Análisis de patrones de paralelismo bajo la óptica de las aplicaciones de cómputo científico sobre *clusters* de nodos *multicore*

Wolfmann Gustavo

Tesis para alcanzar el grado de
Magíster en Ingeniería del Software

Director:
Dr. Fernando G. Tinetti

Co-director:
Dr. Gustavo Rossi

Facultad de Informática
Universidad Nacional de La Plata

10 de diciembre de 2010

Índice general

1. Introducción y Antecedentes **1**

1.1. Objetivos 5

1.2. Estructura de la tesis 7

2. Arquitectura y programación de *clusters* de nodos *multicore* **9**

2.1. Breve evolución de las computadoras paralelas 9

2.2. *Clusters* de computadoras 10

2.2.1. Arquitectura de los *clusters* 10

2.2.2. Programación de aplicaciones bajo *cluster* 12

2.3. Computadoras *multicore* 13

2.3.1. Arquitectura de los *multicore* 13

2.3.2. Programación de aplicaciones en *multicore* 14

2.4. Programación de *clusters* de nodos *multicore* 16

2.5. Resumen del capítulo 19

3. Patrones de paralelismo **21**

3.1. Evolución y estado actual de los patrones de paralelismo . . . 21

3.2. Algunos Patrones de Paralelismo 26

3.2.1. Patrones de división y reagrupación de tareas 27

3.2.1.1. Task Decomposition Pattern 27

3.2.1.2. Group Task Pattern 28
3.2.2. Patrones de distribución de datos 29
3.2.2.1. Data Decomposition Pattern 29
3.2.2.2. Data Sharing Pattern 30
3.2.3. Otros patrones 32
3.2.3.1. Design Evaluation Pattern 32
3.2.3.2. Dense Linear Algebra 34
3.2.3.3. Master / Worker Pattern 35
3.2.3.4. Wavefront Pattern 37
3.2.3.5. Non-Blocking Communications Pattern 38
3.2.4. Reengineering for parallelism 40
3.2.4.1. Reengineering for parallelism pattern 41
3.3. Análisis de los patrones existentes bajo la óptica de las aplicaciones de cómputo intensivo 43
3.4. Resumen del capítulo 47

4. Experimentación de la aplicación de patrones de paralelismo sobre algoritmos seleccionados 49

4.1. Multiplicación de Matrices 51
4.1.1. Memoria Compartida 52
4.1.2. Memoria Distribuida - Comunicaciones colectivas . . . 54
4.1.3. Memoria Distribuida - Patrón *Master/Worker* 55
4.1.4. Memoria Distribuida - Patrón *Wavefront* 59
4.1.5. Memoria Distribuida - Uso de cómputos parciales . . . 63
4.2. Factorización de Cholesky 70
4.2.1. Memoria Compartida 72
4.2.2. Memoria Distribuida - Comunicaciones colectivas . . . 74
4.2.3. Memoria Distribuida - Uso de cómputos parciales . . . 77

4.3. Resumen del capítulo 82

5. Patrones de paralelismo para computación de alto rendimiento (HPC) 83

5.1. Propuestas sobre contenido y estructura de los patrones de paralelismo enfocados al HPC 84

5.2. *Partial Computing Pattern* 87

5.2.1. Nombre 89

5.2.2. Propósito 89

5.2.3. Contexto 89

5.2.4. Fuerzas 91

5.2.5. Solución 91

5.2.6. Ejemplos 94

5.2.6.1. Factorización de Cholesky 94

5.2.6.2. Multiplicación de Matrices 98

5.2.6.3. Rendimientos Obtenidos 102

5.2.7. Patrones Relacionados 103

5.3. Resumen del capítulo 104

6. Conclusiones e impacto de la tesis 107

6.1. Conclusiones Específicas 108

6.2. Conclusiones Generales 110

6.3. Impacto y trabajos futuros 111

6.3.1. En la implementación de programas paralelos 111

6.3.2. En los temas de investigación abiertos 112

6.3.3. Reflexión Final 112

Índice de tablas

4.1. Tiempos de ejecución y *speedup* para multiplicación de matrices bajo memoria compartida en un nodo con 8 *cores* del *cluster* 1. 53

4.2. Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria distribuida. Algoritmo de *Broadcast* bajo Ethernet e Infiniband, utilizando el cluster 2 de pag 51. . . . 55

4.3. Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria distribuida. Algoritmos de *Broadcast* y *Master/Worker* bajo Ethernet e Infiniband, utilizando el cluster 2 de pag 51. 57

4.4. Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria compartida. Algoritmos de *Broadcast* y *Master/Worker* sobre Infiniband, utilizando el cluster 3 de pag 51. 58

4.5. Tiempos de segundos de multiplicación de matrices aplicando el patrón *wavefront*, corriendo en 4 nodos con Ethernet o con Infiniband 62

4.6. Tiempos en segundos de la multiplicación de matrices en 4 nodos aplicando *wavefront* sobre Infiniband con múltiples envíos simultáneos no bloqueantes 63

4.7. Comparación de tiempos para los algoritmos utilizando *broadcast* y *Partial Computing* para multiplicación de matrices . . . 69

4.8. Tiempos de ejecución en segundos para la factorización de Cholesky bajo memoria compartida con Sunperf y MKL. . . . 73

4.9. Tiempos de ejecución en segundos del algoritmo de Cholesky, con *broadcast* y la versión Scalapack, sobre *cluster* 1. Scalapack lanzado con una grilla de 32 procesadores MPI, y *broadcast* con 8 procesos MPI × 4 *threads* OpenMP en cada uno. 76

4.10. Tiempos de ejecución (en segundos) de los tres algoritmos de Cholesky experimentados, usando Ethernet e Infiniband, sobre *cluster* 2 usando 6 nodos. 80

4.11. Tiempos de ejecución (en segundos) de los tres algoritmos de Cholesky experimentados, usando Infiniband, sobre *cluster* 1 usando 6 nodos, 4 *cores* por nodo y también sobre *cluster* 3 usando 8 nodos, 8 *cores* por nodo. 80

5.1. Tiempos de ejecución (segs.) del algoritmo de Cholesky usando *Broadcast* o *Partial Computing*, sobre Ethernet e Infiniband, sobre 6 nodos, 8 *cores* cada uno. 104

5.2. Tiempos de ejecución (segs.) de los tres algoritmos de Cholesky experimentados, usando Infiniband, usando 8 *cores* por nodo. . 104

5.3. Comparación de tiempos para los algoritmos utilizando *broadcast* para la distribución por bandas y *Partial Computing* para multiplicación de matrices. Hasta 64 nodos, 8 *cores* cada uno, Infiniband. 105

Capítulo 1

Introducción y Antecedentes

La resolución de problemas científicos utilizando computadoras ha tenido una tendencia irrevocable en cuanto al tamaño de datos procesados. La disponibilidad de tecnologías cada vez más potentes ha posibilitado que la cantidad de datos procesados sea cada vez mayor, o se insuma un menor tiempo para iguales cantidades de procesamiento. En parte esto se ha debido al aumento en la velocidad de los procesadores y en parte también a las facilidades para distribuir el cómputo entre varios procesadores en paralelo.

En nuestros días, la tasa de crecimiento en la velocidad de los procesadores ha caído, en gran parte debido a que la tecnología utilizada para su construcción, basada en silicio, ha llegado a su límite de posibilidades [ABC+06]. Mientras que no exista un cambio en dicha tecnología, el crecimiento de la potencia de cálculo está centrada en la distribución en paralelo del cómputo, es decir, en paralelizar el procesamiento[ABD+09].

En la actualidad existen dos tipologías básicas en cuanto a la arquitectura paralela de un sistema de computación. Por un lado están las computadoras con múltiples unidades internas de procesamiento (multiprocesadores) y por otro, los sistemas compuestos por varias computadoras independientes que conforman una unidad de cómputo (multicomputadores). En estas últimas, la capacidad de cómputo viene dada por la suma de los procesadores de cada una de las computadores, pero debe tenerse en cuenta que los datos a procesar, deben distribuirse en la memoria de cada equipo individual que conforma el multicomputador.

Históricamente los equipos multiprocesados fueron los primeros en surgir. Como el costo de un computador era muy elevado, el crecimiento de la capacidad de procesamiento se centraba en esa estrategia. Sin embargo, el

advenimiento de la computación masiva dado por el bajo costo de los equipos y de las redes de comunicación para unirlos, indujo a que fuera económicamente viable armar grupos, o *clusters* de computadoras baratas que trabajen simultáneamente. Este fenómeno trajo aparejado un cambio en la tecnología a aplicar en el procesamiento paralelo. En lugar de tener un mismo programa corriendo en varias unidades internas de procesamiento como en los primeras épocas, esta nueva arquitectura de procesamiento impone la existencia de varias instancias de un mismo programa corriendo en máquinas independientes, con sus versiones locales de datos, cooperando entre sí por medio de intercambio de información, utilizando la red de comunicaciones constituida a tales efectos.

Por otro lado, como se señaló antes, la tecnología del silicio ha llegado a un punto tal que hace difícil seguir aumentando la velocidad de los procesadores. Por ello es que en los últimos años la industria ha vuelto a tomar la línea de avance por el lado de los multiprocesadores, es decir, disponer de varias unidades de procesamiento en cada computador, de forma tal que la potencia de cómputo crezca de la mano del paralelismo. El surgimiento de los procesadores *multicore* de bajo costo es tal, que en la actualidad es difícil encontrar procesadores sin esta característica.

Estos hechos provocaron retomar la tecnología de paralelismo de los equipos multiprocesados de antaño, ya que la filosofía es similar, en cuanto a disponer de una única instancia de memoria común a todos los procesadores del equipo. Los múltiples núcleos o *cores* pueden correr un mismo programa o diferentes. En el primer caso, esta arquitectura permite eludir la necesidad de comunicar los procesos por medio de pasos de mensajes entre ellos al disponer de un área común de datos.

Sin embargo, nuevamente el factor económico hace que surga una nueva arquitectura de hardware paralelo, el llamado *cluster* de equipos *multicore*. Este nuevo sistema paralelo está conformado por varias computadoras que trabajan mancomunadamente entre sí, cada una de las cuales dispone a su vez de varias unidades de procesamiento. La capacidad de procesamiento en paralelo es una combinación de múltiples *cores* con múltiples equipos, lo que *a priori* permite inducir que la capacidad de cómputo de este equipamiento se multiplica. Sin embargo, es solo por medio de la programación que se puede aprovechar esa capacidad de cómputo, ya que ambos modelos de memoria poseen diferentes formas de trabajar en paralelo, por lo que se transforma en un modelo híbrido que debe tener en cuenta cuándo y cómo paralelizar dentro o fuera de cada equipo de tal forma de poder usufructuar los beneficios potenciales de esta arquitectura de hardware.

La programación de los equipos *multicore* está basada en el llamado **"Modelo de memoria compartida"**, el cual surge a partir del hardware subyacente sobre el cual deben correr los programas, compuesto por varias unidades de procesamiento (una única integrada con varios núcleos o varias integradas con varios núcleos) que disponen de una memoria para datos común a todas ellas. Bajo este modelo, el paralelismo del programa se logra lanzando varios hilos de ejecución simultáneos de un mismo programa. La implementación más usual en la actualidad dentro del ámbito de aplicaciones de cómputo científico que implementa este modelo, es la denominada OpenMP [ARB], la cual no posee directivas dedicadas a la comunicación de los procesos que corren en paralelo; la comunicación de los procesos se hace por medio de un área común de datos y es el único medio de vinculación entre los distintos hilos de ejecución en paralelo disponible bajo esta tecnología.

Dentro del ámbito de los *clusters*, la programación en paralelo habitualmente se logra utilizando el llamado **"Modelo de memoria distribuida"**. Como la arquitectura de *hardware* subyacente lo impone, el paralelismo se logra lanzando un proceso por máquina, generalmente el mismo, que trabajan como un todo en forma conjunta. Cada proceso dispone de su espacio de memoria exclusivo. Los datos son compartidos por los procesos por medio del paso de mensajes entre ellos[1], o bien, cada proceso dispone de su propio juego de datos distribuido de antemano al inicio del programa, pero realiza una parte del procesamiento total, el cual es alcanzado al unir los resultados de cada proceso. La tecnología de paso de mensajes predominante es la llamada MPI por *Message Passing Interface*[Conb].

Por otro lado, las **aplicaciones de cómputo científico** están en su mayoría desarrolladas por programas escritos en lenguaje FORTRAN o C/C++, existiendo muy pocos desarrollos que utilicen otros lenguajes. Esto es debido a que son aplicaciones de cálculo numérico que requieren de un alto nivel de procesamiento, dado el gran volumen de cómputo que se necesita realizar; dichos lenguajes de programación son los que mejores rendimientos alcanzan. No resulta extraño entonces que las implementaciones de OpenMP y MPI más eficientes estén disponibles para ambos lenguajes. De esta manera, la programación de aplicaciones que se quiera ejecutar en un *cluster* con nodos *multicore* está casi circunscrita a programas FORTRAN o C/C++ que contengan una combinación de directivas de OpenMP y MPI, y en general se denomina "Modelo de programación híbrido MPI/OpenMP" [AMR+05].

Existe otro paradigma de desarrollo de aplicaciones de cómputo científi-

[1]Por ello también se lo suele llamar "Modelo de paso de mensajes".

fico en paralelo, denominado "Computación de Alto Rendimiento" (HTC - *High Throughput Computing*), que se define como el procesamiento realizado por un número grande de tareas independientes entre sí, corriendo sobre equipos individuales, *clusters* o sobre sistemas distribuidos, posibilitando la realización de grandes cantidades de cómputos durante largos períodos de tiempo [SK09] [cona]. Este paradigma se diferencia del HPC (*High Performance Computing*), el cual está ligado al concepto de *supercomputación*, es decir, orientado al gran poder de cómputo intensivo. La tendencia de los últimos años ha sido que las principales supercomputadoras estén diseñadas como *clusters*: arreglos de computadoras homogéneas comunicadas con redes de alta tasa de transferencia y baja latencia [Zel09, org].

El modelo de programación en paralelo del HTC tiene la característica distintiva de poder utilizar recursos computacionales heterogéneos, no solo a nivel de procesador, sino incluso de sistema operativo, los cuales se conectan por medio de redes de alta latencia y bajo ancho de banda. Sin embargo, no se realizan comunicaciones entre los procesos lanzados en paralelo. Esta arquitectura permite ejecutar un gran volumen de procesamiento por medio de corridas que demoren semanas o meses de procesamiento, más allá de lo posible de realizar en *clusters* de tamaño "habitual". De esta forma, se logran altos rendimientos en términos de magnitudes procesadas, dejando de lado la tasa de eficiencia de uso individual de cada equipamiento. Es por ello que esta variante de computación científica queda fuera del alcance de este trabajo.

Desde el punto de vista de la codificación, las aplicaciones secuenciales de cómputo científico presentan en general una marcada ausencia de propiedades deseables desde la óptica de la ingeniería de software, a saber, la modularidad, mantenibilidad, estilo en la programación y escalabilidad entre otras, por cuanto la paralelización se torna en una tarea como mínimo compleja, siendo aconsejable encarar el proyecto con un conjunto de herramientas mínimo que apoyen la tarea. La opción de reescribir la aplicación desde cero con buenas prácticas es casi inaceptable, ya que es aún más difícil conseguir el experto con tiempo disponible que esté dispuesto a transmitir el conocimiento necesario para este proyecto, además del alto tiempo de maduración que las mismas requieren [MMS07, OXJF05].

El aporte de los patrones de diseño dentro de la Ingeniería de Software es indiscutido [YA03]. La existencia de buenas prácticas, encapsuladas en estructuras estandarizadas, son de fundamental ayuda para el diseñador de *software*. Sin embargo, los patrones no están limitados al diseño de *software*, sino también, entre otros, al análisis [Fow96] y a la arquitectura del *software* [BMR+96]. En este orden de cosas, se presentan los patrones de paralelismo

como herramientas de ayuda para el desarrollo de aplicaciones paralelas. En general los **patrones de diseño** son soluciones a problemas frecuentes, de forma tal que se puedan utilizar reiteradamente, cada vez que el problema tratado se presente [GHJV95]. Básicamente capturan la experiencia de resolver un cierto tipo de problema, frente al cual, el patrón brinda la solución del mismo.

La descripción de un patrón posee cierta estructura o formato que debe respetarse y es relativamente estándar y aceptado por la comunidad informática. Dicha estructura está generalmente compuesta por el nombre del patrón, una descripción del tipo de problema al que hace referencia, la solución propuesta y las consecuencias o efectos que la misma provoca. Ciertos autores también proponen incluir características tales como la motivación del patrón, fuerzas intervinientes en la implementación de la solución, la aplicabilidad del mismo, ejemplos de casos donde la solución es alcanzada y referencias a otros patrones relacionados.

Específicamente en el ámbito informático han surgido un gran número de patrones que atacan problemáticas muy diversas, como ser los patrones de diseño, patrones de redes, patrones de seguridad y lógicamente también patrones de paralelismo. Estos últimos están destinados a brindar un abanico de soluciones frente a los problemas habituales que deben enfrentarse al desarrollar un programa que sea ejecutado simultáneamente por varios procesadores, abarcando soluciones desde el nivel de la concurrencia del problema, pasando por la estructura del algoritmo paralelo y llegando al nivel de los mecanismos de implementación del paralelismo [MSM04, KM09].

Los patrones de paralelismo de nivel más abstracto analizan el problema a nivel de concurrencia y están enfocados a la descomposición de tareas y la distribución de datos, como objetivos principales de su temática. Los patrones ligados a la estructura del algoritmo tratan temas sobre la recurrencia, estrategias de *"divide y vencerás"* o la estructura *"pipeline"*, por citar algunos ejemplos. A nivel de implementación se estudian soluciones del tipo *fork/join*, paralelismo sobre lazos, o el modelo SPMD (*"Simple Program Multiple Data"*) [MSM04, KM09].

1.1. Objetivos

Hasta este punto se han presentado los antecedentes de la presente tesis: por un lado un breve resumen del estado actual de la tecnología de *hardware* y

de *software* disponible para el desarrollo de aplicaciones de cómputo científico, como así también las características generales de este tipo de aplicaciones, y por otro lado, la existencia de patrones de paralelismo, como soluciones a problemas generales y habituales en la paralelización de aplicaciones. Esta temática es de reciente data, en particular los patrones de paralelismo, por lo que uno de los objetivos de esta tesis es estudiar el estado del arte de estos temas, bajo el enfoque del tipo de aplicaciones en consideración y analizar su pertinencia, es decir, la aplicabilidad de los patrones de paralelismo y su alcance en la solución al problema del desarrollo de aplicaciones de cómputo científico bajo un *cluster* de nodos *multicore*.

Debido a que el tema de los patrones de paralelismo está aún en desarrollo, es de esperar que a partir del presente estudio surjan fundamentos que motiven cambios en los actuales patrones, los cuales podrían ser particularizados, ampliados, reducidos o modificados para adecuarse al tipo de aplicaciones bajo consideración, por lo que otro objetivo de la tesis es brindar un aporte sobre los elementos que deberían ser tenidos en cuenta a la hora de desarrollar patrones específicos para las aplicaciones bajo estudio; en particular, un tema de vital importancia es la eficiencia en el uso de los recursos computacionales, ya que de nada sirve paralelizar una aplicación y que su *performance* no se vea mejorada acorde a los recursos de *hardware* utilizados, por lo que patrones de paralelismo específicos al dominio bajo estudio deberían tener en cuenta algún tipo de indicación al respecto de la eficiencia.

El alcance y pertinencia de los patrones de paralelismo presentados en este trabajo, ha sido corroborado sobre algunos algoritmos de cómputo científico mediante la realización de experimentos específicos, los cuales consistieron en la ejecución de programas que resulten de la aplicación de distintos patrones a cada uno de los algoritmos elegidos como *"testbed"*. Sus resultados, que son presentados en el capítulo 4, permitieron constatar el alcance de la aplicación de los patrones sobre el nivel de rendimiento obtenido.

Como consecuencia de la realización de estas experimentaciones buscando mejorar los resultados de los algoritmos paralelos, surgió un aporte adicional de la tesis, el cual es la presentación de un nuevo patrón de paralelismo, en una versión preliminar, el cual ha sido denominado *"Partial Computing Pattern"*, debido a que busca optimizar los cómputos en paralelo aprovechando la disponibilidad de datos parciales en tiempos inactivos de los procesadores.

1.2. Estructura de la tesis

El trabajo está organizado de la siguiente manera:

Capítulo 2 destinando a una breve presentación de las características los *clusters* de nodos *multicore*, sus elementos constitutivos, arquitectura y tecnologías disponibles para la programación de aplicaciones paralelas, como así también una breve introducción a las tecnologías de programación OpenMP y MPI, dado que son las mas usuales y difundidas en la actualidad para la programación de aplicaciones científicas en *clusters*.

Capítulo 3 dispone de secciones sobre el estado del arte de los patrones de paralelismo, su evolución y las características principales que estos presentan, y una sección adicional dedicada a la exposición de algunos de los patrones de paralelismo más significativos. Los patrones son expuestos resumidamente, y dentro del universo de patrones de paralelismo existente, fueron seleccionados los que *a posteriori* fueron aplicados en los experimentos desarrollados, a los fines de determinar su impacto en el rendimiento.

Capítulo 4 por su parte presenta los resultados de la experimentación de aplicar diversos patrones de paralelismo a una serie de algoritmos de cómputo científico, principalmente de álgebra lineal, destinados a fundamentar la pertinencia de los patrones de paralelismo bajo en enfoque de la *performance* lograda.

Capítulo 5 tiene una propuesta sobre cambios en los actuales patrones de paralelismo para poder ser útiles en el dominio de aplicaciones bajo estudio. Adicionalmente se presenta una versión preliminar de un nuevo patrón de paralelismo, el *Partial Computing Pattern*, surgido como consecuencia de la búsqueda de mejorar los tiempos de ejecución de los algoritmos del capítulo anterior, al tratar de utilizar tiempos inactivos de los nodos del *cluster*, cuya razón de ser es generalmente la necesidad de realizar sincronizaciones entre nodos.

Capítulo 6 contiene las conclusiones de la tesis sobre la pertinencia de los patrones de paralelismo para las aplicaciones de cómputo científico en *clusters* de nodos *multicore*, tanto sobre temas puntuales como generales. Se incluye también, el impacto de la tesis y un listado de lineas de investigación abiertas en la materia.

Capítulo 2

Arquitectura y programación de *clusters* de nodos *multicore*

2.1. Breve evolución de las computadoras paralelas

Históricamente, los primeros modelos de computadoras que permitían paralelizar aplicaciones fueron los equipos que disponían de múltiples procesadores de forma tal que se pudiera realizar en paralelo más de un procesamiento. Son los denominados equipos "multiprocesadores" (multiprocesador de aquí en adelante), donde, siguiendo la taxonomía de Flynn [Fly72], pueden clasificarse en SIMD (*Single Instruction Multiple Data*), MISD (*Multiple Instruction Single Data*), o MIMD(*Multiple Instruction Multiple Data*), es decir, acorde a la disponibilidad de múltiples juegos de datos e instrucciones.

A lo largo del tiempo surgieron varios diseños en cuanto a la arquitectura de los multiprocesadores, dependiendo de la forma de disponer y compartir la memoria entre los distintos procesadores, encontrando diseños donde la totalidad de la memoria es compartida por todos los procesadores con igual costo de acceso (*Uniform Memory Access* - UMA), o donde toda la memoria es accesible por todos los procesadores, pero con distintos tiempo de acceso, ya que los bancos de memoria están ligados a un procesador en particular y éste tiene acceso directo a su banco, pero indirecto a los restantes (*Non Uniform Memory Access* - NUMA y *cache coherent Non Uniform Memory Access* - ccNUMA)[GKKG03].

Los costos elevados de este tipo de computadoras, hicieron que fueran

poco difundidas, poco estandarizadas y la existencia de un escaso número de recursos humanos disponibles con capacidad de poder utilizar las potencialidades que cada modelo brindaba.

Avanzando en el tiempo, la popularización de las computadoras de escritorio, unido a su bajo costo, y la extensa difusión y facilidad para la creación de redes de área local (LAN) que interconecten estas computadoras, dio lugar a la conformación de redes de computadoras tipo PC, pero que en lugar de trabajar cada una independientemente de las otras, entre todas conforman una unidad computacional que trabaja en conjunto para alcanzar un objetivo común, es decir, la conformación de un multicomputador compuesto por varias computadoras físicamente independientes, también denominado *cluster*. Siguiendo a Flynn, este sería un caso de *Multiple Instruction Multiple Data*-MIMD, pero con memoria distribuida.

La capacidad de procesamiento de los *clusters* está dada por la cantidad y capacidad individual de la computadoras (nodos) que la integran y el rendimiento o capacidad de transferencia de la red que las comunica. En los primeros tiempos, el rendimiento de la red era bajo, por lo que la capacidad global de procesamiento del *cluster* también lo era. La gran difusión que los *clusters* fue ganando, principalmente por su bajo costo, dio lugar a investigaciones e inversiones para mejorar la tecnologías de red que lo conforman, lo que sumado a los avances en la capacidad de procesamiento individual de las CPU's, da lugar a que en la actualidad, gran número de las multicomputadoras más potentes del planeta, según el sitio web `www.top500.org`, estén construidos bajo la arquitectura de un *cluster*.

2.2. *Clusters* de computadoras

2.2.1. Arquitectura de los *clusters*

Un *cluster* es una multicomputadora, conformada por un conjunto de computadoras de escritorio, llamados nodos, comunicados entre sí por una red de área local (LAN), configurados para que todos los nodos puedan correr un mismo programa, aunque con distintas versiones de datos, lo que se denomina "paralelismo de datos" o modelo de ejecución paralelo SPMD (*Single Program Multiple Data*). Como la memoria de cada nodo no es accesible en forma directa por los restantes nodos, se denomina "Modelo de Memoria Distribuida".

La forma de comunicación entre los nodos que componen el *cluster*, tanto para el intercambio de datos, como para la sincronización del procesamiento, es por medio del paso de mensajes. En este tipo de multicomputador, dado que los procesos corren independientes en cada nodo, los procesos se comunican utilizando el modelo de paso de mensajes, donde cada de ellos tiene definidos una serie de puntos en su ejecución en donde envía o queda a la escucha de mensajes enviado/s por otro/s proceso/s, por lo que de esta manera se produce el intercambio de resultados de procesamiento locales a cada nodo y se sincronizan los procesos [GLS99].

Históricamente, PVM (*Parallel Virtual Machine*) ha sido la primera implementación de paso de mensaje difundida de este tipo de multicomputadoras. Con el tiempo, tomó mas popularidad MPI, acrónimo en inglés de *Message Passing Interface*, un protocolo de comunicaciones que implementa el modelo, y que es definido por un foro de notables de la industria [Forb]. En la actualidad, MPI se ha convertido en un estándar *de facto* del modelo de paso de mensajes, principalmente debido a que existen varias implementaciones del protocolo (mayormente *open-source*), y a que dispone de un conjunto de opciones de mensajes más amplio que PVM, lo que brindan más flexibilidad al programador [GKP96].

En cuanto a la tecnología utilizada para conformar la red, lo más usual es que se utilice la tecnología *Ethernet* para interconectar los nodos. El uso de este tipo de redes está muy difundido dado su bajo costo, existiendo una larga trayectoria en su uso. Sin embargo, a los fines de conformar un *cluster*, el alto tiempo de latencia y el bajo ancho de banda que disponen, genera un límite en cuanto a la capacidad del multiprocesador global que conformen. A pesar del avance en cuanto al ancho de banda que esta tecnología está logrando en los últimos tiempos, su alta latencia sigue siendo un obstáculo.

Existen otras tecnologías de red que sobresalen dadas las que mejores prestaciones brindan, gracias a mayores anchos de banda y menores latencias, las cuales son Infiniband y Myrinet [Pfi01, Myr]. El mayor rendimiento viene de la mano de mayores costos, que, si bien son superiores a las redes *Ethernet*, no son tan elevados. Infiniband está entre las más usadas dentro de los supercomputadores más potentes del mundo [org]. Debe tenerse en cuenta el impacto de la arquitectura interna de la red utilizada a la hora de diseñar los algoritmos paralelos, ya que las redes *Ethernet* están basadas en un bus común entre todas las interfaces de red, mientras que Infiniband establece conexiones punto a punto entre los dispositivos de conectividad que conforman la red [Gru10].

2.2.2. Programación de aplicaciones bajo *cluster*

El modelo de ejecución en paralelo en este tipo de multicomputador está basado en que cada nodo del *cluster* ejecuta un mismo programa, pero con un conjunto de datos diferente, es decir, hace paralelismo de datos. Si bien MPI permite que se ejecuten en paralelo programas distintos en cada nodo, lo usual es que esto no sea así, y a lo sumo, el mismo programa tenga distintas ramas de ejecución, acorde al tipo de rol que el programa asigne a un nodo o según los valores que presenten los datos.

El mecanismo que posibilita el intercambio de información dentro de un *cluster* es el envío de mensajes entre los nodos, existiendo dentro de MPI, numerosas primitivas que implementan las diversas modalidades que pueden asumir los mensajes. Siempre existe la necesidad de que específicamente esté programado en el proceso que actúa como emisor, una invocación a un procedimiento de envío, y en el/los procesos receptores, la respectiva invocación a un procedimiento de recepción. Si no existe esa simetría, el proceso global no avanza o se genera un error [1].

Una primera clasificación que se puede dar para los mensajes entre procesos es que sean de tipo colectivo o punto a punto. En el primer caso, los emisores o receptores del mensaje son todos los procesos que conforman el *cluster*, pudiendo ser de orden 1-a-n, n-a-1 o bien de n-a-n. Los mensajes de tipo punto a punto se dan exclusivamente entre dos procesos, un emisor y un receptor determinado.

Otro criterio de clasificación de los mensajes es que sean bloqueantes o no bloqueantes. Para los primeros, las rutinas de comunicación no devuelven el control de ejecución al programa hasta que la comunicación ha concluido, lo que es la semántica habitual en el llamado de funciones o rutinas en los lenguajes de programación, lo que traducido en nuestro entorno, determina que el emisor tiene la confirmación del receptor de haber recibido todos los datos enviados. En cambio, los mensajes no bloqueantes permiten que el programa siga ejecutándose en paralelo con el envío de los datos ya que el llamado a la rutina de comunicaciones devuelve el control al programa principal sin necesidad de que las comunicaciones hayan finalizado.

