Universitat Autònoma de Barcelona

Departament d'Arquitectura de Computadors i Sistemes Operatius

Màster en Computació d'Altes Prestacions

Factores de Rendimiento Asociados a SPMD

Memória del trabajo de investigación del "Màster en Computació d'Altes Prestacions", realizada por Claudio Daniel Márquez Pérez, bajo la dirección de Joan Sorribes y Eduardo César. Presentada en la Escuela de Ingeniería (Departamento de Arquitectura Computadores y Sistemas Operativos)

Bellaterra, Julio de 2011

Iniciació a la recerca i traball fi de màster
Máster en Computación de Altas Prestaciones
Curso 2010-2011

Título	:	Factores de Rendimiento Asociados a SPMD
Autor	:	Claudio Daniel Márquez Pérez
Directores	:	Joan Sorribes, Eduardo César

Departamento Arquitectura de Computadores y Sistemas Operativos
Escuela de Ingeniería (EE)
Universidad Autónoma de Barcelona

Claudio Márquez Joan Sorribes Eduardo César

Resumen

Actualmente existen muchas aplicaciones paralelas/distribuidas en las cuales SPMD es el paradigma más usado. Obtener un buen rendimiento en una aplicación paralela de este tipo es uno de los principales desafíos dada la gran cantidad de aplicaciones existentes. Este objetivo no es fácil de resolver ya que existe una gran variedad de configuraciones de hardware, y también la naturaleza de los problemas pueden ser variados así como la forma de implementarlos. En consecuencia, si no se considera adecuadamente la combinación "software/hardware" pueden aparecer problemas inherentes a una aplicación iterativa sin una jerarquía de control definida de acuerdo a este paradigma. En SPMD todos los procesos ejecutan el mismo código pero computan una sección diferente de los datos de entrada. Una solución a un posible problema del rendimiento es proponer una estrategia de balance de carga para homogeneizar el cómputo entre los diferentes procesos. En este trabajo analizamos el *benchmark* CG con cargas heterogéneas con la finalidad de detectar los posibles problemas de rendimiento en una aplicación real. Un factor que determina el rendimiento en esta aplicación es la cantidad de elementos *nonzero* contenida en la sección de matriz asignada a cada proceso. Determinamos que es posible definir una estrategia de balance de carga que puede ser implementada de forma dinámica y demostramos experimentalmente que el rendimiento de la aplicación puede mejorarse de forma significativa con dicha estrategia.

Palabras Claves: SPMD, Modelo de Rendimiento, Computación de Altas Prestaciones, Sintonización, Balance de Carga.

Abstract

There currently are many 'parallel/distributed' applications that use the SPMD paradigm. Getting a good performance in a parallel application of this type is a major challenge because of the large number of existing applications. This objective is not easily achieved because there are many hardware configurations possible, and also the nature of the problems can be varied as well as its implementation. Consequently, if not adequately consider the combination 'software/hardware' inherent problems can occur without an iterative application defined control hierarchy according to this paradigm. In SPMD all processes execute the same code but they compute a different section of the input data. In this paper we analyze the *benchmark* CG with heterogeneous loads in order to detect possible performance problems in a real application. One factor that determines the performance in this application is the number of elements *nonzero* contained in the array section assigned to each process. We determined that it is possible to define a load balancing strategy, which can be implemented dynamically, and we demonstrate experimentally that the application performance can be significantly improved with this approach.

Key Words: SPMD, Performance Model, High Performance Computation, Tuning, Load Balancing.

Resum

Actualment existeixen moltes aplicacions paral·leles/distribuïdes en les quals SPMD és el paradigma més emprat. Obtenir un bon rendiment en una aplicació paral·lela d'aquest tipus és un dels principals reptes donada la gran quantitat d'aplicacions existents. Aquest objectiu no és fàcil de resoldre donat que existeixen una gran varietat de configuracions de hardware, i també la naturalesa dels problemes pot ser variada així com la forma d'implementar-los. En conseqüència, si no es considera adequadament la combinació "software/hardware" poden aparèixer problemes inherents a una aplicació iterativa sense una jerarquia de control definida d'acord a aquest paradigma. En SPMD tots els processos executen el mateix codi però computen una secció diferent de les dades d'entrada. Una solució a un possible problema de rendiment es proposar una estratègia de balanceig de càrrega per homogeneïtzar el còmput entre els diferents processos. En aquest treball analitzem el *benchmark* CG amb càrregues heterogènies amb la finalitat de detectar els possibles problemes de rendiment en una aplicació real. Un factor que determina el rendiment en aquesta aplicació és la quantitat d'elements *nonzero* continguda en la secció de la matriu assignada a cada procés. Es determina que és possible definir una estratègia de balanceig de càrrega que pot ser implementada de forma dinàmica i es demostra de forma experimental que el rendiment de la aplicació pot millorar-se de forma significativa amb aquesta estratègia.

Paraules Plau: SPMD , Model de Rendiment, Computació d'Altes Prestacions, Sintonització, Balanç de Càrrega.

A Cristo Jesús

por darme una Vida en abundancia, sin Tú ayuda nada habría sido posible.

A mi madre

por cuidarme siempre, por velar que nunca me falte nada y por ser un gran regalo de Dios.

A mi padre

por su apoyo y formación en mi vida, por siempre estar orgulloso de mi.

A mis hermanos

por su gran amistad y por su gran apoyo en todo.

A Eduardo, Joan, Ania y Tomas

por su gran ayuda, por sus enseñanzas y por permitirme estar en este excelente grupo.

A Emilio y Lola

por darme esta gran oportunidad de seguir creciendo.

A Roberto, Cesar, Guifrè, Arindam, Claudia, Alvaro, Andrea, Abel y Ronal

por ser muy buenos compañeros, por su amistad y por ayudar en mi aprendizaje.

A todo los miembros de CAOS

por su buen compañerismo.

A Roberto Uribe-Paredes

por apoyarme en la investigación y por nunca cansarse de ayudarme en lo académico.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En la actualidad existe un importante conjunto de problemas de ciencia e ingeniería que pueden ser estudiados y resueltos gracias a que disponemos de rescursos computacionales más y más potentes . Como por ejemplo de estos problemas podemos mencionar la predicción meteorológica, dinámica de fluidos, simulaciones de materiales y estructuras, genoma humano y comportamiento de vehículos en simulaciones.

Los recursos computacionales que permiten estudiar y resolver problemas como los mencionados se engloban bajo el término "Cómputo de Altas Prestaciones" (HPC) y, en general, consisten de una gran cantidad de procesadores o *cores* (más de medio millón en el *K-Computer*) conectados por redes de alta velocidad y capacidad.

Las aplicaciones que se diseñan para estos sistemas deben tomar en cuenta de forma explícita aspectos relacionados con el uso eficiente de los recursos, ya que, en casi contrario, no se obtendrían las prestaciones necesarias para resolver los problemas en un tiempo razonable.

El obtener un buen rendimiento no sólo depende del problema a resolver, sino también, podemos agregar a esto la dificultad que tiene el implementar una aplicación paralela sobre cualquiera de las variedades del hardware existente. Por lo tanto la implementación de aplicaciones paralelas no es una tarea fácil, más a aun, si busca lograr buenos índices de rendimiento.

Aspectos como la descomposición de tareas, el *mapping*, la concurrencia, la escalabilidad, la eficiencia, la planificación, etc., se suman al conocimiento que debe adquirirse sobre librerías de comunicación y de modelos de programación para diseñar y desarrollar soluciones paralelas [38].

En contraste con una aplicación secuencial encontramos nuevos problema en todas las fases de desarrollo de este tipo de aplicaciones. Las aplicaciones paralelas no sólo son difíciles de desarrollar, sino también el analizar, comprender y depurar. Debido a esto los desarrolladores muchas veces se valen de herramientas adicionales para observar el comportamiento de sus aplicaciones, y posterior a esto comprender los resultados, modificar, recompilar y reiniciar su aplicación para resolver alguna situación de ineficiencia.

Para corregir problemas de ineficiencias, necesitamos de las fases de monitorización, análisis y modificación de código [48]. En la monitorización se obtiene la información o medidas de rendimiento sobre el comportamiento de la aplicación. Posterior a esto, en el análisis de las medidas de rendimiento, se buscan causas de posibles *bottleneck* (cuellos de botella), y soluciones para corregir los problemas de ineficiencia. Finalmente, se aplican los cambios decididos sobre el código para mejorar el rendimiento. Como consecuencia, los usuarios finales están forzados a conocer muy bien la aplicación, las diferentes capas software involucradas y el comportamiento del sistema distribuido sobre el que se ejecuta la aplicación.

Todos estos aspectos hacen que el proceso de mejora de rendimiento sea difícil y costoso, especialmente para usuarios no expertos. Además, los orígenes de los problemas de rendimiento pueden estar en diferentes niveles como lo son: comunicaciones, librería, diseño o implementación, Sistema Operativo, capacidades del hardware, etc.

Por lo tanto se hace necesario usar herramientas automáticas para acelerar y simplificar el proceso de sintonización del rendimiento. Afortunadamente estas herramientas han sido diseñadas con el enfoque de hacer más cómodas las diferentes fases que involucran el análisis y una posterior sintonización [49],[57].

Para poder hacer la sintonización es importante conocer cuál es la mejor vía para analizar y sintonizar la aplicación. Es importante muchas veces obtener características de comportamiento, conocimiento del entorno y de la aplicación. Sin duda, conocer la estructura o paradigma de programación de la aplicación paralela resulta de gran ayuda para poder reconocer problemas comunes de rendimiento asociados a estos [18]. Como por ejemplo cuellos de botella en el proceso *master*, para el caso del *master/worker*.

La mayor parte de los algoritmos pueden ser clasificados en estructuras por sus patrones de interacción [9]. Entre estas estructuras están: *master/worker* ,*pipeline*, SPMD, *divide and conquer*, entre otros. Siendo SPMD (Single Program Multiple Data) el paradigma más utilizado en las aplicaciones paralelas [69] debido a las sencilles del paradigma y a la gran escalabilidad que posee.

Uno de los temas críticos al momento de sintonizar es seleccionar una adecuada estrategia de balance de carga, ya sea estática o dinámica. Cuando la carga de trabajo en una aplicación es previsible y constante en el tiempo, una estrategia de balance de carga estático al inicio de

la ejecución (*static load balancing*) es suficiente. Mientras que un balance de carga dinámico (*dynamic load balancing*) es necesario cuando se desconoce el costo computacional de cada sección de cómputo, por creación dinámica de tareas, por migración de tareas o variaciones en los recursos hardware por cargas externas[62].

Estudios desarrollados por el grupo *Entornos para Evaluación de Rendimientos y Sintonización de Aplicaciones* de la Universidad Autónoma de Barcelona [18], demuestran que es posible obtener mejorar el rendimiento utilizando como vía el desarrollar modelos de rendimiento a partir del conocimiento de estos paradigma de programación (*master/worker* [17], *pipeline* [58]), para ser integrados como fuente de conocimiento en el entorno de sintonización de rendimiento dinámico MATE(Monitoring Analysis and Tuning Environment) [57].

Estos modelos de rendimiento tienen como finalidad la ejecución eficiente de las aplicaciones paralelas, componente importante en estos modelos son la predicciones de rendimiento a través de expresiones analíticas o estrategias (basada en heurísticas, en metodologías, en estadísticas, etc.) , una serie de parámetros correspondientes a puntos de medida para evaluar dichas expresiones y puntos de acción o de sintonización para mejorar el rendimiento.

El desarrollar modelos de rendimiento para aplicaciones bajo el paradigma SPMD con la finalidad de lograr una sintonización dinámica como en [18], puede brindarnos el conocimiento necesario para utilizar el entorno MATE con aplicaciones SPMD, por consiguiente, mejorar el rendimiento de algunos problemas típicos en HPC.

También diseñar estos modelos de rendimiento resulta un gran desafío, ya que se debe escoger bien que es lo relevante que se debe abstraer para definir los alcances y limitaciones.

En general, se pueden hacer modelos de rendimiento a través de un "modelado por simulación"que se basa en una aplicación que emula el hardware y el comportamiento temporal de la aplicación, el simular una aplicación paralela puede ser muy costoso y difícil de utilizar en herramientas de evaluación de rendimiento. Otra forma es a través del "modelado analítico"que consiste en modelar el hardware y el algoritmo a través de expresiones matemáticas, de esta forma la fiabilidad queda mermada a la calidad de los parámetros elegidos. También podemos desarrollar un "modelo mediante la obtención de métricas y trazas"para obtener resultados más precisos, ya que se puede ver su comportamiento en un entorno real. Normalmente esta última forma se usa para extraer los parámetros que luego se utilizarán en modelos analíticos, simulaciones, para validaciones, etc.

El presente trabajo de investigación de máster tiene como eje principal el detectar factores de rendimiento asociados al paradigma SPMD para el posterior desarrollo de un modelo de rendimiento para SPMD. Por lo tanto, buscamos conocer ciertos parámetros necesarios para caracterizar el funcionamiento a través de una aplicación, con el fin de detectar problemas típicos y factores que determinan el funcionamiento de aplicaciones SPMD.

1.1 Objetivos

En HPC reducir el tiempo de ejecución a través del uso eficiente de los recursos de cómputo es uno de los desafíos más importantes, en este caso, en la gran cantidad de las aplicaciones científicas que están implementadas bajo en paradigma SPMD. Por esto, estudiar el funcionamiento, los factores que influyen en el rendimiento y los posibles métodos de sintonización para las aplicaciones implementadas con este paradigma, se presentan como áreas de investigación abierta y de interés en la comunidad científica.

El objetivo a largo plazo consiste en la creación de un modelo de rendimiento para aplicaciones SPMD que permita mejorar sus prestaciones dinámicamente. Este primer año estamos centrados en sentar las bases para lograr esta meta.

Como objetivo a corto plazo nos hemos propuesto, durante este primer año, analizar los factores que influyen en el rendimiento de aplicaciones SPMD. COo este objetivo hemos realizados las siguientes tareas: estudiar el paradigma, analizar una aplicación representativa, detectar factores que influyen en su rendimiento y evaluar estrategias de mejora.

Así hemos investigado otros trabajos relacionados, posterior a esto analizamos una aplicación del tipo Benchmark reconocida por la comunidad científica, perteneciente a los Benchmark de NAS. Por último, buscamos reconocer factores de rendimiento y presentar una propuesta de sintonización.

1.2 Organización del Trabajo

Hemos organizado el contenido de esta memoria en 5 capítulos, comenzando por el Capítulo 1 de introducción.

En el Capítulo 2 se expone el contexto de los Entornos Paralelos de HPC, explicando las diferentes arquitecturas, paradigmas de programación y las aproximaciones existentes para el Análisis de Rendimiento y Modelos de Rendimiento.

Luego, en el Capítulo 3, exponemos en mayor profundidad el paradigma SPMD y los elementos a analizar para alcanzar los objetivos planteados para esta memoria.

En el Capítulo 4 se describirá la aplicación estudiada con la finalidad de entender su funcionamiento y plantear una estrategia de balance de carga.

A continuación, en el Capítulo 5, se presentará una evaluación experimental para demostrar que es posible mover carga en una aplicación SPMD como la presentad, es decir, es posible sintonizarla para mejorar su tiempo de ejecución. También se expondrá el factor de rendimiento detectado para esta aplicación SPMD, lo cual servirá como fundamento para la culminación de toda la investigación a largo plazo.

Finalmente, se presentarán las conclusiones y las lineas de trabajo futuro.

1.3 Trabajos Relacionados

Dado que nuestro trabajo se centra en el desarrollo de modelos de rendimiento asociados a la estructura de la aplicación, con el objetivo de ser integradas en herramientas automáticas de sintonización de rendimiento, centraremos este apartado en la descripción de trabajos relacionados sobre herramientas de análisis y sintonización y, también, sobre la definición de modelos de rendimiento.

1.3.1 Sintonización dinámica

La ejecución de una aplicación paralela puede ser varias veces más rápida que su versión secuencial, pero no siempre el rendimiento es aceptable. Es más es bastante probable que si no se ha efectuado una sintonización apropiada los resultados de rendimiento obtenidos sean decepcionantes. Para hacer una sintonización apropiada, es necesario tener un conocimiento profundo de la aplicación y del entorno de ejecución, así como un alto grado de especialización.

Debido a la complejidad de esta tarea se necesita del apoyo de herramientas para monitorizar la aplicación, con el fin de realizar el seguimiento del rendimiento, analizar la ejecución y obtener factores relevantes, con la finalidad de poder detectar los posibles *bottleneck* (cuellos de botella) en nuestra aplicación.

Entre las herramientas para el apoyo al análisis tenemos un grupo basado en trazas estáticas, entre las cuales podemos mencionar: Vampir [47], Tape/PVM [52] Y XPVM [36], ParaGraph [42], Jumpshot [77] y Pablo [65]. Un enfoque más amplio de apoyo al análisis de rendimiento *post-mortem* es un análisis de funcionamiento automático, mediante la adición de algún grado de conocimiento a la herramienta, esto se ha propuesto en varios estudios,

como KappaPi [31], Paradise [50], Expert [74] y AIMS [76].

Por lo general, estas herramientas trabajan a través de archivos de trazas y con algunos conocimientos heurísticos para descubrir problemas simples y algunos no tan simples. Los principales problemas relacionados con este enfoque son la generación y almacenamiento de archivos de trazas posiblemente de tamaño enorme, otro problema es el *overhead* agregado por la instrumentación necesario para reunir los datos de rendimiento, y también muchas veces estas instrumentaciones deben ser introducidas directamente por el programador.

Es importante tener en cuenta que la mejor manera de analizar y sintonizar, dependerá de características de comportamiento de la aplicación. Cuando las aplicaciones muestran un comportamiento dinámico se requiere de una vigilancia dinámica y técnicas de optimización. Sin embargo, hacer esto tampoco es sencillo, ya que se debe controlar la sintonización de forma dinámica, se deben tomar decisiones de manera eficiente. Los resultados del análisis deben ser exactos y la sintonización se debe aplicar en el tiempo oportuno con la finalidad de abordar eficazmente los problemas. Al mismo tiempo, la intrusión en la aplicación debe reducirse al mínimo para que al monitorizar y sintonizar la aplicación no aparezcan problemas que eran ausentes antes de la instrumentación.

Esto es difícil de lograr si no se tiene información sobre la estructura y el comportamiento de la aplicación, por lo tanto, un enfoque automático a ciegas de sintonización dinámica tendrá un éxito muy limitado, mientras que un enfoque cooperativo de sintonización dinámica puede hacer frente a problemas más complejos, pero la colaboración necesaria por parte del usuario se convierte en un costo adicional.

