholesky. Tener en cuenta que esto incluye el efecto de fallas de *cache* y de la administración de los procesos paralelos. Dicha marca solo se logra, además de utilizar una rutina optimizada, por particionar y coordinar las tareas en forma eficiente.

Lo mismo sucede para el caso de las pruebas de MM. Si bien en este

experimento no se logró obtener un rendimiento agregado entre las CPU's y las GPGPU's, alcanzar los límites del ancho de banda del canal de memoria realizando cómputo y comunicación interna en forma simultánea, solo se logra con una excelente gestión de los procesos paralelos. Recordar que en este caso, la biblioteca utilizada no presenta escalabilidad, la cual es efectivamente lograda a partir de la ejecución paralela de rutinas secuenciales.

El impacto que se estima puede generar este trabajo es una contribución en el desarrollo de programas paralelos de alto rendimiento, en particular en el modelado y análisis de los algoritmos y en acortar el tiempo de desarrollo de los mismos. La rápida adaptabilidad a diversos entornos de ejecución también se verá ayudada.

Cubrir el salto existente entre el modelado y el desarrollo de un programa de esta naturaleza es de gran ayuda, ya que los aplicaciones de uso científico requieren ambas características, alto rendimiento y la correctitud en sus resultados.

También es esperable que contribuya para los casos de paralelización de código heredado. En esta clase de código suele realizarse un análisis preliminar a la paralelización propiamentedicha. Se analizan los datos y las tareas que pueden ser divididas, la relación entre estas, la dependencia entre los datos y la carga computacional de cada sección de código [MMS05]. De dicho proceso podría obtenerse un esquema de tareas y bloques de datos útil para modelar con la herramienta desarrollada y así facilitar el proceso de paralelización del código heredado.

Es deseable que contribuya a la formación de recursos humanos, en especial a la capacitación dentro del área y a la investigación de temas abiertos por parte de nuevos investigadores.

5.2 Líneas abiertas de investigación

Las temas que se estiman quedan abiertos para continuar investigando relacionadas con temas del presente trabajo, las podemos dividir en líneas sobre el modelo en sí mismo y en líneas sobre los algoritmos a modelar. Entre las primeras podemos citar:

- Extender la implementación del modelo a memoria distribuida, en particular, a un *cluster* de computadoras. La dificultad principal en este

punto es la coherencia del Vector de Marcado entre los nodos componentes del *cluster*. Una opción para este caso puede ser dividir el cómputo global en partes para que cada nodo trabaje sobre una de estas. Otra opción para memoria distribuida es utilizar el acceso directo a memoria remota (RDMA) que ciertas implementaciones de MPI usualmente ofrecen, como por ejemplo, la implementación de MVAPICH [tOSU] que usa RDMA para implementar las directivas *one side*, y mantener así la coherencia por medio del uso de un esquema de memoria global en un *cluster*.

- Abrir el código fuente y documentar el *framework* para facilitar el desarrollo por parte de terceros programadores.
- Implementar las Matrices de Incidencia como matrices ralas, lo que permite disminuir el uso de memoria y el procesamiento en la selección de tareas. De esta forma se evitarían los inconvenientes en modelos con matrices grandes pero ralas, como lo acontecido en las pruebas de MM.
- Desarrollar una implementación del modelo en *hardware* reconfigurable, que facilite el desarrollo sobre *hardware* embebido.

En cuanto al tipo de algoritmos a implementar se puede citar:

- Realizar pruebas con algoritmos del tipo iterativos, cuyo número de pasos para resolver el problema es indeterminado al comenzar la ejecución ya que depende de los datos provistos. Para estos algoritmos, con una alta cantidad de pasos predefinidos, el tamaño de las Matrices de Incidencia generadas sería presuntamente prohibitivo, pero determinado precisamente, ya que esta clase de algoritmos tiene un número límite de iteraciones a la cual se llega siempre que el umbral de corte no haya sido alcanzado previamente. La implementación sobre matrices ralas sería conveniente en estos casos.
- Otro tipo de algoritmos distinto de las rutinas de álgebra lineal, como es el caso de algoritmos de grafos, ordenamientos, árboles, etc, clásicos dentro de la ciencia de la computación.
- Algoritmos complejos con múltiples rutinas, como es el caso de algoritmos de que resuelven problemas físicos, químicos, astronómicos, de ingeniería, etc. Es habitual que, si bien en estos casos intervengan muchos componentes algorítmicos, la mayor parte del costo computacional esté destinado a la resolución, por algún método particular apropiado al problema en cuestión, de un sistema de ecuaciones lineales.