Una tercera categoría de mensajes la componen los mensajes destinados a la sincronización de los procesos que se ejecutan en cada nodo. El mensaje

[1] Esto es lo que se denomina *two side communications*. En la definición de MPI-2 también se especifican las *one side communications*, donde se elimina la necesidad explícita de un emisor y un receptor por mensaje.

de barrera (*barrier*), es un ejemplo de mensaje de sincronización colectivo que impide el progreso del procesamiento en un nodo, hasta que todos los restantes nodos alcancen ese punto. Dentro de esta categoría también figuran los mensajes de sincronización de las llamadas no bloqueantes, ya que es necesario disponer de mensajes que permitan confirmar la finalización de las comunicaciones pendientes.

El diseño de los programas bajo este modelo de ejecución tiene que tener en cuenta la distribución de los datos sobre los cuales se realizará el procesamiento, tratando de minimizar las comunicaciones necesarias, ya que éstas se convierten en un cuello de botella para el procesamiento global, al ser varios órdenes de magnitud más lentas que el procesamiento en la CPU. Tener en cuenta que, si bien la replicación (copia) de parte o todos los datos en todos los nodos minimiza las comunicaciones, esto atenta contra la escalabilidad de los algoritmos, ya que los datos replicados ocupan memoria, por lo que esta práctica no es recomendable.

Otro factor a tener en cuenta en el diseño de los algoritmos paralelos bajo esta arquitectura es la tecnología de red que conforma el *cluster*. Es evidente que si se desea optimizar procesamiento, se debe utilizar el *hardware* de la forma en que fue diseñado para utilizarse mejor, lo cual es un punto importante a tener en cuenta para el diseño del algoritmo. Como se dijo anteriormente, entre las tecnologías usuales para la constitución de la red que conforma el *cluster*, hay una distinción respecto del tipo de conexiones: punto a punto o colectivas. Esto plantea ventajas o desventajas dependiendo del caso: en el caso de punto a punto, se deben utilizar varios envíos punto a punto para resolver un mensaje colectivo; para el caso arquitectura con colectivas, un mensaje punto a punto genera una captura del canal común de comunicaciones para ser utilizado por solo dos nodos. El diseño del algoritmo debe tener en cuenta este factor a la hora de establecer las comunicaciones entre los nodos, por que, como se verá más adelante en esta tesis, comunicaciones colectivas en un *hardware* punto a punto, generan rendimientos inferiores al uso de comunicaciones punto a punto.

2.3. Computadoras *multicore*

2.3.1. Arquitectura de los *multicore*

El avance en el poder de cómputo de los procesadores hasta hace unos años se basaba fundamentalmente en el aumento de la velocidad de funciona-

miento del procesador. El límite físico del silicio determinó que esta forma de aumentar el poder de cómputo se viese frenada, por lo que las fábricas de procesadores tuvieron que optar por otros mecanismos para poder mantenerse en la carrera del aumento de la capacidad de procesamiento.

La principal estrategia en la actualidad para este objetivo es la de duplicar / multiplicar los procesadores internos que componen un mismo circuito integrado. Hoy en día resulta difícil conseguir procesadores que no estén compuestos por al menos dos procesadores, los *dual core*, y la tendencia es que crezca el número de procesadores componentes de un *chip*, lo cual genera la necesidad de programar en paralelo para poder utilizar en plenitud la potencialidad de cómputo del integrado.

La arquitectura de estos microprocesadores es tal que cada uno de estos puede actuar en forma independiente en cuanto a los procesos que se ejecuten, ya que tienen su propias unidades de procesamiento, punto flotante, *pipelines*, etc, incluso cada *core* dispone generalmente de su propia memoria *cache*, pero en lo referido a la memoria principal del equipo, es de tipo compartida, por lo que toda la RAM es accesible desde cualquiera de los procesadores integrantes del CPU. Es por esto que este tipo de computadora multiprocesadora se denomina como de "Memoria Compartida".

2.3.2. Programación de aplicaciones en *multicore*

Una de las formas de implementar la programación de un sistema con memoria compartida es por medio de la utilización de *threads*. Los *threads* son hilos de ejecución, es decir, una serie de instrucciones ejecutables que corren en forma independiente y paralela, que son administradas por el sistema operativo formando parte de un mismo proceso, por lo que disponen de un área de recursos comunes entre ellas. Los *threads* son lanzados desde un proceso principal y son ejecutados en paralelo junto con los restantes hilos. Disponen de un área de memoria exclusiva, o privada de cada hilo, y de un área de memoria común entre todos los hilos, la cual actúa como mecanismo de comunicación entre los *threads*. Este hecho a su vez impone la existencia de mecanismos de exclusión para evitar un manejo indeseable de las zonas comunes por varios *threads* en forma simultánea.

Históricamente, debido a que cada fabricante o programador de compiladores implementaba un modelo particular de *threads*, era complicado portar una aplicación programada bajo un entorno hacia otro. La falta de normativa motivó que en la década de 1990 se creara un grupo de estandarización del

tema, del cual nace el estándar de las POSIX *threads* o más comúnmente conocidas ***Pthreads***. El lanzamiento de nuevos hilos es por medio del llamado a una subrutina interna del programa que da lugar a la creación de tantos hilos como previamente se haya pre-establecido. La implementación de las *Pthreads* es por medio de una biblioteca de funciones que permiten crear, manipular, sincronizar y destruir los hilos, el paso de parámetros y la creación de variables con mecanismos de exclusión mutua. Dicha biblioteca generalmente debe ser enlazada en el proceso de compilación del programa ejecutable. Esta forma de implementar el modelo, está casi exclusivamente restringida al lenguaje C/C++, con escasas implementaciones para Fortran [But97].

Debido a que las *pthreads* imponen un manejo detallado del paralelismo, son relativamente complicadas de programar. En virtud de esto es que surge **OpenMP** como implementación alternativa del modelo de memoria compartida. Está basado en un modelo de hilos, los cuales se definen a partir de directivas de compilación, y al generarse el código, el compilador introduce las instrucciones referidas a la generación y administración de los hilos, lo cual evita las llamadas directas a rutinas *Pthreads* que debía hacer el programador anteriormente. Las directivas OpenMP son insertadas en el código secuencial facilitando la programación en paralelo bajo este modelo [CDK+00, CJP07].

Las principales directivas de OpenMP están destinadas a la definición de regiones de código que pueden ejecutarse en paralelo, siendo la más usual, la paralelización de un bucle, lo cual debe hacerse de manera tal que los datos que se utilicen en un hilo no se solapen con datos a usar en otro para poder obtener independencia entre ellos y alcanzar mejores rendimientos. Existen también directivas para posibilitar el uso exclusivo o compartido de variables en cada hilo. Otra directiva permite que el usuario defina el número de hilos en paralelo a ejecutarse, pudiendo definirse un número mayor al número de *cores* disponibles en el equipo, lo cual, cuando los hilos son CPU intensivos, no genera ninguna ventaja dado que los hilos empiezan a competir entre ellos por el uso de los recursos físicos existentes [CDK+00, CJP07].

Es evidente que en este modelo de paralelismo, el envío de mensajes entre hilos no tiene sentido, ya que la existencia de un área de memoria común permite el intercambio de datos entre hilos de forma más eficiente, por lo que el modelo de ejecución en paralelo tiene muy poco en común con el modelo de envío de mensajes de memoria distribuida. Son dos modelos diferentes sin puntos en común en lo referente su programación, por lo que un programa escrito para correr bajo memoria compartida no puede ejecutarse, en paralelo, en un *cluster*. Existen intentos de utilizar memoria distribuida como si fueran

áreas de memoria compartidas, encarnados por proyectos como Co-Array Fortran [Fora], pero esto no elimina la necesidad de replicar de datos y el envío de mensajes para la comunicación de los datos compartidos.

2.4. Programación de *clusters* de nodos *multicore*

En la actualidad, es un hecho que los *clusters* están conformados por nodos *multicore*, por lo que el modelo de paralelismo imperante es una combinación de memoria compartida - memoria distribuida, y para poder aprovechar al máximo sus potencialidades, debe utilizarse un híbrido de paralelismo de memoria compartida, programado vía OpenMP, y de memoria distribuida, programado con MPI [Smi00, Qui04].

Es posible utilizar cada uno de los *cores* de un equipo *multicore* como si fuera una computadora independiente y aplicar solo el modelo de memoria distribuida entre estos, es decir, dentro de una misma computadora coexistir tantos procesos independientes entre sí como *cores* haya disponibles, intercambiando resultados entre sí por medio del envío de mensajes entre éstos. De hecho, muchos programas preparados para correr en paralelo solo utilizan esta estrategia de paralelismo, y el usuario no experto en programación solo puede utilizarlo "como está".

Si bien esta forma de usar los *cores* disponibles es factible, hay un desperdicio de recursos, ya que el envío de mensajes entre procesos dentro de una computadora implica el uso de un mecanismo más complejo que el hecho de compartir un área de memoria. A los fines ejemplificativos, el mensaje tiene como mínimo un destinatario dentro del *cluster* identificado por su nombre de nodo. Dicho nombre es mapeado hacia un identificador de la interfase de red del destinatario, en este caso un *loopback* si use usa TCP/IP, o algo similar para las otras tecnologías de red. Si el mensaje es bloqueante, tanto emisor como receptor deben acordar la finalización de la comunicación, lo cual genera un tiempo inactivo por sincronización. Es evidente la sobrecarga en relación a disponer de variables compartidas.

Utilizar cada *core* para un proceso MPI, además del mal uso del tiempo del procesador, también genera un mal uso de la memoria. Como el espacio de memoria de cada proceso MPI es independiente de los restantes, si un equipo tiene disponibles 4 *cores* y lanza 4 procesos MPI, el máximo espacio de memoria disponible que tendrá cada proceso, eludiendo el *cache* sobre

disco, será de un cuarto de la memoria del equipo. Esto no sucede utilizando *threads* ya que disponen de un área común de memoria.

Por otro lado, Co-Array Fortran es un intento para utilizar el paralelismo de memoria compartida sobre un *cluster* de memoria distribuida [Fora]. El paralelismo está basado en distribuir arreglos de Fortran entre los nodos componentes del *cluster*. Tiene un fundamento básico: programar con memoria compartida es más simple que programar con memoria distribuida. Se soluciona enmascarando la distribución de los datos como si fuera en memoria compartida. Esta alternativa de paralelización también trae aparejado un desperdicio de recursos. Memoria compartida en un *cluster* generalmente implica una replicación de los datos en los nodos participantes, y un mecanismo de mensajes para la actualización de los valores compartidos. Hay ciertas libertades por la implementación utilizada para dicho mecanismo que plantean dudas. ¿Los mensajes son punto a punto o colectivos?. ¿Cuándo se genera la sincronización de los procesos? ¿El tipo de red subyacente es tenido en cuenta? Como éstos, hay muchos otros interrogantes que quedan librados a la implementación del lenguaje. Es evidente que una solución de tipo general siempre atenta contra la optimización en el uso de los recursos.

Es por estas razones que la forma aconsejable de aprovechar el paralelismo que brinda un *cluster* de nodos *multicore*, es programar haciendo un *mix* de instrucciones OpenMP y MPI, lo cual es más complejo, pero más productivo a nivel de rendimiento. Además de la complejidad en la codificación por tener que utilizar dos modelos de paralelismo con sus características sintácticas particulares, también es necesario hacer una división de tareas y datos en cada uno de los modelos de paralelismo, en dos niveles y combinarlos para obtener los rendimientos esperados.

Para utilizar conjuntamente ambos modelos de memoria, pueden plantearse *a priori* utilizar OpenMP dentro de MPI o a la inversa, lo cual se refleja en utilizar n hilos con un proceso MPI dentro de cada uno de los hilos, o a la inversa, un proceso MPI con n hilos paralelos corriendo en su seno. Si bien ambas opciones son técnicamente válidas, la última alternativa es la más recomendable por lo que señalamos anteriormente respecto de cómo se utiliza la memoria en los procesos MPI, y es la forma en que se realizaron todos los experimentos a lo largo de la tesis.

Para describir las ventajas del modelo híbrido de programación en un *cluster* de nodos *multicore*, es frecuente utilizar los conceptos de grano fino y grano grueso [GKKG03]. La granularidad está definida como la relación cómputo / comunicación, y se dice de grano grueso, cuando hay largos perío-

dos de cómputo intercalados con pocas comunicaciones y de grano fino, cuando hay muchas comunicaciones intermedias al cómputo. En general, grano grueso se asocia con una división general de grandes volúmenes de datos, y por grano fino se entiende una división más detallada o profunda de los datos, de volumen menor, ya que con grandes volúmenes de datos se supone que se pueden realizar grandes cantidades de cómputos.

Numerosos autores citan las ventajas y desventajas de la programación bajo el modelo híbrido MPI/OpenMP [SB00, Smi00, AMR+05]. Entre las ventajas encontramos:

- Mejoras de *performance* en código que escale pobremente bajo el modelo MPI puro. En casos en que el área paralelizable sea de *grano fino*, el modelo MPI tiene una pobre *performance* potencial debido a la sobrecarga de comunicación. Un modelo híbrido permite distribuir mejor el cómputo aprovechando la paralelización de *grano grueso* bajo el modelo MPI y la de *grano fino* en el modelo OpenMP.

- Código que usa datos replicados: OpenMP puede utilizar una única copia de datos en cada *thread* que se genere. MPI necesita su propia versión de los datos en cada proceso abierto.

- Problemas de balance de carga. Si la descomposición en paralelo no genera hilos de ejecución con cargas homogéneas de cómputo, se genera un problema de balance carga. Este desbalance, bajo el modelo de MPI genera serios problemas de comunicaciones y de grado de utilización de los nodos cuando se utiliza poca carga de cómputo, dada esto por un tamaño de datos pequeño o bien por la utilización de un algoritmo simple. Los hilos OpenMP podrían absorber estos tareas pequeñas y mejorar el balance.

Como desventajas, estos autores indican los casos en que no estén presentes las condiciones en que es ventajosa la descomposición de *grano fino* o de *grano grueso*. Si no hay condiciones para que existan ventajas de rendimiento en la utilización de alguno de los dos modelos, el modelo híbrido en conjunto no ayuda.

En general se puede ver que las condiciones estructurales del código y los datos son las que dictan el modelo a utilizar. Si es posible descomponer en *grano fino* y en *grano grueso*, aprovechando simultáneamente las ventajas de ambas, el modelo híbrido es el aconsejable. De lo contrario, utilizar el modelo afín a la descomposición que sea factible realizar. Una limitante a

este lineamiento está dada por el volumen de datos a procesar: si el modelo recomendado es el de memoria compartida, pero el procesamiento necesario demora días o semanas para correr en una sola computadora *multicore*, se deberá utilizar el modelo híbrido en un *cluster*, y tratar de dividir datos y tareas de forma tal que la sobrecarga comunicacional impacte lo menos posible en el rendimiento.

En otro orden de cosas, los lenguajes usuales para este tipo de programas son Fortran y C/C++. Son usuales los compiladores de ambos lenguajes que implementan directivas de compilación de OpenMP y hay desarrolladas varias bibliotecas de rutinas que implementan MPI para ambos lenguajes. Como los rendimientos son la razón de ser del paralelismo, la utilización de lenguajes de más alto nivel que los citados no es usual, ya que, si bien hay implementaciones de paralelismo para el modelo compartido o el modelo distribuido en Java, Pyhton o Perl, la diferencia de rendimientos respecto de C/C++ y Fortran es muy significativa, por lo que no dejan de ser implementaciones académicas. La realidad indica que el código paralelo eficiente, es mayormente escrito en Fortran o en C/C++.

2.5. Resumen del capítulo

En este capítulo se ha presentado una breve descripción del sistema de computación paralelo definido como *Cluster* de nodos *multicore*. Se expuso un resumen de su composición a nivel de *hardware*, tanto de computadoras como de red, y se enunciaron las opciones más difundidas y las características de los mecanismos de programación. Se concluyó que el modelo híbrido de programación MPI - OpenMP es el más adecuado para la programación de este tipo de arquitectura para obtener los máximos resultados posibles de rendimiento y se delinearon los factores que determinan cuándo utilizar cada una de dichas tecnologías.

Capítulo 3

Patrones de paralelismo

3.1. Evolución y estado actual de los patrones de paralelismo

El surgimiento de los patrones, como un instrumento que presenta una solución a un tipo determinado de problemas, comienza a tomar difusión con el libro de Alexander et al. [AIS77], donde los autores plantean a partir del análisis de problemas de diseño habituales en el campo de la arquitectura, soluciones genéricas a cada tipo de problema, de forma tal que quien tenga que implementar una solución, pueda basarse en lo aconsejado por el patrón respectivo, pudiendo adaptarse a las situaciones particulares que el problema concreto plantee.

El concepto de patrón que los autores plantean es:

> *"Each pattern describes a problem which occurs over and over in our environment and then describes the core of the solution to that problem in such a way that you can use the solution a million times over without ever doing it the same way twice."*

Se pone entonces énfasis en que la solución planteada no es propia a una situación en particular, sino que al contrario, la describe como una solución genérica, que puede aplicarse con distintos matices según las particularidades de cada caso.

Este concepto de patrón es posteriormente tomado por Gamma et al. en su ya famoso libro "Design Patterns" [GHJV95], donde es trasladado hacia

el ámbito del diseño de software, apoyándose fuertemente en el paradigma de orientación a objetos. Bajo esta óptica es que se plantean soluciones de diseño de software en términos de clases y objetos intervinientes, definición de roles y relaciones entre los objetos (colaboraciones y responsabilidades).

En dicho libro se plantean una serie de patrones de diseño que son clasificados bajo las categorías de: a) patrones creacionales, aquellos relativos a la creación de objetos, b) patrones estructurales, relativos a composición de las jerarquías de clases y objetos, y finalmente, c) patrones de comportamiento, que describen interrelaciones entre objetos.

Si bien no existe un consenso general al respecto de cómo debe estructurarse un patrón, en general está compuesto por cuatro clases de elementos escenciales, a saber:

- El nombre, que debe ser representativo del tipo de problema considerado
- El problema, que es una descripción de los factores que lo conforman su contexto, cuando y donde ocurre y los elementos que intervienen en el planteamiento de su definición.
- La solución, como una serie de elementos de diseño que deben considerarse y sus relaciones. No se describe una solución particular, sino una plantilla que puede ser utilizada en diferentes situaciones.
- Las consecuencias e impactos de la implementación de la solución planteada y donde poner el foco para evitar efectos indeseables de la aplicación.

Dentro de cada una de estas secciones, un patrón puede contener varios ítems, como es el caso de la descripción del problema, donde existen ítems como la intención, la motivación y la aplicabilidad. Estos ítems suelen estar presentes en la presentación de patrones, pero en ocasiones, dependiendo del tipo de patrón en consideración, pueden refundirse para presentarse en conjunto bajo un único elemento descriptor del problema. Una guía respecto de la estructura formal que debe respetar el desarrollo de un patrón puede verse en Meszaros y Doble [MD96].

Los patrones de diseño tuvieron una aceptación tan generalizada dentro del dominio del diseño de software, que pronto la idea se extendió hacia algunas otras áreas del dominio informático, tal como ocurrió con el desarrollo de patrones de comunicaciones, patrones de seguridad, patrones de aplicaciones

empresariales, entre otros, y el área de la programación en paralelo tampoco quedaría atrás con el desarrollo de patrones de paralelismo.

Entre los primeros intentos de patrones para el área del paralelismo surgieron las trabajos de Sing et al. [SSS96] y de Siu et al. [SSGS96]. Sing presenta las primeras experiencias en la aplicación del modelo de programación basado en *templates* dentro del entorno paralelo, mientras que el segundo de los trabajos describe el concepto de patrón de diseño para el desarrollo de aplicaciones paralelas. Este último trabajo introduce la utilización de patrones de diseño para la agilización del proceso de desarrollo de programas paralelos proponiendo mecanismos generales de sincronización y comunicaciones, sin embargo, la propuesta va más allá del concepto de patrón de paralelismo ya que la finalidad es el desarrollo de una biblioteca de funciones que implemente los patrones propuestos.

Otro importante paso en el desarrollo de patrones de paralelismo lo constituyen los trabajos de investigadores de la universidad de Alberta [SSJ+02, SDJ+02], quienes profundizan el estudio de la aplicabilidad de patrones de diseño en el ámbito de la programación en paralelo, a través de conceptos *frameworks* [FSJ99] y de programación generativa [CE00] como herramientas para el desarrollo de programas paralelos. El fruto de estas investigaciones fue la creación de un entorno de trabajo para el desarrollo de aplicaciones paralelas llamado CO_2P_3S. Si bien la herramienta no ha tenido una difusión exitosa, fueron de los primeros en dividir el problema de paralelización de una aplicación secuencial en distintos niveles o capas lógicas, entre nivel de diseño del programa paralelo, nivel de estructura del algoritmo paralelo, y nivel de implementación de los algoritmos para alcanzar mejor rendimiento.

Por otro lado, B. Massingill, T. Mattson y B. Sanders trabajan desde el año 2000 en el desarrollo, en términos de los propios autores, de un lenguaje de patrones de paralelismo, visto como una colección estructurada de patrones que permite guiar al programador en el desarrollo de un programa paralelo. En uno de sus primeros trabajos [MMS00] introducen una división conceptual para el desarrollo de un programa paralelo en cuatro capas, o espacios según los autores, los cuales son:

1. El espacio de "Encontrar la Concurrencia", donde se describen los patrones de mayor nivel de abstracción y que están orientados a resolver la problemática de la concurrencia entre las tareas que conforman el algoritmo.

2. El espacio de "Estructura de Algoritmos", destinado definir los lineamientos generales de los algoritmos que exploten la concurrencia hallada anteriormente.

3. El espacio de "Soporte de Estructuras" de algoritmos, donde se describen las estructuras de datos y de pseudo-instrucciones del algoritmo, que dan los lineamientos de la implementación de los algoritmos.

4. El espacio de "Mecanismos de Implementación", que contiene patrones destinados a describir diversas implementaciones de programación a bajo nivel, a ser utilizadas por los algoritmos.

Esta clasificación de niveles conceptuales sigue una lógica de diseño desde un nivel de abstracción mayor hacia un nivel de implementación cercano al *hardware*. En el proceso de desarrollo de un programa paralelo, un programador debería empezar definiendo en primer lugar la división de tareas y datos a partir del análisis de la concurrencia existente en el problema, para luego definir la estructura del algoritmo paralelo, las estructuras de datos y cómputos intervinientes y finalmente, la implementación. En el trabajo citado arriba ([MMS00]), se presentan patrones a nivel del espacio de concurrencia solamente, entre los que se encuentran patrones destinados a la descomposición de tareas, distribución de datos y la reevaluación del diseño.

El trabajo de estos tres autores se completa en el libro "*Patterns for parallel programming*" [MSM04], donde hacen una presentación completa del lenguaje de patrones manteniendo la clasificación referida. A nivel del espacio de concurrencia, presentan patrones como *Task Decomposition* y *Data Decomposition*, patrones para el análisis de dependencias: *Group Task* y *Order Task Pattern* y un patrón destinado a la evaluación del diseño, *Design Evaluation Pattern*. A nivel del espacio de estructura del algoritmo, se presentan *Divide and Conquer*, *Recursive Data* y *Pipeline Pattern* entre otros. Entre los patrones de estructuras de soporte se presentan *Master/Worker*, *Loop Parallelism* y *Fork/Join*. Finalmente, a nivel de implementación, más que patrones, se presentan mecanismos de implementación de comunicaciones, sincronización, creación de procesos paralelos, etc, sugiriendo implementaciones en los lenguajes de programación habituales en el dominio, es decir, C, Fortran y también en Java.

Otro grupo de investigación de la materia en cuestión es el *Parallel Computing Laboratory* (Parlab), laboratorio de computación en paralelo, de la universidad de California, Berkeley, el cual dispone de un sitio web donde se presenta la visión y el desarrollo sobre los patrones de paralelismo que sostiene dicho grupo [PAR]. El tema es introducido en términos similares a los de

Massigill et al., en el sentido de disponer de un lenguaje de patrones que presentan una estructura jerárquica según un criterio de refinamiento del diseño de un programa paralelo, según lo señalan en el documento de *Our Pattern Language* [KM09]. La jerarquía de patrones parte desde el nivel de patrones para aplicaciones (subdividida en patrones estructurales para aplicaciones y patrones computacionales), siguiendo por patrones de estrategias de algoritmos paralelos, los patrones de estrategias de implementación, finalizando con los patrones de implementación de ejecución paralelo.

En general, los patrones que este grupo presentan son preexistentes, aunque existen algunos originales, como por ejemplo el *Dense Linear Algebra Pattern*. Dicho lenguaje de patrones no está aún concluido, ya que para algunos de los patrones solo se presenta el titulo del patrón, como un casillero a ser completado.

La Universidad de Illinois - Urbana Champaign, también presenta un catálogo de patrones de paralelismo a través de su centro UPCRC (*Universal parallel computing research center*) [UPC]. Está muy relacionado con el catálogo de Berkeley, a tal punto que muchos de sus enlaces están redirigidos hacia el sitio de esta última. Se reconoce que la única diferencia entre ambos centros de investigación es que el grupo de Berkeley pone más énfasis hacia los patrones de más alto nivel, mientras que los investigadores del UPCRC se enfocan sobre patrones de nivel inferiores, es decir cercanos a la implementación.

Perteneciente al centro UPCRC, el profesor Marc Snir expone en su página personal una serie de patrones de paralelismo [Snia], donde se presentan clasificados no ya en una jerarquía de niveles de diseño, sino acorde a su funcionalidad, como lo son las categorías de *Pattern Language*, *frameworks* de paralelismo, modelos climáticos, modelos de programación paralela, etc.

En general podemos señalar que el dominio de los patrones de paralelismo está aún en fase de desarrollo. No son muchos los centros de investigación que se dedican a la materia, y no hay un consenso acabado en cuanto a la clasificación que éstos deben presentar. En el mismo sentido, hay un cierto reconocimiento a la necesidad de desarrollar el área, ya que en los catálogos arriba referidos, existen numerosos patrones presentados solo con su nombre, que todavía faltan ser desarrollarlos. Al igual que en otras áreas de patrones, anualmente se realizan *workshops* sobre patrones de paralelismo, las ParaPLoP, con versiones 2009 y 2010. Todo esto permite visualizar lo abierto que permanece el tema. Como será expuesto más adelante en este capítulo, si se considera que las aplicaciones de cómputo científico son una parte de las

aplicaciones paralelas con sus particularidades y que éstas deben ser tenidas en cuenta a la hora de definir un patrón de paralelismo, la carencia es aun mayor.

3.2. Algunos Patrones de Paralelismo

Luego del raconto del estado actual de los patrones de paralelismo expuesto en la sección anterior, en esta sección se presentarán algunos de los patrones de paralelismo actualmente existentes. No se hará una presentación completa y acabada de los mismos, ya que esto excede el objetivo del trabajo, por lo que solo que se presentarán algunos de los más relevantes. La selección de los patrones presentados aquí se hizo considerando su aplicación en los experimentos que se presentan en el Capítulo 4, referido a la aplicación de patrones de paralelismo en algoritmos de álgebra lineal y su impacto en el rendimiento. Un estudio completo de la materia debería incluir los sitios de catálogos de patrones de la Universidad de California - Berkeley [PAR] y de Illinois - Urbana Champaign [UPC], el sitio personal de Marc Snir [Snia] y el libro de Massingill et al. [MSM04].

En honor a la brevedad, y al alcance dado al presente trabajo, la presentación de cada patrón no se hará en forma completa, sino los puntos más destacables de cada uno de ellos, manteniendo el formalismo en la exposición que originalmente presentan y la referencia a su fuente. No se expondrá la sección de ejemplos de cada patrón por lo extensa que ésta suele ser, pudiendo ser consultada en la versión original del mismo. El resto de la página es intencionalmente dejada en blanco, dada la estructura de presentación de cada patrón.

3.2.1. Patrones de división y reagrupación de tareas

3.2.1.1. Task Decomposition Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

¿Cómo descomponer un problema en tareas que pueden ser ejecutadas en forma concurrente?

Contexto

Una vez que el programador ha comprendido la naturaleza del problema a resolver y ha determinado cual es la parte computacionalmente intensiva del mismo, se deben determinar las tareas que lo componen. En algunos casos, puede comenzarse por definir una serie de tareas relativamente independientes. En otros, es difícil separar la división de tareas de la división de datos, y éste se convierte en el punto inicial.

Fuerzas

- Flexibilidad, para adaptarse a implementaciones diferentes.
- Eficiencia, para alcanzar mejores rendimientos.
- Simplicidad, para poder programar en paralelo con cierta facilidad.

Solución

El criterio principal para descomponer tareas debe ser determinar tareas relativamente independientes que impliquen una mínima carga de administración de las dependencias. Deben determinarse una serie de tareas diferentes que ayuden a solucionar el problema y que sean lo más independientes posibles

---

3.2.1.2. Group Task Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

Una vez divididas las tareas, ¿Cómo reagruparlas para simplificar el manejo de las dependencias?

Contexto

El patrón es aplicable luego de realizar la división de tareas y de datos. Es el primer paso en el análisis de dependencias entre tareas. Si una de las tareas del grupo debe respetar una restricción temporal, dicha restricción debe ser respetada por todas las tareas del grupo. Si un grupo de tareas deben trabajar sobre un conjunto de datos, es necesaria una sincronización para que todas trabajen como una unidad.

Solución

Las dependencias entre tareas pueden agruparse en:

- Dependencia temporal, B depende del resultado de A, entonces B solo puede realizarse una vez finalizada A.
- Tareas simultáneas: un conjunto de tareas deben realizarse simultáneamente para evitar dependencia cíclicas entre sí.
- Independencia, es decir, un conjunto de tareas que pueden lanzarse en cualquier orden ya que no se afectan entre sí.

El reagrupamiento de tareas debe ser tal que simplifique el análisis de dependencias, por ejemplo, aquellas tareas que tienen iguales dependencias pueden agruparse entre sí a los fines de tratar una única dependencia.

3.2.2. Patrones de distribución de datos

3.2.2.1. Data Decomposition Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

¿Cómo dividir los datos en unidades de forma tal que puedan ser procesadas en forma relativamente independiente?

Contexto

Una vez que el programador ha comprendido la naturaleza del problema a resolver y ha determinado cual es la parte computacionalmente intensiva del mismo, ¿Cómo se usan los datos a medida que la solución del problema avanza y se va refinando? Definir si, ¿Empezar descomponiendo por datos o descomponiendo por tareas? Una descomposición inicial por datos es beneficiosa si el cómputo intensivo surge de un gran volumen de datos, o si se aplican operaciones similares a diferentes partes de datos.

Fuerzas

- Flexibilidad, para adaptarse a implementaciones diferentes.
- Eficiencia, para alcanzar mejores rendimientos.
- Simplicidad, para poder programar en paralelo con cierta facilidad.