En resumen la sintonización dinámica consiste en la inserción manual o automática de

cambios en el código fuente de la aplicación en tiempo de ejecución. Si además, la sintonización dinámica es automática permite desvincular al desarrollador de la difícil tarea de sintonizar su aplicación. La sintonización automática consiste en realizar de forma automatizada todas las tareas necesarias y relacionadas con la optimización de la aplicación. Una revisión más extensa sobre herramientas de sintonización se encuentra descrita en [53].

La aproximación más cercana al trabajo conjunto entre modelos y herramientas es MATE [57] (Monitoring, Analysis and Tuning Environment). MATE es una herramienta que implementa sintonización automática y dinámica de aplicaciones paralelas. Este entorno ha sido diseñado e implementado por el grupo de *Entornos para Evaluación de Rendimientos y Sintonización de Aplicaciones* del departamento (CAOS) de la Universidad Autónoma de Barcelona. Su objetivo es mejorar el rendimiento de una aplicación paralela en tiempo de ejecución, adaptándola a las condiciones variables del sistema sobre el que se ejecuta. MATE constituye la principal motivación para la investigación de modelos de rendimiento en nuestro grupo de investigación, esto debido a que una de las fuentes de inteligencia de MATE se obtiene del modelo de rendimiento que tenga implementado. Según el modelo de rendimiento utilizado, en conjunto con los parámetros de entrada y las medidas de la aplicación obtenidas se tomaran las decisiones para obtener un mejor rendimiento en la aplicación.

La Universidad de Manchester en el año 2005 comenzó a desarrollar PerCo [54]. Es un *framework* para el control del rendimiento en entornos heterogéneos. Es capaz de gestionar la ejecución distribuida de aplicaciones usando migraciones, por ejemplo, en respuesta a cambios en el entorno de ejecución. PerCo monitoriza los tiempos de ejecución y reacciona de forma acorde a una estrategia de control para adaptar el rendimiento cuando tienen lugar cambios importantes en el rendimiento. Y está orientada a aplicaciones utilizadas en modelos de simulación científica [46] y búsqueda distribuida en control estadístico.

La Universidad de Maryland en el año 2002 implementó el *framework* Active Harmony [24], este permite la adecuación dinámica de una aplicación a la red y a los recursos disponibles, mediante la adaptación automática de algoritmos, distribución de datos y balanceo de carga. Se basa en un modelo *cliente-servidor*, en donde el cliente es la aplicación y la herramienta es el servidor, la cual tomará las decisiones correspondientes en función de la información recibida del cliente.

Autopilot [66] es una infraestructura software desarrollada por la Universidad de Illinois en 1998 para la sintonización dinámica del rendimiento de entornos computacionales heterogéneos basada en bucles de control cerrados. Se basa fundamentalmente en la aplicación de técnicas de control en tiempo real para adaptar dinámicamente el sistema a las diferentes demandas y disponibilidad de recursos. La creación de esta infraestructura has sido posible a través del conocimiento obtenido al realizar por la misma universidad el entorno de análisis de rendimiento Pablo [65].

RMS [37] (Resource Managment System) es una aproximación basada en gestión de recursos, presentada en el año 2005 por la Universidad del Estado de Mississippi, USA. Permite hacer una gestión de recursos en forma adaptativa según la necesidad, por lo cual, permite mejorar el rendimiento en aplicaciones del tipo *Malleable* o *Adaptativas* [32, 27], las cuales tienen mucha variabilidad en el consumo de recursos.

En [3] el departamento de *Ciencia de la Computación* de la Universidad de Pisa ha desarrollado herramientas de rendimiento basadas en una tecnología de flujo de datos, en la cual se aplican diferentes mecanismos para mejorar el rendimiento y su posterior evaluación sobre cálculos basados en *Skeletons* [29].

En [21] se propone un *framework* que dinámicamente se adapta a los cambios en los recursos en un ambiente distribuido, incluyendo adaptabilidad mediante *load balancing* y *fault tolerance*. También en esta aproximación se afirma que pueden mejorar el rendimiento de una aplicación en un ambiente heterogéno mediante estratégias locales y globales. Este *framework* esta desarrollado por el grupo de la Universidad de Manchester en 2007.

1.3.2 Modelos de rendimiento

Un modelo de rendimiento básicamente es una expresión matemática que representa aspectos específicos y claves dentro del un sistema computacional [40]. Con ellos se puede analizar fácilmente el comportamiento de una aplicación y permiten evaluar posibles ejecuciones con configuraciones diferentes de la aplicación así como también del hardware. En nuestro caso los modelos de rendimiento constituyen el medio por el cual podemos obtener el conocimiento del funcionamiento de la aplicación para posteriormente hacer las correcciones necesarias para incrementar las prestaciones.

A continuación se presentan algunos trabajados desarrollados por el grupo de investigación *Entornos para Evaluación de Rendimientos y Sintonización de Aplicaciones* del departamento del departamento (CAOS), algunos modelos de SPMD desarrollado por otros grupos de investigación y trabajos que apuntan a un objetivo similar.

En [17] se desarrollan una serie de métodos de distribución de datos para *framework master/worker*, en el cual se presenta un modelo de rendimiento que consta en una estrategias para lograr un equilibrio de computo en relación carga computacional, una estrategia para adaptar el número de *workers*, y una expresión analítica que permite la combinación de ambos. Una

de las características principales de este modelo es que ha sido probado en el entorno MATE con buenos resultados. En [19] se presenta un modelo de rendimiento para *pipeline* integrado también al entorno de de sintonización dinámica MATE, obteniendo buenos resultados.

El *framework* SAMBA (Single Application, Multiple Load Balancing) [64], muestra una batería de estrategias de *load balancing* en diferentes aplicaciones SPMD para mejorar el rendimiento de una aplicación, el énfasis central en este *framework* es la identificación de una estrategia adecuada de *load balancing*.

En [15] a través de una serie de herramientas en conjunto con una librería se presenta la generación de modelos de rendimiento para SPMD, modelando las fases de comunicación y de computo en la aplicación. Estos modelos generados están muy ligados a una arquitectura y aplicación especifica, por lo cual son escasamente intercambiables a otras aplicaciones y entornos.

El grupo informático de Universidad de Edinburgh liderado por "Murray Cole" ha desarrollado varios trabajos sobre *performance tuning* usando el conocimiento provisto al implementar las aplicaciones mediante *Skeletons* [23].

Junto con el anterior podemos mencionar a la Universidad de la Laguna, que han trabajado en el desarrollo de herramientas de programación basadas en *frameworks*, paralelizando compiladores y desarrollando modelados de rendimiento para diferentes aplicaciones [29].

En Lyon France, el grupo del Laboratorio de Paralelismo Informático ha desarrollado trabajos entorno a la búsqueda de un mejor rendimiento en modelos *pipeline*, utilizando *clusters* heterogéneos [10]. Esto lo logran mediante mecanismos de replicación, *mapping* y *scheduling*.

También en el departamento CAOS de la Universidad Autónoma de Barcelona en [59], ha desarrollado una Metodología para ejecución eficiente para SPMD, la cual bajo determinadas condiciones permite seleccionar una mejor configuración para obtener mejores prestaciones. Esta metodología utiliza una serie de estrategias basadas en *mapping*, políticas de *scheduling* para lograr un solapamiento entre el cómputo y la comunicación.

Capítulo 2

Sistemas paralelos distribuidos

2.1 Introducción

El objetivo del procesamiento paralelo es mejorar la productividad (cantidad de aplicaciones ejecutadas por unidad de tiempo) y/o la eficiencia (disminuir el tiempo de ejecución de la aplicación). Para poder paralelizar una aplicación es necesario explotar la concurrencia presente en la misma con el objetivo de distribuir el trabajo a cada nodo de cómputo y así conseguir un menor tiempo de ejecución de la aplicación. Es decir, la computación paralela permite que las aplicaciones se puedan dividir en varias tareas pequeñas para una ejecución simultánea. Desarrollar aplicaciones de este tipo no es sencillo, pero los beneficios en cuanto a resultados obtenidos, las hacen indispensables para determinados problemas.

Durante la última década la computación paralela se ha expandido a nuevas áreas, tales como: Financiera, Aeroespacial, Energía, Telecomunicaciones, Servicios de Información, Defensa, Geofísica, Investigación, etc. La evolución del hardware y software necesario para estos nuevos problemas ha generado nuevos desafíos a abordar, y también se ha podido dar solución a problemas antes intratables.

Si bien el hardware de altas prestaciones ha tenido una escalabilidad inmensa desde sus inicios, siguiendo la ley de Moore o incluso superándola durante mucho tiempo, debido a la irrupción de los sistemas basados en *cluster* y *grid*, el software se ha visto frenado por diversas razones, tales como la complejidad de los propios sistemas hardware, la complejidad de su desarrollo, de su implementación y de su mantenimiento. Para que todo esto sea posible, hay que continuar explorando las mejoras posibles a realizar en los sistemas paralelos y/o distribuidos.

Desgraciadamente, sostener buenos índices prestaciones no es simple, más aún, si queremos acercar estos índices a valores óptimos teóricos. Las razones de estas dificultades, tanto para el desarrollo como para una sintonización, están en las complejas interacciones que las aplicaciones soportan con el sistema físico, tales como: el software de sistema, las librerías, Sistema Operativo, las interfaces de programación usadas y los algoritmos implementados.

Si deseamos contruir un modelo de rendimiento para SPMD necesitamos comprender todas estas interacciones, cómo mejorarlas y cómo detectar las ineficiencias. Esto es una tarea demasiado costosa, además estas ineficiencias podrían estar muy ligadas al ambiente analizado. Hoy podemos encontrar mucho estudios de factores o métricas que influyen en el rendimiento [49]. Según lo demostrado en [18] podemos maximizar las prestaciones enfocándonos en los factores asociados a un paradigma y aplicación en concreto. Por lo tanto, en este trabajo nos centraremos en reconocer que factores son los que determinan ineficiencias a nivel de la aplicación y a nivel de paradigmas de programación, sin dejar de lado las diferentes alternativas de construcción de sistemas paralelos.

2.2 Modelos Arquitecturales

Existen muchas formas de construir sistemas paralelos, debido a la variedad de sus características se pueden clasificar en diferentes taxonomías. En función de los conjuntos de instrucciones y datos, y frente a la arquitectura secuencial que denominaríamos SISD (Single Instruction Single Data), podemos clasificar las diferentes arquitecturas paralelas en diferentes grupos (a esta clasificación se la suele denominar taxonomía de Flynn) [68, 4, 55, 71, 72]: SIMD (Single instruction Multiple Data), MISD (Multiple Instruction Single Data), MIMD (Multiple Instruction Multiple Data) y SPMD (Single Program Multiple Data) que es un MIMD con algunas variaciones.

SIMD : En una arquitectura SIMD unos procesos homogéneos (con el mismo código) síncronamente ejecutan la misma instrucción sobre sus datos, o bien la misma operación se aplica sobre unos vectores de tamaño fijo o variable.

MISD : El mismo conjunto de datos es tratado de forma diferente por los procesadores. Son útiles en casos donde sobre el mismo conjunto de datos se deban realizar muchas operaciones diferentes.

MIMD : En este enfoque no sólo distribuimos datos sino también las tareas a realizar entre los diferentes procesadores. Varios flujos (posiblemente diferentes) de ejecución son aplicados a diferentes conjuntos de datos.

SPMD : En paralelismo de datos [44], utilizamos un mismo código, sobre secciones diferentes de datos. Hacemos varias instancias de las mismas tareas, cada uno ejecutando el código de forma independiente como se expone en detalle en el capítulo 3.

Esta clasificación de modelos arquitecturales se suele ampliar para incluir diversas categorías de computadoras que no se ajustan totalmente a cada uno de estos modelos. Una clasificación extendida de Flynn es presentada en [30].

Otra clasificación de los modelos arquitecturales es la basada en los mecanismos de control y la organización del espacio de direcciones [51, 34, 25]. Utilizando este enfoque se puede realizar una clasificación según la organización de la memoria (que puede ser física y lógica). Por ejemplo, disponer de una organización física de memoria distribuida, pero lógicamente compartida.

En el caso de los multiprocesadores podemos hacer una subdivisión entre sistemas fuertemente o débilmente acoplados. En un sistema fuertemente acoplado, el sistema ofrece un mismo tiempo de acceso a memoria para cada procesador. Este sistema puede ser implementado a través de un gran módulo único de memoria, o por un conjunto de módulos de memoria de forma que puedan ser accedidos en paralelo por los diferentes procesadores (UMA). En un sistema ligeramente acoplado, el sistema de memoria está repartido entre los procesadores, disponiendo cada uno de su memoria local. Cada procesador puede acceder a su memoria local y a la de los otros procesadores (NUMA).

El incremento de la complejidad de las aplicaciones de cálculo científico y el incremento en el volumen de la información, ha originado la necesidad de construir sistemas con multiples unidades que intregran estos modelos arquitecturales. Estos sistemas se conocen como *cluster* de cómputo y permiten que las aplicaciones que necesitan mucho tiempo de cómputo se puedan ejecutar en menores tiempos.

2.3 Cluster de Cómputo

Las aplicaciones SPMD en su gran mayoria funcionan en *clusters* aprovechando la escalabilidad y los recursos de cómputo proporcionados por los mismo. Un *cluster* de cómputo es un tipo de sistema de procesamiento paralelo o distribuido, formado por una serie de unidades de cómputo (nodos) que ejecutan una serie de aplicaciones de forma conjunta. Está formando por un conjunto de recursos integrados de computación para ejecutar aplicaciones en menores tiempos de cómputo [16].

Debido al aumento de los sistemas que necesitan redes de comunicación, se ha desencadenado un amplio mercado y una comercialización más extendida. En consecuencia el hardware utilizado en estos sistemas ha reducido su costo, de manera que podemos montar una máquina paralela de forma relativamente económica. De hecho, buena parte de los sistemas actuales de altas prestaciones [1] son sistemas del tipo *cluster*, con redes de interconexión más o menos dedicadas.

En [4, 28] se definen como una colección de computadores conectados por alguna tecnología de comunicaciones como redes (Gigabit) Ethernet, Fiber Channel, ATM, etc. Un *cluster* está controlado por una entidad administrativa que tiene el control completo sobre cada sistema final.

Si quisieriamos clasificarlos según algún "Modelo Arquitectural", los *clusters* de cómputo son sistemas MIMD con memoria distribuida, aunque la interconexión puede no ser dedicada, y utilizar mecanismos de interconexión simples de red local, o incluso a veces no dedicados exclusivamente al conjunto de máquinas. Básicamente es un modelo DM-MIMD (Distributed Memory-MIMD) en donde las comunicaciones entre procesadores son habitualmente son más

lentas.

En un *cluster*, podemos aprovechar los tiempos muertos en algunas unidades de cómputo en otras aplicaciones paralelas, sin que esto perjudique a los usuarios. Esto también nos permite diferenciar entre los *clusters* dedicados (o no), dependiendo de la existencia de usuarios (y por tanto de sus aplicaciones) conjuntamente con la ejecución de aplicaciones paralelas. Así mismo, también podemos diferenciar entre dedicación del *cluster* a una sola o varias aplicaciones paralelas simultaneas en ejecución.

Una ventaja importante que encontramos en los *clusters* es que son bastante escalables en cuanto al hardware. Podríamos decir que es fácil construir sistemas con centenares o miles de máquinas, donde comúnmente las tecnologías de red son las que limitan la capacidad de escalabilidad. Aunque no es habitual disponer de hardware tolerante a fallos, normalmente se incorporan mecanismos por software que invalidan (y/o substituyen por otro) el recurso que ha fallado.

Para que el hardware de red permita la escalalibidad debe estar necesariamente diseñado para el procesamiento paralelo, es decir, minimizar las latencias y maximizar el ancho de banda. Muchas veces encontramos que el software base montado sobre los *clusters* suelen ser diseñados para máquinas específicas, como es el caso del sistema operativo, normalmente no ofrecen posibilidades para llevar un mayor control de los recursos. En estos casos es necesario incorporar toda una serie de capas de servicios *middleware* sobre el sistema operativo, para hacer que los sistemas *cluster* sean eficientes. En sistemas a mayor escala, es necesario incluir toda una serie de mecanismos de control y gestión adicionales, para el *scheduling* [13] y monitorización de los sistemas, como es el caso en los sistemas *grid* [35]. En general esto supone una complejidad muy grande del software de sistema, que aumenta sensiblemente la

complejidad total, y tiene una repercusión significativa sobre las prestaciones en estos sistemas.

Actualmente existe una gran variedad de configuraciones de hardware en los *clusters* debido a que la construcción de estos no tiene mayores restricciones de diseño o de alcanze. Por lo cuál, también la forma de programar un *cluster* puede variar dependiendo de que recursos se desean utilizar o bien que recursos se desean interconectar.

2.4 Modelos de Programación

Las aplicaciones paralelas consisten en un conjunto de tareas que pueden comunicarse y cooperar para resolver un problema. Debido a la fuerte dependencia de las arquitecturas de las máquinas que se usen, y los paradigmas de programación usados en su implementación, no hay una metodología claramente establecida para la creación de aplicaciones paralelas. En [34], se define la creación de una aplicación mediante cuatro etapas: Partición, Comunicación, Aglomeración y *Mapping*.

En la etapa de Partición el cómputo a realizar y los datos a operar son descompuestos en pequeñas tareas. Durante la etapa de Comunicación se establecen las estructuras de datos necesarias, protocolos, y algoritmos para coordinar la ejecución de las tareas. La Aglomeración especifica, si es necesario, que las tareas sean combinadas en tareas mayores, si con esto se consigue reducir los costes de comunicación y aumentar las prestaciones. En el *Mapping* cada tarea es asignada a un procesador de manera que se intenta satisfacer los objetivos de maximizar la utilización del procesador, y minimizar los costes de comunicación.

Estas etapas las podemos subdividir en dos grupos, las dos primeras se enfocan en la

concurrencia y la escalabilidad, y se pretende desarrollar algoritmos que primen estas características. En las dos últimas etapas, la atención se desplaza a la localidad y las prestaciones ofrecidas.

Como resultado de las tareas anteriores, tendremos que tomar la decisión correspondiente al paradigma de programación que usaremos (ver sección 2.5). Nos estamos refiriendo a una clase de algoritmos que tienen la misma estructura de control [41, 56], y que pueden ser implementados usando un modelo genérico de programación paralela.

Finalmente obtendremos prestaciones combinando el paradigma con el modelo de programación sobre una arquitectura, es decir, de la combinación de hardware paralelo, red de interconexión y software de sistema disponibles. A continuación presentaremos la división de los modelos de programación paralela [51, 34].