- Paralelización de código heredado, que luego del análisis de sus componentes, dependencias y cargas, pueda ser modelado usando PEM y por lo tanto ser ejecutardo en paralelo sin necesidad de rearmar desde cero el código.

Apéndice A

Archivos de configuración del *framework*

Se adjuntan ejemplos de archivos de configuración del *framework* desarrollado y usados en algunas de las pruebas expuestas

El *framework*, cuyo ejecutable se llama tst_pn, es llamado desde la línea de comandos con:

```
>tst_pn run_config_mm.xml
```

El archivo `run_config_mm.xml` es el parámetro de ejecución principal, que contiene ciertos valores y referencias a otros archivos de configuración. Muchos de los valores son autoexplicados.

```
<xmldef_run_config>
<pnet_def_file>mmx3x4</pnet_def_file>
<proc_def_file>dice+2_pu.xml</proc_def_file>
<thre_def_file>dice+2x2_cores.xml</thre_def_file>
<bank_def_file>cuat_bank.xml</bank_def_file>
<num_mat>3</num_mat>
<mat_size>24000</mat_size>
<mat_row_blocks>3</mat_row_blocks>
<mat_col_blocks>3</mat_col_blocks>>
<mat_sub_row_blocks>4</mat_sub_row_blocks>
<mat_sub_col_blocks>4</mat_sub_col_blocks>
</xmldef_run_config>
```

El archivo `mmx3x4` contiene la Red de Petri desplegada para el modelo de MM de pag. 93, con división $n = 3$ y $r = 4$. El archivo no es expuesto en este anexo por su longitud.

El archivo `dice+2_pu.xml` contiene los parámetros de definición de los dieciseis procesadores definidos sobre CPU más los dos sobre GPGPU.

```
<xmldef_procs>
<processor>
<pname>Proc1</pname>
<pclass>PetriProcessor</pclass>
<bname>b1</bname>
<gpunum>1</gpunum>
<prdeffile>mmtaskgpu.txt</prdeffile>
</processor>
<processor>
<pname>Proc2</pname>
<pclass>PetriProcessor</pclass>
<bname>b1</bname>
<gpunum>2</gpunum>
<prdeffile>mmtaskgpu.txt</prdeffile>
</processor>
<processor>
<pname>Proc3</pname>
<pclass>PetriProcessor</pclass>
<bname>b1</bname>
<gpunum>0</gpunum>
<prdeffile>mmtaskamd.txt</prdeffile>
</processor>
<processor>
<pname>Proc4</pname>
<pclass>PetriProcessor</pclass>
<bname>b1</bname>
<gpunum>0</gpunum>
<prdeffile>mmtaskamd.txt</prdeffile>
</processor>

...

</xmldef_procs>
```

El archivo `dice+2x2_cores.xml` contiene la asignación o afinidad de cada procesador con los *cores* físicos en el computador.

```
<xmldef_thread_pool>
<thread_master>
<core_num>2</core_num>
<proc>Proc1</proc>
</thread_master>
<thread_master>
<core_num>40</core_num>
<proc>Proc2</proc>
</thread_master>
<thread_master>
<core_num>10</core_num>
<core_num>11</core_num>
<proc>Proc3</proc>
</thread_master>
<thread_master>
<core_num>4</core_num>
<core_num>5</core_num>
<proc>Proc4</proc>
</thread_master>

...

</xmldef_procs>
```

El archivo `chtaskamd.txt` contiene el mapeo de tareas a rutinas a ejecutar, para el caso del algoritmo de Cholesky.

```
<net_procdef>
<singled>
<task><name>potr</name><rutine>cpu_spotrf</rutine></task>
<task><name>gemm</name><rutine>cpu_sgemm</rutine></task>
<task><name>syrk</name><rutine>cpu_ssyrk</rutine></task>
<task><name>trsm</name><rutine>cpu_strsm</rutine></task>
</singled>
<doubled>
<task><name>potr</name><rutine>cpu_dpotrf</rutine></task>
```

```
<task><name>gemm</name><rutine>cpu_dgemm</rutine></task>
<task><name>syrk</name><rutine>cpu_dsyrk</rutine></task>
<task><name>trsm</name><rutine>cpu_dtrsm</rutine></task>
</doubled>
</net_procdef>
```

Bibliografía

[AAD+10] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[ABB+99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[ADD+09] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, Vol. 180, 2009.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[ATN10] Cedric Augonnet, Samuel Thibault, and Raymond Namyst. Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. Technical Report 7240, INRIA, March 2010.