Solución

Si la división por tareas ya se ha realizado, la división de datos deviene de los requerimientos de las tareas. Si dividir datos es el primer paso, se debe considerar las estructuras de datos principales y como pueden dividirse para operar concurrentemente. Ejemplos son las computaciones sobre arreglos o sobre estructuras recursivas de datos.

La flexibilidad se logra a partir de poder definir la cantidad de datos que cada partición tendrá desde parámetros que las determinen. Para lograr eficiencia es importante que la granularidad no sea tan pequeña que la administración de dependencias de datos genere una sobrecarga que reste capacidad de cómputo.

3.2.2.2. Data Sharing Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

Dada una descomposición de datos ¿Cómo compartir los datos entre diversas tareas?

Contexto

Una vez realizado el análisis de dependencias y el respectivo agrupamiento de tareas, el siguiente paso es definir cómo los diversos grupos de tareas pueden compartir datos entre sí. Una vez divididos los datos, cada procesador trabaja con un conjunto de datos, llamados datos locales. Sin embargo puede suceder que parte o la totalidad de éstos datos locales sean necesarios compartir con otras unidades de procesamiento. También suele ser necesario compartir datos que conforman una dependencia: una vez realizada una tarea, su resultante es necesario por otros procesadores.

Fuerzas

Compartir datos tiene impacto sobre la correctitud del cómputo y la eficiencia.

- Debe tenerse en cuenta que la versión del dato a compartir debe ser la apropiada a la etapa del cómputo donde se comparte, para evitar errores en los resultados finales.
- Garantizar que un dato compartido esté disponible puede generar una sobrecarga de sincronización excesiva.
- Tener en cuenta el costo comunicacional al compartir un dato.

El objetivo entonces es permitir compartir un dato minimizando el impacto en la eficiencia.

Solución

Primero: identificar el dato necesario por varias tareas. Esto solo puede lograrse una vez divididos los datos. Luego, determinar la naturaleza del uso el dato:

- solo lectura, en cuyo caso cada tarea puede tener su copia local, es decir, usar replicación.
- lectura - escritura hecha por varias tareas, que pueden a su vez clasificarse como:
    - acumulaciones: el dato compartido es modificado concurrentemente por varias tareas, lo cual implica el uso de alguna técnica de compartición de recursos, como ser, la técnica de "semáforos".
    - múltiple lectura - una escritura: dos copias son requeridas, la primera para preservar el valor inicial y la segunda para actualizar. Cuando la actualización es firme, el valor inicial es descartado y reemplazado por el actualizado.

3.2.3. Otros patrones

3.2.3.1. Design Evaluation Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

La descomposición de datos y tareas y el análisis de dependencias ya están realizados, entonces, ¿Se puede avanzar a la próxima etapa o es necesario reevaluarlos?

Contexto

Ya se han determinado las tareas que se realizan concurrentemente, también el análisis de dependencias, cómo se distribuyen los datos y cómo se reagrupan las tareas, habiendo aplicado *Task Decomposition, Data Decomposition, Group Task* y *Data Sharing Patterns*. Haberlo hecho correctamente influirá en la calidad del diseño paralelo, por lo que reevaluar lo hecho garantizará el desarrollo posterior.

Fuerzas

El diseño debe reevaluarse desde tres dimensiones:

- Pertinencia sobre la plataforma a utilizar.
- Cualidades de diseño: simplicidad, flexibilidad, eficiencia.
- ¿Conexión con la próxima fase de diseño: la interacción entre tareas es sincrónica o asincrónica? ¿Las dependencias son regulares o irregulares?.

Solución

Evaluar el trabajo del diseño desde la perspectiva de las tres dimensiones anteriores:

- Plataforma: tener aunque sea una mínima contemplación sobre la plataforma a utilizar.
    - Contemplar el balance de carga si las tareas definidas son diferentes.

- Contemplar la cantidad de procesadores a los fines de dividir los datos: si la división puede hacerse con relativa libertad, ajustar éstas a la cantidad de procesadores (caso de la multiplicación de matrices sobre memoria distribuida).
- Si hay un volumen grande de datos compartidos, utilizar un modelo de memoria compartida ya que de lo contrario, sobre memoria distribuida, puede generarse una sobrecarga de comunicaciones.
- Tener en cuenta la relación entre tiempo de cómputo y tiempo de comunicaciones y sincronización

- Diseño:
  - Flexibilidad: diseño adaptable a diversas plataformas y tamaño de la máquina paralela:
    - ¿Es flexible el número de tareas generado?
    - ¿El número de tareas es independiente de la política de planificación de lanzamiento de tareas (*scheduling*)?
    - ¿El número de divisiones de los datos puede ser parametrizado?
  - Eficiencia:
    - ¿La carga de trabajo puede ser balanceada en caso de ser necesario?
    - ¿El número de comunicaciones en un sistema distribuido es el mínimo? ¿Puede reducirse aumentando el tamaño de los datos de forma tal que la latencia sea menor?
    - En memoria compartida, ¿Los mecanismos de sincronización son los mínimos posibles?
  - Simplicidad: impacta sobre la mantenibilidad del programa. Para ello se debe evitar complicar la lógica del programa innecesariamente.

- Próxima fase: el diseño dado a este nivel impacta sobre el espacio de definición de la estructura del algoritmo. Si las interacciones entre las tareas son regulares puede utilizarse un patrón *pipeline*, de lo contrario un patrón *master/worker* puede ser más pertinente.

3.2.3.2. Dense Linear Algebra

El patrón puede verse completo en [BMB]

Problema

Existen problemas que pueden resolverse con operaciones de álgebra lineal sobre arreglos o matrices de datos densos. ¿Cómo balancear la distribución de datos y tareas para maximizar el rendimiento?

Contexto

Muchos problemas pueden resolverse en términos de operaciones de vectores y matrices. Esta forma de expresar el problema depende de su estructura y de que se simplifique la expresión de la solución al usar operaciones de álgebra lineal. Se debe balancear costos de comunicación con costos de computación.

Fuerzas

- El solapamiento de cómputo con comunicaciones debe ser evaluado en función de la jerarquía de memoria particular, lo cual se contrapone con el desarrollo de un algoritmo portable.
- La optimización de la operación de álgebra lineal puede imponer una distribución de datos que se contraponga con la óptima para el resto de las operaciones.

Solución

Utilizar bibliotecas de rutinas de álgebra lineal que estén disponibles para la plataforma a usar. BLAS (*Basic Linear Algebra Subprograms*) [Sub] es la definición de rutinas más difundida. Si no existiera alguna implementación de BLAS específica para dicha plataforma, utilizar ATLAS (*Automatically Tuned Linear Algebra Software*) [Sof], que se configura y compila con auto-ajuste de rendimiento, usando solo un compilador C.

Si no se desea o no se puede utilizar bibliotecas, debe tenerse en cuenta la jerarquía de memoria para lograr optimizaciones a la hora de dividir en bloques los datos. Utilizar *loop unrolling* y las instrucciones SIMD disponibles en el procesador.

3.2.3.3. Master / Worker Pattern

Extraído del libro *Patterns for parallel programming* [MSM04]

Problema

¿Cómo puede organizarse un programa cuando en el diseño se impone la necesidad de balancear la carga de trabajo dinámicamente?

Contexto

La carga debe estar balanceada para lograr una máxima eficiencia. El diseño del programa impone la división en numerosas tareas que no tienen igual carga de trabajo cada una. Existen numerosas causas por las que esto puede suceder:

- La carga de cada tarea es variable y/o impredecible.
- La parte computacionalmente pesada del programa no depende de *loops*
- La capacidad de cómputo de los procesadores no es homogénea.

Este patrón es aplicable cuando las dependencias entre las tareas no existe o son débiles.

Fuerzas

- La carga de la tarea varia impredeciblemente.
- Las operaciones de balance imponen una carga comunicacional pesada.
- La programación de un algoritmo de balanceo puede ser complicada e introducir fuente de errores.

Solución

Dividir a los procesadores en dos categorías lógicas: el maestro (*master*) y los trabajadores o esclavos (*worker*). El primero tiene el rol de definir qué tarea van a realizar los esclavos, los cuales envían el resultado de retorno al maestro para que compute el resultado global. De esta forma el balance de carga se hace dinámicamente y en forma automática, ya que el maestro no define de antemano qué va a realizar cada *worker*. Tener en cuenta

que este mecanismo puede imponer una sobrecarga de comunicaciones. Como la distribución de tareas es dinámica y los datos se distribuyen de antemano, cuando se le asigna una tarea a un nodo, es probable que los datos que necesite los deba solicitar a otro nodo.

Un tema importante es cómo determinar que el trabajo completo ha finalizado. Una forma de hacerlo es definir una bolsa de tareas, desde la cual el *master* las va despachando una a una entre sus esclavos, hasta finalizar con todas. El problema puede suceder cuando los esclavos producen nuevas tareas dinámicamente, lo cual debe administrar el *master* en forma correcta. La aplicación del patrón *Divide and Conquer* puede dar lugar a este fenómeno.

Existen variantes de este patrón que permiten al *master* asignarse el rol de *worker* una vez que distribuyó las tareas para aprovechar el tiempo inactivo hasta que los esclavos finalicen. Otra variante está orientada al problema de tener una única bolsa o cola de tareas. Este hecho suele provocar un "cuello de botella" en un entorno distribuido, lo cual suele solucionarse con una doble estructura de granularidad, gruesa para un entorno global, y fina para un entorno local, de forma tal que las tareas sean subdivididas antes de ser ejecutadas.

Es un patrón práctico a los fines de implementar tolerancia a fallos por medio de dos colas, una para las tareas pendientes y otra para las enviadas a realizar: si esta última no se vacía, se supone un fallo y la tarea es reenviada a otro nodo.

Patrones Relacionados

Loop Parallelism, Divide and Conquer

3.2.3.4. Wavefront Pattern

El patrón puede verse completo en [Snib]

Problema

Los datos del problema están distribuidos en forma de grilla representando un plano lógico y la dependencia de datos se asimila al avance de una ola sobre la diagonal principal de la grilla. El cómputo comienza en una esquina y avanza diagonalmente hacia la esquina opuesta. El problema es la distribución de los datos y de tareas entre los nodos procesadores.

Fuerzas

- Mantener balanceada la carga a medida que avance el cómputo en forma diagonal.
- Algunas unidades de procesamiento se mantienen inactivas mientras otros procesan.
- Mantener un eficiente rendimiento de todo el sistema.

Solución

Los resultados que son computados en etapas anteriores deben guardarse para evitar su recómputo en etapas posteriores (*memoization*). Usar una distribución en bloques cíclica de datos para mejorar el balance de carga.

Patrones Relacionados

Divide and Conquer

3.2.3.5. Non-Blocking Communications Pattern

El patrón puede verse completo en [BMK09]

Intención

Realizar cómputos en forma simultánea con comunicaciones se estén realizando.

Aplicabilidad

- Algoritmos que tienen comunicaciones intercaladas con cómputos.
- Existencia de cómputos independientes de los datos transferidos.

Motivación

Las comunicaciones colectivas bloqueantes no devuelven el control del programa a la rutina principal hasta que la comunicación no finalice. ¿Cómo reducir el tiempo de espera en aquellos programas que intercalan cómputos con envíos y recepciones de datos independientes? El desafío es determinar las áreas de cómputo que usan datos independientes de los datos comunicados para poder superponer cómputo y comunicación.

Fuerzas

- Simplicidad: las comunicaciones bloqueantes son simples y tienen la misma semántica que una llamada a una función, en el sentido que hasta que no termina de ejecutarse, no devuelve el control a la llamadora. Comunicaciones no bloqueantes complican la interpretación y el seguimiento del programa ya que existe un punto de invocación de la comunicación y otro de finalización de ésta, con cómputo realizado entre ambos puntos.
- Disponibilidad de la funcionalidad: a veces este tipo de funcionamiento no está disponible dentro de las bibliotecas de funciones disponibles y solo se pueden utilizar las bloqueantes.

Solución

Utilizar comunicaciones no bloqueantes. Estas tienen dos momentos en el tiempo: el primero cuando se lanza y devuelve el control a la rutina

llamadora de inmediato y el segundo, cuando dentro de la llamadora se intenta determinar que realmente haya finalizado la comunicación, permitiendo cómputos independientes intermedios que elevan el rendimiento. Hay varias formas de utilizar las comunicaciones no bloqueantes:

- Lanzar recepciones tempranas, de forma tal que a medida que se vayan realizando los cómputos, se van recibiendo los datos simultáneamente, y estos quedar disponibles para el momento de continuar con los siguientes cálculos.
- *Double Buffering*: se mantienen dos copias de los datos: una para enviar y otra para computar, de forma tal que el envío no interfiere con el cómputo.
- Múltiples envíos / recepciones: particionar un envío voluminoso en varias partes y comenzar a enviarlos / recibirlos, a medida que están disponibles.

El beneficio de esta técnica se incrementa con el número de procesadores intervinientes y decrece con la velocidad de transferencia de la red subyacente.

3.2.4. Reengineering for parallelism

Los autores del libro "*Patterns for Parallel Programming*" escribieron un reporte [MMS07] modelando el uso del lenguaje de patrones descripto en el libro, destinado a la paralelización de un programa serial existente. El trabajo propone una metodología de cómo proceder para paralelizar una aplicación serial existente (*legacy application*). En esa dirección presenta una serie de pasos a seguir en el proceso de transformación a paralelo de una aplicación serial, siguiendo el orden de los niveles jerárquicos de diseño planteados por el lenguaje de patrones dado en el libro y donde en cada paso puede aplicarse uno o varios de los patrones del respectivo nivel. Se indica que éste sería el punto de entrada al lenguaje de patrones paralelos (PLPP - *Pattern Language for Parallel Programming*).

Dicho reporte es presentado utilizando el formalismo de patrones, por lo que es denominado como *Reengineering for parallelism pattern*. No se encuentra muy feliz la presentación como un patrón de lo que mas bien sería una metodología de como paralelizar una aplicación serial, ya que brinda una serie de pasos donde en cada uno de ellos se aconseja aplicar patrones preexistentes. Adicionalmente se plantea la solución a un problema de naturaleza mayor al que consideran los patrones generalmente. Por estos motivos, será considerado a lo largo de la tesis como un pseudo-patrón, más que como un patrón en sí mismo.

Dejando en claro la salvedad anterior, se considera que la inclusión de este trabajo es relevante ya que las aplicaciones *legacy* son muy habituales dentro del ámbito científico. Como es planteado en el trabajo, en numerosas oportunidades, es conveniente paralelizar una aplicación serial existente antes que desarrollar una nueva paralela. Esto se debe a que el conocimiento experto embebido en la aplicación suele ser más difícil de obtener, que sortear las complicaciones que trae aparejado el proceso de paralelización de un código preexistente.

La exposición se hace manteniendo la metodología de presentación de patrones de la sub-sección anterior.

3.2.4.1. Reengineering for parallelism pattern

Problema

Dada una aplicación existente, ¿Cómo mejorar su rendimiento haciendo uso de *hardware* paralelo, y cómo utilizar el PLPP en el proceso?

Contexto

La existencia de una base de programas seriales y complejos de elaborar. La existencia de nuevas computadoras cuyo *hardware* tiene mayores prestaciones, no por el incremento de la velocidad del procesador, sino por la existencia de múltiples procesadores en paralelo. La necesidad de adaptar dichos programas al nuevo *hardware* para poder mejorar su rendimiento.

Fuerzas

- Aplicaciones con cargas de procesamiento lo suficientemente importantes para poder aprovechar el paralelismo.
- La ley de Amdahl [Amd67], que juega en contra de la mejora del rendimiento de las aplicaciones paralelas.
- La existencia de código que funciona y que implica un costo de desarrollo importante. Minimizar los cambios es lo deseable.
- La concurrencia introduce una nueva fuente de errores de programación que deben ser evitados.

Solución

Tener un balance entre los cambios a introducir y la mejora del rendimiento obtenido. Realizar cambios implica la posibilidad de introducir errores y éstos deben minimizarse. La paralelización debe pensarse como una serie de transformaciones que preservan la semántica del programa. Estas transformaciones deben hacerse sobre *"hot spots"*, es decir, lugares puntuales del programa que tengan una alta carga de procesamiento. Identificar los *"hot spots"* utilizando herramientas de *profiling*.

Primera etapa: Paralelización. Identificar los puntos de concurrencia utilizando los patrones del espacio de "Encontrar Concurrencia", aplicando principalmente la división de tareas y la división de datos. Definir la

estrategia principal del algoritmo con los patrones del espacio "Estructuras de Algoritmos" eligiendo el apropiado entre "*Task Parallelism*", "*Divide and Conquer*", "*Geometric Decomposition*", etc. Implementar la estrategia vía la utilización de patrones del espacio "Estructuras de Soporte", como ser SPMD, *Loop Parallelism, Fork/Join*, etc.

Segunda etapa: Evaluación y refinamiento. Luego de cada transformación deben hacerse corridas de prueba a los efectos de comprobar que se mantienen los resultados y el nivel de mejora en el rendimiento. Si éste último no es el aceptable, reevaluar el diseño y considerar alternativas para mejorar.

Se destaca dentro de la solución la propuesta de re-evaluación y refinamiento de las transformaciones practicadas: como el fin de la paralelización es la mejora del rendimiento, el diseño hecho en primeras instancias puede no ser lo suficientemente apropiado para lograr la mejora deseada. La utilización de herramientas de *tracing* y *profiling* son vitales para poder determinar los "*hot spots*" donde poner énfasis en los cambios. Además permiten verificar, que el cambio que se presumía como mejora, lo sea efectivamente. En caso contrario, proceder a modificar la estrategia de paralelización. Por lo tanto, la paralelización se vuelve un proceso de "Prueba - Error".

3.3. Análisis de los patrones existentes bajo la óptica de las aplicaciones de cómputo intensivo

En la sección anterior se presentaron algunos ejemplos de patrones de paralelismo considerados más o menos relevantes para las aplicaciones de cómputo científico, sin llegar a ser un listado completo de los mismos. Dichos patrones han sido desarrollados bajo la impronta de todo patrón de diseño, específicamente, como solución genérica a diversos tipos de problemas en el área del paralelismo. En esta sección se propone analizar si las soluciones planteadas se corresponden con el dominio bajo estudio, es decir, si realmente presentan una solución para el tipo de aplicaciones consideradas y teniendo en cuenta su entorno de ejecución bajo un *cluster* de nodos *multicore*.

Recapitulando sobre las propiedades de nuestras aplicaciones, se puede decir que la característica principal que presentan es que son de cómputo intensivo y requieren un alto nivel de procesamiento dado por el tipo de problema en sí mismo o por el volumen de datos a considerar. Esta es la particularidad por lo cual se busca que los programas corran en paralelo, es decir, obtener un resultado en tiempo razonable.

También se sostuvo en el capítulo de introducción que para obtener los niveles de procesamiento necesarios para la ejecución de estas aplicaciones, hay dos opciones, la señalada como HTC (*High Throughput Computing*) y la señalada como HPC (*High Performance Computing*). Como se dijo en aquella oportunidad, el HPC se centra en cómo utilizar lo más intensivamente posible el *hardware* donde se ejecutan los programas, para lo cual los algoritmos deben adecuarse a la plataforma física donde corren. También se citó que la arquitectura de *cluster* es la predominante hoy en día dentro del HPC.

Al determinar el alcance del trabajo se definió que se referiría al estudio de patrones de paralelismo aplicados a programas paralelos que corran en *clusters* de nodos *multicore*, es decir un conjunto de recursos computacionales homogéneo o altamente homogéneo y con redes de comunicación internas veloces, por lo que quedó definido que la perspectiva de estudio es la del HPC, sin dejar de reconocer que el alcance de este trabajo abarca un subconjunto pequeño de todo el espectro que forma el HPC. Por lo tanto, el análisis de los patrones debe dilucidar si el estado actual de los mismos realmente presentan soluciones para la clase de problemas bajo estudio.

Resalta entre los patrones de paralelismo actuales, vistos bajo la óptica del HPC, la escasa referencia que tienen al rendimiento. Es claro que para hablar de rendimiento se debe tener en cuenta cuál es el problema tratado, bajo qué plataforma de cómputo y de comunicaciones se ejecuta y cuál es la magnitud de la cantidad de datos considerada, entre otros factores. Sin embargo, si se quiere obtener alto rendimiento, dichos elementos deben ser tenidos en cuenta. Esto se contrapone con la concepción general de patrón, en lo referido a que la solución brindada no es una solución específica para un caso en particular, más bien genérica para poder ser adaptada a cada caso. Pero, si lo que se busca es rendimiento, los casos particulares deberían ser tenidos en cuenta, al menos como categorías de circunstancias.

Para que los patrones de paralelismo realmente brinden una solución referida a la eficiencia, deberían tener como mínimo una consideración al tipo de *hardware* donde se ejecuta el programa. Se cita como ejemplo el caso del patrón *Master/Worker*: este patrón es válido en un entorno de memoria compartida, siempre que la cantidad de *cores* disponibles sea tal que la pérdida de un core destinado a actuar como *master*, se compense con la ganancia que genera el balance de carga sobre los *cores* restantes.

Bajo un entorno de memoria distribuida, aplicar un patrón *master/worker* implica una centralización de los datos en el nodo *master*, lo cual promueve una carga de comunicaciones importante, ya que toda tarea hecha por los *workers* deberá recibir desde el *master* previamente los datos, y a su vez, enviar a éste los resultados. Se puede advertir claramente que no es necesario un número grande de nodos para saturar el canal de comunicación del *cluster*, principalmente cuando éste sea de tipo *bus* compartido por todos los nodos, como es el caso de redes tipo *Ethernet*. Si bien estas consideraciones figuran en la sección fuerzas del patrón a modo enunciativo, esto es insuficiente para captar la importancia central que tiene en la aplicación del mismo, al punto de ser el factor determinante que puede definir si se aplica o no para cada caso particular.

De todos los patrones estudiados, solo *Reengineering for parallelism pattern* tiene en cuenta que si el resultado obtenido no es el esperado en función a los recursos empleados, se debe reconsiderar el diseño del algoritmo paralelo, y reformularlo a los fines de poder utilizar más eficientemente el equipamiento e iterar reiteradas veces este proceso hasta alcanzar resultados aceptables. Este es el motivo por lo que dicho patrón es incluido en este estudio a pesar de no ser un patrón estrictamente hablando, sino más bien un procedimiento. Sin embargo, ninguna mejora es factible si los elementos particulares de la plataforma de ejecución no son tenidos en cuenta.

El patrón *Design Evaluation*, presentado en la sección 3.2.3.1, plantea algo en la misma linea de la reevaluación indicada en el párrafo anterior, pero lo hace a nivel del "Espacio de Diseño", según sus autores. Es difícil de entender como puede hacerse una reevaluación de la estructura global del algoritmo paralelo, sin haber llegado a la implementación. A nuestro entender, la reevaluación antes de la implementación es una especulación que solo pueden corroborarse con los hechos y que estos pueden dictaminar lo contrario a lo estipulado previo a las prácticas. Esto nos ocurrió efectivamente en reiteradas ocasiones al realizar los experimentos inherentes a la tesis.

Un posible punto de inclusión de las características particulares de la plataforma sobre la que corre el programa que aplica un patrón de paralelismo, puede ser en la sección de **Fuerzas**. Dicha sección tiene el siguiente objetivo: si la solución propuesta depende del contexto, las fuerzas deben presentar consideraciones a tener en cuenta en la aplicación de la solución que pueden ser contradictorias entre sí o relativizar la solución, por lo que deben balancearse para llegar a una mejor solución. Esta sección tiene relación con la sección de **Contexto** de un patrón, a los fines de poder valorar la importancia de las fuerzas y la aplicabilidad final de la solución [MD96].

Lamentablemente, ninguno de los patrones estudiados presenta en la sección de fuerzas, consideración alguna sobre las características que deben presentar tanto la plataforma de cómputo como la de comunicaciones para ser aplicados en forma exitosa. Tampoco consideraciones sobre rendimiento, *speedup* ni *performance*, de lo que se puede deducir, que al menos desde la perspectiva de la optimización del rendimiento en un *cluster*, el estado actual de los patrones de paralelismo presenta una carencia importante en este sentido. Por lo tanto, y en términos generales, más que una solución al problema de la optimización, estos patrones pueden ser considerados como estrategias de implementación de paralelismo, sin dar una respuesta al rendimiento que se pueda alcanzar, quedando librado a la experimentación individual de cada programa, la efectividad de su aplicación.

Por otro lado, la sección de fuerzas tampoco parece la apropiada para incluir allí detalles específicos sobre *hardware* o configuración de la red. Como se indicó anteriormente, en esta sección deben incluirse los factores que condicionan la aplicabilidad de un patrón, pero a modo de enumeración de los condicionantes en la aplicación del patrón. Quien sea el encargado de llevar adelante la aplicación del patrón, debe tener en cuenta dichos condicionantes en cada caso en particular y debe evaluar la conveniencia de aplicar el patrón en el problema a resolver.

Para el tipo de problemas bajo estudio, el impacto en el rendimiento de la plataforma de ejecución debe tener un mayor espacio del que brinda la sección de fuerzas. Se hace necesario dar indicaciones precisas sobre las condiciones de la plataforma de ejecución y pautas de rendimiento absolutas y comparativas que se puede alcanzar, para que el programador tenga un punto de partida avanzado para el proceso de paralelización y no comenzar con lo que su intuición le dicte al aplicar el patrón *Design Evaluation*, ganando tiempo y experiencia previa y evitando iteraciones en el proceso de prueba y error hasta alcanzar un rendimiento aceptable.

Un ejemplo que puede graficar la situación del párrafo anterior, se presenta ahora como adelanto del experimento expuesto en la sección 4.1.4: en un trabajo anterior referido al impacto de la arquitectura de la red subyacente en un *cluster* sobre los algoritmos [WT09], se aplicó el patrón *wavefront* para la distribución de tareas del algoritmo de multiplicación de matrices. Sin ser el tipo de algoritmo donde es recomendado este tipo de patrón, se experimentó aplicarle los patrones de comunicaciones colectivas y el patrón *wavefront* y comparar rendimiento, llegando a resultados que demostraron que bajo redes *Ethernet* la aplicación del *wavefront* es inapropiado en cuanto al rendimiento obtenido, lo cual es lo esperable. Sin embargo, el mismo algoritmo corriendo sobre una red Infiniband, da mejor rendimiento que para las comunicaciones colectivas, invirtiendo las conclusiones.

Del ejemplo anterior se deduce que hacer una indicación en la sección de fuerzas del patrón *wavefront* del tipo "tenga en cuenta que tipo de red que utiliza en el *cluster*", no daría una indicación apropiada y suficiente para el programador por lo limitado que dicha sección. El tema tiene una importancia tal, que como mínimo debería convertirse en una recomendación que supere a lo que brevemente puede señalarse en la sección de fuerzas. Así, debería indicarse sobre utilizar comunicaciones punto a punto en lugar de colectivas cuando la red que soporta el *cluster* tiene ventajas comparativas para este tipo de comunicaciones, sin importar que el problema donde se aplique sea "apropiado" para el patrón a nivel de diseño estratégico del algoritmo.

El próximo capítulo está destinado a mostrar resultados de pruebas de ejecución de algunos algoritmos clásicos de álgebra lineal aplicándoles diversos patrones de paralelismo a cada uno de ellos, siempre dentro del entorno *cluster*, y con redes de comunicación de tipo *bus* y punto a punto. Se verá que el rendimiento depende fuertemente de la plataforma utilizada, por lo que la aplicación del patrón "recomendado" según el problema, a nivel de la etapa de diseño, se relativiza frente a la plataforma finalmente utilizada.

3.4. Resumen del capítulo

En este capítulo se expuso la evolución y el estado del arte de los patrones de paralelismo. Se describieron las características que estos patrones presentan, como están categorizados y cuales son los centros de estudio de la materia más relevantes en la actualidad. A continuación se presentaron algunos ejemplos de patrones, incluyendo solo los conceptos más relevantes de cada uno de ellos.

También, y luego de un relevamiento del estado del arte de los patrones de paralelismo, se pudo comprobar la falta de referencias que éstos tienen al tema del rendimiento, ya que están orientados a brindar soluciones de paralelismo y no de rendimiento. Tan solo algunos patrones tienen una mínima referencia al tema en la sección de Fuerzas. La utilización de *clusters* para aplicaciones de cómputo intensivo, está fundada en la mejora del rendimiento a obtener, por lo que, el estado actual de los patrones de paralelismo, es solo una buena guía de ejemplos de paralelización, pero no es útil a la hora de ayudar en la obtención de altos rendimientos paralelos.

Capítulo 4

Experimentación de la aplicación de patrones de paralelismo sobre algoritmos seleccionados

En los capítulos anteriores han sido expuestos los elementos y características que conforman los *clusters* de nodos *multicore* y sus modelos de programación, las características de las aplicaciones de cómputo científico que requieren un alto volumen de cómputo y finalmente el estado actual de los patrones de paralelismo y su nivel de aplicabilidad al tipo de aplicaciones bajo estudio en la tesis. El presente capítulo será destinado a la presentación de experimentos que permitirán respaldar las conclusiones referidas a la aplicación de patrones de paralelismo sobre algunos algoritmos clásicos en el ámbito científico, más específicamente de álgebra lineal. Estos algoritmos son frecuentes dentro del tipo de multicomputadoras objeto de esta tesis y son representativos para convalidar los resultados alcanzados.

En virtud de que el ámbito de ejecución de los programas objeto de la presente tesis son los *cluster* de nodos *multicore*, los algoritmos utilizados como test en el trabajo serán probados primero bajo el modelo de memoria compartida, es decir intra-nodo, de forma tal de determinar cual es la mejora en el rendimiento de ejecución al utilizar los nodos disponibles respecto del programa serial.

Los experimentos están destinados a corroborar el impacto de la aplicación de patrones de paralelismo sobre los algoritmos en cuanto a los rendimientos obtenidos. Básicamente las pruebas se destinan a obtener dos tipos de resultados, por un lado la aplicación de diversas estrategias de distribu-

ción de datos y tareas entre los nodos del *cluster* para un mismo algoritmo y su impacto en el rendimiento, dichas estrategias basadas en lo sugerido por diversos patrones, y por otro lado, el impacto del tipo de red que compone el *cluster* sobre los rendimientos, más específicamente, usando redes tipo *Ethernet* y tipo Infiniband.