2.4.1 Memoria Compartida

En este modelo consiste en una colección de procesos accediendo a variables locales y un conjunto de variables compartidas [8, 20]. Cada proceso accede a los datos compartidos mediante una lectura/escritura asíncrona. Este modelo requiere de mecanismos para resolver los problemas de exclusión mutua que se puedan plantear (mediante mecanismos de semáforos o bloqueos), principalmente debido a la concurrencia de acceso a los datos.

Las tareas son asignadas a *threads* de ejecución en forma asíncrona. Estos *threads* poseen acceso al espacio compartido de memoria, con los mecanismos de control citados anteriormente.

Una de las implementaciones más utilizadas para programar bajo este modelo es OpenMP (Open Specifications for Multi Processing) [75], comunmente utilizado en sistemas de tipo SMP [45] o SM-MIMD (Shared Memory-MIMD). En OpenMP se usa un modelo de ejecución paralela denominado *fork-join*, básicamente el *thread* principal, que comienza como único proceso, realiza en un momento determinado una operación *fork* para crear una región paralela de un conjunto de *threads*, que acaba mediante una sincronización por una operación *join*, reasumiéndose el *thread* principal de forma secuencial, hasta la siguiente región paralela.

2.4.2 Paso de Mensajes

Mediante un modelo de paso de mensajes los programas son organizados como un conjunto de tareas con variables locales privadas, que también tienen la habilidad de enviar y recibir datos entre tareas por medio del intercambio de mensajes. Las aplicaciones de paso de mensaje tienen como atributo básico que los procesos poseen un espacio de direcciones distribuido a diferencia de las aplicaciones de memoria compartida. Este es el modelo más ampliamente usado en HPC [51, 73].

El paso de mensajes permite una mayor expresión disponible para los algoritmos paralelos, proporcionando control no habitual, ya que el programador es quien decide cómo y cuándo se efectuaran las comunicaciones, es decir, es el programador quien de forma directa controla el flujo de las operaciones y los datos. También deja al programador la capacidad de tener un control explícito sobre la localidad de los datos, debiendo tomar precauciones para obtener buenas prestaciones a través del manejo de la jerarquía de memoria.

Como la gran responsabilidad en este modelo recae en el programador, la implementación se debe hacer tomando varios aspectos en cuenta, como lo son: una buena distribución de

datos, las comunicaciones entre tareas, sus puntos de sincronización y las operaciones de I/O si las hubiera. La idea es tratar de evitar las dependencias de datos, *deadlocks* y comunicaciones ineficientes, así como implementar mecanismos de tolerancia a fallos [12, 2] para sus aplicaciones.

El medio más usado para implementar este modelo de programación es a través de una librería que implementa la API de primitivas habitual en entornos de paso de mensajes. Hoy en día existen varias aproximaciones [55] de librerías que soportan el modelo de paso de mensajes, algunas de ellas PVM [36], BSP [78] y MPI [39]. Sin embargo, MPI es una especificación, que los entornos que la implementen deben cumplir, dejando al fabricante el cómo implementar el entorno de soporte para la ejecución.

2.5 Paradigmas de Programación

Los paradigmas de programación paralelos son clases de algoritmos que solucionan diferentes problemas bajo las mismas estructuras de control [41]. Cada uno de estos paradigmas definen modelos de cómo paralelizar una aplicación mediante la descripción general de cómo distribuir los datos y de cómo es la interacción entre las unidades de cómputo.

Al momento de diseñar una aplicación paralela es importante conocer que alternativas de paradigmas existen, así cómo conocer sus ventajas y desventajas, puede ser que dependiendo del problema sea innecesario utilizar un tipo de paradigma o bien la utilización de un determinado paradigma disminuiran las prestaciones en comparación a otro. Hay diferentes clasificaciones de paradigmas de programación, pero un subconjunto habitual [16] en todas ellas es:

2.5.1 Divide and Conquer

En este paradigma el problema es dividido en una serie de subproblemas. Se realizan tres operaciones: dividir, computar y unir. La estructura del programa es del tipo árbol, siendo los subproblemas las hojas que ejecutarán los procesos o *threads*. Cada uno de estos subproblemas se soluciona independientemente y sus resultados son combinados para producir el resultado final. En este paradigma, se puede realizar una descomposición recursiva hasta que no puedan subdividirse más y la resolución del problema se hace a través de operaciones de partición (*split* o *fork*), computación, y unión (*join*). También se trabaja con parámetros similares a los que posee una algoritmo de tipo árbol, ejemplos de estos son: la anchura del árbol, el grado de cada nodo, la altura del árbol y el grado de recursión.

2.5.2 Pipeline

Este paradigma está basado en un aproximación por descomposición funcional. La aplicación se subdivide en subproblemas o procesos y cada subproblema se debe completar para comenzar el siguiente proceso, uno tras otro. En un *pipeline* los procesos se denomina etapas y cada una de estas etapas resuelve una tarea en particular. También es importante destacar que en este paradigma se logra una ejecución concurrente si las diferentes etapas del *pipeline* están llenas o existe en ellas un flujo de datos. Esto no es sencillo de lograr, debido a que la salida de una etapa es la entrada de la siguiente. Uno de los parámetros importantes aquí es el número de etapas, por lo cual la eficiencia de ejecución de éste paradigma es dependiente del balanceo de carga a lo largo de las etapas del *pipeline*.

2.5.3 Master-Worker

El paradigma *master-worker* también es conocido como *task-farming* o *master-slave*. Este paradigma se compone de dos tipos de entidades: un *master* y varios *workers*. El *master* es

el encargado de descomponer el problema en trabajos más pequeños (o se encarga en dividir los datos de entrada en subconjuntos) y distribuir las tareas de la aplicación a los diferentes *workers*. Posterior a esto se realiza una recolección de los resultados parciales para procesar el resultado final. Los *workers* sólo cumplen la función de recibir la información, procesarla y enviar los resultados al *master*.

2.5.4 SPMD

En el paradigma Single Program Multiple Data (SPMD), para un determinado número de unidades de cómputo se ejecutan sobre estas un mismo código, pero sobre un subconjunto distinto de los datos de entrada. Es decir, partir los datos de la aplicación entre los nodos (o máquinas) disponibles en el sistema. El paradigma SPMD describe una comportamiento iterativo en el cual antes de finalizar una iteración se realiza una fase de sincronización, en esta fase el intercambio de información se hace entre los nodos vecinos. La información de intercambio dependerá de la naturaleza del problema. En la literatura también se conoce paralelismo geométrico, descomposición por dominios o paralelismo de datos.

En el capítulo 3 se muestra un enfoque más detallado de este paradigma, exponiendo su funcionamiento y los aspectos que afectan su rendimiento.

Capítulo 3

Single Program Multiple Data

3.1 Introducción

La mayor parte de la aplicaciones paralelas están diseñas bajo el paradigma SPMD, esto debido a sus propiedades de escalabilidad. Con la finalidad de aprovechar al máximo las ventajas de este paradigma y tratar de minimizar las ineficiencias, debemos complementar el conocimiento del problema con el conocimiento del paradigma SPMD.

Cuando se quiere lograr un buen rendimiento en una aplicación SPMD, debemos considerar muchos aspectos, como los siguientes: las comunicaciones, el tipo de problema, la dependencia de datos, la estructura interna de almacenamiento, etc. También debemos estar muy atentos en mantener una relación aceptable entre el cómputo y la comunicación, para no tener un exceso de comunicaciones generado por un exceso de recursos.

Siguiendo la línea de obtener un "Modelo de rendimiento", se hace necesario conocer las propiedades, limitaciones y factores que determinarán el rendimiento en nuestra aplicación. A continuación se presentarán las características más relevantes del paradigma SPMD.

3.2 SPMD

En el paradigma SPMD cada proceso ejecuta básicamente el mismo código, pero cada proceso computa una sección diferente de los datos de entrada a la aplicación [69]. Por lo cual, se debe hacer una división de los datos de entrada entre los procesos disponibles. Este paradigma también se conoce como paralelismo geométrico, descomposición de dominio y paralelismo de datos. En la figura 3.1 presenta una representación esquemática de este paradigma.

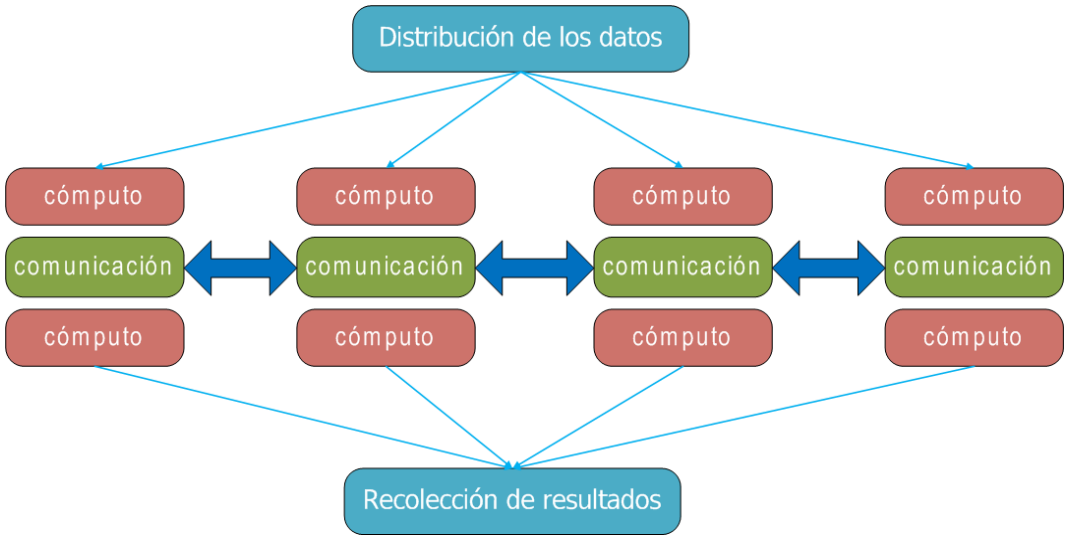


Figura 3.1: Estructura básica de un programa SPMD.

La estructura básica es geométrica regular, con interacción limitada espacialmente. Esta estructura permite que los datos se distribuyan uniformemente entre los procesos (ver figura 3.2a), donde cada uno será responsable de un área definida del total de los datos de entrada.

En las aplicaciones SPMD generalmente la comunicación se realiza entre procesos vecinos como se aprecia en la figura 3.2b. El tamaño de los datos a comunicar será proporcional al tamaño de los límites de la sección a computar, mientras que la carga de cómputo será proporcional al volumen del problema.

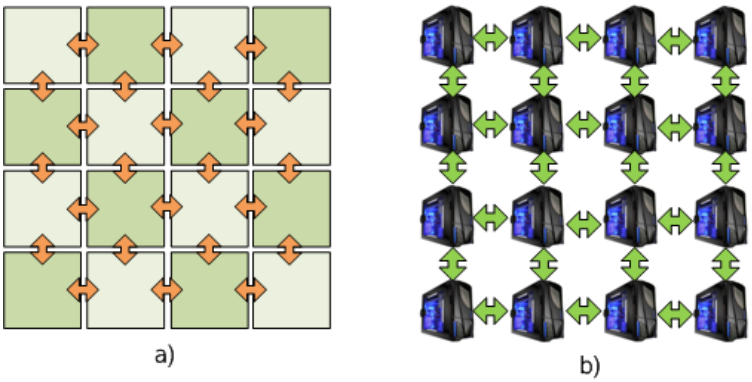


Figura 3.2: Distribución de datos homogénea y comunicación entre vecinos en SPMD.

Dependiendo de la aplicación puede ser necesario realizar periódicamente sincronizaciones globales entre todos los procesos. Comúnmente las aplicaciones SPMD poseen una fase de inicialización en donde se pueden leer de disco los datos de entrada, hacer una repartición inicial de los datos a cada proceso o bien autogenerar los datos de entrada.

Las aplicaciones SPMD suele ser muy eficientes si el entorno es homogéneo y si los datos están bien distribuidos sobre los procesos. Si la carga de trabajo o la capacidad de cómputo entre los diferentes procesos es heterogénea, entonces el paradigma requiere el apoyo de alguna estrategia de *load balancing* (balanceo de carga) que sea capaz de adaptar la distribución de los datos durante la ejecución de la aplicación.

Este paradigma también es muy sensible a la caída de algún proceso. Por lo general, la caída de un solo proceso es suficiente para causar un bloqueo a nivel general en la resolución del problema (local o global, esto dependerá de la aplicación), por lo que ningun de los proceso podrá avanzar más allá de un punto de sincronización global.

El grado de dependencias de datos entre los diferentes procesos va a determinar la complejidad de las comunicaciones durante la ejecución. Cuando se usan cientos o miles de procesadores, se debe poner especial énfasis en la programación, ya que la aplicación debe estar muy bien estructurada para evitar problemas de *load balancing* y problemas de *deadlock* [15].

A menudo, las aplicaciones SPMD poseen fases de cómputo y de intercambio de información (comunicación). Estas fases dentro de la aplicación se repetirán hasta el final de la ejecución siendo separadas por puntos de sincronización lógica a las cuales llamaremos iteraciones. En la figura 3.3 podemos apreciar dos iteraciones, cada una compuesta por una fase de cómputo y una fase de intercambio.

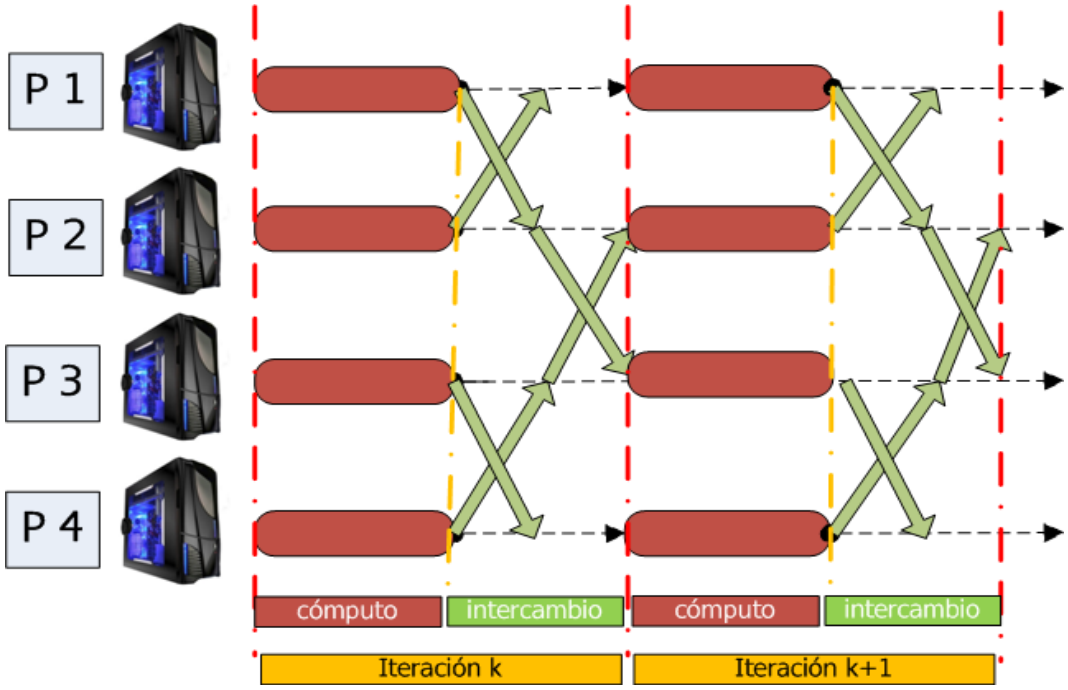


Figura 3.3: Fases de cómputo e intercambio dentro de una iteración.

Debido al comportamiento iterativo, mientras más sincrónica sea la aplicación será más

fácil hacer estimaciones de tiempo de ejecución. Sin embargo, gran parte de aplicaciones SPMD no están diseñadas con un alto grado de sincronismo, generalmente la sincronización entre las fases sólo se aplica por las dependencias de datos y por paso de mensajes explícitos para satisfacer estas dependencias.

Para una aplicación con características de bajo sincronismo global no necesariamente todos los procesos ejecutan la misma fase al mismo tiempo. Si el desbalance de la carga de trabajo en cada fase difiere mucho al resto de los procesos, entonces tendremos una sobreestimación del tiempo de ejecución total, ya que se incrementarán los costos de cada fase.

La forma en que se distribuyen los datos repercute significativamente en el rendimiento de la aplicación, más aún si la aplicación tiene características irregulares, en donde la cantidad de carga de trabajo cambia durante la ejecución de la aplicación [70, 67, 11]. Por lo tanto se hace necesario tener políticas adecuadas de particionamiento de datos y de balance de carga.

3.3 Particionamiento de datos y balance de carga

Para aplicaciones en las cuales la carga de trabajo requerida por cada sección de datos es previsible y constante en el tiempo, el cálculo de una adecuada partición de datos al inicio de la ejecución (*static load balancing*) es suficiente para mejorar las prestaciones en una aplicación SPMD.

De lo contrario, se deberá proveer de algún tipo de mecanismo de adaptación de la carga durante el tiempo de ejecución para evitar desequilibrios durante el procesamiento de datos y las fases de comunicación. No es sencillo decidir cuándo y cómo particionar, calcular la nueva

partición y hacer la redistribución de los datos para la nueva partición, por lo tanto lograr mejores prestaciones bajo estos aspectos es una tarea muy compleja.

Podemos mencionar tres aspectos importantes que necesitamos tener en cuenta al momento de seleccionar una estrategia de particionamiento de datos [60]:

***Overhead* por desequilibrio de carga** : hay procesos se mantienen sin trabajar mientras esperan a que los procesos vecinos terminen su tarea de cómputo.

***Overhead* por comunicación** : cada mensaje requiere de un envío y recepción entre los procesos participantes, también requiere de un tiempo de inicio o de establecer la conexión más un cierto tiempo proporcional a la longitud de los datos a enviar. Además, cada mensaje está sujeto una latencia que puede agravar el desequilibrio de la carga.

Complejidad de la implementación : Los algoritmos y estrategias utilizadas pueden variar dependiendo de la cantidad necesaria de esfuerzo del programador. En general, cuanto más complejo sea el algoritmo de particionamiento, mejor será el resultado de equilibrio de carga y la eficiencia de la comunicación.

Existen varios métodos para el particionamiento global de datos entre procesos. Una elección adecuada dependerá de la naturaleza de los datos y de los cálculos que se realizan [60]. En general, las estrategias más simples se aplican a tipos de problemas bien definidos, mientras que las estrategias más complejas se pueden aplicar a una gama más amplia de problemas.

Para seleccionar la mejor estrategia de *load balancing*, debemos conocer bien nuestra aplicación con la finalidad de ver a qué modelo de particionamiento se adapta, y también decidir si el grado de complejidad de la aplicación y del particionamiento permite un tipo de *load balancing* estático o dinámico.

