UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Comparación del uso de GPGPU y cluster de multicore en problemas con alta demanda computacional

Autores: Montes de Oca, Erica Soledad
Director: Naiouf, Marcelo
Codirector: De Giusti, Laura
Asesor profesional: --
Carrera: Licenciatura en Informática

Resumen

La presente Tesina de Grado tiene como objetivo la investigación y el estudio de las plataformas de memoria compartida GPU y cluster de Multicore para la resolución de problemas con alta demanda computacional. Se presentan soluciones al problema planteado con el fin de comparar rendimiento en sus versiones secuencial, paralela con memoria compartida, paralela con pasaje de mensajes, paralela híbrida y paralela en GPU. Se analiza la bondad de las soluciones en relación al tiempo de ejecución y aceleración, y se introduce el análisis de consumo energético.

Palabras Claves

Programación paralela.
Multicore. Cluster de Multicore.
GPU. GPGPU. CUDA.
Problemas de alta demanda computacional.
N-Body.

Conclusiones

Se ha podido demostrar los beneficios del uso de la GPU en problemas con características similares al caso de estudio planteado en el presente trabajo. Los tiempos de ejecución obtenidos son considerablemente inferiores comparados con las diferentes soluciones implementadas para la arquitectura CPU. Además, los experimentos realizados desde el punto de vista del consumo energético favorecen el uso de la GPU en problemas del estilo de N-Body.

Trabajos Realizados

Se estudiaron los fundamentos del procesamiento paralelo para desarrollar, analizar y evaluar distintas soluciones para el problema de alta demanda computacional: N-Body. Se emplearon varios modelos de comunicación: memoria compartida (con Pthread en CPU y CUDA en GPU), pasajes de mensajes (con MPI) y diferentes soluciones híbridas (con MPI-Pthread). Además,se analizó el problema desde el punto de vista energético, introduciendo de esta manera, los fundamentos de consumo energético.

Trabajos Futuros

Estudiar, desarrollar, analizar y evaluar soluciones híbridas utilizando MPI-CUDA.
Implementar y realizar un estudio de escalabilidad y eficiencia en soluciones para Cluster de GPUs.
Extender el estudio del consumo energético en GPU.
Avanzar en el análisis del uso de otros algoritmos (Treecode y FMM) para la solución del problema de aplicación.

Fecha de la presentación: Diciembre 2012

Comparación del uso de GPGPU y cluster de Multicore en problemas con alta demanda computacional

Tesina de Grado de Licenciatura en Informática

Alumna: A.P.U. Montes de Oca Erica S.
Director: Dr. Naiouf Marcelo
Codirector: Dra. De Giusti Laura

Facultad de Informática
Universidad Nacional de La Plata

"Tu tiempo es limitado, de modo que no lo malgastes viviendo la vida de alguien distinto. No quedes atrapado en el dogma, que es vivir como otros piensan que deberías vivir. No dejes que los ruidos de las opiniones de los demás acallen tu propia voz interior. Y, lo que es más importante, ten el coraje para hacer lo que te dicen tu corazón y tu intuición. Ellos ya saben de algún modo en qué quieres convertirte realmente. Todo lo demás es secundario"

Steve Jobs

Agradecimientos

En primer lugar a Dios, porque es gracias a Él que "...vivo, me muevo y existo...".

A mis padres que me han apoyado, aconsejado y enseñado los valores morales que me han formado como la persona que soy hoy.

A mi director Marcelo Naiouf y codirector Laura De Giusti por su paciencia, su tiempo y su conocimiento invertido en mí.

A Franco, Emanuel, Fabiana y Enzo por su tiempo y ayuda.

Al III LIDI por aceptarme dentro de su familia y hacer que este trabajo de Grado sea posible.

A la Facultad de Infomática, Universidad Nacional de La Plata, por todo el conocimiento dado durante estos años de estudio de manera gratuita.