En lo referente a la optimización de los algoritmos bajo un entorno de memoria compartida, se toma como antecedente un trabajo anterior en el cual se experimentó la utilización de bibliotecas de rutinas de álgebra lineal para el algoritmo de multiplicación de matrices en un *cluster* de nodos *multicore* [TW08]. En dicho trabajo se alcanzó un *speedup* casi óptimo en relación al número de cores disponibles en uno de los nodos que conforma el *cluster*, repitiéndose los resultados habiendo utilizando varias implementaciones de la biblioteca BLAS. Se hizo con tan solo configurar el número de *threads* OpenMP que utilizan dichas rutinas.

La simple configuración de los hilos parelelos a utilizar por parte de las implementaciones de la biblioteca BLAS, determina que el programador deba poner el énfasis casi exclusivamente sobre la distribución de datos a nivel del modelo de memoria compartida. Sin embargo, para cada algoritmo se corroborará la proyección de estos resultados de forma tal que, bajo el tipo de plataforma de ejecución definida para esta tesis, el proceso de paralelización debe abocarse a la distribución de datos y tareas entre los nodos del *cluster*, es decir, la optimización bajo el modelo de memoria compartida.

Los algoritmos seleccionados para los experimentos son:

1. Multiplicación de matrices, dada su simplicidad, amplia difusión y nulo nivel de dependencia de datos [GL96].
2. Factorización de matrices Cholesky, dado su amplia difusión y dependencia de datos de complejidad intermedia [GL96].

En toda ocasión se utilizaron números de punto flotante de simple precisión. Esta precisión es suficiente a los efectos de determinar el impacto por la aplicación de los patrones de paralelismo. La correctitud de los cómputos ha sido verificada en todos los experimentos vía la utilización de matrices cuyo resultado es conocido de antemano. Para el caso de la multiplicación de matrices, se utilizaron matrices cuya multiplicación de como resultante una matriz cuyos valores sean todos n, el rango de las matrices utilizadas, en punto flotante. Para el caso del algoritmo de Cholesky, la multiplicación de los factores obtenidos de igual a la matriz factorizada.

Los experimentos han sido realizados utilizando tres clusters distintos:

1. Cluster 1, conformado por seis computadoras, dos de las cuales están configuradas con dos procesadores Intel Xeon 5420 *quad-core* cada una, ocho GBytes RAM. Las restantes cuatro, con dos CPU's AMD Opteron 2200 *dual-core* cada una, cuatro GBytes RAM. La conectividad de red está formada por dos redes, una red *Ethernet*, con un *switch* de un Gbit/sec, y una red Infiniband con un switch Flextronics e interfaces de red 4x, con un ancho de banda nominal de 20 Gbits/sec. Todas las computadoras tienen Centos 5.3 como sistema operativo y la implementación de MPI usada es Open MPI 1.3.4. Compilador Sun studio 12.1, usando bibliotecas Sun Sunperf como implementaciones de las bibliotecas BLAS y LAPack.

2. Cluster 2, conformado por siete computadoras con dos CPU's Intel Xeon 5420 *quad core* cada una, ocho GBytes RAM, también con dos redes de comunicación, una *Ethernet* de 100 Mbits y la otra Infiniband de 20 Gbits/sec. Centos 5.3 como sistema operativo y Mvapich 1.1 como implementación MPI. Compilador Intel ifort 11.0, y la implementación de las bibliotecas BLAS y LAPack usada es MKL de Intel.

3. Cluster 3, conformado por 64 computadoras con dos CPU's Intel Xeon 5420 *quad core* cada una, dieciséis GBytes RAM, y red Infiniband 20 Gbit/sec de conexión. También Centos 5.3 como sistema operativo y Open MPI 1.4.1 como biblioteca de rutinas MPI. El compilador utilizado es Intel ifort 11.1 y la biblioteca MKL como implementación de BLAS y LAPack.

Como puede apreciarse, si bien los *clusters* tienen características similares en cuanto al tipo de CPU utilizado, están configurados de forma tal que son combinaciones de distintos compiladores e implementaciones de bibliotecas BLAS, LAPack y MPI, lo cual brinda un marco de prueba variado a los efectos de garantizar cierta generalidad en los resultados.

Las próximas secciones están destinas a la presentación de los resultados experimentales sobre los algoritmos seleccionados.

4.1. Multiplicación de Matrices

El primero de los algoritmos utilizados para la experimentación es el algoritmo de multiplicación de matrices. La elección de este algoritmo está

fundada en que es de amplia utilización y fácil implementación, tiene un nivel de dependencia de datos bajo (*embarrassingly parallel*) y por lo tanto, se pueden aplicar diversas distribuciones de datos bajo el modelo de memoria compartida.

Los experimentos realizados son, en primer lugar, el computar utilizando exclusivamente memoria compartida, y luego extendiendo el uso de los recursos bajo la combinación de memoria compartida y distribuida, con diversos esquemas de distribución de datos y tareas, siempre como consecuencia de la aplicación de un patrón de paralelismo, de forma tal de determinar su impacto en la mejora de los rendimientos.

4.1.1. Memoria Compartida

Para estos experimentos serán utilizadas las rutinas de multiplicación de matrices incluidas en la biblioteca de rutinas de álgebra lineal BLAS, específicamente la rutina SGEMM, bajo las implementaciones de los paquetes MKL de Intel [INT] y Sun Performance Library de Sun [Sun]. Fueron aplicadas las conclusiones del trabajo arriba señalado, respecto de los pobres rendimientos obtenidos por la programación "cruda" del algoritmo con su famoso triple ciclo anidado, y, si se dispone de una rutina que lo implemente, ésta suele tener mejor *performance* [TW08].

La utilización de los *cores* de procesamiento disponibles en cada nodo se hizo por medio de asignar diversos valores al parámetro que las bibliotecas de álgebra lineal disponen para configurar el número de *threads* que utilizaran en un entorno *multicore*, estandarizado como una simple llamada a la rutina `SET_OPENMP_NUM_THREADS(x)`, donde x indica el número de *threads* a utilizar, y el máximo valor que x debería tomar es el número de *cores* disponibles en el nodo, ya que valores mayores no causan efectos positivos sobre el rendimiento.

En toda ocasión se utilizan matrices cuadradas y los rangos de estas se determinan para que cubran una serie de tamaños variados, teniendo un límite de máximo rango, al que utilice la máxima memoria disponible en el nodo, es decir, teniendo en cuenta que el sistema operativo consume una parte, de forma tal que el sistema operativo evite hacer *swap* de memoria sobre el disco.

Los resultados se presentan en el cuadro 4.1. Para dichos cómputos se utilizó un nodo de 8 *cores* del *cluster* 1 y la biblioteca *Sunperf* como implementación BLAS y configurando el parámetro que indica el número de

Rango Matriz	*Threads*	Tiempo (segs.)	*Speedup*
2000	1	1.515	1.000
2000	2	0.772	1.962
2000	4	0.420	3.607
2000	8	0.251	6.035
4000	1	12.078	1.000
4000	2	6.096	1.981
4000	4	3.209	3.764
4000	8	1.825	6.618
8000	1	96.918	1.000
8000	2	48.853	1.984
8000	4	25.547	3.793
8000	8	14.328	6.764
12000	1	323.229	1.000
12000	2	163.204	1.981
12000	4	84.094	3.843
12000	8	46.362	6.972
16000	1	778.290	1.000
16000	2	390.527	1.993
16000	4	202.332	3.847
16000	8	111.971	6.951

Tabla 4.1: Tiempos de ejecución y *speedup* para multiplicación de matrices bajo memoria compartida en un nodo con 8 *cores* del *cluster* 1.

threads a utilizar por dicha biblioteca como se indicó anteriormente. Como puede verse, los valores del *speedup* son casi los óptimos, salvo para el caso de 8 *threads* con un rango de matriz menor, cosa que se revierte a medida que se incrementa el rango.

Estos resultados permiten concluir, que dentro del modelo híbrido OpenMP-MPI, lo referido a la optimización del procesamiento en memoria compartida para algoritmos que usen multiplicación de matrices está aceptablemente resuelto por las bibliotecas de álgebra lineal usuales, debiendo ponerse el énfasis en la forma de distribuir datos y procesamiento entre los nodos que componen el *cluster* para lograr una optimización global.

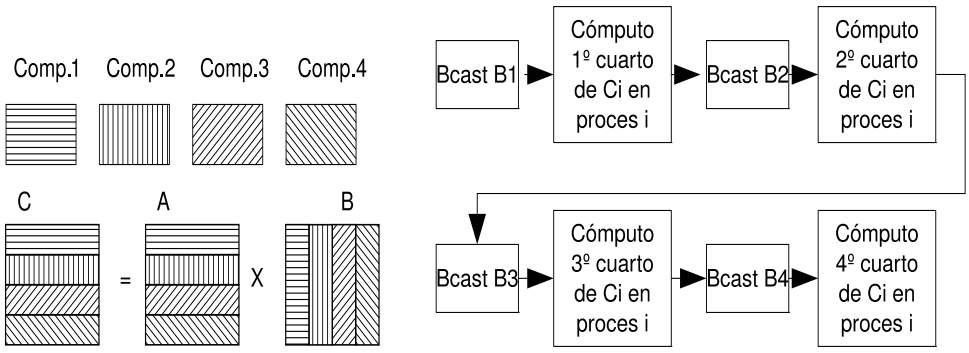
4.1.2. Memoria Distribuida - Comunicaciones colectivas

Las primitivas implementadas por la bibliotecas de paso de mensajes MPI que utilizan comunicaciones colectivas tienen como rutina principal a la rutina *broadcast*. Esta funciona de forma tal, que un nodo envía un mensaje a todos los restantes nodos que conforman el grupo colectivo al cual pertenece el nodo que envía, tal como puede verse en *Collective Communication Patterns* [CKS+09]. En este trabajo, que contiene varias soluciones de comunicaciones colectivas, se puede encontrar un patrón específico referido a la comunicación de datos entre todos los componentes del computador paralelo, con un emisor y todos los restantes nodos como receptores.

La semántica del *broadcast* que el protocolo MPI define, es la denominada "bloqueante" [SOHL+96], lo cual impone que hasta que todos los nodos hayan confirmado la recepción del mensaje, el emisor no puede continuar con la ejecución de su código. Las principales bibliotecas que implementan MPI, además imponen una barrera entre los restantes nodos, por lo que el *broadcast* actúa generalmente como un punto de sincronización entre todos los nodos. Comunicaciones colectivas no bloqueantes no están disponibles por disposición del comité que define MPI, sin embargo algunos autores lo proponen para futuras versiones basados en algunos resultados positivos [HLR07].

En un trabajo previo se alcanzaron resultados que demostraban un *speedup* casi óptimo para la multiplicaciones de matrices en *cluster* de nodos *multicore* [TW09]. Para un cómputo de la forma $C = A \times B$, donde A, B y C son matrices cuadradas, el algoritmo es mostrado esquemáticamente en la Fig.(4.1), suponiendo un *cluster* formado por cuatro nodos. La parte izquierda de la figura muestra la distribución por bandas de los datos entre los nodos que conforman el *cluster*, en bandas fila para las matrices A y C, y bandas columnas para la matriz B, mientras que la parte derecha de la figura muestra la secuencia de procesos que realizan el cómputo total, donde cada nodo computa su respectiva banda fila de C, recibiendo por vez, vía *broadcast*, una banda columna de B y computando en cada iteración, en función de esto, una cuarta parte de su banda.

Es claro que bajo este algoritmo, el número de bandas en que se divide cada matriz y el número de *broadcasts* que se ejecutan es igual al número de nodos que componen el *cluster*. Por otro lado, dado el alto nivel de independencia de datos existente para computar cada bloque de la matriz C, es evidente que bajo la óptica de aplicación de patrones de paralelismo, la distribución de datos es quien impone la secuencia de distribución de tareas.



(a) Distribución de datos (b) Secuencia de procesos

Figura 4.1: Algoritmo de multiplicación de matrices

Se ejecutaron varias pruebas de este algoritmo bajo diversos *clusters* y con distintos rangos de matrices. Los resultados de uno de los mismos se presentan en la tabla 4.2 utilizando la red *Ethernet* y la red Infiniband. Adicionalmente a lo largo de la sección existen otros resultados ya que el algoritmo de *Broadcast* es tomado como punto de referencia para la comparación de los algoritmos alternativos expuestos.

Nodos	Rango	Bcast Ether.(1)	Bcast IB (2)	(1)/(2)
4	2400	0.456	0.297	1.535
4	4800	2.008	1.179	1.703
4	9600	10.652	7.128	1.494
4	14400	30.758	22.685	1.356
6	2400	0.449	0.311	1.444
6	4800	1.888	1.041	1.814
6	9600	9.200	5.335	1.724
6	14400	24.703	16.068	1.537

Tabla 4.2: Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria distribuida. Algoritmo de *Broadcast* bajo Ethernet e Infiniband, utilizando el cluster 2 de pag 51.

4.1.3. Memoria Distribuida - Patrón *Master/Worker*

Una de las primeras alternativas estudiadas al algoritmo de comunicaciones colectivas de la sección anterior, es la utilización del patrón *master/worker*

[MSM04] para la asignación de tareas entre los nodos componentes del *cluster*. Dados los buenos resultados obtenidos en la sección 4.1.1 en cuanto al rendimiento logrado para el modelo de memoria compartida, el presente algoritmo se apoya en dichos resultados y utiliza dicha estrategia para el cómputo intra-nodo, enfocándose en la distribución entre los nodos de las tareas y los datos siguiendo lo indicado por el patrón.

Se plantea una salvedad respecto a la aplicación del patrón. El algoritmo desarrollado para esta sección no se adecúa estrictamente al patrón *master/worker* ya que la distribución de los datos no responde al esquema definido. En lugar de que los datos estén centralizados en el nodo *master* y que éste los envíe a los nodos *workers* por cada tarea que deben hacer, los datos están distribuidos entre los nodos que componen el *cluster* con la finalidad de posibilitar la escalabilidad del algoritmo: la cantidad de memoria RAM disponible por el nodo *master* impone un límite máximo al tamaño de matrices que el algoritmo puede tratar sin hacer *swap* con el disco fijo, por lo que una distribución de datos permite escalar en cuanto al tamaño de los mismos.

La distribución de datos practicada para este algoritmo es la misma que para el algoritmo de comunicaciones colectivas, es decir la graficada en la Fig.(4.1a). A diferencia del algoritmo de *broadcast*, ahora las comunicaciones en lugar de ser colectivas, son punto a punto, de forma tal que el nodo que necesita datos, se lo solicita a su poseedor y éste se lo envía únicamente al solicitante. De esta forma, el algoritmo tiene la siguiente estructura de pasos:

1. Cada vez que hay un nodo *worker* disponible, éste notifica al *master* su situación y queda en estado de espera de los datos de la banda fila de la matriz B que el poseedor de la misma debe enviarle para que pueda realizar el cómputo.

2. El *master* notifica al poseedor de la banda solicitada que el receptor está esperando por sus datos y éste último entonces, se los envía al *worker* sin intermediación del *master*.

3. El destinatario recibe y computa junto con su banda fila de A, el bloque respectivo en la banda de C correspondiente, y, dado que la distribución de la matriz C es igual a la de la matriz A, no necesita enviar el resultado a ningún otro nodo.

Esta variante de la aplicación del patrón *master/worker* tiene la ventaja adicional de disminuir el tráfico de red respecto de la versión estándar de

Nodos	Rango	Bcast Eth.(1)	Mas/Wor Eth. (2)	(2)/(1)	Bcast IB (3)	Mas/Wor IB (4)	(4)/(3)
4	2400	0.456	0.528	1.16	0.297	0.400	1.35
4	4800	2.008	2.613	1.30	1.179	1.899	1.61
4	9600	10.652	16.686	1.57	7.128	13.343	1.87
4	14400	30.758	51.648	1.68	22.685	41.723	1.84
6	2400	0.449	0.505	1.13	0.311	0.323	1.04
6	4800	1.888	2.093	1.11	1.041	1.016	0.98
6	9600	9.200	10.363	1.13	5.335	5.359	1.00
6	14400	24.703	28.323	1.15	16.068	18.131	1.13

Tabla 4.3: Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria distribuida. Algoritmos de *Broadcast* y *Master/Worker* bajo Ethernet e Infiniband, utilizando el cluster 2 de pag 51.

master/worker, ya que por cada cómputo de un bloque de C, solo una banda fila de B es necesaria enviar desde su poseedor hasta el destinatario final sin necesidad de intermediación por parte del *master*.

Un detalle de implementación está dado por el hecho de que, como la cantidad de nodos disponibles en el *cluster* es bajo, asignar el rol de *master* a un nodo completo implica un impacto fuerte en la caída de los recursos disponibles para realizar el cómputo propiamente dicho. Por otro lado, en este caso en particular, la tarea del nodo *master* es de solo dar la orden de envío de datos y sincronizar todo el proceso, lo cual no es una carga computacional pesada. Estos factores posibilitan que el rol de *master* pueda ser asignado a un solo *thread* en uno de los nodos, dejando disponible para cómputo los restantes. Dicho de otra forma, hay un nodo que toma el rol de *master* por un solo *core* de los n disponibles y también el rol de *worker* por los restantes $n - 1$ *cores*.

Otro punto destacable de la implementación de este algoritmo, es que las comunicaciones entre los nodos del *cluster* dejan de ser colectivas, para pasar a ser punto a punto. En términos teóricos esto impacta en forma diferente según sea el esquema de red del *cluster*, ya sea utilizando *Ethernet* o Infiniband. Dadas las características de la arquitectura de ambos tipos de redes, es de suponer que se obtendrán resultados relativos mejores sobre Infiniband que sobre *Ethernet*. Este supuesto fundamenta que las pruebas del presente algoritmo se hayan realizado por duplicado sobre un mismo *cluster*, haciendo uso de una u otra red en cada caso, para constatarlo.

Nodos	Rango	Bcast IB (1)	Mas/Wor IB (2)	(2)/(1)
4	2400	0.213	0.396	1.86
4	4800	0.717	0.860	1.20
4	9600	4.346	5.798	1.33
4	14400	12.260	17.691	1.44
6	2400	0.232	0.291	1.25
6	4800	0.637	0.714	1.12
6	9600	3.100	3.858	1.24
6	14400	9.662	11.907	1.23
8	2400	0.264	0.232	0.88
8	4800	0.624	0.422	0.68
8	9600	2.555	2.088	0.82
8	14400	7.151	5.977	0.84

Tabla 4.4: Tiempos de ejecución en segundos para multiplicación de matrices bajo memoria compartida. Algoritmos de *Broadcast* y *Master/Worker* sobre Infiniband, utilizando el cluster 3 de pag 51.

Los resultados de los experimentos pueden verse en tablas 4.3 y 4.4. La primera muestra resultados utilizando 4 o 6 nodos y *Ethernet* o Infiniband para diversos rangos de matrices. La segunda tabla es similar pero con hasta 8 nodos y solo para Infiniband. Para todos los casos experimentados se tomaron los tiempos para el algoritmo de comunicaciones colectivas (*broadcast*), como punto de referencia para determinar una posible mejora en el presente algoritmo. Se computó el cociente entre el tiempo insumido para la versión *master/worker* sobre el tiempo insumido para la versión *broadcast* a los efectos de poder tener un índice de la eficiencia relativa del algoritmo *master/worker*. Los rangos de las matrices han sido tomados de forma tal que al dividirse las matrices por el número de nodos y por 8 *threads* utilizados internamente dentro de cada uno de éstos, de como resultado un número entero, de forma tal que todos *threads* tengan una carga igual.

Dejando por sentado que los *test* fueron realizados sobre *clusters* de pequeña dimensión y que las conclusiones no pueden ser generalizadas, puede observarse que, como se presumía de antemano, el algoritmo de *master/worker* es más ineficiente relativamente que el de *broadcast* bajo *Ethernet* que bajo Infiniband. Sin embargo, a medida que crece el número de nodos del *cluster* utilizados, la ineficiencia relativa decrece, al punto que utilizando 8 nodos, el algoritmo de *master/worker* es más eficiente que el de *broadcast*

bajo Infiniband, incluso destinando un *thread* al rol de sincronización. Dicho de otra forma, para el caso particular expuesto en la tabla 4.4, con 8 nodos, 63 procesadores utilizando *master/worker* son más eficientes que 64 utilizando comunicaciones colectivas.

De los resultados de esta sección puede deducirse que el costo de la sincronización colectiva es gravoso y creciente con el número de nodos involucrados. Determinar las causas de este fenómeno está fuera del alcance del presente trabajo, pero desde el punto de vista de los patrones de paralelismo, puede verse que la aplicación de un patrón no determina por si solo la eficiencia de un algoritmo, sino que en conjunción con los restantes factores que intervienen en el desarrollo del cómputo. La aplicación de *master/worker*, a pesar de su esquema de comunicaciones complejo, reditúa en un algoritmo más eficiente cuando el número de nodos no es bajo y se ejecuta sobre Infiniband.

4.1.4. Memoria Distribuida - Patrón *Wavefront*

Un algoritmo alternativo para la multiplicación de matrices puede definirse aplicando el patrón *wavefront* de la sección 3.2.3.4. Dado que, en la multiplicación de matrices el nivel de independencia de las tareas es alto ya que cada tarea no depende de ninguna otra previa, el orden en que éstas se realicen y la forma de comunicar los datos son los factores que puede determinar una mejora en los resultados en términos de eficiencia.

A pesar de la independencia de datos que el algoritmo de multiplicación de matrices tiene, *a priori*, puede considerarse que aplicar el patrón *wavefront* acarreará una pérdida en el rendimiento dado que se cambia un esquema de comunicaciones colectivas por uno de comunicaciones "punto a punto" mucho más sofisticado en cuanto a la determinación de la tarea que debe hacerse en cada momento del tiempo y del nodo encargado de realizarla (*scheduling*). Esta complejidad supone un incremento en el tiempo insumido. A pesar de este supuesto inicial, se implementó la versión con *wavefront* para poder cuantificar la caída en el rendimiento, y cuanto según se utilice *Ethernet* e Infiniband.

El algoritmo parte de una distribución de datos similar al caso del algoritmo con comunicaciones colectivas, es decir, para un cómputo $C = A \times B$, las matrices A y C distribuidas por bandas filas y la matriz B por bandas columnas. Considerando que n es el rango de las matrices y p es la cantidad de nodos que compone el *cluster*, cada nodo tiene $n/p \times n$ datos, pero ahora distribuidos cíclicamente en dos sub-bandas, como puede verse en la

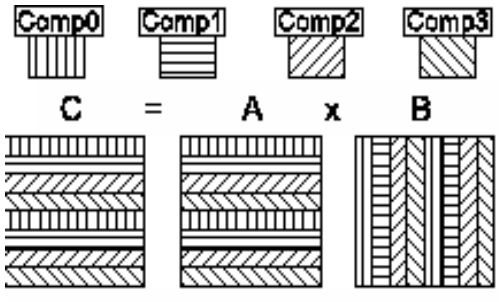


Figura 4.2: Distribución de datos para la multiplicación de matrices con patrón *wavefront*

Fig.(4.2). Esta distribución, con cada nodo disponiendo de dos sub-bandas, está orientada a disminuir el desbalance que provocaría si solo fuera una banda entera, como para el algoritmo de *broadcast*, al avanzar el procesamiento en forma diagonal sobre la distribución de datos.

La tareas son divididas formando una grilla 2D, donde cada una de éstas computa un bloque cuadrado de rango $n/(p \times 2)$. Los datos son enviados desde el nodo dueño de la banda respectiva de B hacia el dueño de la banda respectiva de A para que realice el cómputo. En este algoritmo, en lugar de comunicar cada banda por medio de un mensaje *broadcast* a todos los restantes nodos del *cluster*, la comunicación es de tipo "punto a punto", como para el caso de *master / worker*.

La secuencia de tareas forma una especie de "ola" que avanza desde la esquina superior izquierda hacia la inferior derecha, como puede verse en la Fig.(4.3), donde se suponen 4 nodos conformando el *cluster*, los cuales se

	0	1	2	3	0	1	2	3
0	0	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8	9
3	3	4	5	6	7	8	9	10
0	4	5	6	7	8	9	10	11
1	5	6	7	8	9	10	11	12
2	6	7	8	9	10	11	12	13
3	7	8	9	10	11	12	13	14

Figura 4.3: Distribución de tareas para la multiplicación de matrices con patrón *wavefront*

numeran de 0 a 3. Cada diagonal tiene un número que representa el avance del cómputo por cada "ola". En dicha figura, los números en la columna fuera de la matriz indican el número del nodo poseedor de la respectiva banda fila de A y C, acorde a la distribución de datos señalada. Los números en la fila superior fuera de la matriz indican el número del nodo poseedor de la respectiva banda columna de B. La secuencia de procesamiento es la siguiente:

1. Las tareas se ejecutan comenzando por la "ola" o diagonal numerada con 0, donde el nodo 0, es el único que computa y no requiere comunicación al ser el poseedor de la banda fila y columna necesarias para el cómputo.

2. Se continua con las tareas de la diagonal 1, donde se ejecutan dos tareas, necesitando los envíos desde el nodo 0 al 1 y desde el 1 al 0 para computar ambas.

3. Siguiendo de esta forma se llega a las tareas de la diagonal 13 donde son necesarios los envíos desde los nodos 2 a 3 y 3 a 2, para finalizar con la única tarea de la diagonal 14, donde no es necesaria la comunicación y solo el nodo 3 computa.

Se evidencia en este algoritmo la complejidad que presenta el esquema de comunicaciones. Por ejemplo, las tareas de la diagonal 7 necesitan de 8 comunicaciones, y si bien no hay dependencia de datos, para poder concretarse cada envío/recepción, ambos nodos deben estar disponibles, es decir, deben haber acabado todas las tareas que tenían asignadas previamente. Este hecho puede verse en la Fig.(4.4), donde se muestra la traza de ejecución del algoritmo corriendo en el cluster 1 de página 51, utilizando Infiniband. La traza de ejecución fue obtenida utilizando la herramienta "*Analyzer*" de la *suite* de compilación "*Oracle Solaris Studio*" [Ora] y se corresponde con una ejecución que utiliza solo 4 nodos, donde las áreas blancas representan el tiempo utilizado en el cómputo propiamente dicho, las áreas celestes (grises), los tiempos de espera para completar las comunicaciones y la lineas negras entre las 4 bandas horizontales representa la dependencia entre los nodos que genera la espera en completar una comunicación.

En la implementación del algoritmo se utilizaron comunicaciones no bloqueantes, en una secuencia *<envío no bloqueante - recepción no bloqueante - cómputo usando un valor previo - espera hasta recibido>* que permite de esta forma solapar comunicaciones con cómputo. A pesar de este solapamiento que en principio posibilita agilizar el cómputo, puede verse claramente en la

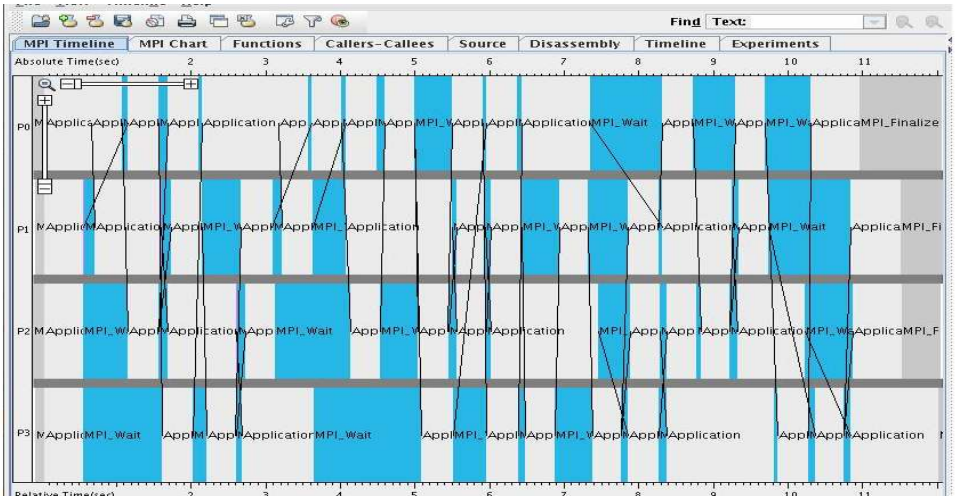


Figura 4.4: Traza de ejecución del algoritmo de multiplicación de matrices con el patrón *wavefront*.

Rango Matriz	Wavefr. Ether.	Bcast. Ether.	Wave/Bcast Ether.	Wavefr. Infin.	Bcast. Infin.	Wave/Bcast Infin.
2000	0.886	0.610	1.452	0.691	0.628	1.100
4000	3.090	2.310	1.338	2.483	2.022	1.223
8000	17.204	12.647	1.360	14.378	11.271	1.276
12000	56.186	39.219	1.433	45.145	36.088	1.251

Tabla 4.5: Tiempos de segundos de multiplicación de matrices aplicando el patrón *wavefront*, corriendo en 4 nodos con Ethernet o con Infiniband

Fig.(4.4) que existen momentos de espera importantes para poder completar la comunicación antes de poder proseguir con los cómputos. En la tabla 4.5 se muestran los tiempos de ejecución sobre el citado *cluster* utilizando *Ethernet* e Infiniband como red de comunicaciones, siempre con la presencia de los tiempos para el algoritmo que usa *broadcast* para fines de comparación.

Puede entonces verse que el algoritmo con el patrón *wavefront* es más ineficiente que el que utiliza *broadcast*. Como es de esperar, la caída del rendimiento del algoritmo bajo *Ethernet* es mayor a la caída bajo Infiniband, un 40 % aproximado del primero, contra un 25 % en promedio del último.

Este problema fue reportado en un trabajo previo [WT09], tratando de reducir los tiempos de espera para completar las comunicaciones en este algoritmo. Se pudo mostrar que estas esperas pueden subsanarse utilizando múltiples envíos simultáneos no bloqueantes en lugar de uno solo como hasta

Rango Matriz	Wavefr. Infin.	Bcast. Infin.	Wave/Bcast Infin.
2000	00.584	00.625	0.934
4000	01.993	02.001	0.996
8000	11.748	11.195	1.049
12000	36.852	35.880	1.027

Tabla 4.6: Tiempos en segundos de la multiplicación de matrices en 4 nodos aplicando *wavefront* sobre Infiniband con múltiples envíos simultáneos no bloqueantes

ahora, en una secuencia *<múltiples envíos no bloqueantes - múltiple recepción no bloqueante - cómputo - múltiples esperas hasta que un mensaje es recibido>*. Con este cambio en la implementación del algoritmo de *wavefront* pudieron lograrse tiempos similares a los del algoritmo de *broadcast* bajo Infiniband como queda reflejado en tabla 4.6 con diferencias entre ambos algoritmos de $\pm$ 5 % en los tiempos de ejecución. De esta forma pudo comprobarse que, aplicar un patrón que *a priori* se presenta como inapropiado para el algoritmo del problema, en la práctica, dicho supuesto puede revertirse, dependiendo del tipo de *hardware* sobre el que se ejecute la implementación, por lo tanto la pertinencia de la aplicación del patrón está altamente condicionada por la plataforma de *hardware* donde se ejecute el programa.