3.3.1 Particionamiento regular

Este tipo de particiones se aplica particularmente a datos con geometría simple por ejemplo matrices multidimensionales, las cuales pueden ser particionadas bajo en un patrón regular y repetitivo. Podemos dividirlas por fila, por columna, o por bloques, como se puede apreciar en las figura 3.4.

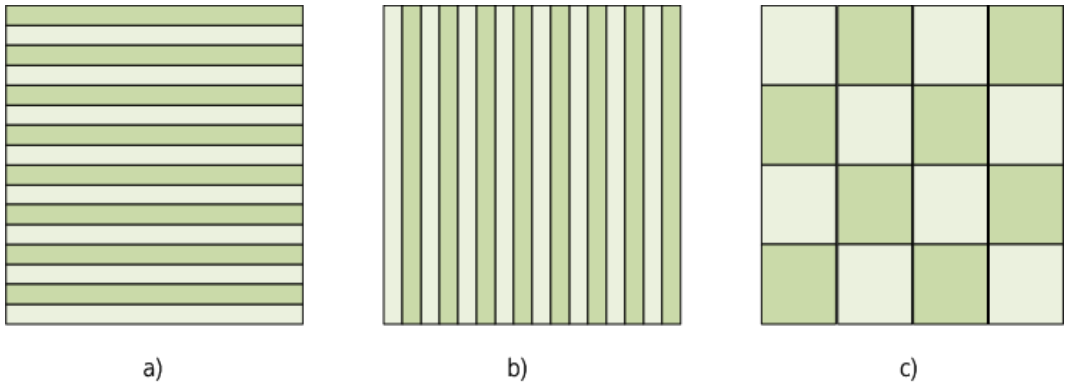


Figura 3.4: Particiones regulares, a) por filas, b) por columnas y c) por bloques.

De estos tres enfoques, la partición por bloques a menudo se prefiere sólo cuando se necesitan los datos cercanos para iniciar cada una de las fases de cómputo. Para hacer estos cálculos generalmente se requiere de una comunicación entre los procesos fronteras respecto a la partición. En la figura 3.4 se aprecia que para la partición a y b cada partición tiene entre 1 y 2 fronteras, mientras que en c tenemos de 3 a 4 fronteras por partición.

En las particiones de las figuras 3.4a y 3.4b, cada región tiene sólo dos vecinos. Sin embargo, cada una de las fronteras tiene una gran superficie, por lo que este tipo de partición tendría un *overhead* agregado en las comunicaciones. Cuanto mayor sea la superficie de una partición mayor es el tamaño de la frontera y por consiguiente mayor es la probabilidad de realizar intercambios con los procesos vecinos.

3.3.2 Particionamiento irregular

Las particiones de este tipo se utilizan cuando haciendo una partición normal (regular) causaría un desequilibrio muy grande a nivel de carga de trabajo. Esto puede depender del tipo de problema, de los datos de entrada, o bien de las características heterogéneas de las unidades de cómputo y comunicación.

Por ejemplo, en una aplicación de simulación de flujos de aire sobre un prototipo de aeroplano, el espacio se puede dividir como una colección de planos horizontales de diferentes distancias verticales entre sí. Los datos y cálculos para cada uno de estas secciones resultantes se asignan a un proceso distinto. Las secciones finas se utilizarían en secciones sobre el aeroplano y las secciones más gruesas donde hay poco movimiento de aire como en la figura 3.5*a*. En la partición resultante se aprecia la gran superficie de las fronteras, por lo que este tipo de partición tendría un *overhead* por el tamaño de las comunicaciones.

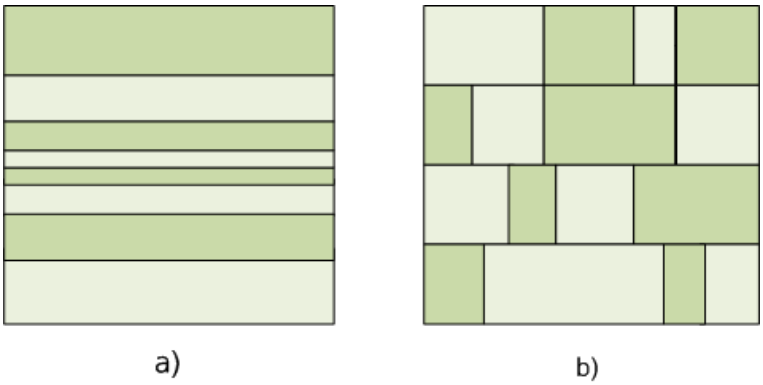


Figura 3.5: Particiones irregulares.

Para hacer los cálculos de cuál es el tamaño de partición adecuado, requiere de un nivel de experticia mayor. Primero necesitamos estimar la densidad de la carga de trabajo en cada proceso, esto no se puede medir en relación al tamaño del área a computar, sino por el carga computacional de dicha área.

Como mencionamos anteriormente, cuanto más complejo es el algoritmo de particionamiento, mejor será el resultado de equilibrio de la carga y de la eficiencia en la comunicación. Lograr esta eficiencia puede estar limitada por la complejidad de los patrones de comunicación y los intercambios de información necesarios.

También se pueden particionar los datos en secciones horizontales y verticales con áreas diferentes, con lo cual se podría llegar a un equilibrio de la carga computacional y minimizar los costos de comunicación. En la figura 3.5*b* cada sección tiene un vecino derecho y un vecino izquierdo y un número variable de los vecinos arriba y abajo. En este caso cada proceso debe enviar una menor cantidad de datos. Sin embrago, no siempre es posible hace esto, algunas veces existen restricciones inherentes a la aplicación, como se muestra en la aplicación expuesta en el capítulo 4.

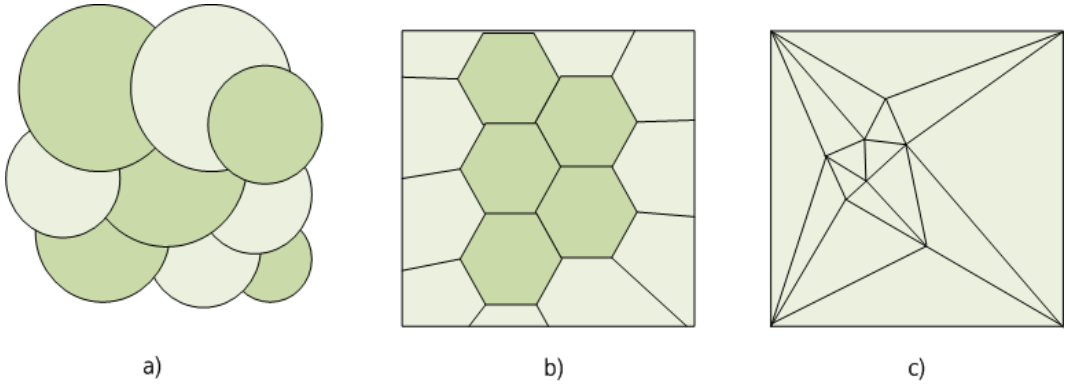


Figura 3.6: Otros tipos de particionamiento.a)*covering radius*, b)*Voronio diagram* y c)*Delaney traingulation.*

Existen otros métodos de partición como vemos en las figura 3.6, por ejemplo, para las aplicaciones basada en espacios métricos se puede hacer una división en función de un *covering radius* (radio covertor), en diagramas de *Voronio*, o según triangulación de Delaney [63, 33]. Bajo una triangulación de Delaney cada sección tiene 3 secciones vecinas como máximo. Mientras que en las otras la cantidad de vecinos puede depender de la cantidad de centros y del radio cobertor en función de la dispersión de los elementos en el espacio.

3.3.3 Carga de trabajo aleatoria

Hay un grupo de aplicaciones en las cuales los datos van cambiando de lugar durante la ejecución o bien al iniciar la aplicación la posición de la carga de trabajo es impredecible. En estos casos, normalmente cada elemento guarda la información sobre su posición respecto de los datos globales en alguna tipo de estructura especial para esto (p.e. estructuras de indexación) o bien para optimizar su funcionamiento se utilizan subrutinas de librerías especiales.

Un ejemplo son los registros de una base de datos, ordenados por un campo en particular. Aquí el valor de cada registro guarda también la ubicación respecto de los datos globales.

Uno de los principales problemas en este tipo de aplicaciones son las complejas estructuras que se deben mantener para optimizar los accesos a datos. Por lo general en una estructura de indexación mientras más información es cargada para localizar los datos, esta se vuelve menos eficiciente.

Del mismo modo mientras más información se almacene la estructura también crecerá, por lo cual las búsquedas se verán mermadas. También el actualizar estas estructuras en tiempo de ejecución supone un gran problem a tratar.

3.3.4 Balance de carga dinámico

Uno de los temas críticos al momento de implementar eficientemente una aplicación es seleccionar una adecuada estrategia de balance de carga dinámico. Un balance de carga dinámico es necesario, por ejemplo, cuando se desconoce el costo computacional de cada sección de cómputo, por creación dinámica de tareas, por migración de tareas o variaciones en los recursos hardware por cargas externas[62].

Como mencionamos en el principio de este apartado 3.3 necesitamos decidir cuándo y cómo: particionar, calcular la nueva partición y hacer la redistribución de los datos para la nueva partición. Además de conocer qué tipo de elementos vamos a mover. Existe varias taxonomías que clasifican estrategias de *load balancing*, pero cada clasificación tiene como enfoque problemas concretos.

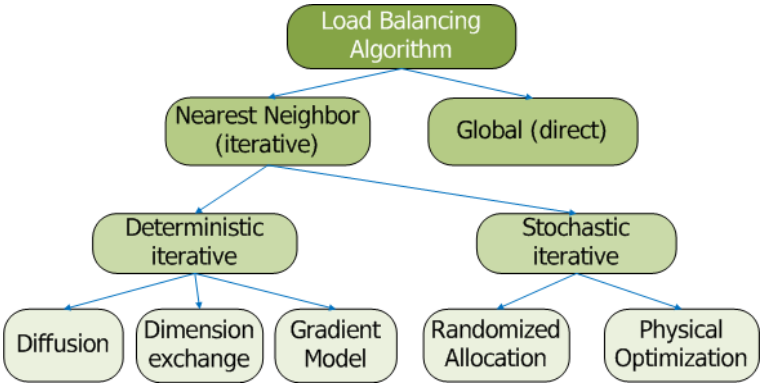


Figura 3.7: Clasificación de estrategias de balance de carga dinámico propuesta en [22].

Por ejemplo, en [22] se proponen una clasificación de métodos basados en consecutivas mejoras locales con la finalidad de conseguir una mejora global (ver figura 3.7).

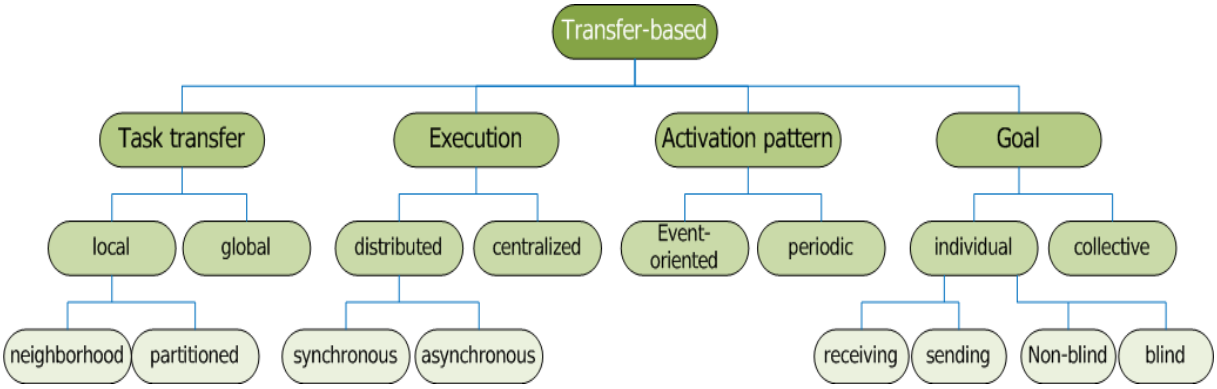


Figura 3.8: Clasificación de estrategias *transfer-based* propuesta en [64].*task: corresponde a una sección de datos o unidad de trabajo en SPMD*

En [64] se propone una serie de estrategias para distribución de carga en aplicaciones SPMD. En este caso se presenta en forma amplia diferentes criterios de clasificación. En la figura 3.8 se muestra sólo la clasificación propuesta en [64] para algoritmos basados en transferencia de carga cuando se detectan *overloads* o *underloads*.

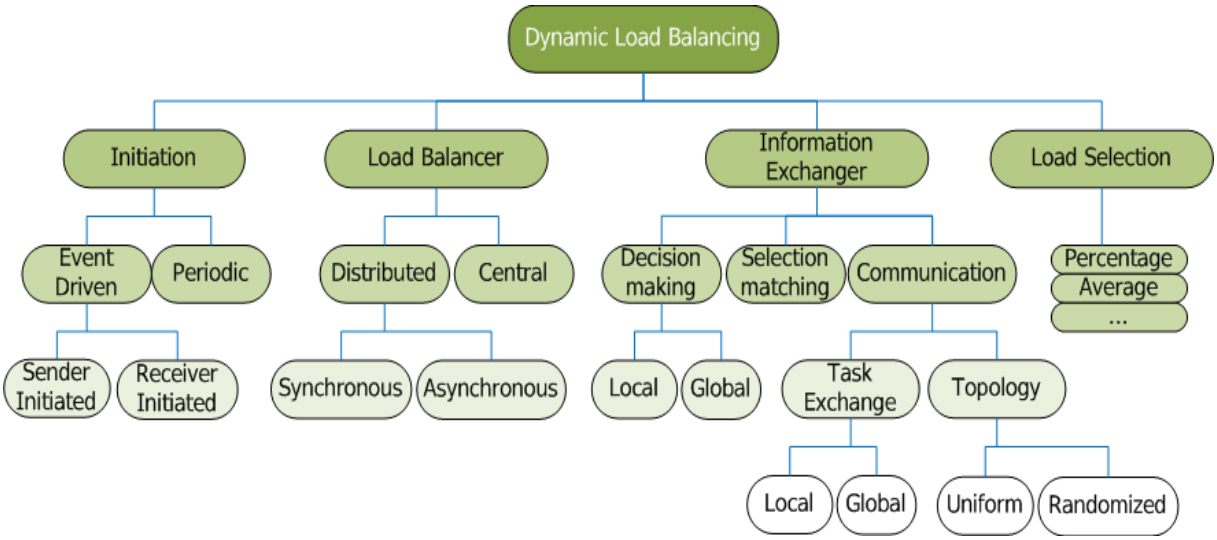


Figura 3.9: Taxonomía de balance de carga dinámico según [62].

En la taxonomia propuesta en [62] se presentan una clasificación basada en cuatro subestrategias que representan el proceso de balance de carga dinámico. En la figura 3.9 se pueden ver las subestrategias de *initiation*,*location*,*exchange* y *selection*.

En el balance de carga dinámico se pueden utilizar los mismos algoritmos que están disponibles para el cálculo de una partición estática. Sólo se debe proporcionar el algoritmo con información sobre la distribución actual de la carga de trabajo. Una vez que la nueva distribución se ha calculado, los datos debe ser redistribuido entre los diferentes procesos.

La elección del algoritmo de balance de carga depende en gran medida de la naturaleza de los datos de entrada.

Capítulo 4

Aplicación Gradiente Conjugado

4.1 Introducción

El gradiente conjugado (CG) es quizás el algoritmo más conocido para resolver sistemas lineales dispersos simétricos y definidos positivos [61], inicialmente propuesto por Magnus Hestenes y Eduard Stiefe en 1952. Aunque el interés actual arranca a partir de 1971, cuando se plantea como un método iterativo.

Es ampliamente utilizado para resolver sistemas de n ecuaciones lineales del tipo $Ax = k$ con n incógnitas, cuando A es positiva definida y simétrica. Una matriz es definida positiva si y solo si todos los menores principales son estrictamente positivos, es decir, si son positivos los determinantes de las submatrices cuadradas que se forman desde el vértice superior izquierdo agregando un elemento de la diagonal en forma sucesiva. Sin embargo, existen aproximaciones para su utilización en el caso en que la matriz A no sea simétrica.

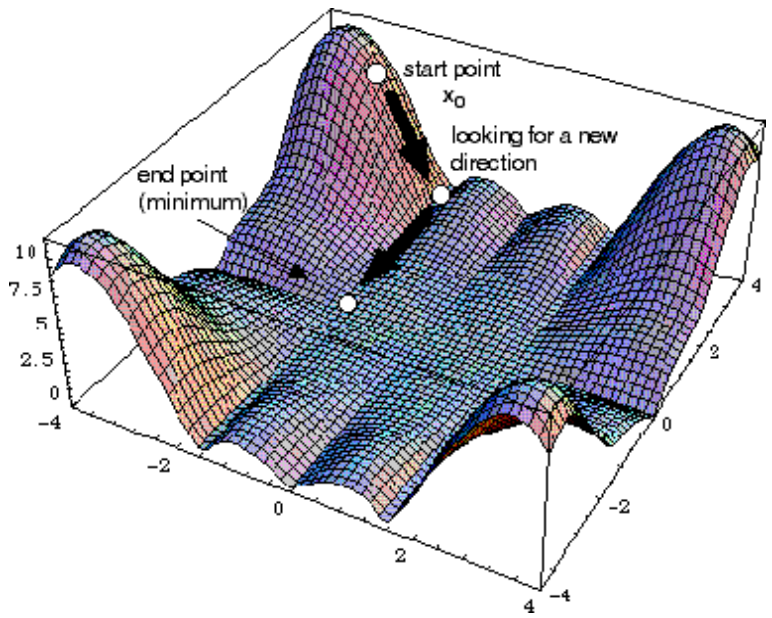


Figura 4.1: Cálculo de direcciones por el método gradiente conjugado.

La idea básica en que descansa el método del gradiente conjugado consiste en construir una base de vectores ortogonales y utilizarla para realizar la búsqueda de la solución en forma más eficiente, esto se repite calculando nuevas direcciones hasta encontrar el punto de convergencia (ver figura 4.1).

Este método garantiza que se encontrará la solución en n pasos [43], siempre y cuando no hayan errores de redondeo(aritmética exacta). También necesita poca memoria para ser realizado, ya que en cada iteración del algoritmo básico es necesario realizar un producto matriz-vector, tres actualizaciones vectoriales y dos productos escalares entre vectores.

Si bien existen varias opciones de aplicaciones SPMD para estudiar problemas de rendimiento como las analizadas en [14], para este estudio hemos decidido utilizar el *benchmark Conjugated Gradient* (CG) perteneciente a la *suite NAS Parallel Benchmark* (NPB) desarro-

llado por la *NASA Advanced Supercomputing* (NAS) [6].