Muchas gracias

Índice

Objetivos

La presente Tesina de Grado tiene como objetivo la investigación y el estudio de las plataformas de memoria compartida GPU y cluster de Multicore para la resolución de problemas con alta demanda computacional. Presentando una solución al problema planteado con el fin de la comparación de rendimiento en su versión secuencial, paralela con memoria compartida, paralela con memoria distribuida, paralela híbrida y paralela en GPU, pudiendo sacar una conclusión sobre conveniencias de las soluciones con relevancia en tiempo de ejecución y consumo energético.

Capítulo 1

Introducción

"Divide et vinces...
Divide y vencerás"[1]

1.1. Procesamiento paralelo

La gran cantidad de datos que algunas aplicaciones necesitan procesar junto con el alcance de la cantidad máxima de transistores en un microchip, ha hecho que la única forma de tratar algunos problemas computables sea a través del procesamiento paralelo. Los problemas científicos de alta demanda computacional tales como: diseño de vehículos (automóviles, aeronaves, motores), control de sistemas de tiempo real de manejo de múltiples señales, diseños de circuitos de alta complejidad, diseño y simulación del comportamiento de estructuras, modelo del clima, procesamiento de imágenes y reconstrucción 3D, bioinformática, datamining, servidores de altas prestaciones, etc. [GGKK03], requieren respuestas en un período de tiempo razonable. Claro está, que el procesamiento secuencial para tales casos, ocasionaría soluciones en días, meses, o años.

El intento de aumentar la velocidad en el procesamiento secuencial en la conocida máquina de Von Neumman, está orientado por un lado a mejorar el hardware existente, aumentando su capacidad, o agregar más recursos de ejecución, dando lugar al procesamiento paralelo.

1 Emperador Julio César

1.1.1. Definición de procesamiento paralelo

Se define al **procesamiento paralelo** como la *ejecución de la solución de un problema de manera concurrente o simultánea sobre diferentes componentes físicos*. El procesamiento del algoritmo paralelo para dicho problema se ejecuta de forma coordinada y cooperante, descomponiéndose en múltiples procesos en varios procesadores. Busca tratar de realizar un procesamiento de la información de manera eficiente en el menor tiempo posible [HB84].

1.1.2. Ventajas del uso del procesamiento paralelo

El aumento de los procesadores tiene la ventaja de aproximarse al paralelismo inherente de los problemas del mundo real. Dichos problemas, a demás suelen requerir el procesamiento de grandes cantidades de datos bajo restricciones de tiempo que solo pueden cumplirse utilizando paralelismo.

Los procesadores que se agregan, sean homogéneos o heterogéneos, son menos complejos, por lo tanto, menos costosos. En la actualidad, las máquinas monoprocesador han desaparecido del mercado, lo que implica que inevitablemente si se desea aprovechar al máximo la potencia de la arquitectura subyacente es necesario implementar soluciones paralelas.

Es de esperarse que si se pueden ejecutar varias instrucciones en paralelo, aún cuando éstas no se ejecutan más rápido, el tiempo total del procesamiento sea menor que el tiempo total de la ejecución análoga secuencial [DGT98].

1.1.3. Limitaciones del procesamiento paralelo

A pesar de las ventajas vistas en el apartado anterior, el paralelismo presenta varias limitaciones a considerar:

a) En este nuevo escenario los procesos paralelos requieren realizar comunicación de datos entre sí.

b) Los procesos deben esperarse entre sí para poder continuar con la ejecución de la aplicación (sincronización).

c) El máximo grado de concurrencia alcanzable se encuentra determinado por el problema concreto a paralelizar.

d) Alta dependencia al hardware subyacente.

e) Los métodos de debugging del procesamiento secuencial no sirven en procesamiento paralelo.

f) El no determinismo de la ejecución, dificulta la comprensión y el debugging de los algoritmos.

g) El desarrollador de algoritmos paralelos se encuentra frente a una serie de dificultades a resolver, tales como: no determinismo, sincronización, comunicación, división y distribución de los datos, balance de carga, tolerancia a fallos, heterogeneidad, manejo de memoria compartida y distribuida, deadlocks.