[BLA01] Basic Linear Algebra Subprograms Technical Forum Standard. Technical report, University of Tennessee, 2001.

[BLKD07] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report 191, LAPACK Working Note, September 2007.

[Boa] The OpenMP Architecture Review Board. Openmp. the openmp api specification for parallel programming. http://openmp.org/.

[BSM10] Luiz F. Bittencourt, Rizos Sakellariou, and Edmundo R. M. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 0:27–34, 2010.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CL08] Christos G. Cassandras and Stéphane Lafortune. *Introduction to Discrete Event Systems*. Springer Science+Business Media, LLC, Boston, MA, 2008.

[CLR09] H. Casanova, A. Legrand, and Y. Robert. *Parallel algorithms*. Chapman & Hall/CRC numerical analysis and scientific computing. CRC Press, 2009.

[Cora] The AMD Corporation. The amd core math library. http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/.

[Corb] The NVIDIA Corporation. The cublas library. https://developer.nvidia.com/cuBLAS.

[Corc] The NVIDIA Corporation. The cuda toolkit. http://www.nvidia.com/object/cuda_home_new.html.

[Des87] George R. Desrochers. *Principles of Parallel and Multiprocessing*. McGraw-Hill, Inc., New York, NY, USA, 1987.

[DFLL11] Jack Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile lu factorization. Technical Report 259, LAPACK Working Note, December 2011.

[Dia09] Michel Diaz. *Petri Nets: Fundamental Models, Verification and Applications*. ISTE Ltd - John Wiley & Sons, Inc., London, Hoboken, 2009.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[FLBGR13] Joao Vicente Ferreira Lima, Francois Broquedis, Thierry Gautier, and Bruno Raffin. Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In *25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Porto de Galinhas, Brazil, October 2013.

[Fow01] Martin Fowler. Reducing coupling. *IEEE Software*, 18(4):102–104, 2001.

[FSJ99] Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. *Building Application Frameworks: Object-oriented Foundations of Framework Design.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[GFLMR13] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Boston, Massachusetts, États-Unis, May 2013.

[gtx12] GTX 680 Kepler whitepaper. Technical report, NVIDIA Corporation, 2012.

[HLYD11] Azzam Haidar, Hatem Ltaief, Asim YarKhan, and Jack Dongarra. Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures. Technical Report 243, LAPACK Working Note, March 2011.

[HMU03] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley series in computer science. Pearson Education International, 2003.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Hog08] JD Hogg. A dag-based parallel cholesky factorization for multicore systems. Technical report, Technical Report RAL-TR-2008-029, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2008.

[IA06] Marian Iordache and Panos Antsaklis. *Supervisory Control of Concurrent Systems: A Petri Net Structural Approach.* Birkhäuser Boston, Boston, 2006.

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems.* Springer, 2009.

[KD06] Jakub Kurzak and Jack J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. Technical Report 178, LAPACK Working Note, September 2006.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[KLFD13] Jakub Kurzak, Piotr Luszczek, Mathieu Faverge, and Jack Dongarra. Lu factorization with partial pivoting for a multicore system with accelerators. *IEEE Trans. Parallel Distrib. Syst.*, pages 1613–1621, 2013.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[LTN+09] Hatem Ltaief, Stanimire Tomov, Rajib Nath, Peng Du, , and Jack Dongarra. A scalable high performant cholesky factorization for multicore with gpu accelerators. Technical Report 223, LAPACK Working Note, November 2009.

[MMS05] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Reengineering for parallelism: An entry point for plpp (pattern language for parallel programming) for legacy applications. In *Proceedings of the Twelfth Pattern Languages of Programs Workshop (PLoP 2005)*, 2005.

[Pom] Frank Pommereau. The snakes toolkit. https://www.ibisc.univ-evry.fr/~fpommereau/SNAKES/.

[Pom09] Franck Pommereau. *Algebras of coloured Petri nets*. Habilitation thesis, University Paris-East, Créteil, 11 2009.