Esta aplicación es ampliamente utilizada por la comunidad científica concerniente al HPC, además posee un carácter irregular en cuanto a su patrón de comunicación y a su implementación. Por lo que las acciones para mejorar rendimiento pueden no ser triviales.

Un *benchmark* es un software de referencia específico a un tipo de aplicación, comúnmente se utilizan con el objetivo de analizar el rendimiento de un sistema de destino y para hacer comparaciones con otros sistemas. Pueden ser una aplicación real, o bien una aplicación que sintetiza algunas características de una aplicación real. Dependiendo de la aplicación, el *benchmark* pueden enfocarse en los recursos individuales como el acceso de memoria, unidades aritméticas, unidades lógicas, entrada/salida o puede enfocarse en sobrecargar el sistema en su conjunto.

Un característica importante del *benchmark* CG es que soporta grandes problemas para matrices *sparse* cuadradas. Este tipo de problema es aplicado en diversas áreas, tales como: modelado de flujos de fluidos, simulación de flujo de aire, ingeniería de yacimientos, ingeniería mecánica, análisis de dispositivos semiconductores, modelos de reacción nuclear y simulación de circuitos eléctricos. Por lo que si deseamos estudiar una aplicación real, este *benchmark* se ajusta nuestras necesidades.

4.2 *Benchmark* CG

El *benchmark* CG resuelve un sistema lineal disperso desestructurado por el método antes mencionado del gradiente conjugado. Este *benchmark* utiliza el método de la potencia inversa para encontrar una estimación del mayor autovalor (valor propio) de una matriz *sparse* de dos dimensiones, simétrica, positiva y definida, con un patrón aleatorio de elementos *nonzeros* [7].

Este *benchmark* está implementado en el lenguaje de programación Fortran [5] y con el estándar de paso de mensajes MPI [39]. Al iniciar la aplicación no hay un repartición general de la matriz global de entrada, en cambio, cada proceso genera la sección de la matriz global que le corresponde. Y el número de procesos asignados debe ser potencia de dos, sinó será imposible la compilación del *benchmark*.

Cada proceso calcula cuál es la sección de la matriz global a crear, la sección asignada dependerá del identificador del proceso, del tamaño global de la matriz y de la cantidad total de procesos asignados a la ejecución. Estas secciones de la matriz global se almacenan en matrices del tipo *sparse* y entre los diferentes procesos son muy parecidas en cuanto a la cantidad de elementos *nonzero*.

El tamaño de problema es variable de acuerdo a una clasificado según una serie de configuraciones específicas, en los *benchmark* NPB a estas configuraciones se les denominan "clases" de problemas.

Size	**n**	**niter**	**nonzer**	λ
S	1.400	15	5	10
W	7.000	15	8	12
A	14.000	15	11	20
B	75.000	75	13	60
C	150.000	75	15	110
D	1.500.000	100	21	500
E	9.000.000	100	26	1.500

Cuadro 4.1: Parámetros para las diferentes clases de problemas en CG.

En CG los tipos de problemas están agrupados en siete tamaños de problema diferentes como se detalla en la tabla 4.1. En donde n define el tamaño del sistema, *niter* el número de iteraciones externas de la implementación y λ corresponde a una constante utilizada en el algoritmo para definir el desplazamiento a través de la diagonal principal [7].

Con la finalidad de comprender el funcionamiento de este tipo de aplicaciones SPMD y posibles problemas de rendimiento, a continuación detallaremos la forma en que se particionan los datos, la estructura de almacenamiento y el patrón de comunicación.

4.2.1 Particionamiento de datos

El particionamiento de datos se basa simplemente en dividir la matriz global en bloques de igual cantidad de filas e igual cantidad de columna para cada proceso (figura 4.2).

Figura 4.2: Particionamiento de bloques cuadrados en CG para 16 procesos.

Los bloques generados no siempre serán cuadrados, como se puede ver en las figura 4.3 esto dependerá de la cantidad de procesos asignados a la resolución del problema.

Figura 4.3: Particionamiento de bloques cuadrados en CG para 8 procesos.

Debido a este tipo de particionamiento el tamaño de los bloques asignados será proporcional al tamaño de la matriz global. Y gracias a que los datos son almacenados en matrices *sparse* este *benchmark* puede procesar grandes volúmenes de información sin acceder a memoria secundaria.

4.2.2 Matrices *sparse* en CG

El tamaño de los problemas matemáticos que pueden ser abordados en un momento determinado generalmente está limitado por los recursos informáticos disponibles. Ejemplos de estos problemas pueden se una limitación por la velocidad de los procesadores o por la cantidad de memoria disponible.

Hay muchas clases de problemas matemáticos que dan lugar a matrices, donde una gran cantidad de los elementos son cero. En este caso, tiene sentido tener un tipo de matriz especial para manejar esta clase de problemas, donde sólo se almacenan los elementos *nonzero* de la matriz. Esto no sólo reduce la cantidad de memoria para almacenar la matriz, sino que también se puede aprovechar el conocimiento a priori de las posiciones de los elementos *nonzero* para focalizar y acelerar los cálculos.

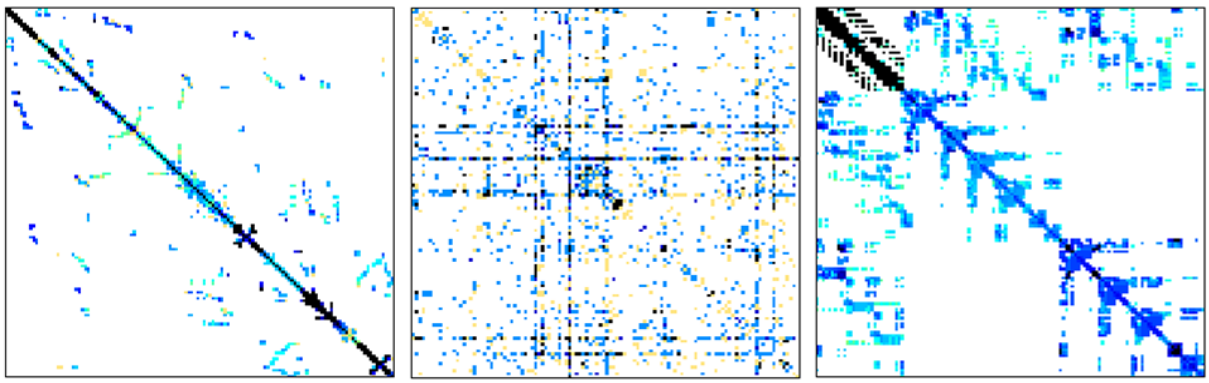


Figura 4.4: Ejemplos de matrices *sparse* (*University of Florida sparse matrix collection* [26]).

Una matriz que almacena sólo los elementos *nonzero* se llama generalmente *sparse*. En la figura 4.4 podemos ver diferentes tipos de matrices en las cuales la mayor cantidad de elementos son cero. Estas matrices pertenecen a la colección de matrices *sparse* de la *University of*

Florida [26], correspondientes a problemas de *Automobile shock absorber assembly*, *Metabolic network* y *Threaded connector/contact* respectivamente.

Existen muchos formatos de como almacenar una matriz *sparse*, tales como : DNS *Dense format*, BND *Linpack Banded format*, CSR *Compressed Sparse Row format*, CSC *Compressed Sparse Column format*, COO *Coordinate format*, ELL *Ellpack-Itpack generalized diagonal format*, DIA *Diagonal format*, BSR *Block Sparse Row format*, MSR *Modi ed Compressed Sparse Row format*, SSK *Symmetric Skyline format*, NSK *Nonsymmetric Skyline format*, LNK *Linked list storage format*, JAD *The Jagged Diagonal format*, SSS *The Symmetric Sparse Skyline format*, USS *The Unsymmetric Sparse Skyline format*, VBR *Variable Block Row format* [68].

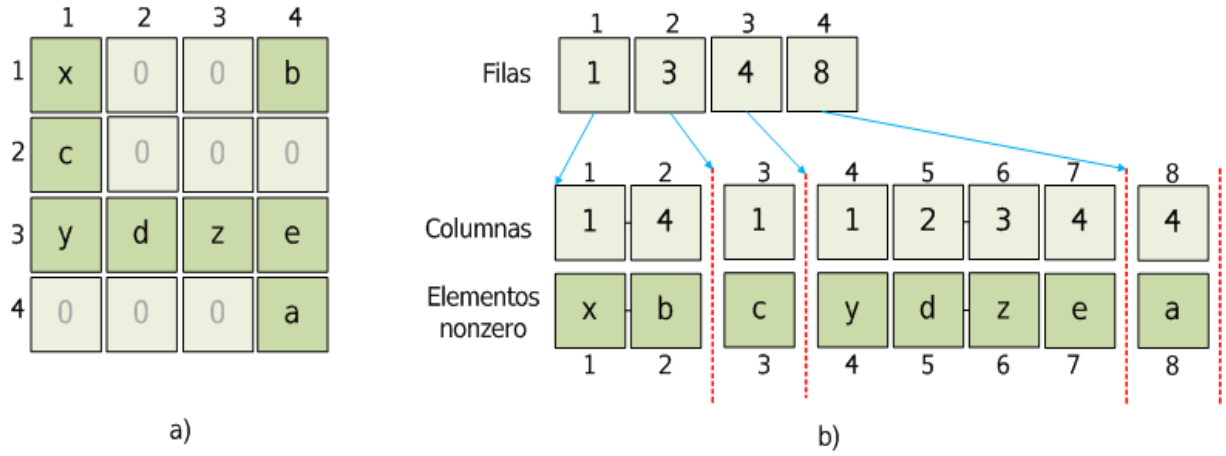


Figura 4.5: Almacenamiento de una matriz *sparse* CSR.

En el caso de *benchmark* CG cada proceso genera su propia sección de la matriz global en el formato CSR (*Compressed Sparse Row format*). En la figura 4.5 se almacenada una matriz (*a)*) en formato CSR, para lo cual se utilizan 3 vectores (*b)*). El primer vector contiene la posición en la cual comienzan los elementos *nonzero* de una determinada fila respecto al

vector de elementos *nonzero*. El segundo vector almacena la columna a la que pertenece cada elemento *nonzero*. Y el tercer vector almacena los elementos *nonzero*.

4.2.3 Comunicaciones

El patrón de comunicación dentro de cada iteración se compone una serie de intercambios de vectores fila, vectores columna y escalares. Estos intercambios se realizan en las siguientes etapas: comunicación con los procesos vecinos horizontales, comunicación con los procesos transpuestos a la diagonal y dos intercambios con vecinos horizontales.

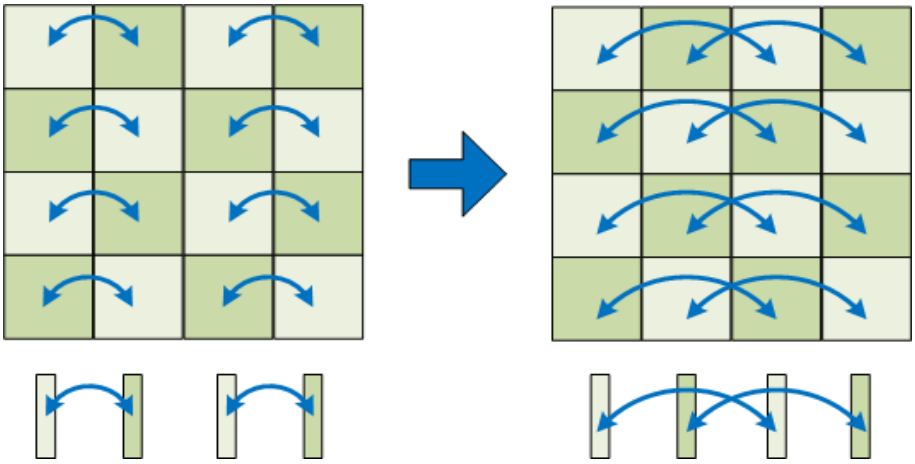


Figura 4.6: Intercambio horizontal de vectores.

En la primer intercambio que realiza el CG dentro de una iteración, cada proceso envía y recibe un vector columna. Esto se realiza entre los vecinos alineados horizontalmente como se ve en la figura 4.6.

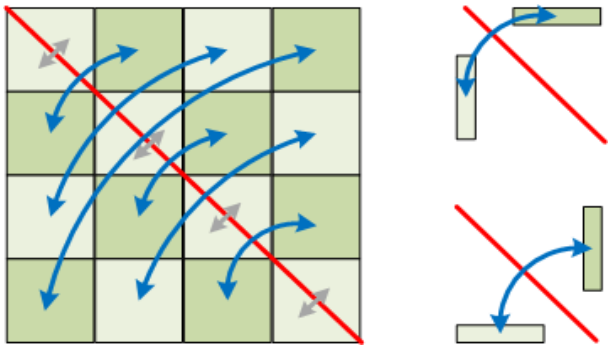


Figura 4.7: Intercambio de vectores transpuesto a la diagonal.

El segundo intercambio se realiza entre procesos transpuestos a la diagonal principal como se muestra en la figura 4.7. A diferencia del primer intercambio aquí cada proceso envía un vector columna y el receptor espera un vector fila. En este intercambio los procesos ubicados sobre la diagonal se envían a sí mismos los datos necesarios para continuar con el cómputo, esto siempre y cuando los bloques sean cuadrados.

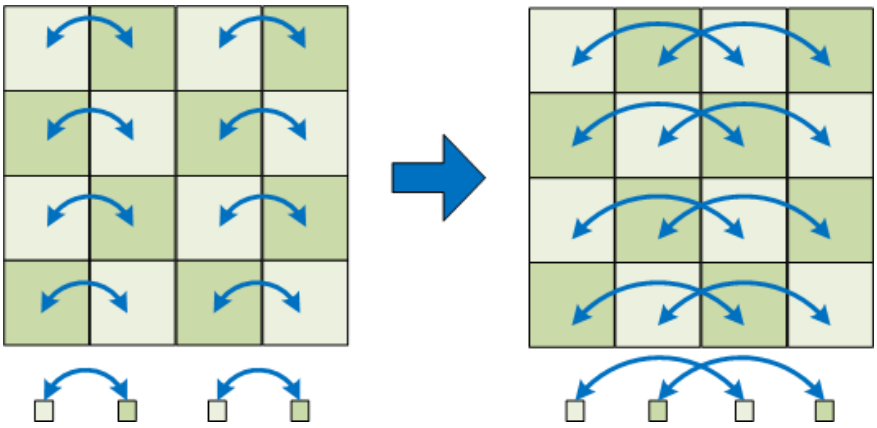


Figura 4.8: Intercambio horizontal de escalares.

El tercer y cuarto son similares al primero, envíos a los vecinos horizontales, pero con la diferencia que sólo se envía un elemento (figura 4.8).

Ahora bien, si quisiéramos representar gráficamente las comunicaciones de las tres figuras anteriores, sería algo similar a la figura 4.9.

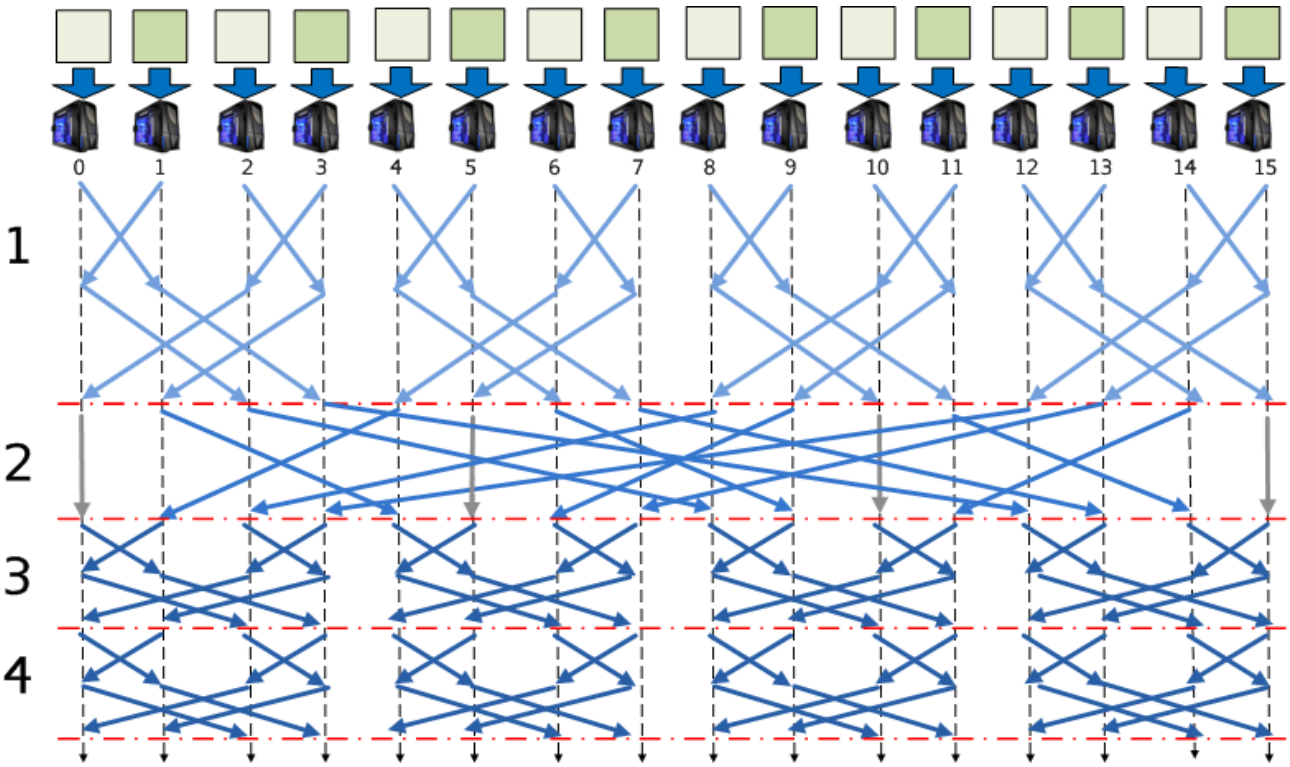


Figura 4.9: Fase de comunicación para 16 procesos.

4.3 Definición del Problema de Investigación

Un aspecto importante para obtener un buen rendimiento en una aplicación SPMD es conseguir una alineación de las fases entre los diferentes procesos. Es decir, lograr que todos procesos finalicen su fase de cómputo en un tiempo similar para realizar una sincronización sin tiempos muertos.