1.1.4. Fuentes de overhead en algoritmos paralelos

Se denomina fuente de overhead a todo aquello que incrementa el tiempo de ejecución del algoritmo paralelo en tiempo no útil, es decir, tiempo mal gastado que no es utilizado en el procesamiento de la aplicación.

Pueden identificarse tres fuentes de overhead en los programas paralelos:

1. Creación de procesos y scheduling;
2. Comunicación;
3. Sincronización.

Lo deseable es minimizar dichas fuentes de overhead que son inevitables en todo algoritmo paralelo. La interdependencia que existe entre las mismas implica que minimizar una de ellas, puede traer como consecuencia el incremento de otra.

El overhead en la creación de procesos puede reducirse creando un proceso por cada procesador, lo que significaría que no habría retardos por context switching. Sin embargo, en el caso que por alguna razón el proceso incurriera en un delay, el procesador quedaría ocioso, lo que no ocurriría si existiera más procesos asignados a dicho procesador.

El caso de la comunicación entre los procesos en entornos de pasaje de mensajes, el envío de mensajes a través de la red y su recepción implican un retardo de tiempo que pueden llegar a ser muy significantes. Una solución es enmascarar los retardos si se cuenta con más procesos para ejecutar. En los casos de entornos de memoria compartida

los fallos de caché pueden ser considerados de manera similar que al pasaje de mensaje en lo que a la problemática se refiere. La reducción de los accesos a memoria principal se intenta lograr dividiendo los datos que cada procesador va ha usar en diferentes partes, evitando las variables compartidas y los false sharing (ocurre cuando dos variables se encuentran almacenadas en la misma línea de caché, y son accedidas por diferentes procesadores, y luego uno de los procesadores modifica dicha variable provocando la invalidación de la línea de caché).

Cuando los procesos trabajan juntos con el fin de resolver un problema es inevitable que deba haber entre ellos alguna sincronización. Las sincronizaciones más comunes en los algoritmos paralelos son: las secciones críticas, las barreras y el pasaje de mensajes. La minimización de este tipo de overhead está basada en el uso eficiente de los protocolos de sincronización y la reducción de los delays [And00].

1.2. Concepto de Sistema Paralelo

Un sistema paralelo es la conjunción de un algoritmo paralelo con una máquina paralela. Como se mencionó anteriormente, el algoritmo paralelo se encuentra muy fuertemente ligado a la arquitectura paralela para la que se desarrolla. Por lo tanto, no puede desligarse el algoritmo de la arquitectura, por lo tanto, las optimizaciones y el rendimiento de la aplicación están íntimamente relacionadas a la arquitectura subyacente.

1.2.1. Concurrencia y paralelismo

El procesamiento concurrente comprende la ejecución de dos o más procesos simultáneamente. Cada proceso es un programa secuencial que es ejecutado por múltiples threads. La concurrencia es un concepto de software, que implica la determinación de la comunicación y sincronización entre los procesos en ejecución simultánea.

El paralelismo puede ser caracterizado como un caso particular de concurrencia, en el que los procesos que se ejecutan concurrentemente lo hacen sobre un conjunto de componentes físicos homogéneos o heterogéneos con un espacio de direcciones de memoria compartido o distribuido.

1.2.2. Cómputo distribuido y paralelismo

Aunque el procesamiento paralelo y el distribuido se encuentran relacionados, no se deben confundir los conceptos (a pesar de que muchos autores en la literatura igualan los conceptos). Un sistema paralelo busca reducir los tiempos de cómputo de las aplicaciones. En un sistema distribuido las tareas que se ejecutan se encuentran en máquinas autónomas, y su fin es compartir información y recursos. El espacio de direcciones de memoria es distribuido y las aplicaciones que se ejecutan son de múltiples usuarios y dinámicas.