[PZ91] Louchka Popova-Zeugmann. On time petri nets. *Elektronische Informationsverarbeitung und Kybernetik*, 27(4):227–244, 1991.

[RR10] Thomas Rauber and Gudula Rünger. *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2010.

[RV09] Yves Robert and Frédéric Vivien, editors. *Introduction to Scheduling*. Chapman and Hall/CRC Press, 2009.

[SB09] S. Sriram and S.S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Signal Processing and Communications. Taylor & Francis, 2009.

[SOHL+98] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[SOW13] Janet Soler, Javier Ortiz, and Gustavo Wolfmann. Strategies to optimize the LU Factorization Algorithm on multicore computers. In *Proceedings of the VI Latin American Symposium on High Performance Computing - HPCLatAm 2013*, Mendoza, Argentina, 2013.

[TDB10] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parallel Comput.*, 36(5-6):232–240, June 2010.

[THW99] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*, pages 3–14. IEEE, 1999.

[tOSU] the Ohio State University. The MVPICH: MPI over Infiniband, 10GigE/iWarp and RoCE. http://mvapich.cse.ohio-state.edu/.

[tUoTa] the University of Tennessee. The MAGMA project. Matrix Algebra for GPU and Multicore Architectures. http://icl.cs.utk.edu/magma.

[tUoTb] the University of Tennessee. The PLASMA project. Parallel Linear Algebra for Scalable Multicore Architectures. http://icl.cs.utk.edu/plasma.

[TW08] Fernando G. Tinetti and Gustavo Wolfmann. Análisis de paralelización con memoria compartida y memoria distribuida en clusters de nodos con múltiples núcleos. In *XIV Congreso Argentino de Ciencias de la Computación - CACIC 2008*, Chilecito, Argentina, Octubre de 2008.

[TW09] Fernando Tinetti and Gustavo Wolfmann. Parallelization analysis on clusters of multicore nodes using shared and distributed memory parallel computing models. In *CSIE 2009, 2009 WRI World Congress on Computer Science and Information Engineering, March 31 - April 2, 2009, Los Angeles, California, USA, 7 Volumes*, pages 466–470, 2009.

[TW11] Fernando G. Tinetti and Gustavo Wolfmann. Broadcast and partial computing algorithms for cholesky factorization on a cluster of multicore computers. In *The 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-11)*, 2011.

[Wan98] J. Wang. *Timed Petri Nets: Theory and Application*. The International Series on Discrete Event Dynamic Systems. Springer US, 1998.

[WDG13] Gustavo Wolfmann and Armando De Giusti. Parallel asynchronous modelization and execution of cholesky algorithm using petri nets. In *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-13)*, 2013.

[WDG14a] Gustavo Wolfmann and Armando De Giusti. Algorithm model and execution based on petri nets in an heterogeneous parallel computer. In *First HPCLATAM - CLCAR Latin American Joint Conference, CARLA 2014*, Valparaíso, Chile, 2014.

[WDG14b] Gustavo Wolfmann and Armando De Giusti. Petri net based algorithm modelization and parallel execution on symmetric multiprocessors. In *The 2014 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA-14)*, 2014.

[Wei] Eric W. Weisstein. Tetrahedral Number from MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/TetrahedralNumber.html.

[Wol10] Gustavo Wolfmann. First results in the parallelization of cholesky factorization algorithm over a cluster of multicore computers using partial computing. In *HPC 2010 - 39º JAIIO (Jornadas Argentinas de Informática e Investigación Operativa)*, Buenos Aires, Argentina, Setiembre 2010.

[WT08] Gustavo Wolfmann and Fernando G. Tinetti. Cómputo intensivo en clusters de nodos multicore: Análisis de speed up y eficiencia. In *37º JAIIO (Jornadas Argentinas de Informática e Investigación Operativa)*, Santa Fe, Argentina, Setiembre 2008.

[WT09] Gustavo Wolfmann and Fernando G. Tinetti. The impact of network architecture in cluster parallel algorithms design: Matrix multiplication on infiniband. In *HPC 2009 - 38º JAIIO (Jornadas Argentinas de Informática e Investigación Operativa)*, Mar del Plata, Argentina, Agosto 2009.

[Yar12] Asim YarKhan. *Dynamic Task Execution on Shared and Distributed Memory Architectures*. Doctor of philosophy thesis, University of Tennessee, 12 2012.

[YKD11] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.