4.1.5. Memoria Distribuida - Uso de cómputos parciales

En esta sección se presentará un algoritmo alternativo para la multiplicación de matrices. Ha sido el resultado de buscar alternativas en la distribución de datos y la división de tareas que disminuyan los tiempos de ejecución y está basado en una estrategia de cómputo de valores parciales. Este caso servirá de base para la sección ejemplos en la presentación de un patrón inexistente al momento, desarrollado en el capítulo 5, llamado *Partial Computing*, el cual surge como un producto de los estudios y experimentos realizados para el desarrollo de la presente tesis.

El concepto principal del los "Cómputos Parciales" es aplicar una división de tareas en un nivel más profundo que el tradicionalmente practicado, por medio de subdividir las tareas de procesamiento en partes que computen resultados parciales y mantener dichos resultados hasta el momento en que sean necesarios para calcular los valores finales. Para que esta subdivisión sea

conveniente desde el punto de vista del rendimiento, es necesario que estén disponibles parte de los datos en el momento en que el procesador esté ocioso.

Siguiendo la estrategia indicada, se considera en primer lugar el esquema de distribución de datos y tareas que tiene el algoritmo que utiliza comunicaciones colectivas, sección 4.1.2, para poder determinar la carga de comunicaciones que tiene. Tomando matrices A, B, C de rango n, que realicen el computo $C = A \times B$, utilizando p nodos de procesamiento y los datos de las matrices distribuidos en bandas, horizontales para las matrices C y A, y verticales para la matriz B, donde cada banda tiene n/p filas por n columnas para las primeras y n filas n/p columnas para las ultimas. Cada nodo posee una banda fila para A y C, y una banda columna de B.

El procesamiento que da por resultado cada banda de la matriz C, se hace en el nodo poseedor de dicha banda, por lo que el nodo k, dueño de la banda fila k de C, es quien calcula los valores resultantes de dicha banda. Como las bandas de A son de dimensión $(n/p) \times n$ y las de B son $n \times (n/p)$, el producto a nivel de bandas de $A \times B$ es de dimensión $(n/p) \times (n/p)$, y será llamado $C_{i,j}$ a cada sub-bloque de C resultante del cómputo $C_{i,j} = A_i \times B_j$.

Dado que los datos de la fila i de A y C residen en el mismo nodo, el cómputo evoluciona enviando los datos de la columna j de B a todos los nodos, para que cada nodo haga el cómputo del sub-bloque $C_{i,j}$ correspondiente, salvo que $i = j$, en cuyo caso no hay envíos de datos ya que residen en el mismo nodo; en la práctica se utiliza una llamada a la rutina *broadcast* tomando como origen al nodo j. Entonces, cada banda fila de C es el resultado de p tareas de procesamiento, cada una de ellas a partir de los datos de la banda fila de A correspondiente y una banda columna de B.

Acorde a la distribución de datos definida anteriormente, el movimiento de datos para computar una banda fila de C es de $p-1$ bloques de $n \times (n/p)$ valores. El total de bloques movidos por el conjunto de nodos para computar toda la matriz es:

$$p \times (p-1) = p^2 - p \tag{4.1}$$

valor que será considerado más adelante.

Se presenta a continuación un algoritmo alternativo basado en cómputos parciales, razón que fundamenta el nombre de esta sección. Suponga que el número de nodos p es un número cuadrado (esta restricción será eliminada más adelante), e inicialmente será tomado como ejemplo un valor de cuatro. La distribución de datos difiere respecto al caso anterior, es decir, se aplica otra forma de dividir los datos entre los nodos componentes del *cluster*.

La partición de datos, que es igual para las matrices A, B y C, resulta de dividir cada matriz en p bloques cuadrados de $(n/\sqrt{p})$ filas por $(n/\sqrt{p})$ columnas, es decir, cada matriz es dividida en $\sqrt{p} \times \sqrt{p}$ bloques, dos filas por dos columnas para el ejemplo con $p = 4$, como puede verse en la Fig.(4.5), llamando a esta distribución *Square Data Distribution* [VB05].

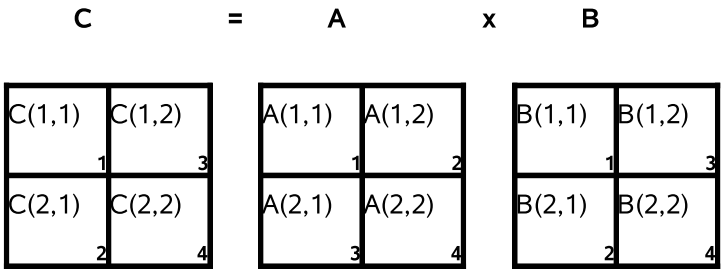


Figura 4.5: Distribución de datos para mutiplicación de matrices bajo "*Partial Computing*"

La distribución de los bloques de datos entre los nodos es tal que cada nodo tiene solo un bloque de A, de B y de C. La Fig.(4.5) muestra el esquema de distribución para las tres matrices. En dicha figura, el número que indicado en la esquina inferior derecha de cada sub-bloque señala su nodo poseedor. Puede verse que el esquema de distribución de B es el mismo que el de C. El esquema de distribución de A es diferente en cuanto a que los números de nodos reflejan la traspuesta de los números de nodos de la distribución de B y C, de forma tal que el nodo poseedor del bloque $C_{i,j}$ también posee el bloque $B_{i,j}$ y el bloque $A_{j,i}$. La importancia de este formato de distribución reside en que minimiza el número de envíos de bloques, como podrá verse mas adelante.

Para el caso del algoritmo de comunicaciones colectivas, cada sub-bloque resultante de la matriz C, es el resultado de un único proceso de multiplicación de una banda fila de A, por una banda columna de B, $C_{i,j} = A_i \times B_j$. Aquí, $C_{i,j}$ es el resultado de $\sqrt{p}$ procesos de cómputo parcial:

$$C_{i,j} = \sum_{k=1}^{\sqrt{p}} A_{i,k} \times B_{k,j}$$

donde ahora, para computar $C_{i,j}$, hay una carga adicional, la cual es realizar la suma de los resultados parciales $A_{i,k} \times B_{k,j}$. Dicha suma la hace el nodo poseedor del bloque $C_{i,j}$ a medida que los resultados parciales son recibidos.

Como se dijo anteriormente, en el presente algoritmo cada bloque de datos en las matrices A, B y C, tiene $(n/\sqrt{p} \times n/\sqrt{p}) = n^2/p$ valores, por lo que tienen la misma cantidad de valores que tiene cada bloque del algoritmo con *broadcast*, tanto para las bandas filas como para las bandas columnas. En aquel algoritmo el total de valores que forman el bloque enviado es $n \times (n/p) = n^2/p$, pudiendo entonces considerar que el movimiento de un bloque para el primer algoritmo tiene la misma carga en volumen de datos que para el algoritmo actual. Esto permite que pueda compararse el movimiento de datos de ambos algoritmos a nivel de movimientos de bloques.

El algoritmo se completa realizando los cómputos $A_{i,k} \times B_{k,j}$, para lo cual se debe enviar el bloque $A_{i,k}$ desde el nodo donde reside hacia el nodo dueño del bloque $B_{k,j}$, éste realiza la multiplicación a nivel de bloques, y finalmente envía el resultado al nodo dueño del bloque $C_{i,j}$, para que éste último realice la suma de bloques con los cómputos parciales.

La cantidad de movimientos de bloques necesarios para realizar el computo final, manteniendo el caso en que $p = 4$, es:

- Para cada bloque en la diagonal principal de C, como la distribución de los nodos responde al esquema de una trasposición, solo un bloque necesita ser movido. Por ejemplo, para el caso del bloque $C_{1,1}$:

  $$C_{1,1} = A_{1,1} \times B_{1,1} + A_{1,2} \times B_{2,1}$$

  por lo que se necesita computar $A_{1,1} \times B_{1,1}$, hecho por el nodo 1 sin necesidad de ningún movimiento de datos, y computar $A_{1,2} \times B_{2,1}$, hecho por el nodo 2, sin movimientos tampoco, y sumar ambos resultados en el nodo 1, el dueño del bloque $C_{1,1}$. Entonces, solo una tarea de comunicación de un bloque es necesario realizar, desde el nodo 2 hacia el nodo 1, para transmitir el resultado del cómputo parcial de $A_{1,2} \times B_{2,1}$. En general, es posible probar que el número de envíos de bloques para el cómputo de cada bloque de la diagonal principal de C es $\sqrt{p} - 1$.

- Para bloques fuera de la diagonal principal de C, por ejemplo el caso del bloque $C_{2,1}$, se tendrá que:

  $$C_{2,1} = A_{2,1} \times B_{1,1} + A_{2,2} \times B_{2,1}$$

  por lo que se necesita computar $A_{2,1} \times B_{1,1}$, hecho por el nodo 1, y $A_{2,2} \times B_{2,1}$, hecho por el nodo 2, y finalmente el nodo 2, el dueño del bloque $C_{2,1}$, sume ambos resultados. El número total de bloques

movidos para bloques de C fuera de la diagonal principal, es de tres para este caso. Generalizando, también puede demostrarse que el total de bloques movidos para este caso es $2 \times (\sqrt{p} - 1) + 1$.

El número total de bloques movidos para todo el procesamiento es la suma de los bloques movidos para ambos tipos de casos: para el primero hay $\sqrt{p}$ bloques en la diagonal principal, mientras que para el segundo caso hay $p - \sqrt{p}$ casos. Entonces, el número total (tn) es:

$$\begin{aligned} tn &= \sqrt{p} \times (\sqrt{p} - 1) + (p - \sqrt{p}) \times (2 \times (\sqrt{p} - 1) + 1) && (4.2) \\ &= 2p \times (\sqrt{p} - 1) && (4.3) \end{aligned}$$

lo cual es inferior a $p^2 - p$ determinado para el primer algoritmo. En el ejemplo, se tiene 12 envíos para comunicaciones colectivas contra 8 para este algoritmo. Ahora puede verse la importancia de la trasposición en el número de los nodos dueños de los bloques de A, ya que esto permite minimizar la cantidad de movimientos de bloques hecho bajo el algoritmo de *partial computing*. De no realizar esta transposición, el número de envíos de bloques es mayor y el algoritmo pierde eficiencia.

El menor número de bloques comunicados entre nodos que impone este algoritmo, tiene como contrapartida el hecho que el nodo receptor del cómputo parcial debe hacer el cómputo adicional de sumar los resultados parciales, hecho que no es necesario para el algoritmo de comunicaciones colectivas. El costo computacional de una operación de suma de matrices es relativamente bajo y los resultados finales de ejecución son positivos a pesar de esta carga adicional, como se verá más adelante, en tabla 4.7.

Relajando la restricción de que el número total de nodos p sea un número cuadrado, lo que se debe hacer es factorizar p, como $p = r \times c$, de forma tal que la diferencia absoluta entre r y c sea la mínima posible, y $r < c$. Ahora A es dividida en $r \times c$ bloques, B en $c \times r$ bloques y C en $r \times r$ bloques, como puede verse en la Fig.(4.6), graficando el caso de 6 nodos, por lo que $p = 6$, $r = 2$ y $c = 3$.

La distribución de datos entre los nodos para las matrices A y B tienen el mismo criterio que para el caso en que p es un número cuadrado: la numeración de los nodos poseedores de los bloques refleja una operación de trasposición. Hay una diferencia en el criterio para la matriz C. Esta matriz tiene un número de bloques $r^2 < r \times c$, por lo que, a mayor r, bajo la restricción $r < c$, más cercano al caso de p cuadrado se está. Es fácil observar que en este caso, no todos los nodos son poseedores de un bloque de C: esta matriz

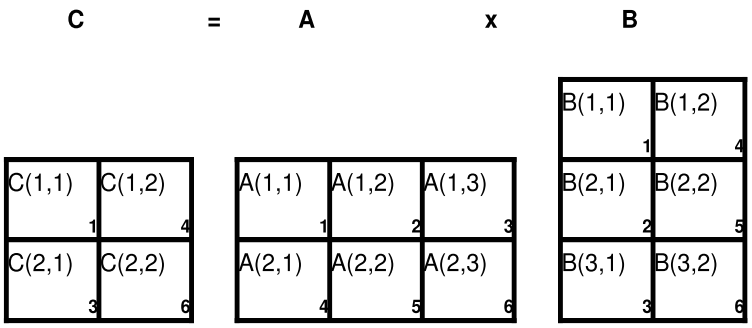


Figura 4.6: Distribución de datos para multiplicación de matrices bajo "*Partial Computing*" cuando el número de nodos procesadores es un número no cuadrado.

es dividida en menos bloques que A y B, por lo que se plantea entonces el problema de cómo asignar la distribución de los bloques de C entre los nodos del *cluster*.

Manteniendo el criterio de que los datos son enviados desde el nodo dueño del bloque de A hacia el nodo dueño del respectivo bloque de B para ejecutar el procesamiento parcial, y que éste resultado es luego enviado al nodo dueño del respectivo bloque de C para hacer la acumulación de los resultados parciales, la distribución de los datos de C, que minimice las comunicaciones, debe tener un esquema de distribución similar que el de B, salvo por el hecho que el número de filas de B es mayor al de C. Entonces, se mantiene para C la misma distribución por filas que para B, salvo para la última, que toma la misma distribución de la última fila de B; dicho de otra forma, es como si se dejara de lado las últimas $c - r$ filas de B salvo la última, que se mantiene, para distribuir los bloques de C. En el ejemplo de Fig.(4.6), C tiene la misma distribución de B salvo por la segunda fila de B.

Este esquema de distribución de datos entre los nodos genera que se mantenga el mismo procedimiento de conteo de bloques enviados que para el caso en que p es cuadrado, es decir, $c - 1$ envíos de bloques para bloques de la diagonal principal y $2 \times (c - 1) + 1$ envíos de bloques para los restantes bloques. Es fácil determinar que, a pesar de tener en este caso un número de envíos mayor al caso en que p es cuadrado, es menor que para el algoritmo de comunicaciones colectivas y distribución bandeada de datos. Mientras menor sea la diferencia entre r y c, más próximo se está al caso donde p es un número cuadrado.

En la cuadro 4.7 se presentan los resultados de ejecución del algoritmo con cómputos parciales junto con el de comunicaciones colectivas (*broadcast*) para varios rangos de matriz. El test ha sido ejecutado el *cluster* 3 de página 51, haciendo uso de hasta 64 nodos, utilizando el modelo híbrido de programación de *clusters* indicado anteriormente. La red usada para comunicaciones entre los nodos es Infiniband. Se utilizó el compilador Intel, la biblioteca de álgebra lineal MKL y openMPI como biblioteca de comunicaciones, y se configuró el número de *threads* a utilizar por la biblioteca MKL en ocho.

Nodos (1)	**Rango Matriz (2)**	**Bcast (secs.) (3)**	**Part. Comp. (secs.) (4)**	**Mejora (4)/(3)**	**Promedio mejora**
4	10800	7.117	6.799	0.955	
4	14400	13.785	13.950	1.012	
4	18000	26.431	26.292	0.995	0.975
9	10800	4.133	3.624	0.877	
9	14400	7.748	7.023	0.906	
9	18000	13.234	12.742	0.963	0.920
16	10800	3.882	2.594	0.668	
16	14400	5.907	4.591	0.777	
16	18000	9.874	7.982	0.808	0.738
25	10000	2.708	1.835	0.678	
25	15000	5.453	3.850	0.706	
25	20000	9.816	7.093	0.723	0.700
36	9000	3.226	1.448	0.449	
36	18000	7.521	4.419	0.588	
36	27000	17.673	11.337	0.641	
36	36000	32.937	23.689	0.719	0.584
49	14700	5.228	3.001	0.574	
49	24500	12.967	8.821	0.680	
49	34300	27.228	18.693	0.687	0.630
64	19200	10.171	4.042	0.397	
64	25600	13.089	6.624	0.506	
64	32000	21.382	10.988	0.514	
64	38400	31.232	17.434	0.558	0.478

Tabla 4.7: Comparación de tiempos para los algoritmos utilizando *broadcast* y *Partial Computing* para multiplicación de matrices

Como puede verse en la tabla donde se exponen los tiempos para ambos algoritmos, cuando el número de nodos utilizados en el *cluster* es bajo, los resultados comparativos entre ambos son similares, pero a medida que crece el número de nodos, la diferencia a favor del algoritmo con cómputos parciales se va incrementando, como queda en evidencia para el caso de 64 nodos. Aplicando las fórmulas de las Ec.(4.1) y (4.3) referidas a la cantidad de bloques movidos en ambos algoritmos, para el caso en que $p = 64$, puede verse que para la primera se obtiene el valor de 4032, mientras que para la segunda, 448. Si bien éste no es el único argumento por el cual puede explicarse la diferencia de tiempos entre ambos algoritmos, la diferencia de cantidad de bloques movidos es significativa.

El algoritmo de esta sección reconoce sus raíces en algoritmos preexistentes de álgebra lineal, donde los procesadores son distribuidos en una grilla 2-D. La diferencia es que en aquellos [DPRV08], el fundamento de la distribución es la maximización de la localidad de datos y por ende, de la reutilización de los mismos. En el presente caso, se hace desde la perspectiva de minimizar el volumen de comunicaciones utilizadas.

4.2. Factorización de Cholesky

El segundo de los algoritmos utilizados en las experimentaciones corresponde al algoritmo de factorización de Cholesky, de amplia difusión en la resolución de sistemas de ecuaciones lineales [GL96, Pre07]. Dicho algoritmo tiene la finalidad de factorizar una matriz A, cuadrada, simétrica y definida positiva, de rango n de la siguiente forma:

$$A = L \times L^T$$

donde L es una matriz triangular, lo cual, dado un sistema de ecuaciones

$$A \times x = b$$

lo transforma en un sistema equivalente:

$$L \times y = b$$

donde $y = L^T \times x$. El sistema se resuelve, resolviendo primero para y y luego para x, lo cual computacionalmente es más simple dada la condición triangular de L.

Para determinar los valores de la matriz L, suponiendo que es triangular inferior, se aplican las siguientes fórmulas:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \qquad 1 \leq j < i \leq n \qquad (4.4)$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \qquad 1 \leq i \leq n \qquad (4.5)$$

donde la fórmula de la Ec.(4.5) se aplica para los elementos de la diagonal principal de L y la de Ec.(4.4) para los restantes elementos.

Este algoritmo ha sido seleccionado para experimentar en lo referido a la aplicación de patrones de paralelismo dado que tiene una dependencia de datos, a diferencia de la multiplicación de matrices que no la tiene, y que está implementado por las bibliotecas de rutinas de álgebra lineal(LAPack) por medio de la rutina xpotrf, (donde la x se reemplaza por s o d según se quiera utilizar *single* o *double* precisión), la cual también está optimizada para entornos *multicore*, de forma tal de tener estas implementaciones como punto de referencia para nuestros experimentos.

Las ecuaciones (4.4) y (4.5) imponen un orden de computación tal que para calcular los elementos de una fila i, primero deben computarse los valores l_{ij} con $j < i$ y luego el elemento de la diagonal principal l_{ii}. A su vez, para computar cada l_{ij} deben primero estar calculados todos los valores de la fila i hasta la columna $j-1$ y todos los valores de la fila j hasta la columna j, lo cual impone una fuerte dependencia de datos. Sin embargo esta dependencia deja ciertos grados de libertad en cuanto al orden en que pueden desarrollarse las tareas.

Aplicando el patrón de división de tareas al algoritmo, considerando que el cómputo de cada valor l_{ij} como una tarea[1], puede determinarse el pseudo-algoritmo de la siguiente tabla y gráficamente en la Fig.(4.7) para el supuesto que $n = 5$.

[1]Se puede considerar que l_{ij} hace referencia a un valor individual o a un bloque cuadrado de la matriz L.

1	$i = 1$
2	mientras $(i \leq n)$
3	calcular l_{ii}
4	calcular los valores de la columna i para las filas $k, k = i + 1 \ldots n$
5	$i = i + 1$
6	volver a 2

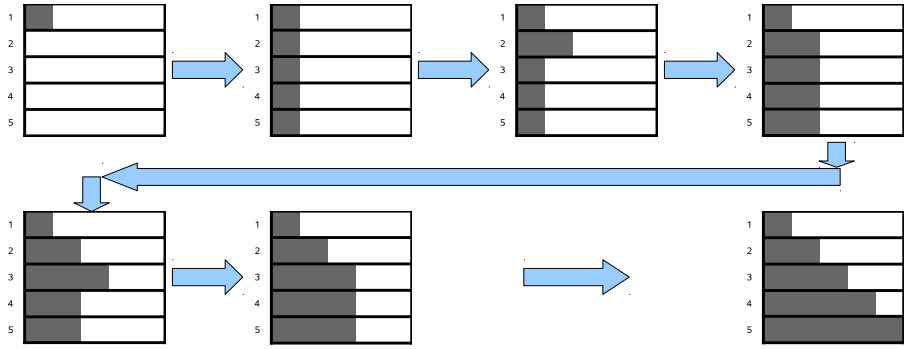


Figura 4.7: Avance de tareas para el algoritmo de Cholesky para $n = 5$

4.2.1. Memoria Compartida

Al igual que se hizo para el algoritmo de multiplicación de matrices, para la optimización del algoritmo en memoria compartida, se realizaron algunos *tests* sobre la rutina que calcula el algoritmo perteneciente a la biblioteca LAPack. Perteneciente a dicha biblioteca, la rutina `spotrf` permite calcular la factorización de Cholesky sobre una matriz simétrica, definida positiva, para números reales de simple precisión.

Como se planteo anteriormente, algunas implementaciones de la biblioteca LAPack, con solo indicar que se trabaja bajo un ambiente de memoria compartida bajo tecnología OpenMP, el cómputo se paraleliza entre los *cores* que se determinen ejecutar, por cuanto se procedió a realizar experimentos destinados a obtener el *speedup* que se logra al configurar la ejecución de ésta forma.

El experimento consistió en hacer una llamada a la rutina `spotrf` configurada para trabajar con 1,2,4 y 8 *cores*, para diferentes tamaños de matriz, y a su vez, repetido en dos *clusters*, con diferentes implementaciones de LAPack. Los resultados pueden verse en la tabla 4.8. La columna de tiempos con la implementación de LAPack de la empresa Sun (Sunperf) fue ejecutada

Rango Matriz	*Threads*	Tiempo Sunperf	*Speedup*	Tiempo MKL	*Speedup*
2000	1	0.330	1.000	0.184	1.000
2000	2	0.187	1.765	0.104	1.769
2000	4	0.103	3.204	0.061	3.016
2000	8	0.057	5.789	0.045	4.089
4000	1	2.561	1.000	1.345	1.000
4000	2	1.437	1.782	0.676	1.990
4000	4	0.748	3.424	0.358	3.757
4000	8	0.401	6.387	0.205	6.561
8000	1	20.916	1.000	10.156	1.000
8000	2	11.788	1.774	5.063	2.001
8000	4	6.235	3.355	2.552	3.980
8000	8	3.439	6.082	1.369	7.419
12000	1	70.771	1.000	33.387	1.000
12000	2	37.339	1.895	16.608	2.010
12000	4	20.579	3.439	8.382	3.983
12000	8	10.782	6.564	4.430	7.537

Tabla 4.8: Tiempos de ejecución en segundos para la factorización de Cholesky bajo memoria compartida con Sunperf y MKL.

en un nodo del *cluster* 2 (pag 51), mientras que el experimento utilizando lo biblioteca de la empresa Intel, MKL, fue ejecutado en un nodo del *cluster* 3.

A partir de estos resultados puede concluirse que para tamaños de matrices menores el *speedup* es relativamente mejor para 4 *threads* que para 8, sin embargo, con tamaños de matrices más grandes, el *speedup* con 4 y 8 *threads* es cercano al óptimo, por lo que, al igual que para el caso de multiplicación de matrices, se puede afirmar que, dentro del modelo de memoria compartida bajo tecnología OpenMP, si se dispone de una rutina que implemente el algoritmo que se intenta paralelizar, es decir una implementación de LAPack optimizada para memoria compartida, con solo invocar la ejecución de dicha rutina con el parámetro adecuado en lo referido a la utilización de los múltiples *cores*, el algoritmo alcanza un *speedup* cercano al óptimo. Al igual que para el caso de multiplicación de matrices, la atención debe enfocarse en cómo paralelizar en memoria distribuida.

4.2.2. Memoria Distribuida - Comunicaciones colectivas

Como en todo proceso de paralelización de un algoritmo bajo memoria distribuida, la primera tarea a realizar es la división de tareas y la distribución de datos entre los nodos que componen el *cluster*. Tomando como punto de partida la división de tareas expuesta en la Fig.(4.7), resta determinar la distribución de datos para poder definir el algoritmo distribuido. En ese sentido, puede verse que la mejor forma de distribuir datos es dividir la matriz A en bandas filas, tantas como nodos conformen el *cluster*, de forma tal que si el rango de la matriz es n y existen p nodos, cada banda tiene n/p filas por n columnas.

El algoritmo distribuido tiene la siguiente secuencia: realizando p pasos de iteración, en la iteración i, el nodo i computa la parte de la diagonal principal de A que posee, luego el resultado es enviado a los restantes nodos para que computen la respectiva parte de la columna i que poseen. Esto posibilita realizar en paralelo el cómputo de la columna i por parte de los nodos respectivos. El cómputo puede hacerse con llamadas a rutinas de las bibliotecas BLAS y LAPack, las cuales, como se ha visto, están optimizadas para entornos *multicore*. En particular, se utilizan las rutinas `spotrf` y `ssyrk` para el cómputo del bloque en la diagonal principal, y, `sgemm` y `stsrm`, para el cómputo de los bloques de la columna i.

La secuencia de pasos del algoritmo es exhibida en la siguiente tabla:

Paso	En el nodo i	En nodo j, con $j > i$
1	for (var $i = 1$, lim p, incr 1)	
2	computo de bloque en la diag. principal mediante llamada a ssyrk y spotrf	
3	broadcast de la banda i	recibe la banda i
4		computa su parte de la columna i usando sgemm y stsrm
5	volver a 1	volver a 1

La división de tareas planteada determina que el cómputo del elemento de la diagonal principal es hecha por un solo nodo, debiendo los restantes estar a la espera de este resultado para poder proseguir con su cómputo. Esta dependencia de datos genera un tiempo ocioso por parte de los nodos que no realizan el cómputo del elemento de la diagonal principal, lo cual provoca la caída del *speedup* que puede lograrse. Otro factor que atenta contra el

speedup está dado por la forma triangular de L que determina que una vez computadas las primeras bandas-fila, los nodos que las poseen dejan de tener actividad de cómputo de allí en adelante, por lo que esta distribución de tareas genera un marcado desbalance de carga.

El efecto del desbalance de carga del algoritmo con *broadcast* es exhibido en la Fig.(4.8) donde se presenta la traza de ejecución de este algoritmo, obtenidas con la misma herramienta (*Analyzer*) utilizada anteriormente, para el caso en que $p = 6$. Las áreas blancas corresponden a tiempos utilizados para cómputo, y las celestes (grises), a tiempos de comunicación. Como puede verse, el nodo 1 al terminar de computar su bloque en la diagonal principal ya no tiene más tareas que realizar. El mismo patrón de comportamiento ocurre en los restantes nodos. Solo el nodo 6, el último, tiene carga de trabajo a lo largo de todo el algoritmo. Es evidente que este algoritmo tiene un pésimo balance de carga, existiendo propuestas para mejorarlo [TR05]. Sin embargo se lo mantendrá a los fines de comparaciones con otras implementaciones paralelas.

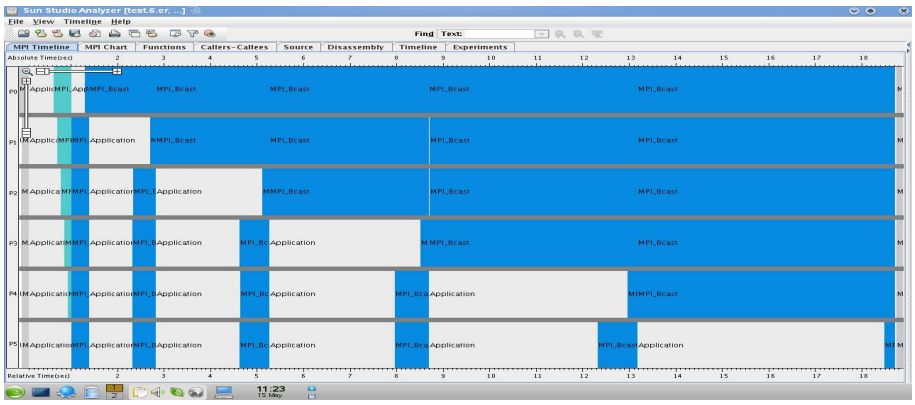


Figura 4.8: Traza de ejecución del algoritmo distribuido con *broadcast* para la factorización de Cholesky para $p = 6$

Los tiempos de ejecución de los experimentos con este algoritmo se exponen en la tabla 4.9. Adicionalmente al algoritmo con *broadcast* se experimentó con la versión ScaLAPACK del algoritmo de Cholesky. ScaLAPACK (*Scalable Linear Algebra package*[Sca]) es la biblioteca de referencia para rutinas de álgebra lineal en paralelo, una implementación paralela de la tradicional biblioteca LAPack. Ha sido diseñada para correr en computadoras bajo el modelo de memoria distribuida y utiliza paso de mensajes (MPI) para la comunicación entre los procesadores. Estos se distribuyen en un formato de una grilla 2-D y los datos son divididos en forma de bloques cuadrados y distribuidos sobre la grilla en un formato cíclico de bloques para lograr un

Rango Matriz	**Scalapack nb=64**	**Scalapack nb=256**	**Broadcast**
12000	13.47	16.69	13.53
18000	40.52	64.05	38.08
24000	92.40	161.64	82.75

Tabla 4.9: Tiempos de ejecución en segundos del algoritmo de Cholesky, con *broadcast* y la versión Scalapack, sobre *cluster* 1. Scalapack lanzado con una grilla de 32 procesadores MPI, y *broadcast* con 8 procesos MPI × 4 *threads* OpenMP en cada uno.

buen balance de carga, escalabilidad y reutilizar al máximo los datos en cada procesador[CDPW92].