1.3. Programación de propósito general en GPU

Las Unidades de Procesamiento Gráfico en sus inicios fueron utilizadas como un coprocesador para la CPU consiguiendo la generación de imágenes interpretadas de una escena en tres dimensiones, fueran plasmadas en una imagen bidimensional [Lue08].

El avance de la industria de los videojuegos, permitió que este hardware acelerador de gráficos evolucionará hasta el punto de ser visto con gran interés en ámbitos académicos e industriales como una solución a la necesidad de mayor poder de cómputo.

Es entonces, cuando nace en el 2003 el término GPGPU, computación de Propósito General en Unidades de Procesamiento Gráfico, utilizando lenguajes tales como assembler o específicos para el desarrollo de aplicaciones gráficas como: GLSL, Cg o HLSL. Muchos de los cómputos realizados sobre gráficos tales como operaciones sobre matrices y vectores podían ser adaptadas a problemas de propósito general. La programación era una tarea compleja y requería un alto grado de conocimiento sobre el proceso de renderizado de gráficos, por lo que el desarrollo de aplicaciones insumía mucho tiempo y esfuerzo al deber adaptar el problema no gráfico en términos gráficos.

A partir del 2004, los esfuerzos académicos centrados en la exploración, la investigación y el desarrollo en GPU logran que emerga con mayor ímpetu, en busca de la implementación de soluciones en diversas áreas de interés: simulaciones físicas, procesamiento de señales, visualización, bioinformática, gestión de bases de datos, problemas numéricos, etc.

GPGPU comenzó entonces a tomar lugar dentro de la computación de alto desempeño, al punto que se dejó de discutir sobre su rendimiento. Aunque aún existían ciertos obstáculos como la precisión de los resultados y la facilidad de la programación, el desarrollo en éste nuevo modelo de programación paralela tomó fuerza y determinación para lograr un avance y un mejoramiento que consiguiera su expansión. Las compañías

fabricantes de GPU comenzaron a modificar los procesadores de streams para que no solo ejecutaran operaciones gráficas sino también cómputo de propósito general. En 2007 Nvidia resuelve los problemas anteriormente mencionados con el lanzamiento de CUDA (Compute Unified Device Architecture), AMD/ATI con Brook+ (actualmente sin soporte), y reicentemente OpenCL, que permiten el desarrollo de aplicaciones más fáciles de mapear en las GPUs [FX10].

> *"CUDA es una plataforma de computación completa para C/C++/Fortran en la GPU. Cuando empezamos a crear CUDA, teníamos muchas opciones con relación a lo que podríamos construir. El punto clave, según afirmaron los clientes, fue que no tenían que aprender un lenguaje o API completamente nuevos. Algunos de ellos estaban contratando desarrolladores de juegos porque sabían que las GPU eran rápidas pero no sabían cómo llegar a ellas. Era esencial brindar una solución que fuera fácil, que se pudiera aprender en una sesión y que pudiera superar el código de su CPU."*
>
> Ian Buck, Gerente General, NVIDIA

En un principio GPGPU fue resistido por muchos que lo calificaban como un “hacking” de las GPUs por la utilización que se hacía de las mismas, ya que siendo procesadores con un propósito específico, se ejecutaban en ellas aplicaciones totalmente diferentes para las que fueron diseñadas. Sin embargo, la investigación y el desarrollo en esta arquitectura se ha abierto paso, y puede decirse que con los resultados de performance alcanzados se espera que el desarrollo en las mismas crezca por los avances en la tecnología y la investigación [KH10] [Pic11].