Esto se puede conseguir teniendo la misma cantidad de carga de trabajo entre las unidades de cómputo disponible. Si bien, en el *benchmark* CG cada proceso genera bloques relativamente homogéneos en cuanto a la carga de trabajo, en una aplicación real las matrices *sparse* no necesariamente son homogéneas, sino que es bastante probable que no lo sean como podemos ver en los ejemplos de la figura 4.4 pertenecientes a la "*University of Florida sparse matrix collection*" [26].

Por lo tanto, nos centraremos en estudiar las prestaciones en el *benchmark* CG con matrices heterogéneas, debido a que este *benchmark* representa a un grupo de problemas reales que funcionan de forma similar. Es más, si quisiéramos conocer cómo se comportaría una aplicación real con problemas de desbanlace de carga, podemos obtener una referencia de su comportamiento desbalanceando el *benchmark* CG.

Así mismo, si quisiéramos estudiar cómo hacer una sintonización o un rebalanceo de la cargar de trabajo en una aplicación real, obtendríamos u importante acercamiento definiendo políticas de balance de cargar en este *benchmark*.

4.3.1 Factor de rendimiento analizado

En esta aplicación la carga de trabajo está definida por la cantidad de elementos *nonzero* dentro del bloque. Hacer una repartición geométrica en bloques de igual tamaño en una matriz *sparse* no necesariamente es la mejor opción, ya que la cantidad de elementos dentro de cada bloque no será homogéneo como se puede ver en la figura 4.10.

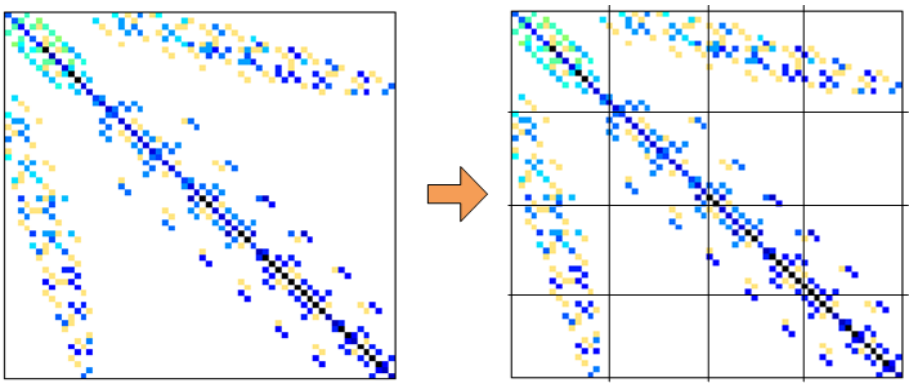


Figura 4.10: Repartición de una matriz *sparse* en bloques de tamaño homogéneo.

Una distribución de este tipo generaría un grave desbalance en la aplicación. En la figura 4.10 podemos apreciar que existe una heterogeneidad en cuanto a los elementos en cada bloque a procesar, algunos procesos tendrán muy pocos, incluso algunos no tendrán qué procesar, mientras que otros tendrán muchos elementos.

De igual manera se generaría una falta de sincronización entre las diferentes fases de la aplicación. Si observamos la figura 4.11 los bloques de las esquinas están más cargados y los centrales tienen un problema de falta de carga de trabajo, si estos bloques repartidos en 16 procesos provocarán que quién haya recibido los mas cargados irán retrasando al resto. Debido a la dependencia de datos, en la misma figura también se exponen los tiempos muertos o tiempo de espera de los procesos con una carga inferior al más cargado.

Figura 4.11: Traza resultante del procesamiento de una matriz desbalanceada.

Para obtener una similitud en los tiempos de cómputo se necesita de una repartición homogénea de la cantidad de elementos *nonzero* entre los diferentes procesos de la aplicación y no una repartición en bloques de tamaños homogéneos. Esto con la finalidad de lograr sincronismo en las fases de cómputo y comunicación.

4.3.2 Balance de carga en CG

Si en este *benchmark* tuviéramos una matriz heterogénea y quisiéramos redistribuir la carga, necesitamos buscar una forma de particionar la matriz de manera tal que exista una homogeneidad de elementos *nonzero* en todos los bloques.

La manera que parece más sencilla de mover carga en un matriz consiste en hacer un particionamiento con bloques diferentes. Es decir, asignar diferentes cantidades de filas y columnas entre los diferentes procesos.

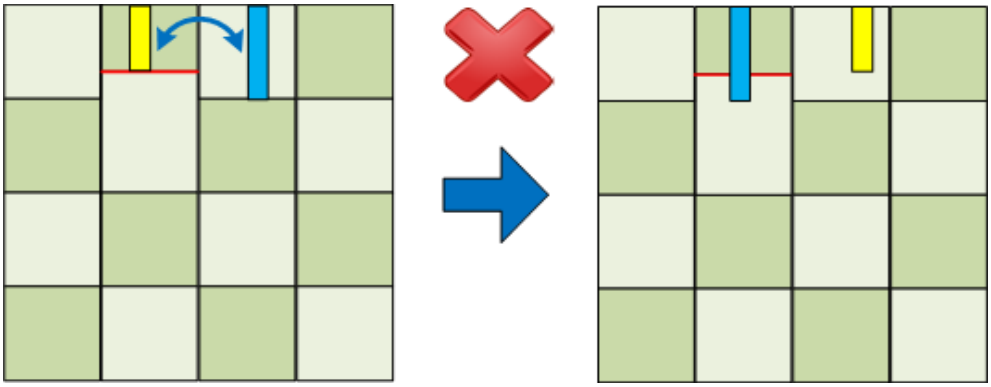


Figura 4.12: Problema con las comunicaciones horizontales al sólo mover filas entre dos bloques.

Como vemos en la figura 4.12, esto no es sencillo, por ejemplo si se disminuye el tamaño de filas de un bloque, en las comunicaciones horizontales un proceso enviará un vector columna de dimensión menor al que espera el receptor y de igual forma recibirá un vector columna mayor al que espera.

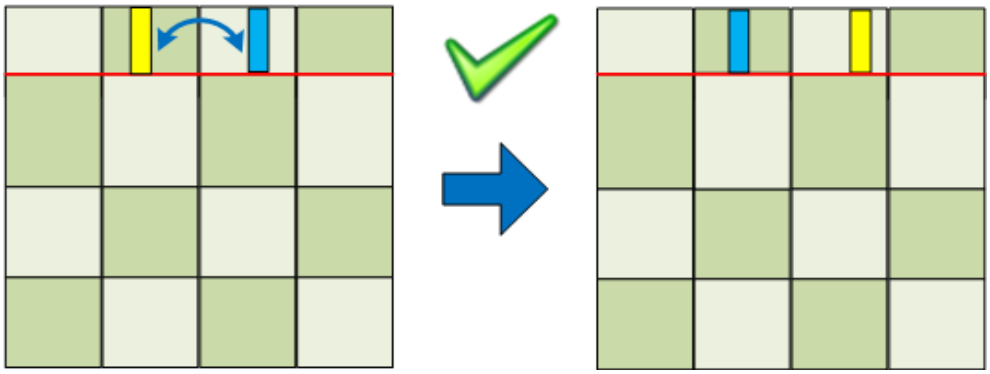


Figura 4.13: Comunicación horizontal al mover filas en los bloques alineados horizontalmente.

La forma correcta sería mover igual cantidad de filas en todos los bloques que estén alineados horizontalmente (ver figura 4.13).

También en las comunicaciones transpuestas a la diagonal existiría un problema similar al primero. En la figura 4.14 se expone que en las comunicaciones transpuestas un proceso enviará un vector columna de dimensión menor al vector fila que espera el receptor y de igual forma recibirá un vector fila mayor al que espera.

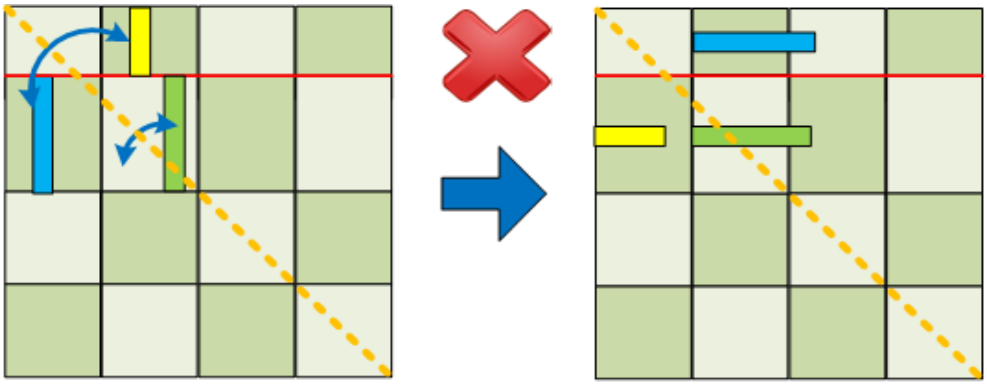


Figura 4.14: Problema con las comunicaciones transpuestas al sólo mover filas en todos los bloques alineados horizontalmente.

Para mantener el funcionamiento de la aplicación en las comunicaciones transpuestas necesitamos mover igual cantidad de filas en todos los bloques que estén alineados horizontalmente e igual cantidad de columnas en todos los bloques que estén alineados verticalmente. La cantidad de filas y columnas movidas debe ser idénticas a fin de mantener una partición simétrica a la diagonal (ver figura 4.15).

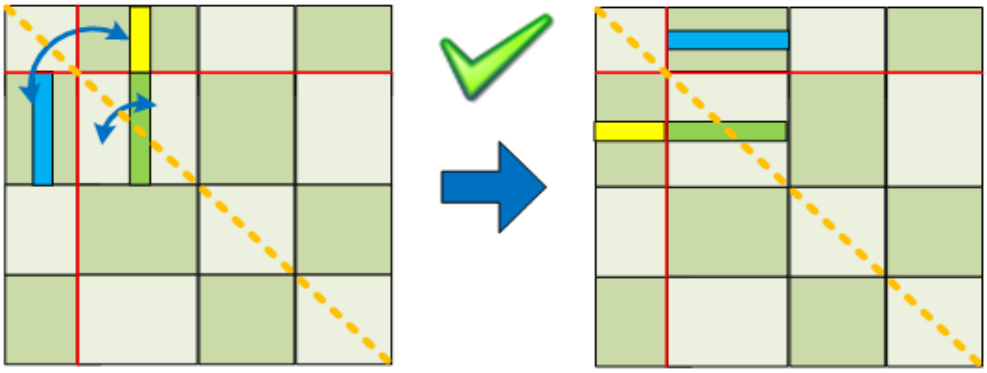


Figura 4.15: Comunicación transpuesta a la diagonal al mover filas y columnas de forma simétrica a la diagonal.

Haciendo una redistribución de bloques según esta política se podrían corregir problemas de desbalance de cargar en un SPMD similar al CG. Una redistribución de bloques podría conseguir una mejor distribución de los elementos *nonzero*, y por consiguiente se podría obtener un mejor rendimiento en la aplicación.

En resumen se puede mover carga en esta aplicación mediante el criterio de mover filas y columnas manteniendo un particionamiento simétrico a la diagonal principal como se muestra en la figura 4.16. Es decir, que al mover filas en un proceso, se deben mover igual cantidad de filas a los procesos vecinos horizontales y también mover esa misma cantidad de columnas a los procesos transpuestos.

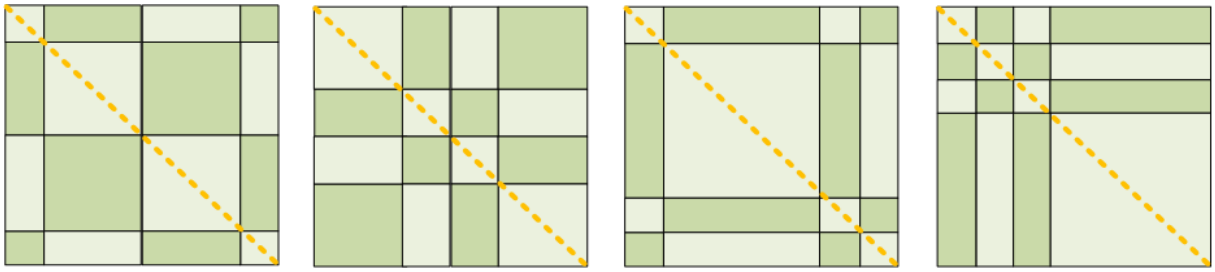


Figura 4.16: Algunos ejemplos de posibles distribuciones de carga.

La decisión de a quienes mover carga va a estar guiada por buscar que los procesos con más elementos *nonzero* cedan filas y columnas a los que poseen menos y siempre manteniendo una repartición simétrica a la diagonal como los ejemplos de la figura 4.16.

Capítulo 5

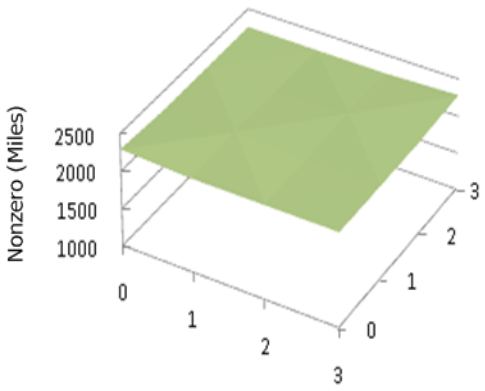
Experimentación y Resultados

Hasta el momento hemos presentado una serie de puntos a tener en consideración al momento de sintonizar una aplicación SPMD para tener una base de conocimiento apuntando hacia el objetivo a largo plazo planteado inicialmente.

Luego, hemos mencionado que la forma de particionar los datos tienen un alto grado de importancia en el rendimiento de una aplicación SPMD. Dependiendo del tipo de partición cambiarán los diferentes enfoques a analizar en una aplicación para mejorar su rendimiento, tales como: el volumen de las comunicaciones, la cantidad de procesos vecinos, la carga de cómputo de cada bloque, el grado de sincronismo y la relación cómputo/comunicación.

Por último, hemos analizado el *benchmark* CG, representativo de un tipo de aplicaciones SPMD. Esto con la finalidad de entender el comportamiento de una aplicación real. También se ha planteado el caso de su funcionamiento con una entrada de datos heterogénea en cuanto a la cantidad de elementos *nonzero* y también hemos propuesto una política de balance de carga con la finalidad de mejorar las prestaciones.

n/n	**0**	**1**	**2**	**3**
0	2.294	2.249	2.251	2.253
1	2.249	2.281	2.242	2.248
2	2.251	2.242	2.283	2.247
3	2.253	2.248	2.247	2.284



Cuadro 5.1: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), repartición original.

Sin embargo, para realizar la presente investigación, se han realizado una serie modificaciones a la sección de la inicialización de la aplicación, ya que por defecto se crean reparticiones homogenéas como se muestra en la figura 5.1. Se ha cambiado el mecanismo en el que cada proceso crea su sección de matriz a computar por la capacidad de leer de disco la sección de la matriz de datos que le corresponde. Esto con la finalidad de experimentar con matrices en donde la carga de trabajo no está distribuida homogéneamente más la posibilidad de hacer diferentes ejecuciones con la misma matriz.

La experimentación se ha dividido en dos escenarios (ver 5.2.1 y 5.2.2), y se presentan tres formas de particionamiento de datos en cada de uno de estos. La primera de ellas por bloques homogéneos y las siguientes por bloque heterogéneos utilizando la política de balance de carga planteada en el punto 4.3.2 para equilibrar la cantidad de elementos *nonzero* de cada bloque.

Al visualizar los resultados experimentales con las tres configuraciones de particionamiento de datos en cada escenario, se puede ver como la carga de trabajo de los procesos de la aplicación se van tornando cada vez más homogéneos. Los valores de tiempo presentados corresponden al promedio de tres ejecuciones de la aplicación. Observando la variación en el tiempo de ejecución se puede inferir que para una matriz heterogénea, una repartición de datos no homogénea de filas y columnas se puede reducir el tiempo de ejecución.

5.1 Entorno de experimentación

Los experimentos expuestos a continuación tienen como objetivo general comprobar la mejora del rendimiento de la aplicación CG, y realizar una sintonización estática empleando las solución planteada en 4.3.2.

Los experimentos han sido ejecutados en un *cluster* homogéneo y dedicado, compuesto por 32 nodos cuya configuración se muestra en la tabla 5.2. La configuración hardware ha sido determinante a la hora de plantear nuestros experimentos.

Cluster IBM	32 Nodos IBM x3550 Serie con doble procesador Intel(R) Xeon(R) Dual-Core
IBM x3550 Nodes	2 x Dual-Core Intel(R) Xeon(R) CPU 5160 @ 3.00GHz 4MB L2 (2x2) 12 GB Fully Buffered DIMM 667 MHz Hot-swap SAS Controller 160GB SATA Disk Integrated dual Gigabit Ethernet

Cuadro 5.2: Configuración *cluster* IBM y sus respectivos nodos.

Con la finalidad de entender mejor los procesos y sus diferentes fases de cómputo y de comunicación en la aplicación, para nuestro experimentos hemos utilizados sólo 1 proceso

por nodo. Si bien en el *cluster* IBM cada nodo posee 4 cores divididos en 2 procesadores y sabiendo que las comunicaciones *interchip* e *intercore* son más rápidas que hacerlas a través de la red *Ethernet*, al utilizar 1 proceso por nodo tenemos una mayor homogeneidad en los tiempo de comunicación para todos los procesos de la aplicación.

En cuanto al software utilizado, los experimentos fueron realizados con la librería OpenMpi en su versión 1.4.1, y la versión 3.3.1 de los *NAS Parallel Benchmark* (NPB).

5.2 Análisis de resultados

Los resultados obtenidos están orientados a cubrir los objetivos inicialmente planteados en la investigación. El tamaño de las matrices es de 150.000 filas y 150.000 columnas en ambos experimentos, la diferencia entre los experimentos está en la cantidad de elementos *nonzero* y la distribución de estos. También para ambos experimento se utilizaron 16 procesos, por lo tanto, 16 nodos del *cluster* IBM.

5.2.1 Escenario 1

Para el primer experimento tenemos una matriz cuadrada de 150.000x150.000, es decir, una matriz de 22.500.000.000 de elementos, de los cuales 24.959.106 son elementos *nonzero*.