Para un *cluster* en particular, con p procesadores, el usuario de ScaLAPACK debe configurar los valores r y c, tal que $p = r \times c$, en otras palabras, es necesario definir el número de filas y columnas que conforman la grilla de procesadores. También debe definirse el tamaño del bloque de datos, nb, el cual impacta en la optimización del balance de carga y la reusabilidad de los datos, acorde al tamaño de memoria y de la *cache* que tengan los nodos procesadores en cada caso en particular. En los experimentos se definió una grilla de 32 procesos, de 8 filas por 4 columnas, y se probaron diversos tamaños de bloques, hallándose el óptimo para un tamaño de $nb = 64$.

El diseño de ScaLAPACK está orientado al balance de carga y la reusabilidad de datos. A nuestro entender, dado la actual conformación de *clusters* de nodos *multicore*, la implementación de dicho biblioteca tiene una falencia referente a que no ha utilizado el modelo híbrido de memoria compartida y distribuida: utiliza solo paso de mensajes entre procesos MPI. De esta forma, cada *core* de un procesador *multicore* debe ser considerado como un nodo procesador independiente generando comunicaciones MPI entre procesos internos de un mismo equipo, lo cual provoca competencia por la única memoria *cache* del equipo. Las observaciones oportunamente planteadas sobre las ventajas de la utilización del modelo de programación híbrido OpenMP - MPI, de la sección 2.4, son pertinentes en este punto.

Volviendo a los experimentos, puede verse en la tabla 4.9 que a pesar de la mala distribución de carga del algoritmo de *broadcast*, los tiempos de ejecución son similares a los de ScaLAPACK, por lo que puede concluirse que dividir los datos y tareas para alcanzar un buen balance de carga, como se aconseja en el *Design Evaluation Pattern* de página 32, al menos para este

caso, impone un esquema de comunicaciones que aumenta los tiempos finales de ejecución, por lo que no termina siendo una buena solución al problema del rendimiento.

4.2.3. Memoria Distribuida - Uso de cómputos parciales

Similar a lo sucedido para el algoritmo de multiplicación de matrices, en la búsqueda de opciones de distribución de tareas que permitan una mejora en el rendimiento del algoritmo distribuido, se hicieron pruebas con la utilización de cómputos parciales con el objetivo de mejorar el balance de carga, la utilización de tiempos de espera de comunicaciones solapado con cómputos, y por ende, obtener un impacto positivo en el rendimiento final del algoritmo. La estrategia de cómputo es la misma que se expuso al comienzo de la sección respectiva para el algoritmo de multiplicación de matrices (sección 4.1.5).

Se plantea un ejemplo para el algoritmo de Cholesky de cómo realizar cómputos parciales. Se usará una matriz de rango 8. Teniendo en cuenta la Ec.(4.4) para calcular los valores de la parte inferior de la matriz L, al expandir el lado derecho de la ecuación para el valor $l_{8,7}$, se obtiene:

$$l_{8,7} = \left(a_{8,7} - l_{8,1} \times l_{7,1} - l_{8,2} \times l_{7,2} - \ldots - l_{8,5} \times l_{7,5} - l_{8,6} \times l_{7,6}\right) / l_{7,7} \quad (4.6)$$

Considérese también el supuesto de que la segunda iteración a concluido, por lo que, a este momento, los valores disponibles en la Ec.(4.6) para computar $l_{8,7}$ son $a_{8,7}$ y $l_{8,1}, l_{7,1}, l_{8,2}$ y $l_{7,2}$, por lo que parte de la diferencia planteada en dicha fórmula puede ser calculada. Genéricamente, si la iteración h del algoritmo ha concluido, $1 \leq h \leq j-1$, todos los valores de las columnas hasta h ya han sido computados, por lo que la fórmula para l_{ij} puede ser desplegada como:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{h} l_{ik} \times l_{jk} - \sum_{k=h+1}^{j-1} l_{ik} \times l_{jk}\right) / l_{jj} \qquad 1 \leq h \leq j-1 \quad (4.7)$$

donde todos los valores del primer sumatorio (hasta h) ya han sido computados y los valores del segundo sumatorio están pendientes de calcular. Siguiendo la distribución de datos entre los nodos del *cluster* en forma de bandas filas, si el procesador i está ocioso, éste podría computar el resultado del primer sumatorio de la Ec.(4.7) con solo recibir desde el nodo j la parte respectiva de la fila j, desde la columna 1 hasta la h.

Por otro lado, *partial computing* también puede aplicarse sobre la fórmula que define los valores de la diagonal principal, Ec.(4.5), y en este caso, sin

necesidad de comunicaciones adicionales. Puede verse que en dicha fórmula solo están involucrados datos de una misma fila, por lo que dada nuestra distribución de datos, puede realizarse cómputos parciales sin agregar comunicaciones.

Al igual que para la fórmula de los valores inferiores de L, la existencia de un sumatorio en la fórmula es lo que posibilita aplicar cómputos parciales. Puede verse que el término cuadrático del sumatorio en la Ec.(4.5) puede desplegarse para la fila i, considerando que la iteración h ha concluido, como el sumatorio de valores ya calculados más el sumatorio de los pendientes:

$$\sum_{k=1}^{i-1} l_{ik}^2 = \sum_{k=1}^{h} l_{ik}^2 + \sum_{k=h+1}^{i-1} l_{ik}^2 \tag{4.8}$$

donde los valores l_{ik}, para $k = 1 \ldots h$ del lado derecho de la ecuación ya han sido computados, y los valores para $k = h+1 \ldots i-1$ están pendientes, por lo que el primer sumatorio del lado derecho de la fórmula puede calcularse por adelantado.

Dada la existencia de la posibilidad de aplicar cómputos parciales, resta determinar el momento donde es conveniente para hacerlo, es decir, determinar la oportunidad donde el procesador queda inactivo. Dicha oportunidad surge, suponiendo finalizada la iteración $k-1$, cuando los procesadores p con $p = k+1 \ldots n$ están a la espera del bloque l_{kk} en proceso de cómputo por parte del procesador k, es decir, en el paso tres de la tabla que presenta el algoritmo con comunicaciones colectivas. En dicho momento, los procesadores p, con $p = k+1 \ldots n$, están ociosos y pueden computar el termino l_{pk-1}^2 indicado en Ec.(4.8) y restarlo de los resultados acumulados previos en el bloque de la diagonal principal respectiva (a_{pp}), como se indica en la Ec.(4.5).

Se desprende de lo analizado hasta el momento, que existen dos posibilidades de aplicar "cómputos parciales" a la factorización de Cholesky, uno por cada una de las fórmulas que lo determina. Este hecho motivo que se realicen dos experimentos de la aplicación de *partial computing* sobre dicho algoritmo, el primero aplicando solamente sobre la fórmula de los valores de la diagonal principal, y el segundo, agregando al primero, la aplicación sobre elementos de la parte triangular inferior de L.

El primer experimento realizado, de aplicar *partial computing* sobre los valores de la diagonal principal es relativamente sencillo de implementar, ya que, como fue dicho anteriormente, no requiere comunicaciones adicionales y fue denominado como *Diagonal Partial Computing*. El algoritmo puede verse expresado en la siguiente tabla, que es igual a la del algoritmo con *broadcast*,

solo modificando el paso 2 para los nodos j:

Paso	Nodo i	Nodo j, con $j > i$
1	for (var $i = 1$, lim p, incr 1)	
2	computo de bloque en la diag. principal mediante llamada a ssyrk y spotrf	computar l^2_{ji-1} y restar de a_{jj} llamada a ssyrk
3	broadcast de la banda i	recibe la banda i
4		computa su parte de la columna i invocando a sgemm y stsrm
5	volver a 1	volver a 1

El segundo experimento, denominado *Full Partial Computing*, se hizo agregándole al anterior algoritmo, la aplicación de *Diagonal Partial Computing* sobre los valores de la parte inferior de la matriz L. Este algoritmo es más complejo de implementar debido a la necesidad de comunicaciones adicionales que impone. Existen, a su vez, muchas posiciones donde puede aplicarse y si se aplicaran en todas, la carga de comunicaciones se torna compleja. La Ec.(4.7) muestra que las posiciones en las últimas filas y columnas de L poseen más valores involucrados en su cómputo, por lo que son mejores candidatas a realizar cómputos por adelantado.

Los factores indicados en el párrafo anterior motivaron que se decida simplificar la implementación de este *test*, y solo como una cuestión experimental, aplicar *partial computing* en una sola posición de la parte inferior de L, la posición del penúltimo bloque de la última fila, es decir el bloque l_{pp-1}, para determinar si tiene algún efecto positivo sobre los tiempos utilizados. De acuerdo a la Ec.(4.4), y teniendo en cuenta que hay p nodos procesadores, dicho valor se computa como:

$$l_{pp-1} = \left(a_{pp-1} - \sum_{k=1}^{p-2} l_{pk} \times l_{p-1k} \right) / l_{p-1p-1} \tag{4.9}$$

por lo que los valores involucrados pertenecen a la última y ante-última filas, y el cómputo parcial se hace sobre el producto que contiene el sumatorio a medida que cada iteración va finalizando y por ende, la respectiva columna queda computada.

Lamentablemente, como puede comprobarse en la Fig.(4.8), los nodos $p-1$ y p son los que menos tiempo ocioso tienen, por lo que no son candidatos a realizar el cómputo por adelantado. Adicionalmente, el primer nodo es el

Rango Matriz	Bcast Eth.	Diag Part Comp Eth.	Full Part Comp Eth.	Bcast IB.	Diag Part Comp IB.	Full Part Comp IB.
12000	10.99(1.00)	9.41(.856)	8.91(.811)	7.29(1.00)	5.82(.798)	5.21(.715)
18000	30.53(1.00)	25.73(.843)	23.74(.778)	22.42(1.00)	17.56(.783)	15.77(.703)
24000	62.95(1.00)	52.03(.827)	47.84(.760)	48.48(1.00)	37.50(.773)	33.87(.699)

Tabla 4.10: Tiempos de ejecución (en segundos) de los tres algoritmos de Cholesky experimentados, usando Ethernet e Infiniband, sobre *cluster* 2 usando 6 nodos.

que está prácticamente todo el algoritmo sin uso, por lo que se decidió que éste sea el nodo que realiza el cómputo por adelantado. Esto, sin embargo, implica que los datos de las filas $p-1$ y p deben transmitirse hacia el nodo 1 para poder multiplicarse, y el resultado, ser enviado desde dicho nodo hacia el nodo p.

La implementación del presente algoritmo se logró agregando al algoritmo de *broadcast*, los pasos que posibilitan que los nodos p y $p-1$ envíen los bloques l_{ph} y l_{p-1h} hacia el nodo 1 cada vez que la iteración h finaliza, para que este realice $l_{nh} \times l_{n-1h}$ y envié el resultado al nodo 1. De esta forma, la multiplicación es hecha por adelantado en un nodo ocioso.

Los resultados de los experimentos se exponen en tablas 4.10 y 4.11, mientras que los gráficos de la traza de ejecución de cada experimento son presentados en las Figs. (4.9) y (4.10) respectivamente. En ésta última puede verse que el nodo 1 mantiene actividad luego de computar el bloque l_{11}, cosa que no ocurre para el caso en que no se aplica *Full Partial Computing*, como puede verse en la primera de las figuras. Las mejoras en los tiempos de ejecución trepan hasta un 30 % sobre el algoritmo de *broadcast*, utilizando el *cluster* 2 con 6 nodos, y hasta un 35 % sobre utilizando el *cluster 3* con 8 nodos.

Rango Matriz	Bcast (1).	Diag Comp Bcst (1)	Full Part Comp (1)	Bcast (3)	Diag Comp Bcst (3)	Full Part Comp (3)
12000	16.95(1.00)	12.90(.761)	11.31(.667)	4.77(1.00)	4.34(.901)	3.43(.719)
18000	47.31(1.00)	37.43(.791)	32.79(.693)	13.19(1.00)	10.61(.804)	8.55(.648)
24000	105.17(1.00)	83.86(.797)	73.88(.702)	27.20(1.00)	21.83(.802)	17.65(.649)

Tabla 4.11: Tiempos de ejecución (en segundos) de los tres algoritmos de Cholesky experimentados, usando Infiniband, sobre *cluster* 1 usando 6 nodos, 4 *cores* por nodo y también sobre *cluster* 3 usando 8 nodos, 8 *cores* por nodo.

Se reconoce que esta implementación de *Full Partial Computing* debería ser general y alcanzar a todos los nodos que puedan realizar cómputos luego de terminadas sus tareas, es decir, que el nodo i, luego de la iteración i, comience a realizar cómputos adelantados correspondientes a la fórmula 4.4. Implementar la aplicación general es dejado para futuras investigaciones dado que el código se vuelve muy complejo, ya que involucra muchas comunicaciones adicionales y se convierte en un código paralelo con estructura MIMD (*Multiple Instruction Multiple Data*). Esto no le resta validez al objetivo de la experimentación, en lo referente a la aplicabilidad de los patrones de paralelismo.

De todas formas, pudo comprobarse que, como se viene sosteniendo, los algoritmos con esquemas complejos de comunicaciones "punto a punto", son relativamente ineficientes cuando la red que soporta el *cluster* es *Ethernet*, mientras que si se utiliza Infiniband, los resultados son como mínimo iguales o bien mejores a los algoritmos que utilizan comunicaciones colectivas, acorde a lo que muestra la tabla 4.10.

Adicionalmente, de los resultados positivos obtenidos aplicando *Partial Computing*, puede deducirse que la optimización de los algoritmos está muy ligado a la estructura que éste tenga, al grado de detalle que se alcance en la distribución de datos y tareas, y fundamentalmente, a la capacidad que tenga el equipo de desarrollo de aplicar *Reengeneering for Parallelism Pattern*, desarrollado en la sección 3.2.4.1, para determinar los *hot spots* que el algoritmo paralelo tenga y puedan ser optimizados.

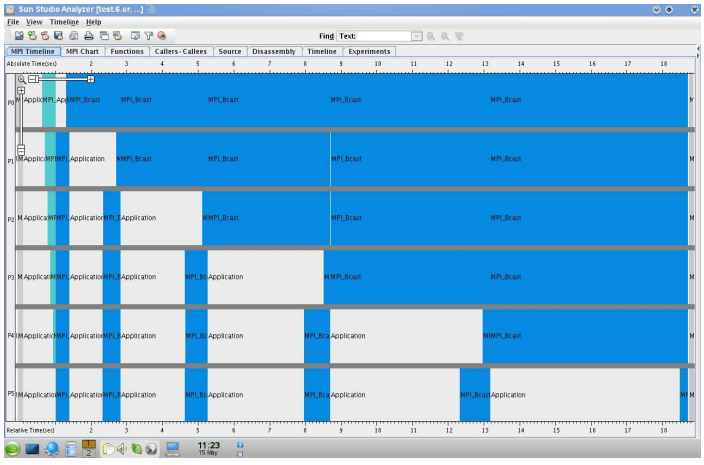


Figura 4.9: Traza de ejecución del algoritmo de Cholesky utilizando *Diagonal Partial Computing* con 6 nodos y n=18000.

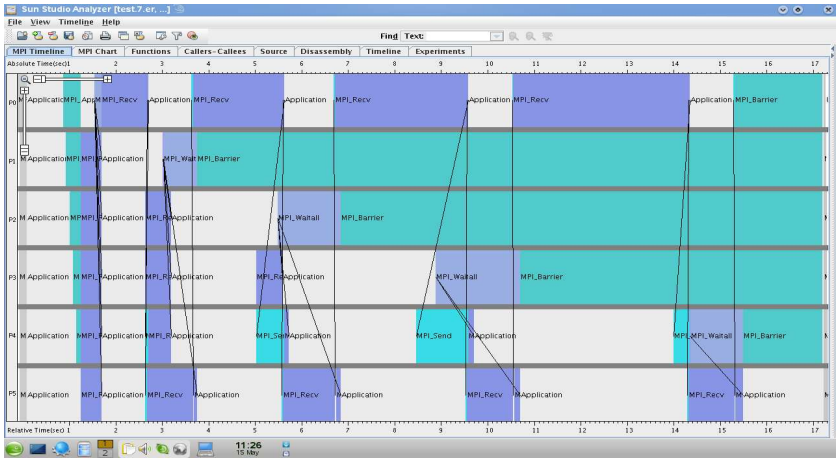


Figura 4.10: Traza de ejecución del algoritmo de Cholesky utilizando *Full Partial Computing* con 6 nodos y n=18000.

4.3. Resumen del capítulo

A lo largo de este capítulo han sido presentado numerosos ejemplos de aplicación de patrones de paralelismo a dos problemas clásicos del álgebra lineal: la multiplicación de matrices y la factorización de Cholesky, obteniéndose varios algoritmos que resuelven cada uno de estos problemas, según el patrón aplicado. También se pudo comprobar, que un patrón que a nivel de diseño del algoritmo es desaconsejable de aplicar por la estructura del algoritmo, puede aplicarse y dar rendimientos aceptables, gracias a la arquitectura del *hardware* subyacente. Por lo tanto, la aplicación a nivel de diseño del algoritmo sin una comprobación en ejecución de los patrones no genera resultados directos en cuanto al rendimiento obtenido. Más aun, un mismo patrón tiene resultados positivos o negativos según sea la arquitectura de red que conforma el *cluster*.

Puede destacarse que el resultado del proceso de paralelización de aplicaciones depende fuertemente de la capacidad de determinar *hot spots* donde poner el acento en la paralelización eficiente del algoritmo. Disponer de herramientas de *tracing* y *profiling* apoya esta tarea de forma fundamental, sin las cuales sería casi imposible de realizar. Se convalida fuertemente el contenido del pseudo-patrón *Reengineering for parallelism pattern*, el cual se recomienda tener en cuenta al trabajar en la paralelización de un algoritmo, ya que esto es un proceso de refinamiento iterativo.

Capítulo 5

Patrones de paralelismo para computación de alto rendimiento (HPC)

A lo largo del presente trabajo se han presentado las características del los *clusters* de nodos *multicore* (cap. 2), el estado actual del desarrollo de los patrones de paralelismo (cap. 3) y por último, las experimentaciones realizadas referidas a la aplicación de los patrones de paralelismo sobre los algoritmos de multiplicación de matrices y de factorización de Cholesky (cap. 4).

Este capítulo está dedicado a exponer las características que a nuestro entender deben tener los patrones de paralelismo orientados a la computación de alto rendimiento. Cabe aclarar, que si bien es usual que la acepción "computación de alto rendimiento" puede referirse tanto al HPC y como al HTC, en el entorno de este capítulo, dicha acepción quedará restringida al HPC exclusivamente, el cual es el marco de esta tesis.

Se presentará en primer lugar, la propuesta de modificaciones en la estructura y los contenidos de los actuales patrones de paralelismo para adecuarlos al tipo de problemas bajo estudio en esta tesis. A continuación, y como resultado adicional de las experimentaciones realizadas, se expone una propuesta de un nuevo patrón de paralelismo, de desarrollo propio, llamado "*Partial Computer Pattern*", el se ha sido elaborado acorde a las sugerencias aportadas.

5.1. Propuestas sobre contenido y estructura de los patrones de paralelismo enfocados al HPC

En este punto del desarrollo de la tesis, es manifiestamente evidente la necesidad del desarrollo de patrones de paralelismo para HPC, como un tipo particular de patrones de paralelismo, donde se brinden soluciones al problema de la optimización de los algoritmos paralelos, y que además tenga en cuenta el tipo de plataforma de *hardware* donde se ejecutan los programas paralelos. Esta conclusión surge a partir de lo expuesto en la sección 3.3 donde se realizó el análisis de los patrones de paralelismo. En aquella oportunidad se destacó la ausencia de información en los actuales patrones sobre el tema de rendimientos, como así también, las carencias sobre el impacto de las características particulares de las redes de comunicaciones que componen el *cluster* y las relaciones entre la estructura del algoritmo, la red y el rendimiento.

Por todo ello, si el problema a considerar es el rendimiento de un algoritmo paralelo, deben existir patrones que traten específicamente dicho problema, cuyas soluciones, al ser aplicadas en un algoritmo paralelo, impacten positivamente en el rendimiento logrado. Por lo tanto, este tipo particular de patrones de paralelismo debe tener, a más del contenido actual, información sobre los rendimientos obtenidos y las características del *hardware* utilizado.

La pregunta subyacente es ¿Cómo reflejar esta información altamente específica, respetando el concepto y la estructura de un patrón? Un punto fundamental en este sentido, es mantener el concepto de que un patrón es una solución de tipo generalizada, lo cual impone dejar de lado particularidades de cada situación puntal, y aún así, se aporten lineamientos generales relacionados con el tema del rendimiento.

Por otro lado, a partir de las experimentaciones realizadas, se evidenció que el rendimiento de un algoritmo paralelo tiene una fuerte dependencia del tipo de computador paralelo sobre el cual se ejecuta, y del tipo de red que configure el *cluster*, si éste fuera el caso. Por lo tanto, estos dos factores deben tenerse en cuenta a la hora de desarrollar un patrón de paralelismo para HPC.

Entonces, ¿Cómo se podría generalizar el impacto de la plataforma de ejecución en el programa paralelo y hacer la presentación de este factor dentro de los patrones de paralelismo? Dicho de otra manera, ¿Qué se propone sobre

el estado actual de los patrones de paralelismo para que se adapten a los requerimientos del HPC?

En dicha dirección, es que se presenta una categorización de las arquitecturas usuales de *hardware* que componen un computador paralelo, en orden creciente según la capacidad de procesamiento:

1. **Modelos de memoria compartida**: el volumen de procesamiento es tal que puede ser ejecutado por un equipo que dispone de todos sus recursos conformando una unidad de cómputo, donde principalmente la memoria es accesible en forma directa por todos los procesadores y no hay necesidad de redes de comunicaciones para comunicar los procesadores. Son los equipos *multi-many core* de la actualidad.

2. **Modelos de memoria distribuida**: conformados por equipos homogéneos unidos por medio de redes de alto ancho de banda y baja latencia. Cuando el volumen de procesamiento supera la capacidad de un solo equipo, por más grande que ésta sea, es necesario interconectar varios para poder incrementar la capacidad de procesamiento. Las redes más usuales en esta arquitectura tienen un esquema de comunicaciones de tipo *bus*, o de tipo punto a punto y las diferencias entre éstas influyen en el rendimiento del programa paralelo. Son los típicos *clusters*.

3. ***Multicluster/Grid***: cuando los requerimientos de procesamiento son incluso mayores, es necesario interconectar varias unidades del tipo señalado en el punto anterior, encontrándonos aquí con equipamiento heterogéneo unido por redes de alta latencia y bajo ancho de banda. *Grid Computing* es la rama de la computación orientada a esta configuración de equipamiento.

Se destaca que, aunque existan casos en que la capacidad de procesamiento de un equipo *multicore* sea superior al de un *cluster*, el componente de la arquitectura del *hardware* es lo que los diferencia, al menos bajo esta clasificación.

Es evidente que la optimización del uso de estos recursos es diferente para cada una de estas categorías. Si la totalidad de los datos a procesar ocupa un volumen tal que la memoria de un equipo es suficiente para almacenarlos evitando el *swap* en disco, la mejora en el rendimiento debe enfocarse, por un lado, a optimizar el uso de la jerarquía de memoria y por otro evitar al mínimo posible los *delays* o tiempos inactivos de los procesadores esperando

resultados procesados por otros. De tal manera, en un entorno de computadoras *multicore*, aplicar patrones a nivel de diseño, referidos a la partición de datos y a la distribución de tareas, tienen bajo impacto en el rendimiento, ya que es posible utilizar al máximo el potencial dado por todos los procesadores. En este caso toman relevancia patrones pertenecientes al espacio de implementación, acorde a Massingil et al. para su *Pattern Language*.

En un entorno de *cluster*, la distribución de datos es vital para minimizar los tiempos de comunicaciones y debe hacerse sin dejar de lado la escalabilidad. La arquitectura de la red influye en la determinación del tipo de algoritmo más conveniente a utilizar. Aquí, lo que debe minimizarse es la comunicación entre los nodos que conforman el *cluster*, por lo que la forma en que los datos estén distribuidos determina las tareas que cada nodo puede llevar adelante, de manera tal que la división de tareas no puede hacerse independientemente de la de datos. En este caso, la distribución de datos determina la división de tareas.

En un entorno de *multicluster / grid* como el definido anteriormente, los costos comunicacionales son altos en términos de tiempo, pero como el objetivo es procesar grandes volúmenes de datos, puede dejarse de lado la optimización del uso de los recursos locales de cada componente de la *grid*: aquí no es importante el *speedup* ni la escalabilidad lograda, sino más bien el *throughput* global obtenido. Se hace entonces necesario poner el foco en la distribución de tareas entre los componentes de la grid para lograr el volumen de procesamiento requerido.

Si el tiempo de respuesta en dicho entorno no es el deseable, se agrega más poder de cómputo, sumando recursos computacionales y replicando los datos. Bajo estas circunstancias la distribución de tareas determina la de datos ya que el cómputo debe realizarse en forma imperiosa, y no importa si los datos deben replicarse: aquí lo importante es que el cómputo final se alcance en el tiempo esperado, por lo tanto, desde el punto de vista de los patrones, es el caso inverso al caso anterior: la distribución de tareas condiciona la de datos.

En un entorno de *cluster*, también se ha visto que el tipo de red que lo conforma tiene un impacto en el rendimiento alcanzable según se utilicen en el algoritmo, comunicaciones con primitivas colectivas o con primitivas punto a punto. Se obtienen los mejores resultados relativos cuando se usan comunicaciones colectivas bajo redes *Ethernet* y comunicaciones punto a punto bajo redes Infiniband.

Con estos elementos en mente, se sugiere que los patrones de paralelismo orientados al HPC deberían hacer referencia a la plataforma paralela donde

se ejecuta el programa, e indicar que, si bien conceptualmente son patrones de "paralelismo en general", la utilización de cada uno de ellos alcanzarán buenos rendimientos, solo si se los aplican bajo un determinado contexto de problema y sobre una/s plataforma/s determinada/s.

Por ello es que se propone que los patrones de paralelismo para HPC incluyan en la sección **Ejemplos**, información sobre rendimientos alcanzados en cada uno de los tipos de plataformas de ejecución arriba mencionadas (*multicore, cluster, multicluster/grid*), y se haga referencia al *speedup* logrado, al menos para los casos de memoria compartida y *cluster*.

Puede objetarse a esta propuesta que los resultados que se obtienen siempre son particulares al *hardware* y *software* de base utilizados, lo cual se reconoce. Pero la presentación de rendimientos en el patrón, aunque sea de algunos casos particulares pero reales, es de gran ayuda para el programador de HPC, ya que le aporta una idea sobre los rendimientos que puede obtener de la aplicación que haga del patrón en su caso particular. Por ejemplo, en un algoritmo con fuerte dependencia de datos en un *cluster*, el *speedup* reportado puede tomarse como límite para las aspiraciones de mejora de los tiempos a lograr.

Es claro que el rendimiento final que se obtenga dependerá de como el programador aplique el patrón, y por lo tanto, el aporte que le puede brindar la información de rendimientos obtenidos en la sección Ejemplos no es suficiente. Pero se sostiene que tener la referencia de un resultado real permite partir desde un escalón más alto en la tarea de obtener un algoritmo paralelo eficiente. De hecho, si se sigue lo aconsejado por el *Reengineering for parallelism pattern*, se dispondrá de información útil para el proceso de "prueba y error". Al menos, al comenzar el proceso, el programador no deberá probar opciones que tenga reportadas como ineficientes. En estos hechos se fundamenta la propuesta realizada.

Como ejemplificación al respecto, la próxima sección presenta a una versión preliminar de un nuevo patrón, *"Partial Computing"*, donde se incluye en la estructura del patrón contenidos referidos al rendimiento obtenido en los experimentos previos.

5.2. *Partial Computing Pattern*

En esta sección contiene la propuesta de un nuevo patrón de paralelismo. El mismo es resultado del trabajo de experimentación realizado a lo largo de

la tesis, y se presenta siguiendo la propuesta de la sección anterior, respecto de incluir rendimientos obtenidos bajo las plataformas de *multicore* y *cluster*.

La estructura del patrón respeta lo usual en el dominio, salvando el agregado recién mencionado que se incluye en la sección **Ejemplos**. Allí se presentan los datos y resultados obtenidos de los experimentos realizados a lo largo del desarrollo de la tesis. Se reconoce el faltante de datos para la categoría de *multicluster/grid* ya que no fueron realizados experimentos bajo dicha arquitectura de multicomputador por caer fuera del alcance de la presente tesis.

Por otro lado, a los efectos de presentar el patrón lo más completo posible, la sección Ejemplos presenta los casos de prueba expuestos a lo largo de la presente tesis, por lo que téngase en cuenta que la duplicación de los ejemplos es solo a los fines de completitud.

Cabe destacar, que la presentación de este patrón cumple con el rol de servir de ejemplo respecto de lo sugerido para que los patrones de paralelismo puedan considerarse orientados al HPC, y también, con el doble rol de ser un patrón inédito, surgido luego del trabajo experimental desarrollado a lo largo de la tesis.

A continuación, la propuesta de patrón.

5.2.1. Nombre

Partial Computing Pattern

5.2.2. Propósito

Obtener altos rendimientos en aplicaciones de cómputos intensivos por medio de dividir el procesamiento en tareas donde cada una de ellas obtiene valores parciales, los cuales deben reservarse hasta el momento de computarse el resultado final del procesamiento.

La paralelización por división de datos está tradicionalmente enfocada a que cada nodo de procesamiento trabaje sobre un bloque de datos y obtenga el resultado definitivo sobre dicho bloque. Este patrón está basado en una división de tareas de grano mas fino, tal que el resultado del procesamiento del bloque no sea un valor definitivo, sino uno parcial que debe ser mantenido y/o acumulado hasta que se realice todo el procesamiento necesario para obtener el valor final, de forma tal que pueda realizarse parte de los cómputos en oportunidad previa al cómputo del resultado final.

5.2.3. Contexto

Las aplicaciones de cómputo intensivo, como lo son las de álgebra lineal, tienen una especificación precisa sobre la secuencia de pasos que las componen [GVL96]. La paralelización de dichos algoritmos ha tomado tradicionalmente como punto de partida dicha especificación secuencial, para luego realizar una división de datos que permita paralelizar cómputos [B$^+$97, Sca], de forma tal que cada procesador trabaje sobre una parte de los datos y contribuya con una fracción del cómputo total [Akl89, MSM04, Lei91].

Los algoritmos secuenciales de cómputo intensivo, por lo general concentran la mayor parte del esfuerzo de procesamiento en determinadas secciones del código, mayormente *loops* sobre grandes arreglos de datos. Los primeros intentos de paralelización mantuvieron dichos lazos pero cada procesador trabajando sobre una parte de los datos, dando lugar al modelo SIMD (Single Instruction Multiple Data) o procesamiento vectorial [Fly72].

Este estilo de paralelización impone la existencia de puntos en el programa paralelo donde se realiza una sincronización entre los procesos [KCS+09] para unificar el avance del procesamiento e intercambiar datos procesados [CKS+09] entre ellos, especialmente en algoritmos que presentan una serie de etapas, cada una de éstas, basada en valores previos obtenidos. *Partial Computing pattern* se presenta con un enfoque diferente, ya que se basa en que cada procesador hace tareas más chicas computando cada una de dichas tareas valores parciales, y no la misma tarea sobre menor cantidad de datos, por lo que el foco es orientado sobre las operaciones como base de mejora de procesamiento. Este patrón tiene similitud con *Task Decomposition pattern* [MSM04] pero con una granularidad más fina. Actualmente, el proyecto PLASMA [oT] sigue la misma base conceptual en cuanto a la división de tareas, manteniendo el esquema tradicional de dependencia de tareas sobre los resultados finales.

Dos requisitos son necesarios que existan para aplicar *Partial Computing* en forma beneficiosa: la disponibilidad de tiempo inactivo en un procesador y la disponibilidad de parte de los datos necesarios para el cómputo. Es en este escenario, donde puede hacerse un procesamiento por anticipado, reservando el resultado para la oportunidad en que todos los datos estén disponibles para realizar el cómputo final.

La dependencia de datos es una de las razones principales para que un procesador quede en estado inactivo, ya que los datos deben estar disponibles para ser procesados. Si estos datos fueron calculados en etapas anteriores por otros procesadores, es necesario comunicarlos, por lo que el receptor queda a la espera de estos para poder continuar. El propósito de este patrón es aprovechar el tiempo inactivo siempre que se disponga de datos para poder ir calculando valores parciales por anticipado.

Otra causa de inactividad por parte de un procesador es la necesidad de sincronizar procesamiento, lo cual generalmente implica un intercambio de información. En arquitecturas de memoria distribuida los tiempos de comunicación son varios órdenes de magnitud más lentos que los de procesamiento, lo cual genera inactividad en las CPU. Un paliativo a esta inactividad es la aplicación del patrón *Overlapping Communication and Computation Pattern* [BMK09], a pesar de lo cual, los tiempos inactivos persisten, básicamente por los diferentes ordenes de magnitud referidos.

En ciertas oportunidades, distribuir los datos bajo la consideración de cómputos parciales, posibilita una reducción en los tiempos de comunicación, como se verá en la sección 5.2.6, para el caso de la multiplicación de matrices.

5.2.4. Fuerzas

- El nivel de **dependencia de datos** del algoritmo. *Partial computing* solo puede aplicarse si parte de los datos de los cuales depende una tarea, están disponibles con anticipación al momento del cómputo del valor final. Cuando el algoritmo tiene una dependencia de datos tal que siempre se necesiten solo los últimos datos calculados, se dificulta la aplicación del patrón.

- La existencia en la fórmula de procesamiento de **operaciones sobre conjuntos de datos** (como ser el sumatorio o el multiplicatorio) en donde poder realizar el despliegue de dichas fórmulas y determinar los puntos del programa donde aplicar el cómputo por anticipado. En ocasiones, el reordenamiento de la fórmula desplegada pueda dar lugar a esquemas de comunicaciones más ágiles, como se verá en la sección de ejemplos para el caso de la multiplicación de matrices.

- La existencia de puntos en el algoritmo paralelo con **procesadores inactivos**, donde estos no realicen cómputo alguno, a la espera de la finalización de alguna comunicación y/o sincronización. Es en estos puntos donde la aplicación del algoritmo genera beneficios en el rendimiento, ya que en lugar de estar en estado *idle*, el procesador va anticipando cómputos. De no existir este tipo de puntos en el algoritmo, no hay posibilidad de generar mejoras en el rendimiento. En el caso de algoritmos *embarrassingly parallel*, donde no existan dependencias de datos, se complica la aplicación del patrón, ya que se dificulta la determinación de puntos en que algún procesador esté inactivo.

- El esquema de **comunicaciones de datos** entre los procesadores. Los *delays* por comunicaciones son los que permiten de realizar cómputos parciales, por lo que la distribución de datos impacta sobre la posibilidad y/o oportunidad en que puede aplicarse *Partial Computing*. Existe un fuerte correlato con el patrón *Overlapping Communication and Computation* [BMK09].

5.2.5. Solución

La aplicación de este patrón presenta dos etapas: a nivel del espacio de diseño del algoritmo y a nivel del espacio de implementación, siguiendo el concepto de distintos niveles de pertinencia de los patrones de paralelismo

planteado por Mattson et al.[MSM04]. A nivel de diseño, el objetivo es definir las tareas que pueden subdividirse en cómputos parciales. A nivel de implementación, hay que determinar los puntos del algoritmo que presentan las condiciones de factibilidad para la aplicación, es decir, los puntos donde hay un procesador inactivo y conjuntos parciales de datos disponibles.

La primera etapa, a nivel del diseño del algoritmo, consiste practicar una división de tareas y datos por medio del análisis de la formulación matemática que tenga el problema para poder determinar que tareas pueden ser subdivididas a los fines de computar resultados parciales. Un elemento clave en esta etapa es determinar si la fórmula opera sobre un conjunto de datos con una operación que tiene la propiedad matemática de asociatividad. La operación $\otimes$ es asociativa si, aplicada a a, b, c, cumple:

$$a \otimes b \otimes c = (a \otimes b) \otimes c = a \otimes (b \otimes c)$$

La adición, multiplicación, comparaciones lógicas y la concatenación de *strings* son ejemplos de operaciones asociativas, a las cuales puede aplicarse *partial computing* sobre una parte de los datos. Existen operaciones que no son asociativas, pero cumplen con solo una de las igualdades de la ecuación anterior, la cual permite hacer una subdivisión de tareas pero no en cualquier orden. La división y la exponenciación son ejemplos de estas operaciones. Tener en cuenta estas características a la hora de definir como aplicar el patrón para preservar el resultado correcto.

No solo en el álgebra lineal se presentan operaciones asociativas, sino muchos otros dominios también, citando el ejemplo de los algoritmos de ordenamiento. Estos se basan en operaciones de comparación, que pueden aplicarse sobre un conjunto parciales de datos, para luego combinar los resultados parciales en uno final. El algoritmo de *Quicksort* es un ejemplo de combinación de resultados parciales, y también lo es el algoritmo *Shellsort* [Wei97] que específicamente utiliza comparaciones sobre conjuntos parciales de datos, cuyos resultados se combinan en la última etapa de procesamiento para llegar al ordenamiento final.

Una vez definidos los puntos del programa donde puede aplicarse *partial computing* por el despliegue de las fórmulas intervinientes, la segunda etapa consiste en determinar la oportunidad de hacerlo. Para ello es necesario llegar al nivel de implementación y encontrar en la ejecución del algoritmo paralelo los puntos en que algún procesador quede inactivo para poder aplicar el patrón en forma beneficiosa. Es aquí donde las herramientas de *tracing* y *profiling* son imprescindibles para poder detectar en forma fehaciente, los puntos del programa en que los procesadores no trabajan. Si bien *a priori* pueden

determinarse estos puntos en forma teórica, la ayuda que brindan estas herramientas es vital para poder detectar la oportunidad real donde exista/n algún/os procesador/es inactivo/s y destinarlo/s al cómputo de resultados parciales.

El proceso de aplicar el patrón *partial computing* es iterativo ya que consiste en una mejora incremental del algoritmo paralelo hasta lograr un resultado de rendimiento aceptable. Se puede considerar que este patrón ayuda al refinamiento del rendimiento del algoritmo paralelo, tal cual lo recomendado por el patrón *Reengineering for parallelism*[MMS07], al cual nos remitimos.

Puede suceder, en un esquema de memoria compartida, que el procesador inactivo donde se deban realizar los cómputos anticipados, necesite de datos previamente calculados, pero residentes en otro nodo, por lo que en este caso se hace necesario una comunicación de datos. Tener en cuenta esta posible sobrecarga como contrapeso de las mejoras obtenidas por el cómputo anticipado. Aplicar el patrón *Overlapping Computation and Communication* para evitar los *delays* agregados por esta causa.

Un inconveniente importante se plantea si no se cuenta con una herramienta de *profiling* paralelo para el seguimiento de la aplicación a nivel de ejecución. Si bien una ausencia de este tipo de herramientas no imposibilita el seguimiento manual de la ejecución del programa, se torna muy difícil reemplazar una herramienta automática por trabajo humano que temporice la ejecución de un algoritmo paralelo. En este caso hay que proceder a una determinación "manual" de los tiempos que insume cada parte del algoritmo y proceder por "prueba y error" estimando los puntos en que el procesador se vuelve inactivo y comprobar la mejora en el rendimiento realizando numerosas corridas de *tests* hasta encontrar el refinamiento adecuado del algoritmo que genera las mejoras aceptables en el rendimiento.

Resumiendo, el proceso de utilizar el patrón *Partial Computing* debe pasar primero por las etapas tradicionales división de tareas y distribución de datos, para luego, al determinar los puntos donde el algoritmo puede modificarse para mejorar el rendimiento, ser específicamente aplicado en esos puntos. No tiene sentido aplicar *partial computing* sin haber realizado un primera división de tareas y determinado los puntos de mejora. De no proceder acorde a lo aconsejado, los resultados de su aplicación pueden incluso llegar a ser contraproducentes respecto al rendimiento obtenido.

Otra variante en la aplicación del patrón puede surgir cuando el algoritmo sea *embarrasingly parallel*, es decir, no existan dependencias de datos y por lo tanto no existan procesadores inactivos durante la ejecución del programa.

En este caso, a nivel de diseño del algoritmo, se pueden desplegar las operaciones asociativas y plantear un orden diferente de cómputo, el cual de lugar a una ventaja en algún factor que mejore el rendimiento, sin que sea necesario llegar al nivel de utilización de los recursos informáticos. En la sección de ejemplos, se expone el caso de la multiplicación de matrices con una descomposición y reacomodamiento de las operaciones realizadas tal que la carga comunicacional del algoritmo es menor, lo cual genera un ahorro en el factor que es más lento y un resultado global positivo. En estos casos, la aplicación del patrón se resume a solo la primera etapa de las dos mencionadas: no es necesario utilizar herramientas para detectar los puntos de aplicabilidad del patrón, ya que lo que cambia es el diseño global del algoritmo y los beneficios de rendimiento son directos.

5.2.6. Ejemplos

Dada la larga tradición existente en la aplicación del patrón *Data Decomposition*, los ejemplos son presentados resaltando las mejoras que *partial computing* permite alcanzar rediseñando el algoritmo paralelo, luego de una primera distribución de datos.

5.2.6.1. Factorización de Cholesky

La factorización de Cholesky resuelve un problema clásico del álgebra lineal [GL96, Pre07], la factorización de una matriz A, simétrica y definida positiva, de rango n, como:

$$A = L \times L^T$$

donde L es una matriz triangular inferior. Cada elemento de L es calculado utilizando las siguiente fórmulas, según sea su posición en la matriz:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} \times l_{jk} \right) / l_{jj} \qquad 1 \leq j < i \leq n \qquad (5.1)$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik^2}} \qquad 1 \leq i \leq n \qquad (5.2)$$

lo cual genera una gran dependencia de datos entre los valores a calcular, ya que solo el valor l_{11} no depende de valores previamente calculados.

Algunos algoritmos paralelos de resolución de este problema utilizan una distribución de datos para la matriz A en forma de bandas fila entre los nodos procesadores [TR05]. El procesamiento es llevado adelante en la matriz triangular, avanzando de fila en fila, comenzando por la esquina superior izquierda hacia la inferior derecha. Cada vez que un elemento de la diagonal principal de la fila k queda calculado, todos los valores de dicha columna pueden calcularse, ya que todos los valores necesarios para ello ya están determinados[1].

Con esta distribución de datos, el nodo poseedor de la última fila calculada, la fila k, debe enviar a los nodos poseedores de las restantes filas, todos los valores de su fila, para que estos puedan continuar con sus cómputos.

El mismo algoritmo puede utilizarse si en lugar de valores individuales se toman bloques de datos, usando las rutinas de las bibliotecas BLAS y LAPACK, como lo hace la implementación en paralelo de éstas para *clusters*, ScaLAPACK [Sca]. Las filas y columnas están compuestas de bloques de datos, y en la diagonal principal, dicho bloque es cuadrado.

El procesamiento de este algoritmo computa valores finales, tanto para los valores o bloques de la diagonal principal, como para los valores o bloques de la parte triangular inferior de la matriz L. Sin embargo, cómputos parciales pueden aplicarse en cualquiera de las dos fórmulas. Es fácil de observar en las dos fórmulas anteriores la existencia de una substracción, donde a valores originales de A se le resta un sumatorio de valores calculados, la cual puede irse completando en términos parciales a medida que avanza el proceso de cómputo, gracias a la asociatividad que presenta la substracción.

Para el caso de los elementos de la parte inferior de L, se puede considerar que, si la iteración h a finalizado, $1 \leq h \leq j-1$, entonces, todos los bloques de las columnas hasta h ya están calculados, por lo que el valor de l_{ij} puede ser desplegado como:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{h} l_{ik} \times l_{jk} - \sum_{k=h+1}^{j-1} l_{ik} \times l_{jk} \right) / l_{jj} \qquad 1 \leq h \leq j-1 \quad (5.3)$$

donde todos los valores del primer sumatorio (hasta h) están disponibles y el sumatorio puede completarse, y que los valores del segundo sumatorio están aún pendientes. De acuerdo a nuestra distribución de datos, si el procesador i está inactivo, podría computar el resultado parcial del primer sumatorio de

[1]Estos valores son los que pertenecen a la fila k, y los valores de las filas anteriores hasta la columna k

la Ec. (5.3) solo con recibir del procesador j la respectiva parte de su fila, hasta la columna h.

Algo similar ocurre con los elementos de la diagonal principal. El término cuadrático del sumatorio en la Ec. (5.2), para la fila i, considerando que la iteración h ha concluido, puede desplegarse también como:

$$\sum_{k=1}^{i-1} l_{ik}^2 = \sum_{k=1}^{h} l_{ik}^2 + \sum_{k=h+1}^{i-1} l_{ik}^2 \tag{5.4}$$

donde los términos l_{ik} para $k = 1 \ldots h$ en el lado derecho de la ecuación ya han sido calculados, y los términos para $k = h+1 \ldots i-1$ están pendientes, por lo que el primer sumatorio del lado derecho de la ecuación puede computarse por adelantado. La distribución de datos por bandas filas hace que no sea necesario ninguna comunicación de datos adicional.

A nivel de diseño, se ha visto entonces, la existencia de dos puntos del algoritmo donde puede aplicarse cómputos parciales. Para completar la aplicación del patrón, resta determinar la oportunidad donde realizar los cómputos parciales, para que el resultado de la aplicación devenga en una mejora del algoritmo.

A nivel de implementación, el uso de herramientas de *tracing* y *profiling* toma importancia. La figura (5.1) muestra el seguimiento de la ejecución

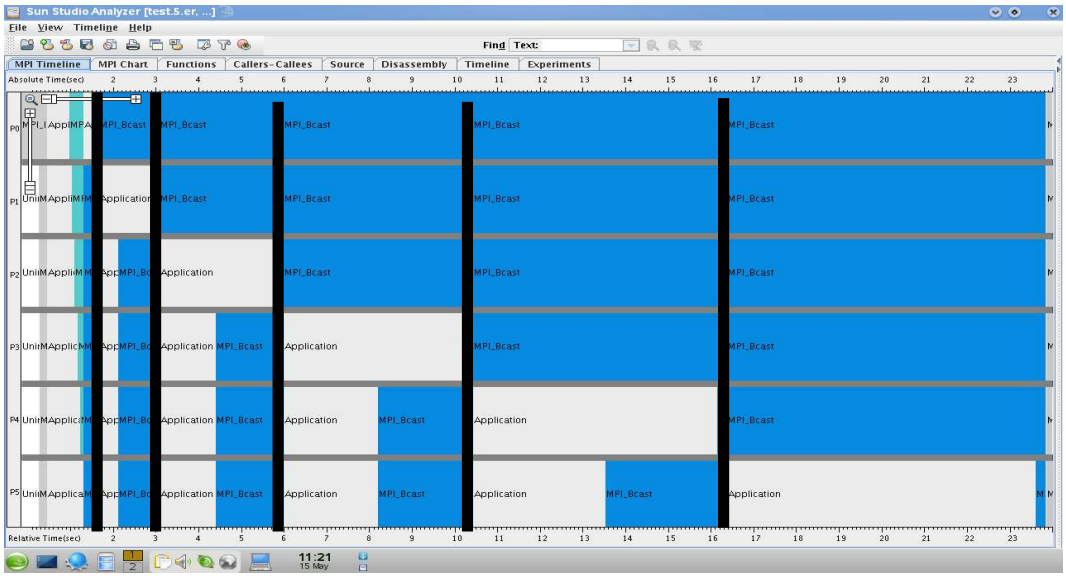


Figura 5.1: Gráfico de la ejecución del algoritmo de Cholesky sin aplicar *Partial Computing* para 6 nodos y n=18000.

del algoritmo previo a la aplicación de *partial computing* utilizando la herramienta *Analyzer* perteneciente a la *suite Sun performance library* de la empresa Sun corp.[Sun]. La ejecución se realizó sobre un *cluster* con 6 nodos y $n = 18000$. Las áreas blancas grafican los tiempos en que el procesador estuvo computando, y las azules (grises), los tiempos de espera para que concluyan las comunicaciones.

Las lineas negras verticales indican el punto del programa donde se completa el cómputo y comunicación del elemento de la diagonal principal y comienza el cómputo de los elementos de la respectiva columna por parte de los nodos poseedores de las restantes filas, como se señaló anteriormente.

Numerosos tiempos de inactividad pueden verse en dicho gráfico, principalmente debido a dos causas:

1. Cuando un nodo está computando el elemento de su diagonal principal, los nodos siguientes están a la espera de este resultado.

2. Una vez que un nodo completó el cómputo del elemento en su diagonal principal, ya no tiene más tareas que realizar.

La primera razón da lugar a que los nodos siguientes puedan realizar un cómputo por anticipado de los elementos de sus respectivas diagonales principales, acorde a lo indicado en la Ec. (5.4). La segunda razón da lugar a la aplicación de *partial computing* para los elementos fuera de la diagonal principal, acorde a la Ec. (5.3). En este último caso, como es necesario una comunicación de datos adicional, se implementó enviándole datos al nodo 1 para que compute valores parciales y los devuelva a su respectivos nodos.

El resultado de la aplicación de ambas prácticas se exhibe en la Fig. (5.2). Allí puede verse que, por un lado, que el nodo 1 tiene más actividad de cómputo que para el caso anterior, y por otro lado, para el momento en que un nodo está computando el elemento de su diagonal principal, los nodos restantes hacen cómputos por adelantado, reflejado en que las áreas azules (grises) son más angostas, es decir, los procesadores tienen menor tiempo inactivo.

Las mejoras en cuanto a tiempo se detallan en la sección rendimientos de este patrón, pero se puede adelantar, que para los *tests* realizados, alcanzan al 35 % de disminución del tiempo insumido. Mayor información sobre estos ensayos puede verse en [Wol10].

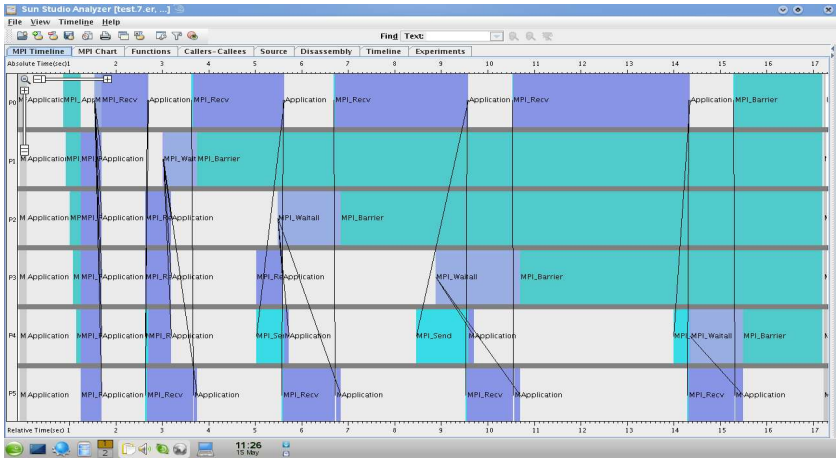


Figura 5.2: Gráfico de la ejecución del algoritmo de Cholesky aplicando *Partial Computing* para 6 nodos y n=18000.

5.2.6.2. Multiplicación de Matrices

El caso de la multiplicación de matrices es particular, ya que es un algoritmo en el que no existen dependencia de datos, por lo que si cada tarea computa un elemento (valor o bloque) de la matriz resultante, puede plantearse cualquier orden para la ejecución de las tareas con un resultado correcto, y por lo tanto, no existen tiempos de inactividad en los procesadores del multicomputador originados en la espera de un resultado anterior calculado por el mismo algoritmo. Sin embargo, es incluido como ejemplo de un algoritmo en el que *partial computing* se aplica solo a nivel de diseño, generando un beneficio en el algoritmo paralelo consistente en la reducción del tráfico de comunicaciones de datos entre los nodos.

Considerensé matrices cuadradas A, B, C de rango n, que intervengan en cómputo de la multiplicación $C = A \times B$, p nodos de procesamiento, y los datos distribuidos por bandas en cada matriz, bandas filas para las matrices C y A, y bandas columnas para la matriz B, cada banda de n/p filas por n columnas para las primeras y n filas por n/p columnas para las últimas, como se puede ver en la figura 5.3.a)[2]. Cada nodo posee una banda fila para el caso de las matrices A y C, y una banda columna de la matriz B, llamando *Banded Data Distribution* a esta distribución de datos.

El procesamiento es realizado en cada nodo, computando un bloque de C por vez, $C_{i,j} = A_i \times B_j$, y como los datos de la fila i de A y C residen en el

[2]Los números pequeños en el gráfico indican el nodo poseedor de los datos

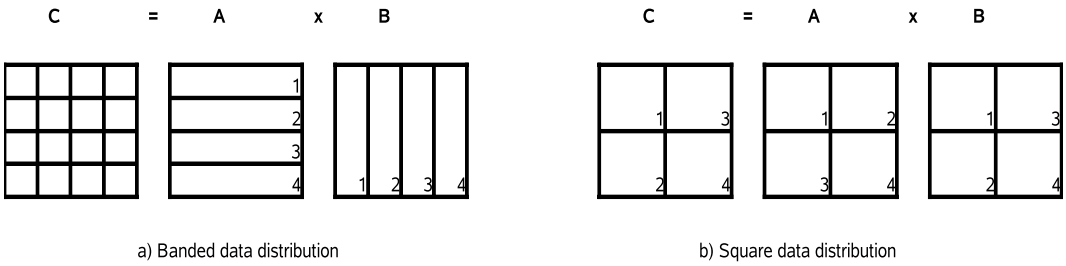


Figura 5.3: Distribución de datos para la multiplicación de matrices.

mismo nodo, la columna j de B debe enviarse hacia el nodo poseedor de la fila i de A, salvo para el caso en que $i = j$. Puede verse que cada bloque de C, $C_{i,j}$, tiene rango n/p.

Este algoritmo determina que cada banda fila de C es el resultante de p procesos de multiplicación en un nodo, y que el total de valores recibidos por dicho nodo para realizar el procesamiento se corresponde con $p-1$ mensajes enviados por los restantes nodos, a razón de $n \times n/p$ valores por cada mensaje. La carga total de datos comunicados es $p^2 - p$ valores.

Un algoritmo alternativo considerando cómputos parciales, solo a nivel de diseño del algoritmo, es el siguiente:

supongase que el número de nodos p es un número cuadrado, cuatro por ejemplo (esta restricción será eliminada más adelante). Las matrices A, B y C son divididas en bloques cuadrados de $\sqrt{p}$ filas y $\sqrt{p}$ columnas, dos filas por dos columnas para nuestro ejemplo, como puede verse en la figura 5.3.b), llamando *Square Data Distribution* [VB05] a esta distribución de datos. Cada bloque de C, $C_{i,j}$, es el resultado de $\sqrt{p}$ procesos de cómputo parcial:

$$C_{i,j} = \sum_{k=1}^{\sqrt{p}} A_{i,k} \times B_{k,j}$$

por lo que la distribución de datos entre los nodos cumple con que cada nodo tiene un bloque de A, B y C, y la distribución de B es la misma que la de C y a su vez, ambas son la traspuesta en cuanto al número de nodo, respecto de la distribución de A, es decir, la distribución de la fila i de A es la misma que la columna i de B y C, como puede verse en la figura 5.3.b).

Cada bloque de las matrices A, B y C tiene $n/\sqrt{p}$ filas por $n/\sqrt{p}$ columnas, por lo que la cantidad de valores de cada bloque es la misma que para la distribución de datos por bandas. Sin embargo, ahora el conteo para

las comunicaciones es diferente. Siguiendo el ejemplo en que $p = 4$, la carga comunicacional es:

- Para bloques en la diagonal principal, como la distribución de datos responde a un criterio de traspuesta, solo un bloque necesita enviarse por cada bloque de C a computar. Siguiendo el ejemplo, para el caso del bloque $C_{1,1}$, se necesitan computar $A_{1,1} \times B_{1,1}$, lo cual lo hace el nodo 1, y $A_{2,2} \times B_{2,2}$, hecho por el nodo 2, y finalmente, sumar ambos resultados parciales en el nodo 1, por ser el poseedor del bloque $C_{1,1}$, por lo que solo un bloque de datos necesita ser enviado. En general, es posible probar que el número de comunicaciones necesarias para los bloques en la diagonal principal es $\sqrt{p} - 1$.

- Para bloques fuera de la diagonal principal, en nuestro ejemplo, solo se necesitan tres envíos. Por ejemplo, para el caso del bloque $C_{2,1}$, se necesita computar $A_{2,1} \times B_{1,1}$, en el nodo 1, y $A_{2,2} \times B_{2,1}$, en el nodo 2, y sumar ambos resultados en el nodo 2, el poseedor del bloque $C_{2,1}$. El número total de envíos para bloques fuera de la diagonal principal, para nuestro caso de $p = 4$, es de 6 envíos (tres por cada nodo). Generalizando, también puede demostrarse que la cuenta para este tipo de bloques es $2 \times (\sqrt{p} - 1) + 1$.

El número total de envíos necesarios para completar todo el procesamiento, es la suma de envíos para ambos tipos de bloques: para el primer caso hay $\sqrt{p}$ bloques de este tipo y para el segundo caso hay $p - \sqrt{p}$ bloques, por lo que el total de bloques enviados a lo largo de todo el procesamiento (tn) es:

$$\begin{aligned} tn &= \sqrt{p} \times (\sqrt{p} - 1) + (p - \sqrt{p}) \times (2 \times (\sqrt{p} - 1) + 1) \\ &= 2p \times (\sqrt{p} - 1) \end{aligned}$$

lo cual es menor que los $p^2 - p$ bloques determinados para el primer algoritmo. En nuestro ejemplo, 12 envíos *versus* 8 para el segundo algoritmo, siendo esta diferencia cada vez mayor a medida que p crece.

Relajando la restricción de que el número de nodos p sea un número cuadrado, lo que se hace es factorizar p como $p = r * c$ de tal forma que la diferencia absoluta entre r y c sea la menor posible, y $r < c$. Ahora, la matriz A se divide en bloques de r filas por c columnas, la matriz B en c filas por r columnas y la matriz C en r x r, como se puede observar en la Fig. (5.4) para el caso en que $p = 6$, por lo que $r = 2$ y $c = 3$.

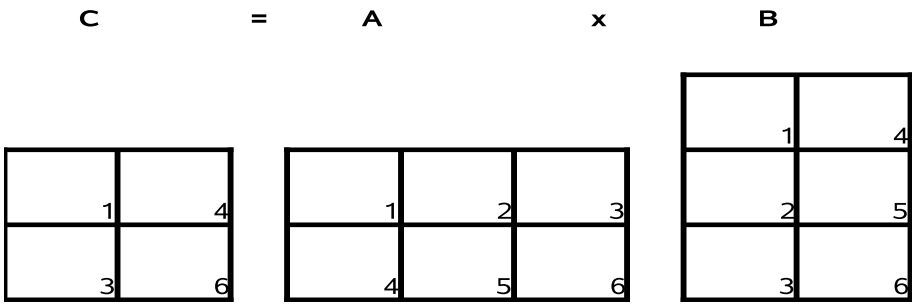


Figura 5.4: Distribución de datos para un número de procesadores no cuadrado.

La distribución de datos para las matrices A y B siguen el mismo criterio que antes, en cuanto a la trasposición del número de los nodos.La diferencia se plantea para la matriz C, que tiene un número de bloques diferente, r^2, para el cual, cuando mayor sea r, más cercano se está del caso en que p es cuadrado.

Como consecuencia de lo anterior, no todos los nodos tienen un bloque de C: como esta matriz se dividió en un número de bloques menor que para las matrices A y B, entonces, ¿Cómo es la forma en que se asignan los bloques de C entre los nodos?. Siguiendo el supuesto de que los valores son enviados desde el nodo poseedor de un bloque de A hacia el nodo poseedor del bloque de B respectivo para realizar el cómputo parcial, la distribución de los bloques de C sigue el mismo criterio que para B, exceptuando que el número de filas de B supera al número de filas de C, por lo que en el esquema de distribución de B se suprimen las filas existentes en exceso, manteniendo la primera ni la última, es decir, son eliminadas $c - r$ filas del medio de la matriz, y de esta forma queda definido el esquema de distribución de C. En la figura anterior se ve que la distribución de C es igual a la de B sin su segunda fila.

Este patrón de distribución mantiene la misma estructura de conteo que para el caso en que p es cuadrado, es decir $c - 1$ envíos para bloques en la diagonal principal y $2 * (c - 1) + 1$ envíos para los restantes bloques. Puede verse que a pesar de que el número de envíos en mayor que para el caso de un número cuadrado de nodos, aún sigue siendo menor que para el caso de la distribución por bandas planteado inicialmente. A menor diferencia entre r y c, más cercano se está al caso en que p es cuadrado. La sección rendimientos muestra las ventajas de este algoritmo.

5.2.6.3. Rendimientos Obtenidos

5.2.6.3.1. Memoria Compartida El estado del arte actual para computadoras con procesadores *multicore* presenta un número de *cores* que todavía es relativamente bajo y que todos los cores son homogéneos. Por otro lado, la mayoría de las bibliotecas de rutinas de álgebra lineal tienen implementado la optimización del uso de los *cores* disponibles, por lo que, en la medida que sea posible, y teniendo en cuenta que las aplicaciones de cómputo científico generalmente derivan en algún tipo de rutina de álgebra lineal, utilizar estas bibliotecas configurando el número de *cores* disponibles hasta alcanzar el número máximo de estos.