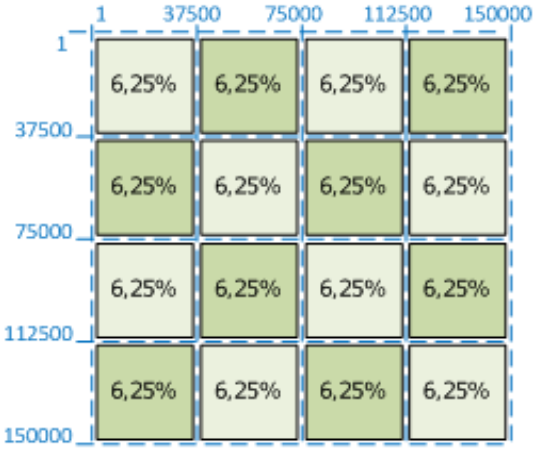
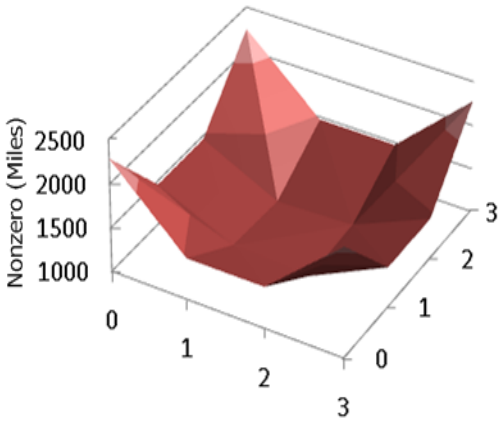


Figura 5.1: Porcentaje asignado a cada proceso del total del área de la matriz de entrada, repartición homogénea.

Si hacemos una repartición por bloques homogéneos, cada proceso tiene igual cantidad de filas y columnas. Como se muestra en la figura 5.1 el porcentaje del área total del problema se divide en 16 bloques, donde cada bloque corresponde a un 6,25 % de tamaño total de la matriz % de entrada.

Ahora si analizamos la cantidad de elementos *nonzero* en cada uno de los bloques vemos que la sección central de la matriz sufre de una falta de elementos *nonzero*, mientras que los extremos poseen una mayor cantidad de elementos *nonzero*. En la tabla 5.3 observamos la cantidad de elementos *nonzero* distribuidos en 16 bloques, en azul las zonas menos cargadas y en rojo las zonas más cargadas; en el gráfico de la derecha se ve representado esto.

n/n	0	1	2	3
0	2.285	1.470	1.472	2.246
1	1.457	1.068	1.031	1.455
2	1.458	1.032	1.071	1.456
3	2.245	1.469	1.470	2.275



Cuadro 5.3: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), repartición homogénea.

Bajo esta distribución homogénea vemos que el bloque más cargado posee 2.285.138 elementos *nonzero* y el menos cargado posee 1.030.963. Aquí apreciamos un desbalance muy significativo, y en la ejecución de la aplicación tendríamos procesos con más del doble de trabajo que algunos de sus pares. El gráfico 5.2 muestra este grado de desbalance en la carga de trabajo.

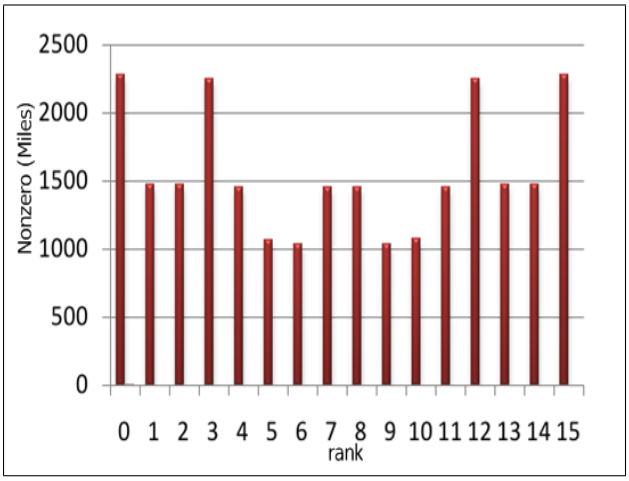


Figura 5.2: Carga de trabajo en cada proceso, repartición homogénea.

Debido este grado de desbalance, si deseamos mejorar las prestaciones para esta entrada de datos la repartición no debe ser homogénea. Por lo cual, podemos hacer un balance de carga utilizando las políticas propuestas en el punto 4.3.2 para mejorar las prestaciones. Sin embargo, como no conocemos la distribución de los elementos *nonzero* dentro de cada bloque, no podemos anticipar exactamente cuántas filas o columnas se necesitan para obtener el mejor rendimiento en este experimento.

Proponemos el uso de dos configuraciones de particionamiento heterogéneo basándonos en la carga de trabajo que posee cada bloque expuesto en la tabla 5.3. En la primera configuración movemos 2.500 filas de los 4 bloques dela segunda fila hacia los bloques superiores y 2.500 filas de los 4 bloques de la penúltima fila hacia los bloques inferiores a estos (respetando su respectivo movimiento de columnas para mantener la simetría).

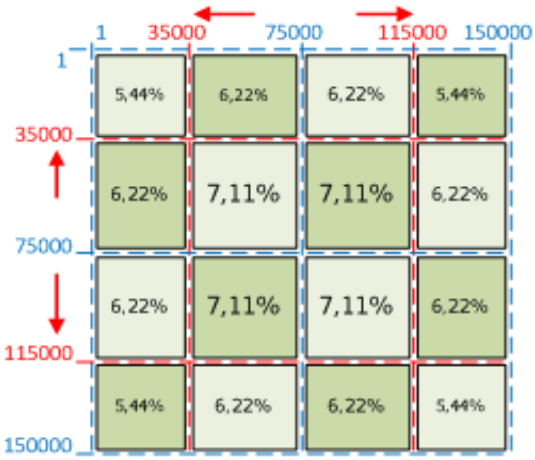
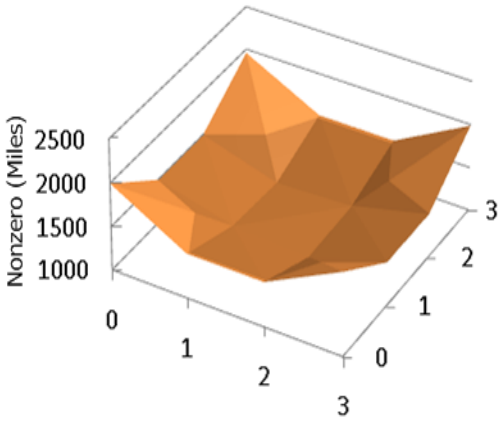


Figura 5.3: Porcentaje asignado a cada proceso del total del área de la matriz de entrada, repartición 1.

En la figura 5.3 se muestra el resultado de la repartición por bloque después de la reconfiguración, exponiendo la variación en el porcentaje del área total de la matriz de entrada que

cada proceso recibiría.

n/n	0	1	2	3
0	1.993	1.511	1.513	1.957
1	1.499	1.276	1.236	1.497
2	1.500	1.237	1.280	1.496
3	1.957	1.510	1.510	1.987



Cuadro 5.4: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), repartición 1.

Esta nueva distribución de elementos *nonzero* dentro de cada bloque mostrada en la tabla 5.4 permite que el bloque más cargado ahora poseerá 1.993.197 elementos *nonzero* y el bloque menos cargado 1.236.082. La distancia entre estos dos bloque se ve reducida, pero no lo suficiente para explotar en mayor medida el rendimiento (ver figura 5.4).

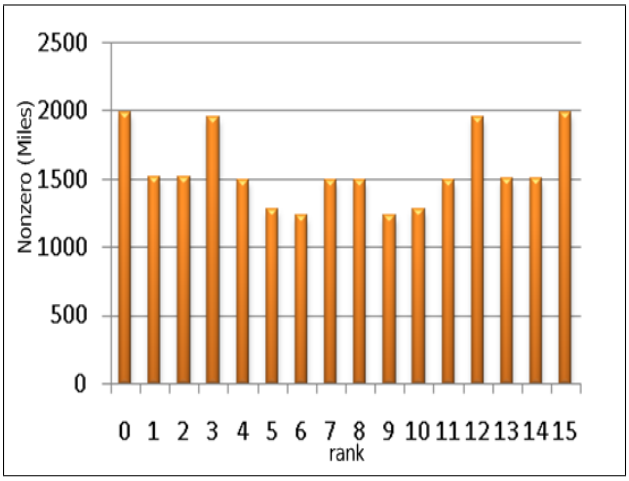


Figura 5.4: Carga de trabajo en cada proceso, repartición 1.

Según la tabla 5.4 vemos que aún la carga de trabajo se sigue concentrando en los bloques de las esquinas, por lo que el ajuste realizado no ha sido suficiente.

En la segunda configuración que utilizaremos moveremos filas y columnas en mismo sentido que la configuración anterior, pero ahora la cantidad de elementos *nonzero* será 6.250 con la finalidad de realizar un cambio considerable en la carga de trabajo.

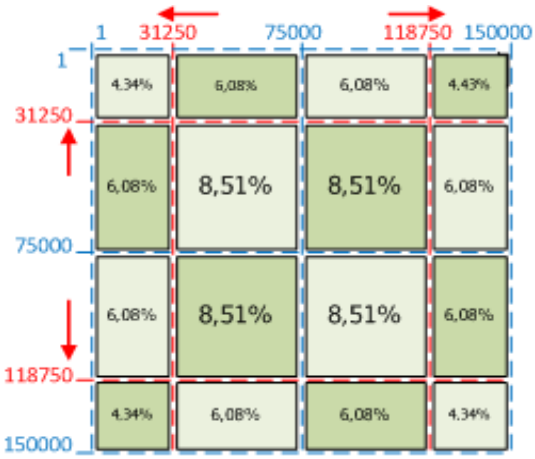
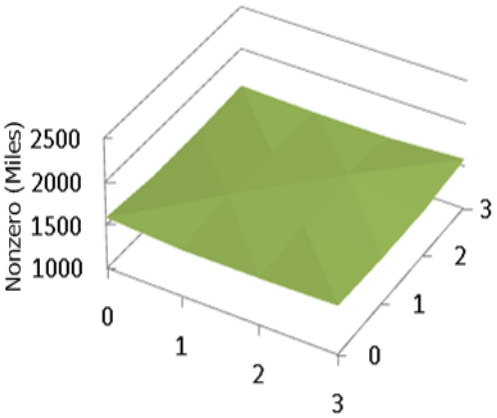


Figura 5.5: Porcentaje asignado a cada proceso del total del área de la matriz de entrada, repartición 2.

Tal como se muestra en la figura 5.5 vemos una gran variación en los porcentajes del área total de la matriz asignada a cada bloque. Por ejemplo los bloques de los extremos han perdido casi un 30 % de su área original, mientras que los bloques centrales has ganado un 36 % aproximadamente del área de sus vecinos.

n/n	0	1	2	3
0	1.591	1.537	1.539	1.557
1	1.526	1.626	1.585	1.523
2	1.527	1.585	1.632	1.521
3	1.557	1.534	1.534	1.586



Cuadro 5.5: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), repartición 2.

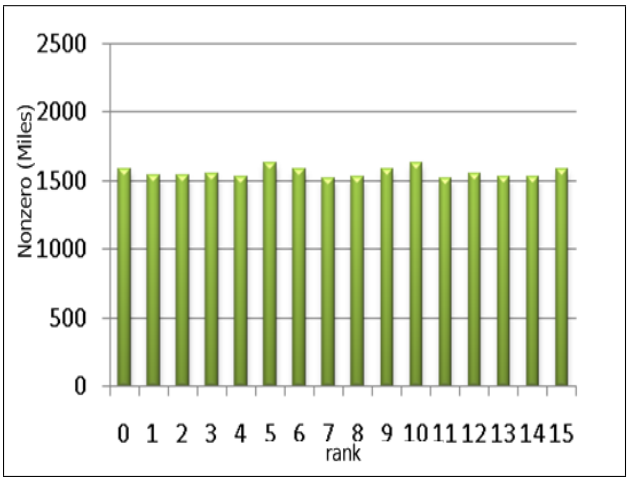


Figura 5.6: Carga de trabajo en cada proceso, repartición 2.

A través de la tabla 5.5 y la figura 5.6 observamos que la diferencia de la carga de trabajo para cada bloque es similar. Esto queda manifiesto en la tabla 5.5, en color azul y rojo se ve que los bloques más cargados y menos cargados están bastante próximos en cuanto a la cantidad de elementos *nonzero*.

En los tiempo de ejecución presentados en la tabla resumen 5.6 se ve reflejado claramente el efecto que tiene el equilibrar la carga de trabajo sobre el rendimiento de esta aplicación SPMD. También vemos que mientras más se aproximan a la media los valores máximos y mínimos de elementos *nonzero* más balanceada estará nuestra aplicación en cuanto a carga de trabajo.

Repartición	**homogénea**(miles)	**primera**(miles)	**segunda**(miles)
máximo nonzeros	2.285	1.993	1.632
media nonzeros	1.560	1.560	1.560
mínimo nonzeros	1.031	1.236	1.521
desviación estándar	454	268	36
Tiempo de ejecución (seg)	50,60	47,35	43,39
Ganancia de tiempo	**-**	**6,43 %**	**14,25 %**

Cuadro 5.6: Tabla resumen de las ejecuciones bajo diferentes reparticiones de la matriz de entrada.

Cambiando la repartición de datos también vemos que el grado de dispersión se ve reducido, ya que en un comienzo la dispersión es 453.955,78 elementos *nonzero* para luego cambiar a 268.102,94 y 36.466,92 para la primera y segunda repartición respectivamente. El grado de dispersión de la carga de trabajo se ve reflejado en la ganancia obtenida alcanzando un 14,25 % en tiempo.

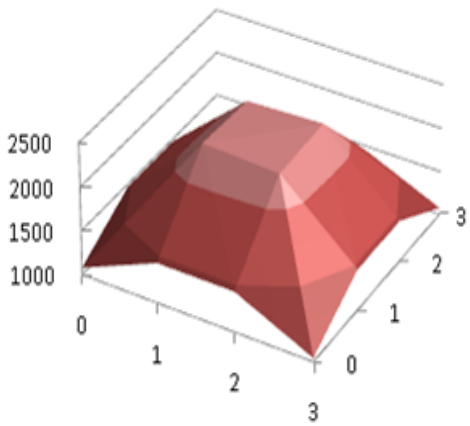
En este primer escenario, se puede describir la influencia que tiene una repartición homogénea de elementos *nonzero* en el rendimiento de la aplicación.

5.2.2 Escenario 2

En este segundo experimento tenemos una matriz cuadrada de igual tamaño que la anterior (150.000x150.000), y la misma cantidad de elementos (22.500.000.000), pero esta vez la cantidad de elementos *nonzero* es 24.960.156 y también la distribución de estos elementos es distinta.

Haciendo una repartición homogénea de filas y columnas la cantidad de elementos *nonzero* en esta nueva matriz vemos que los extremos de la matriz sufren de una falta de elementos *nonzero*, mientras que sección central posee una mayor cantidad de elementos *nonzero*. En la tabla 5.7 observamos la cantidad de elementos *nonzero* distribuidos en 16 bloques, en azul las zonas menos cargadas y en rojo las zonas más cargadas; el gráfico de superficie de la derecha muestra el alto grado de desbalance.

n/n	**0**	**1**	**2**	**3**
0	1.078	1.456	1.458	1.042
1	1.471	2.272	2.234	1.469
2	1.472	2.235	2.274	1.470
3	1.042	1.455	1.457	1.075

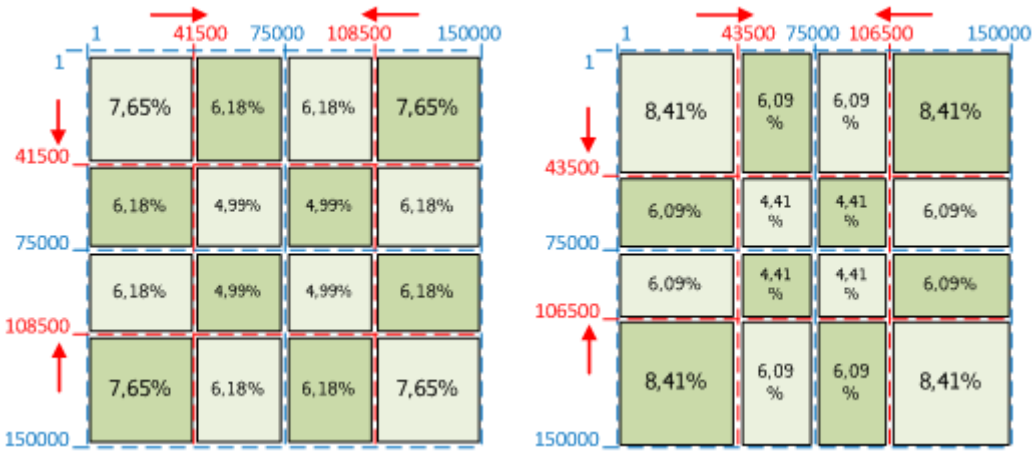


Cuadro 5.7: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), repartición homogénea.

En esta distribución homogénea el bloque más cargado posee 2.274.099 elementos *nonzero* y el menos cargado posee 1.041.537. Aquí el desbalance es evidente. El gráfico 5.2 nos da a entender este grado de desbalance en la carga de trabajo y nos hace presumir que en la

ejecución de la aplicación tendríamos procesos con mucho más trabajo que algunos que otros.

Para este caso también podemos hacer un balance de carga utilizando las políticas propuestas en el punto 4.3.2. Para lo cual tenemos dos configuraciones de particionamiento heterogéneo basándonos en la carga de trabajo que posee cada bloque expuesto en la tabla 5.7. En ambas configuraciones buscaremos reducir el tamaño de los bloques centrales como se puede ver en la figura 5.7. En la primera configuración moveremos 4.000 filas de los 4 bloques superiores hacia los bloques de la segunda fila y 2.500 filas de los 4 bloques inferiores hacia los bloques anteriores a estos (también con su respectivo movimiento de columnas para mantener la simetría).



(a) Repartición 1 (b) Repartición 2

Figura 5.7: Porcentaje asignado a cada proceso del total del área de la matriz de entrada.

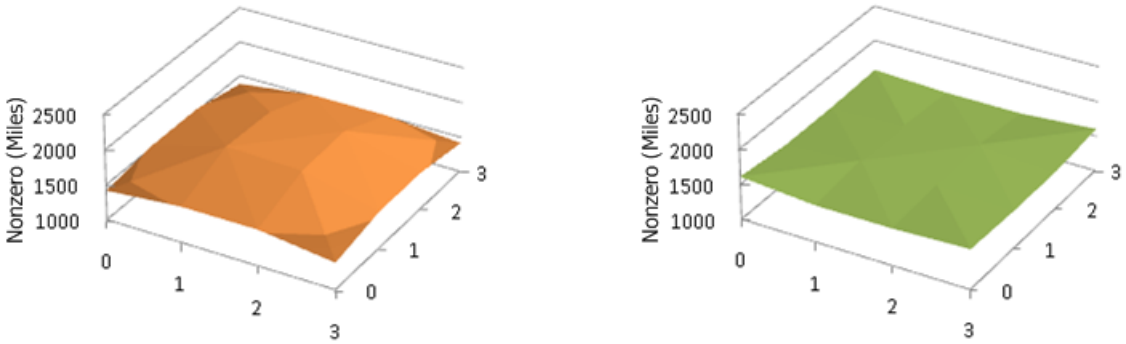
La nueva distribución de elementos *nonzero* dentro de cada bloque para ambas particiones es mostrada en la tabla 5.8. Para la primera repartición el bloque más cargado ahora posee 1.818.764 elementos *nonzero* y el bloque menos cargado 1.378.665. En la segunda el bloque más cargado posee 1.612.181 y el menos cargado 1.521.567.

n/n	0	1	2	3
0	1.419	1.513	1.515	1.380
1	1.526	1.819	1.784	1.524
2	1.528	1.784	1.819	1.527
3	1.379	1.513	1.516	1.415

n/n	0	1	2	3
0	1.609	1.522	1.524	1.567
1	1.534	1.612	1.578	1.535
2	1.536	1.578	1.610	1.536
3	1.566	1.524	1.525	1.605

Cuadro 5.8: Distribución de elementos *nonzero* en cada bloque (Miles de elementos), reparticiones heterogéneas.

A través de los gráficos de superficie mostrados en la figura 5.8 se aprecia la corrección del desbalance de la carga hecha en ambas reparticiones. En cuanto a la cantidad de elementos *nonzero*, las diferencias entre los bloques con las máximas y mínimas cargas de trabajo se ve reducida bajo estas configuraciones.

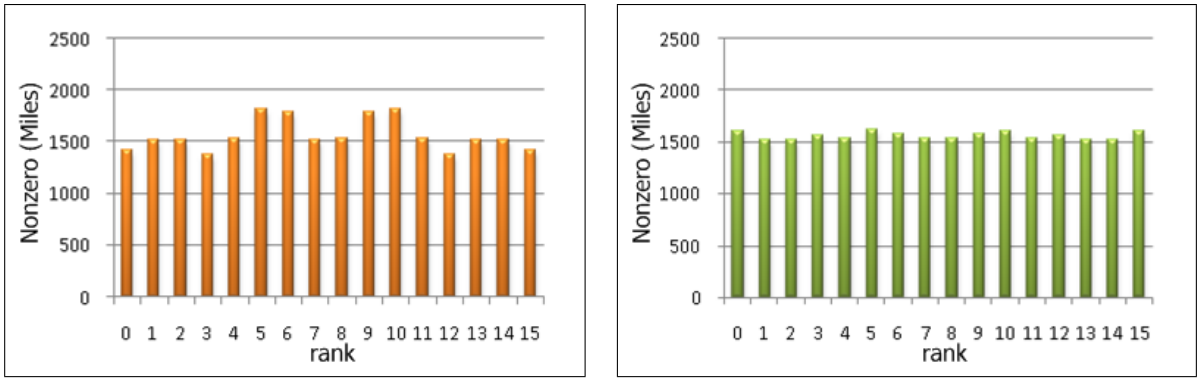


(a) Repartición 2

(b) Repartición 3

Figura 5.8: Gráfico de distribución superficial de elementos *nonzero* (Miles de elementos), reparticiones heterogéneas.

Del mismo modo la figura 5.9 nos muestra que la diferencia de la carga de trabajo para cada bloque se corrige en ambas reparticiones.



(a) Repartición 2 (b) Repartición 3

Figura 5.9: Carga de trabajo en cada proceso, reparticiones heterogéneas.

De la misma forma que en el escenario anterior los tiempo de ejecución de la tabla resumen 5.6 muestran claramente la mejora en el rendimiento de esta aplicación SPMD. De igual manera cuando los valores máximos y mínimos de elementos *nonzero* se aproximan más a la media tendremos nuestra aplicación más balanceada en cuanto a carga de trabajo.

Por lo tanto el grado de dispersión de los elementos *nonzero* en cada bloque tiene un efecto directamente sobre el rendimiento. En la primera y segunda repartición tenemos 153.547 y 34.801 elementos nonzero respectivamente, mientras que bajo una repartición homogénea el grado de dispersión era de 447.708.

Repartición	**homogénea**(miles)	**primera**(miles)	**segunda**(miles)
máximo nonzeros	2.274	1.819	1.612
media nonzeros	1.560	1.560	1.560
mínimo nonzeros	1.042	1.379	1.522
desviación estándar	448	154	35
Tiempo de ejecución (seg)	51,22	45,67	43,50
Ganancia de tiempo	**-**	**10,83 %**	**15,07 %**

Cuadro 5.9: Tabla resumen de las ejecuciones bajo diferentes reparticiones de la matriz de entrada.

Bajo este enfoque de balance de carga estático es ha obtenido una ganancia de 15,07 % en tiempo. Por lo cual a través de este experimento confirmamos una repartición heterogénea de elementos *nonzero* en esta aplicación afecta directamente al rendimiento, debido a que la carga de cómputo está dada por el número de elementos *nonzero*.

Capítulo 6

Conclusiones y Trabajo Futuro

El mejorar el rendimiento de los sistemas computacionales es una motivación importante para el HPC y para el desarrollo de nuevas arquitecturas. Las diferentes arquitecturas paralelas y la diversidad de aplicaciones SPMD existentes hoy en día exigen poner atención al rendimiento para una determinada combinación aplicación/hardware.

Actualmente el paradigma SPMD es el más utilizado para desarrollar aplicaciones paralelas. Existen diversas aproximaciones para corregir problemas de rendimiento, pero en general no existe una estrategia definida. La forma de mejorar las prestaciones va a depender de la naturaleza del problema y de su implementación sobre un determinado hardware.

Esta tarea no es fácil, ya que se debe monitorizar la aplicación para obtener información de su comportamiento, identificar los cuellos de botella, identificar los factores que afectan el rendimiento y posteriormente modificar parámetros críticos de la aplicación para mejorar su rendimiento.

Lo importante de una ejecución eficiente en SPMD, es conocer la carga cómputo de cada

proceso y el volumen de comunicaciones, con la finalidad de alinear las diferentes fases de cómputo y comunicación. Para lo cual se hace necesario conocer la aplicación y determinar qué factores influyen directamente sobre esta.

6.1 Conclusiones

El principal objetivo de este trabajo es sentar bases de conocimiento para desarrollar un modelo de rendimiento para aplicaciones SPMD que permita mejorar sus prestaciones dinámicamente.

En este trabajo hemos estudiado el funcionamiento de la aplicación tipo *benchmark* CG y hemos analizado su funcionamiento con cargas de datos heterogéneas en un *cluster* homogéneo. Debido al tipo de comunicaciones horizontales y transpuestas diagonales, en esta aplicación el movimiento de carga debe hacerse respetando una serie de políticas que permitan que el tamaño de los datos enviados y recibidos sean iguales entre los diferentes proceso.

Luego, hemos analizado la influencia de la cantidad de elementos *nonzero* como factor de rendimiento en esta aplicación. Por lo cual, la carga de trabajo no está dada por la cantidad de filas y columnas asignados a cada proceso, sino por la cantidad de elementos *nonzero* que posee cada bloque.

También hemos propuesto una estrategia para mejorar el rendimiento de aplicaciones paralelas con un comportamiento similar al *benchmark* CG. Es decir, una aplicación que trabaje con matrices *sparse*, que tenga intercambios transpuestos a la diagonal principal de la matriz y una repartición por bloques cuadrados homogéneos.

Observando este factor, si deseamos obtener un mejor rendimiento principalmente necesitamos equilibrar la cantidad de elementos *nonzero* en todos los procesos. Es decir, disminuir el grado de dispersión de las cargas de trabajo tratando de que en cada proceso se aproxime a la media de elementos *nonzero*, esto se ve directamente reflejado en reducir la diferencia de elementos *nonzero* entre el proceso más cargado y el proceso menos cargado.

Finalmente mediante la experimentación hemos podido comprobar a través de esta estrategia que se puede mejorar el rendimiento de la aplicación obteniendo una ganancia en tiempo de ejecución dependiente del grado de dispersión de la carga de trabajo.

6.2 Trabajos Futuros

En cuanto a los trabajos futuros los hemos divido en un grupo a corto plazo y un grupo de trabajos a largo plazo. Ambos grupos enfocadosde cara al objetivo de desarrollar un modelo de rendimiento para aplicaciones SPMD que permita mejorar sus prestaciones dinámicamente.

A corto plazo tenemos planteado analizar otra aplicación SPMD o bien alguna librería para crear aplicaciones de carácter matemático basadas en el paradigma SPMD. Esto con la finalidad de ampliar el conocimiento hacer del comportamiento de las aplicaciones SPMD.

También nos hemos propuesto estudiar otros factores de rendimiento como lo son el tamaño de las comunicaciones y analizar cuál es el límite máximo de movimiento de carga entre bloques sin que se vea mermada la relación cómputo/comunicación.

A largo plazo tenemos el proponer estrategias de balance de carga en otras aplicaciones,

quizás con un tipo de entrada de datos diferente a las matrices *sparse*.

Finalmente definiremos un modelo de rendimiento para SPMD y lo integraremos a MATE para mejorar las prestaciones mediante sintonización dinámica y automática.

Bibliografía

[1] Top500 supercomputer site [online]. http://www.top500.org.

[2] Shahnaz Afroz, Hee Yong Youn, and Dongman Lee. Performance of message logging protocols for nows with mpi. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, PRDC '99, pages 252–, Washington, DC, USA, 1999. IEEE Computer Society.

[3] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Co-design of distributed systems using skeleton and autonomic management abstractions. In *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 403–414. Springer Berlin / Heidelberg, 2009.

[4] George S. Almasi and Allan Gottlieb. *Highly parallel computing (2. ed.)*. Addison-Wesley, 1994.

[5] J. W. Backus. *The Fortran Automatic Coding System for the IBM 704 EDPM*. 1956.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[8] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[9] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *Euro-Par 2005*, volume 3648 of *Lecture Notes in Computer Science (LNCS)*, pages 761–770, 2005.

[10] Anne Benoit, Loris Marchal, Yves Robert, and Oliver Sinnen. Mapping pipelined applications with replication to increase throughput and reliability. In *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pages 55–62, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Vicente Blanco. *Análisis, predicción y visualización del rendimiento de métodos iterativos en HPF y MPI*. PhD thesis, Universidad de Santiago de Compostela, 2002.

[12] Douglas M. Blough and Peng Liu. Fimd-mpi: A tool for injecting faults into mpi applications. *Parallel and Distributed Processing Symposium, International*, 0:241, 2000.

[13] Cristina Boeres, Vinod E. F. Rebello, Cristina Boeres, and Vinod E. F. Rebello. Towards optimal static task scheduling for realistic machine models: theory and practice, 2003.

[14] Jurgen Brehm. Performance analysis of single and multiprocessor computing systems, 1999.

[15] Jurgen Brehm, Patrick H. Worley, and Manish Madhukar. Performance modeling for spmd message-passing programs, 1996.

[16] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[17] E. Cesar, A. Moreno, J. Sorribes, and E. Luque. Modeling master/worker applications for automatic performance tuning. *Parallel Comput.*, 32:568–589, September 2006.

[18] Eduardo Cesar. *Definition of Framework-based Performance Models for Dynamic Performance Tuning*. PhD thesis, Universitat Autonoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2006.

[19] Eduardo Cesar, Joan Sorribes, and Emilio Luque. Modeling pipeline applications in poetries. In Jose Cunha and Pedro Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 643–643. Springer Berlin / Heidelberg, 2005.

[20] Satish Chandra, James R. Larus, and Anne Rogers. Where is time spent in message-passing and shared-memory programs? *SIGOPS Oper. Syst. Rev.*, 28:61–73, November 1994.

[21] Keping Chen, Kenneth R. Mayes, and John R. Gurd. Autonomous performance control of distributed applications in a heterogeneous environment. In *Proceedings of the 1st international conference on Autonomic computing and communication systems*, Autonomics '07, pages 14:1–14:5, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[22] Xu Chengzhong and Francis C. M. Lau. A survey of nearest-neighbor load balancing algorithms. In *Load Balancing in Parallel Computers*, volume 381 of *The Kluwer International Series in Engineering and Computer Science*, pages 21–35. Springer US, 1997.

[23] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30:389–406, March 2004.

[24] Cristian, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[25] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[26] Timothy A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994.

[27] Travis Desell, Kaoutar Maghraoui, and Carlos Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, 10:323–337, 2007. 10.1007/s10586-007-0032-9.

[28] Hank Dietz. Linux parallel processing howto, 1998.

[29] Antonio Dorta, Pablo Lopez, and Francisco de Sande. Basic skeletons in llc. *Parallel Computing*, 32(7-8):491 – 506, 2006. Algorithmic Skeletons.

[30] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23:5–16, February 1990.

[31] Antonio Espinosa, Tomàs Margalef, and Emilio Luque. Integrating automatic techniques in a performance analysis session (research note). In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 173–177, London, UK, 2000. Springer-Verlag.

[32] Dror G. Feitelson and Larry Rudolph. Towards convergence in job schedulers for parallel supercomputers. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 1–26, London, UK, 1996. Springer-Verlag.

[33] Steven Fortune. Voronoi diagrams and delaunay triangulations. *Discrete & Computational Geometry*, 1995.

[34] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[35] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[36] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, USA, 1994.

[37] Sheikh K. Ghafoor, Tomasz A. Haupt, Ioana Banicescu, and Ricolindo L. Carino. A resource management system for adaptive parallel applications in cluster environments. 2005.

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

[39] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22:789–828, September 1996.

[40] Alex Guevara. Integración de politicas de distribución de datos para aplicaciones tipo master/worker en la librería eskel . Master's thesis, Universitat Autònoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2008.

[41] Per Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency - Practice and Experience*, 5(5):407–423, 1993.

[42] Michael T. Heath and Jennifer E. Finger. Paragraph: A performance visualization tool for mpi. 1999.

[43] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[44] W. Daniel Hillis and Guy L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29:1170–1183, December 1986.

[45] Jay P. Hoeflinger. Extending openmp to clusters, 2006.

[46] Mohamed Hussein, Ken Mayes, Mikel Lujan, and John Gurd. Adaptive performance control for distributed scientific coupled models. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 274–283, New York, NY, USA, 2007. ACM.

[47] Forschungszentrum J, Interner Bericht, Wolfgang E. Nagel, Alfred Arnold, Michael Weber, W. E. Nagel, A. Arnold, M. Weber, and K. Solchenbach. Vampir: Visualization and analysis of mpi resources, 1996.

[48] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

[49] Josep Jorba. *Análisis automático de prestaciones de aplicaciones paralelas basadas en paso de mensajes.* PhD thesis, Universitat Autònoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2006.

[50] Sanjeev Krishnan and Laxmikant V. Kale. Automating parallel runtime optimizations using post-mortem analysis. In *Proceedings of the 10th international conference on Supercomputing*, ICS '96, pages 221–228, New York, NY, USA, 1996. ACM.

[51] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[52] Eric Maillet. Tape /pvm an efficient performance monitor for pvm applications. user guide, 1996.

[53] Andrea Martínez. Sintonización dinámica de aplicaciones mpi. Master's thesis, Universitat Autònoma de Barcelona. Departament d'Arquitectura de Computadors i Sistemes Operatius, 2010.

[54] Kenneth R. Mayes, Mikel Luján, Graham D. Riley, Jonathan Chin, Peter V. Coveney, and John R. Gurd. Towards performance control on the Grid. *Philosophical Transactions of the Royal Society: Series A*, 363(1833):1975–1986, 2005.

[55] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[56] Dan I. Moldovan. *Parallel Processing: From Applications to Systems.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[57] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. Mate: Monitoring, analysis and tuning environment for parallel distributed applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 19:1517–1531, August 2007.

[58] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef, and E. Luque. Dynamic pipeline mapping (dpm). In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 295–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[59] Ronal Muresano, Dolores Rexachs, and Emilio Luque. A tool for efficient execution of spmd applications on multicore clusters. *Procedia Computer Science*, 1(1):2599 – 2608, 2010. ICCS 2010.

[60] nCUBE company. ncube 2 supercomputers parallel programming principles. Technical report, Foster City, CA., 1993.

[61] Leonid Oliker, Xiaoye Li, Parry Husbands, and Rupak Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *SIAM Rev.*, 44:373–393, March 2002.

[62] A. Osman and H. Ammar. A scalable dynamic load-balancing algorithm for spmd applications on a non-dedicated heterogeneous network of workstations (hnow), 2003.

[63] Roberto Uribe Paredes, Claudio Márquez, and Roberto Solar. Construction strategies on metric structures for similarity search. *CLEI Electron. J.*, 12(3), 2009.

[64] A. Plastino, C. C. Ribeiro, and N. Rodríguez. Developing spmd applications with load balancing. *Parallel Comput.*, 29:743–766, June 2003.

[65] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable performance analysis: The pablo performance analysis environment. 1993.

[66] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst.*, 18:175–187, September 2001.

[67] Gudula Runger. Parallel programming models for irregular algorithms. In Karl Heinz Hoffmann and Arnd Meyer, editors, *Parallel Algorithms and Cluster Computing*, volume 52 of *Lecture Notes in Computational Science and Engineering*, pages 3–23. Springer Berlin Heidelberg, 2006.

[68] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2, 1994.

[69] Luis Moura E Silva and Rajkumar Buyya. Parallel programming models and paradigms. 1998.

[70] Roberto Solar, Remo Suppi, and Emilio Luque. High performance distributed cluster-based individual-oriented fish school simulation. *Procedia CS*, 4:76–85, 2011.

[71] Aad J. Van Der Steen and Jack J. Dongarra. Overview of recent supercomputers, 1996.

[72] Aad J. van der Steen and Jack Dongarra. Handbook of massive data sets. chapter Overview of high performance computers, pages 791–852. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[73] Barry Wilkinson and Michael Allen. *Parallel Programming*. Prentice Hall, 1999.

[74] Felix Wolf and Bernd Mohr. Automatic performance analysis of mpi applications based on event traces. In *Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, Euro-Par '00, pages 123–132, London, UK, 2000. Springer-Verlag.

[75] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49:421–439, November 2003.

[76] Jerry C. Yan. Performance tuning with aims - an automated instrumentation and monitoring system for multicomputers. In *Hawaii International Conference on System Sciences*, pages 625–633, 1994.

[77] Omer Zaki, Ewing Lusk, and Deborah Swider. Toward scalable performance visualization with jumpshot. *High Performance Computing Applications*, 13:277–288, 1999.

[78] Andrea Zavanella and Alessandro Milazzo. Predictability of bulk synchronous programs using mpi. In *Proceedings of the 8th Euromicro conference on Parallel and distributed processing*, EURO-PDP'00, pages 118–123, Washington, DC, USA, 1999. IEEE Computer Society.