Nuestros experimentos en este entorno dan resultados de *speedup* cercanos al óptimo al utilizar estas bibliotecas según lo indicado, por lo que se aconseja no aplicar ningún patrón. Revisar en el futuro cuando el número de *cores* sea alto, y/o no homogéneos, en cuyos casos el *speedup* puede caer.

No se dispone de resultados para casos en que no sea posible utilizar las bibliotecas de álgebra lineal. Bajo dichas circunstancias, seguir las mismas recomendaciones que para el modelo de memoria compartida. Tener en cuenta que el máximo *speedup* posible viene dado por el número de cores disponibles en el equipo, y evitar los tiempos de espera por resultados, utilizando los núcleos inactivos para realizar cómputos parciales. Este punto debería completarse con investigaciones futuras.

5.2.6.3.2. Memoria Distribuida / *Cluster* Los resultados que se exponen fueron obtenidos de corridas en un *cluster* conformado por 64 computadoras con dos CPU's Intel Xeon 5420 *quad core* cada una, dieciséis GBytes RAM, y red Infiniband 20 Gbit/sec de conexión. También Centos 5.3 como sistema operativo y Open MPI 1.4.1 como biblioteca de rutinas MPI. El compilador utilizado es Intel ifort 11.1 y la biblioteca MKL como implementación de BLAS y LAPack.

Factorización de Cholesky

En tabla 5.1 se exponen los resultados comparativos según el tipo de red que se utilice, en un *cluster* de 6 nodos. Se presenta el algoritmo con solo *broadcast* como referencia y de la aplicación del patrón realizando los cómputos parciales sobre ambas fórmulas, según lo expuesto anteriormente. Puede observarse como la mejora en los resultados de aplicar el patrón es menor para el caso en que la red *Ethernet* es utilizada, obteniéndose mejoras mayores cuando se utiliza Infiniband.

En tabla 5.2 se presentan los resultados de la aplicación del patrón acorde a los algoritmos expuestos en la sección de Ejemplos, utilizando solo Infiniband, con hasta 48 nodos. La primera columna de tiempos presenta los resultados del algoritmo con solo *broadcast*, la segunda, con *partial computing* solo para los bloques de la diagonal principal, y la tercera, con *partial computing* adicional sobre la última fila de la matriz. Como puede verse, la mejora alcanza porcentajes importantes y crecientes con el número de nodos utilizados en el *cluster*, alcanzando un pico máximo de la mitad del tiempo insumido por el algoritmo con solo *broadcast*.

Multiplicación de matrices

Los resultados de ejecución usando *partial computing* pueden verse en la tabla 5.3, utilizando hasta 64 nodos, con red Infiniband. Los resultados mejoran a medida que se incrementa el número de nodos participantes del *cluster*, al igual que para el caso del algoritmo de *Cholesky*. Las mejoras llegan hasta cerca del 50 % del tiempo original.

5.2.7. Patrones Relacionados

- **Task Decomposition Pattern**: este patrón es el punto de partida para la aplicación de *partial computing*. El procesamiento parcial es hecho principalmente subdividiendo las tareas que realizan operaciones que verifican algún tipo de asociatividad.
- **Data Decomposition Pattern**: La distribución de datos definida a nivel de diseño puede afectar la aplicabilidad de *partial computing*. Tener en cuenta la posibilidad de subdividir operaciones asociativas al momento de distribuir datos.
- **Overlapping Communication and Computation Pattern**: El conjunto de datos sobre el que se realizan cómputos parciales puede residir en un nodo diferente del que realice el cómputo, por lo que es recomendable transferir los datos en simultaneo con la realización de cómputos.
- **Design Evaluation Pattern**: Luego de la división de tareas y distribución de datos inicial, la reevaluación del rendimiento es lo que permite determinar cuales serán los puntos del algoritmo donde se realizaran los cómputos parciales.

Rango Matriz	**Bcast. Ethern.**	**Partial Comp Ethern.**	**Bcast. Infinib.**	**Partial Comp Infinib.**
12000	10.99(1.00)	08.91(.811)	07.29(1.00)	05.21(.715)
18000	30.53(1.00)	23.74(.778)	22.42(1.00)	15.77(.703)
24000	62.95(1.00)	47.84(.760)	48.48(1.00)	33.87(.699)

Tabla 5.1: Tiempos de ejecución (segs.) del algoritmo de Cholesky usando *Broadcast* o *Partial Computing*, sobre Ethernet e Infiniband, sobre 6 nodos, 8 *cores* cada uno.

Nodos	**Rango Matriz**	**Broadcast**	**Diag Part Comp**	**Full Part Comp.**
8	12000	04.77(1.00)	04.34(.901)	03.43(.719)
8	18000	13.19(1.00)	10.61(.804)	08.55(.648)
8	24000	27.20(1.00)	21.83(.802)	17.65(.649)
12	14400	06.15(1.00)	05.39(.876)	03.95(.642)
12	24000	23.77(1.00)	15.35(.646)	11.61(.488)
12	36000	63.87(1.00)	45.75(.716)	37.07(.580)
24	14400	04.72(1.00)	04.50(.953)	03.36(.712)
24	24000	14.25(1.00)	11.97(.840)	09.00(.632)
24	38400	52.08(1.00)	42.54(.817)	35.16(.675)
48	38400	34.88(1.00)	30.88(.885)	25.06(.718)
48	57600	129.45(1.00)	75.02(.580)	63.83(.493)

Tabla 5.2: Tiempos de ejecución (segs.) de los tres algoritmos de Cholesky experimentados, usando Infiniband, usando 8 *cores* por nodo.

5.3. Resumen del capítulo

En la primera parte de este capítulo se presentó la propuesta de adecuación del estado actual de los patrones de paralelismo para que sean útiles dentro del dominio del HPC. Se propuso que los patrones deben tener dentro de su sección de **Ejemplos**, información de rendimientos y *speedup* obtenidos al ejecutar los algoritmos de la sección sobre tres categorías de computadoras paralelas. La información aportada es de gran ayuda para el programador de este tipo aplicaciones, ya que le permite tener un punto de referencia en su tarea de obtener altos rendimientos.

Como resultado de los experimentos realizados a lo largo de la tesis se

Nodos (1)	**Rango Matriz (2)**	**Bcast (secs.) (3)**	**Part. Comp. (secs.) (4)**	**Mejora (4)/(3)**	**Promedio mejora**
4	10800	7.117	6.799	0.955	
4	14400	13.785	13.950	1.012	
4	18000	26.431	26.292	0.995	0.975
16	10800	3.882	2.594	0.668	
16	14400	5.907	4.591	0.777	
16	18000	9.874	7.982	0.808	0.738
36	9000	3.226	1.448	0.449	
36	18000	7.521	4.419	0.588	
36	27000	17.673	11.337	0.641	
36	36000	32.937	23.689	0.719	0.584
64	19200	10.171	4.042	0.397	
64	25600	13.089	6.624	0.506	
64	32000	21.382	10.988	0.514	
64	38400	31.232	17.434	0.558	0.478

Tabla 5.3: Comparación de tiempos para los algoritmos utilizando *broadcast* para la distribución por bandas y *Partial Computing* para multiplicación de matrices. Hasta 64 nodos, 8 *cores* cada uno, Infiniband.

introdujo en la segunda parte del capítulo, una propuesta de un patrón inédito, el *Partial Computing Pattern*. La estrategia del patrón consiste en la subdivisión de tareas que permiten obtener resultados parciales de cómputo que *a posteriori* se utilizan para la determinación del resultado final, pero que, dadas las características del algoritmo donde se aplique, mejoren los rendimientos al utilizar tiempos inactivos de los procesadores. Este patrón ha sido presentado acorde a la propuesta dada en la primera parte del capítulo, por lo que aporta, no solo en cuanto a patrón inédito, sino también como ejemplo de patrón pertinente al dominio de las aplicaciones objeto de la tesis.

Capítulo 6

Conclusiones e impacto de la tesis

En el capítulo inicial se planteó como objetivos de la tesis el estudio del estado del arte de los patrones de paralelismo, bajo el enfoque de las aplicaciones de cómputo científico corriendo bajo entornos de *clusters* de nodos *multicore*. Se busca analizar la aplicabilidad y el alcance de estos patrones a la solución del problema de desarrollo de aplicaciones de cómputo científico bajo el tipo de plataforma de ejecución en consideración. A lo largo de la tesis se vio que este enfoque queda incluido dentro del dominio del *High Performance Computing* - HPC.

Dado lo reciente que es este tema, también se planteó como objetivo adicional analizar la pertinencia de los patrones de paralelismo enfocándolos hacia el HPC, y brindar un aporte sobre los elementos que deberían ser tenidos en cuenta a la hora de desarrollar patrones específicos para las aplicaciones bajo estudio; en particular, considerar un tema fundamental en el HPC, el cual es el rendimiento en el uso de los recursos computacionales.

El estado del arte de la tecnología determina que en la actualidad, el entorno de ejecución adecuado para este tipo de aplicaciones es un *cluster* de nodos *multicore*, con un modelo híbrido de programación memoria compartida/memoria distribuida, implementado por medio de la utilización conjunta de las tecnologías OpenMP - MPI.

Por otro lado, también se estudiaron las principales características que presentan los patrones de paralelismo actualmente. En general están referidos a soluciones de paralelización y poco tienen en cuenta cuestiones de implementación. Se los suele categorizar en diversos niveles según el nivel de abstracción que se esté considerando, a saber, el nivel del espacio de concurrencia, el nivel de estructura del algoritmo paralelo y el nivel de implemen-

tación, estos últimos orientados al desarrollo de herramientas de bajo nivel para la implementación de concurrencia.

El análisis del estado del arte de los patrones de paralelismo fue desarrollado en el capítulo 3, destacándose en aquella oportunidad, el escaso nivel de información que presentan los patrones en cuanto al aspecto del rendimiento de la solución y la casi nula referenciación al entorno de ejecución de las soluciones y su impacto en el rendimiento final.

Bajo estas premisas se realizaron experimentos consistentes en la ejecución de diferentes implementaciones en paralelo de algoritmos de álgebra lineal de amplia difusión. Se aplicaron diversos patrones de paralelismo en el proceso de creación del algoritmo paralelo, obteniendo resultados para el modelo de memoria compartida, para el modelo de memoria compartida y distribuida en forma conjunta. También se hizo la distinción de utilizar *Ethernet* o Infiniband como *hardware* de red que conforman el *cluster*.

Las experimentaciones fueron útiles para validar el estado del arte de los patrones de paralelismo, en particular, si estos brindan soluciones a la problemática del HPC para el estado del arte del *hardware*. En base al análisis de los resultados obtenidos por los experimentos, se propusieron modificaciones en la estructura y en el contenido de los patrones actuales, para que puedan cumplir con el objeto de brindar soluciones en el terreno del rendimiento, lo cual fue desarrollado en el capítulo anterior.

Se presentan a continuación las conclusiones de la tesis. Primero se exponen conclusiones específicas de temas puntuales como resultado del análisis individual de los experimentos realizados, para luego, a partir de éstas, determinar conclusiones más generales.

6.1. Conclusiones Específicas

6.1.a) A nivel memoria compartida, si el algoritmo puede derivar en la utilización de rutinas de álgebra lineal, se aconseja utilizar bibliotecas de rutinas que tengan implementadas optimizaciones para entornos *multicore*, como de hecho muchas de estas bibliotecas ya lo hacen, ya que el *speedup* que logran es cercano al óptimo, y no merece dedicar esfuerzos la paralelización a este nivel.

6.1.b) Tener en cuenta la tecnología que utiliza la red que conforma el *cluster* al momento de determinar cómo distribuir datos y tareas. Basado en

los resultados de los experimentos, se puede concluir, que si se está en presencia de un *cluster* con red *Ethernet*, lo aconsejado es la utilización de patrones de comunicaciones colectivas, ya sea porque brinden mejores resultados, o bien porque, a resultados similares con otros esquemas de comunicaciones, el código paralelo es más simple: a igual resultado, tomar la opción mas sencilla de implementar. Por otro lado, si la red subyacente es de tipo Infiniband, utilizar patrones de comunicaciones punto a punto. En este caso, la implementación del algoritmo paralelo es más compleja, pero los resultados de rendimiento son superiores.

Puede pensarse que esta conclusión no aporta demasiado si se la analiza desde un punto de vista más general dado por el concepto *"utilicemos los mecanismos como están diseñados para funcionar mejor"*, lo cual no es nada nuevo. Sin embargo, ningún patrón de paralelismo hace referencia a esta idea. El concepto de patrón de diseño está ligado a ser una solución general, donde los detalles de implementación quedan a nivel de la aplicación del patrón en cada caso en particular. Sin embargo, para el HPC, la implementación de la solución no es un detalle menor, sino la esencia del problema que estudia.

6.1.c) El estado actual de los patrones de paralelismo hace pensar que más que una solución al problema de la optimización, están orientados a ser estrategias de implementación de paralelismo, sin dar una respuesta al rendimiento que se pueda alcanzar, quedando librada la efectividad de su aplicación a la experimentación individual de cada programa. Debe recordarse que el fin de paralelizar es poder obtener mejores rendimientos, por lo que paralelizar sin mejoras no tiene sentido.

6.1.d) Utilizar cómputos parciales en la medida de lo posible, como forma de ganar independencia entre las tareas en que se divide el algoritmo y mejorar el balance de cargas, lo que permite aumentar el *speedup* y solapar comunicaciones con cómputo.

6.1.e) Para la paralelización de una aplicación serial existente, aplicar siempre que sea posible el pseudo-patrón *Reengineering for parallelism pattern*, de la sección 3.2.4, junto con una herramienta de *tracing* y *profiling*. Los resultados que estas herramientas muestran, dan un panorama de la realidad de la paralelización, y que suelen diferir con los supuestos que pueden pensarse de antemano y sin estas herramientas. La determinación de los *hot spots* reales donde poner énfasis en la mejora del algoritmo es solo posible con estas herramientas. La visualización gráfica de la evolución del cómputo permite también analizar otros puntos

de mejora del algoritmo, no necesariamente referidos a la carga computacional, sino más bien, a la secuencia y modalidad de invocaciones de rutinas comunicacionales y al *scheduling* de las tareas, como pudo observarse para el caso de la aplicación del patrón *wavefront* en la multiplicación de matrices

6.2. Conclusiones Generales

A nuestro entender, los patrones de paralelismo, en su estado actual, son buenos en dos aspectos, principalmente:

- Constituyen un excelente punto de partida para el estudio del paralelismo, ya que brindan un panorama completo, jerarquizado e interrelacionado de los conceptos existentes en el dominio.
- Como una herramienta de comunicaciones entre programadores y arquitectos de *software* paralelo ya que permite expresar en forma concisa, ideas complejas.

Por todo lo analizado a lo largo de este trabajo, y recordando que el objetivo de un patrón es el de brindar una solución genérica a un determinado problema, y que el núcleo de la problemática del paralelismo en aplicaciones de cómputo científico es la mejora en los rendimientos del procesamiento, se concluye que los patrones de paralelismo ***no*** están pensados para la mejora de los rendimientos, y por ende, tampoco para la optimización de aplicaciones de cómputo científico, el tipo de aplicaciones objeto de esta tesis. De hecho, el *pseudo* patrón *Reengineering for parallelism pattern*, aconseja que para optimizar, el programador debe seguir un proceso de "prueba y error" hasta que logre resultados aceptables, por lo que más que una solución, deja librado a la experiencia e imaginación del programador, como lograr la mejora.

Puede quizás argumentarse que un patrón no debe dar una solución específica a un problema, y que obtener rendimientos es algo particular del algoritmo considerado y de la plataforma de ejecución. Pero, basado en este argumento, tampoco puede dejarse de lado que el objetivo principal de la paralelización es el rendimiento, y que los patrones de paralelismo enfocados al dominio del HPC, deben tener al rendimiento como elemento central de sus consideraciones.

Las propuestas al respecto del contenido y estructura de los patrones específicos fueron presentadas en el capítulo anterior. Se resumen en que la

sección de **Fuerzas** en los patrones es insuficiente para incluir en ella las consideraciones respecto del tipo de plataforma de ejecución donde la aplicación del patrón genera mejores resultados de rendimiento. Los patrones de paralelismo orientados al HPC deben poseer una sección nueva llamada **"Rendimientos Obtenidos"**, preferentemente dentro de la sección de **Ejemplos**, donde incluir aspectos tales como el impacto en el rendimiento de la conformación del *hardware* y la tecnología de red utilizados, y en general, todo otro tipo de factores que puedan influir en el rendimiento, junto con mesuras de estos efectos, de forma tal que el programador pueda tener una idea más profunda de los rendimientos que pueda alcanzar con los recursos informáticos que dispone, antes de comenzar con el proceso de la paralelización.

Adicionalmente, esta última conclusión se puede extender hacia todo el dominio del HPC y del HTC, ya que ambas, a su manera, tienen como objeto, la mejora de los rendimientos. Es claro que la optimización para HTC no se logra aplicando la misma receta que para el HPC, pero de todas formas, ninguna de las dos ópticas es tenida en cuenta en el estado actual de los patrones de paralelismo.

Si los patrones de paralelismo tuvieran en la actualidad una sección de rendimientos como la planteada, sería de gran ayuda al proceso de ingeniería de software de este tipo de aplicaciones, con el consecuente ahorro de recursos en el desarrollo de aplicaciones paralelas o en la paralelización de aplicaciones seriales existentes, al haber capturado las experiencias previas en cuanto a los resultados obtenidos y las plataformas utilizadas.

6.3. Impacto y trabajos futuros

6.3.1. En la implementación de programas paralelos

Para el programador que trabaja en el desarrollo de aplicaciones paralelas de cómputo científico, se aporta el siguiente resumen de sugerencias:

1. Determinar el volumen de cómputo necesario y en función de este valor, seleccionar el tipo de plataforma puede ejecutarse.
2. Aplicar los patrones de *task decomposition* y *data decomposition* a nivel del espacio de concurrencia para hacerse una idea precisa de la estructura del algoritmo, y luego, acorde a la plataforma determinada

en el punto anterior, definir cómo implementar la división de tareas y la distribución de datos aplicando *task / data grouping* según sea más beneficioso. Tener en cuenta las conclusiones específicas dadas en la sección 6.1

3. Iterar en ciclos de "prueba-error" hasta lograr un rendimiento aceptable, utilizando herramientas de *tracing* y *profiling*.

6.3.2. En los temas de investigación abiertos

En cuanto a temas de investigación donde es necesario profundizar, esta tesis brinda el punto de partida para el estudio e investigación de los siguientes temas:

1. Validación de los resultados obtenidos bajo otros entornos de red, Myrinet por ejemplo.

2. La modelización teórica del patrón *Partial computing* y del cómputo solapado con comunicaciones no bloqueantes.

3. La generación de una rama de los patrones de paralelismo, referida a los patrones de HPC.

4. La necesidad de apertura de nuevas ramas en la Ingeniería de Software, referidas al desarrollo de aplicaciones de cómputo científico paralelas, tanto nuevas, construidas desde cero, como la paralelización de aplicaciones seriales existentes.

6.3.3. Reflexión Final

Para finalizar esta tesis, unas reflexiones finales. Durante mucho tiempo durante el transcurso del desarrollo de este trabajo, no estuvo clara la respuesta a la pregunta ¿Este trabajo es de Ingeniería de Software o de HPC?

La respuesta "oficial" es que pertenece a ambas áreas, a su intersección. La respuesta en épocas iniciales era la trivial: es un trabajo dentro del dominio del HPC, por lo cuanto no queda duda de su pertenencia a esa área. Y es de Ingeniería de Software porque se habla de patrones. La duda era, ¿Cuáles son los puntos que conforman la intersección de ambos dominios?

Existe un gran esfuerzo por parte de profesionales del área de ingeniería de software por propagar las bondades de los patrones sobre los informáticos de HPC [UPC, ABC+06], sin embargo estos últimos no incorporan su utilización. Básicamente porque los únicos patrones que se pueden aplicar son los de paralelismo, y estos no se enfocan a solucionar el problema principal del HPC. Tratar de paralelizar una aplicación de cómputo científico existente, solo sirve, si se mejora el rendimiento. Por lo tanto, ¿Qué es lo que está fallando?

Lo que está fallando es la falta de tiempo de desarrollo de la ingeniería de software en el dominio. La computación paralela, ha dejado de ser un ámbito reservado a un grupo muy reducido de gente, y es cada vez más popular, estimando que en pocos años, quedarán muy pocos tópicos para hablar de computación sin hablar de paralelismo. Sin embargo, la ingeniería de software le ha dedicado muy pocos recursos a su estudio. Existe un gran desarrollo de metodologías y procedimientos para sistemas de gestión, y también, para sistemas embebidos, sistema ubicuos, sistemas web, y otra clase de sistemas. Es mínimo el aporte "efectivo" para aplicaciones de cómputo científico de altas prestaciones.

Por lo tanto, la intersección entre ambas áreas, más que real, es potencial. Hay un gran vacío que llenar en cuanto metodologías de desarrollo de sistemas paralelos de alto rendimiento. Partiendo desde el diseño de un sistema, los patrones de paralelismo surgen desde este lugar. Sin embargo, se vio su insuficiencia para el dominio del HPC. La etapa de desarrollo está basada en "prueba y error", tal como se aconseja en el pseudo-patrón "Reengineering for parallelism pattern": insostenible. Testing: ínfimo.

La intersección existe, como existe para otros dominios de la computación. Solo que falta llenarla. La ingeniería de *software* debe estudiar mejor las características del *software* de HPC para poder desarrollar herramientas útiles a ese dominio. El desarrollo de *software* de HPC tiene mucho para mejorar utilizando técnicas de ingeniería de *software*.

Bibliografía

[ABC+06] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ABD+09] Krste Asanovic, Rastislav BodÃk, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David A. Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine A. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[AIS77] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, 1977.

[Akl89] Selim G. Akl. *The design and analysis of parallel algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Amd67] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings AFIPS Conference Vol. 30*, pages 483–485. AFIPS Press, 1967.

[AMR+05] Rocco Aversa, Beniamino Di Martino, Massimiliano Rak, Salvatore Venticinque, and Umberto Villano. Performance prediction through simulation of a hybrid mpi/openmp application. *Parallel Comput.*, 31(10-12):1013–1033, 2005.

[ARB] OpenMP ARB. The openmp api specification for parallel programming. http://www.openmp.org.

[B+97] L. Blackford et al. Scalapack users' guide. Technical report, SIAM, Philadelphia, 1997.

[BMB] Eric Battenberg, Tim Mattson, and Dai Bui. Dense linear algebra pattern. http://parlab.eecs.berkeley.edu/wiki/ media/patterns/dense.pdf.

[BMK09] Aaron Becker, Phil Miller, and Laxmikant V. Kale. Patterns for overlapping communication and computation. In *ParaPLoP 2009 Workshop on Parallel Programming Patterns*, June 2009.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, August 1996.

[But97] David R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, 1997.

[CDK+00] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[CDPW92] Jaeyoung Choi, Jack Dongarra, Roldan Pozo, and David W. Walker. Lapack working note 55: Scalapack: A scalable linear algebra library for distributed memory concurrent computers. Technical report, Knoxville, TN, USA, 1992.

[CE00] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[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.

[CKS+09] Nicholas Chen, Rajesh Kumar Karmani, Amin Shali, Bor-Yiing Su, and Ralph Johnson. Collective communication patterns. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[cona] *Condor Project Manual*. http://www.cs.wisc.edu/condor/.

[Conb] Open MPI Consortium. Open source high performance computing. http://www.open-mpi.org.

[DPRV08] Jack Dongarra, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Matrix product on heterogeneous master-worker platforms. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 53–62, New York, NY, USA, 2008. ACM.

[Fly72] M. Flynn. Some computer organizations and their affectiveness. *IEEE Trans. on Computers*, 21(9), 1972.

[Fora] Co-Array Fortran. Home page. http://www.co-array.org/.

[Forb] MPI Forum. Message passing interface forum. http://www.mpi-forum.org/.

[Fow96] Martin Fowler. *Analysis Patterns: Reusable Object Models.* Addison-Wesley Professional, October 1996.

[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.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of reusable object-oriented software.* Addison-Wesley, Reading, Mass., USA, 1995.

[GKKG03] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition).* Addison Wesley, 2 edition, January 2003.

[GKP96] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. Pvm and mpi: A comparison of features. *Calculateurs Paralleles*, 8:137–150, 1996.

[GL96] G.H. Golub and C.F.V. Loan. *Matrix computations.* Johns Hopkins studies in the mathematical sciences. Johns Hopkins University Press, 1996.

[GLS99] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface.* Scientific and engineering computation. MIT Press, 1999.

[Gru10] Paul Grun. Introduction to infiniband for end users. Technical report, InfiniBand Trade Association, 2010.

[GVL96] Gene H. Golub and Charles F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[HLR07] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2007. ACM.

[INT] INTEL. Intel math kernel library. http://software.intel.com/en-us/intel-mkl/.

[KCS+09] R. Kumar Karmani, N. Chen, B. Su, A. Shali, and R. Johnson. Barrier synchronization pattern. In *Workshop on Parallel Programming Patterns (ParaPLOP)*, 2009.

[KM09] Kurt Keutzer and Tim Mattson. Our pattern language (opl): A design pattern language for engineering (parallel) software. In *ParaPLop 2009*, Jun. 2009.

[Lei91] Frank T. Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1991.

[MD96] Gerard Meszaros and Jim Doble. Metapatterns: A pattern language for pattern writing. In *In 3rd Pattern Languages of Programming conference*, pages 4–6, 1996.

[MMS00] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Patterns for finding concurrency for parallel application programs. In *7th. Pattern Languages of Programs Conference (PLOP 2000)*, 2000.

[MMS07] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Reengineering for parallelism: an entry point into plpp for legacy applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(4):503–529, 2007.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[Myr] Myrinet. Myrinet open specifications and documentation. http://www.myri.com/open-specs/.

[Ora] Oracle. Oracle solaris studio. http://www.oracle.com/technetwork/server-storage/sunstudio/overview/index.html.

[org] Top 500 org. Top 500 supercomputers sites. http://www.top500.org.

[oT] The University of Tennessee. The plasma project. http://icl.cs.utk.edu/plasma.

[OXJF05] Jeffrey Overbey, Spiros Xanthos, Ralph Johnson, and Brian Foote. Refactorings for fortran and high-performance computing. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, SE-HPCS '05, pages 37–39, New York, NY, USA, 2005. ACM.

[PAR] PARLAB. Parallel computing laboratory. univ. cal.- berkeley. http://parlab.eecs.berkeley.edu/wiki/patterns.

[Pfi01] Gregory Pfister. An introduction to the infiniband architecture. In Hai J., Toni C., and Buyya R., editors, *High Performance Mass Storage and Parallel I/O*, pages 617–632. IEEE Press and Wiley Press, 2001.

[Pre07] W.H. Press. *Numerical recipes: the art of scientific computing*. Cambridge University Press, 2007.

[Qui04] M.J. Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

[SB00] Lorna A. Smith and Mark Bull. Development of mixed mode mpi / openmp applications. In *Scientific Programming, 9(2-3/2001):83-98.Workshop on OpenMP Applications and Tools (WOMPAT 2000)*, pages 6–7, 2000.

[Sca] ScaLAPACK. Home page. http://www.netlib.org/scalapack.

[SDJ+02] Macdonald S., Szafron D., Schaeffer J., Anvik J., Bromling S., and Tan K. Generative design patterns. In *17th IEEE International Conference on Automated Software Engineering (ASE*, pages 23–34, 2002.

[SK09] Carlos Sosa and Brant Knudson. *IBM System Blue Gene Solution: Blue Gene/P Application Development*. IBM RedBooks, 2009.

[Smi00] Lorna A. Smith. Mixed mode mpi / openmp programming. Technical report, Edimburgh Parallel Computing Centre, 2000.

[Snia] Marc Snir. Repositorio personal de patrones de paralelismo. http://www.cs.illinois.edu/ snir/PPP/.

[Snib] Marc Snir. The wavefront pattern. http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf.

[Sof] Automatically Tuned Linear Algebra Software. Home page. http://math-atlas.sourceforge.net/.

[SOHL+96] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[SSGS96] Stephen Siu, Mauricio De Simone, Dhrubajyoti Goswami, and Ajit Singh. Design patterns for parallel programming. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1996, Sunnyvale, California, USA*, pages 230–240, 1996.

[SSJ+02] Bromling S., MacDonald S., Anvik J., Schaeffer J., Szafron D., and Tan K. Pattern-based parallel programming. In *PROCEEDINGS OF THE 2002 INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING*, pages 257–265, 2002.

[SSS96] Ajit Singh, Jonathan Schaeffer, and Duane Szafron. Views on template-based parallel programming. In *CASCON '96: Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, page 35. IBM Press, 1996.

[Sub] Basic Linear Algebra Subprograms. Home page. http://www.netlib.org/blas/.

[Sun] Sun. Sun performance library. http://developers.sun.com /sunstudio/.

[TR05] Fernando G. Tinetti and Fernando Romero. Factorización de matrices cholesky: Paralelización y balance de carga. In *XI Congreso Argentino de Ciencias de la Computación (CACIC)*, Concordia, Entre Ríos, 2005.

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

[TW09] Fernando G. Tinetti and Gustavo Wolfmann. Parallelization analysis on clusters of multicore nodes using shared and distributed memory parallel computing models. In *2009 World Congress on Computer Science and Information Engineering - CSIE 2009, Los Angeles, USA, ISBN 978-7695-3507-4*, pages 466–470, 2009.

[UPC] UPCRC. Universal parallel computing research center - uiuc. http://www.upcrc.illinois.edu/patterns.html.

[VB05] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47:67–95, January 2005.

[Wei97] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C, 2/E.* Addison-Wesley, 1997.

[Wol10] Gustavo Wolfmann. First results in the parallelization of cholesky factorization algorithm over a cluster of multicore computers using partial computing. In *39 Jornadas Argentinas de Informatica - JAIIO 2010*, 2010.

[WT09] Gustavo Wolfmann and Fernando Tinetti. The impact of network architecture in cluster parallel algorithms design: Matrix multiplication on infiniband. In *38 Jornadas Argentinas de Informatica - JAIIO 2009*, 2009.

[YA03] Sherif Yacoub and Hany Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Zel09] M.V. Zelkowitz. *Advances in Computers: Computer Performance Issues.* Advances in Computers. Elsevier Science, 2009.