Capítulo 2

Arquitecturas paralelas

"El hardware no cumple su cometido sin software que lo utilice; el software no tiene razón de ser sin hardware que lo sustente"[1]

Las plataformas de cómputo paralelo pueden ser clasificadas mediante dos grandes criterios: una organización lógica y una organización física. En el primer caso, dicha organización se refiere a la forma que el programador ve a la plataforma. En cambio, la organización física clasifica a la plataforma desde el punto de vista del hardware. Dentro de estos dos grandes conjuntos existe una subclasificación. Para las plataformas clasificadas según el punto de vista del programador las mismas se subclasifican por: su mecanismo de control (Taxonomía de Flynn) y su modelo de comunicación. En la clasificación por organización física, se distinguen por: su espacio de direcciones (memoria compartida, memoria distribuida, y combinación de memoria compartida y distribuida), su red de interconexión y su granularidad.

1 Anónimo

2.1. Taxonomía de Flynn

Una taxonomía o esquema de clasificación es un sistema de reglas que permite realizar una clasificación de manera particular. La clasificación realizada por Flynn fue uno de las primeros esquemas de las arquitecturas seriales y paralelas que se han definido. Dicha taxonomía se encuentra fundamentada en dos conceptos: el flujo de instrucciones y el flujo de datos, centrándose en cómo las instrucciones se ejecutan sobre los datos en una computadora [Das89] [Qui94] [DGT98].

2.1.1. SISD (Single Instruction, Single Data)

Estas computadoras son las de procesamiento secuencial, en la que las instrucciones se ejecutan en serie, es decir, solo una instrucción puede ser procesada en una unidad de tiempo.

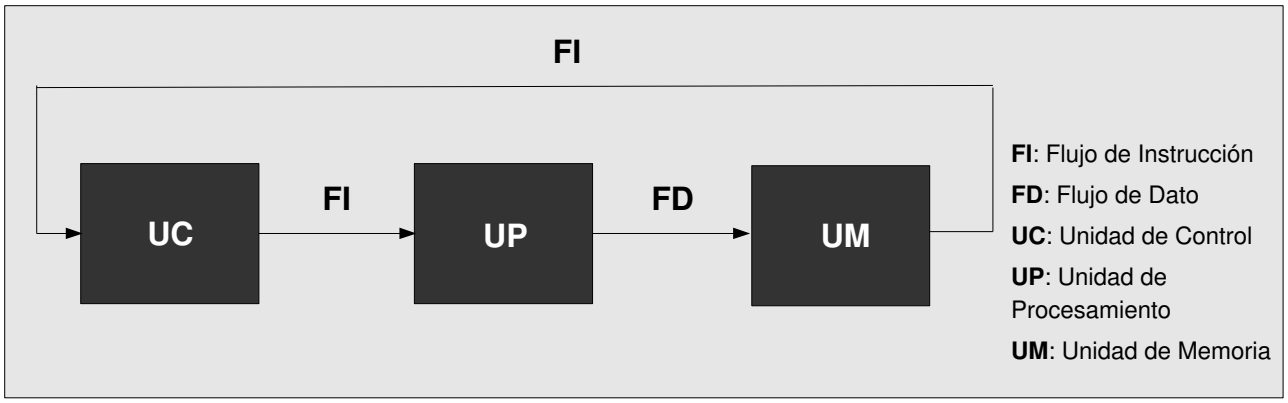


Fig. 2.1. Arquitectura SISD Monoprocesador

En la fig. 2.1. puede apreciarse que es la Unidad de Procesamiento la que lleva a cabo la ejecución de las instrucciones sobre los datos, mientras que la Unidad de Control decodifica dichas instrucciones. La Unidad de Memoria recibe y almacena los datos, y proporciona datos cuando se realiza una operación de lectura.

2.1.2. MISD (Multiple Instruction, Single Data)

Las computadoras que se encuentran dentro de esta categoría son aquellas que ejecutan múltiples instrucciones sobre un mismo flujo de datos, es decir, un mismo dato se encuentra siendo procesado en múltiples Unidades de Procesamiento.

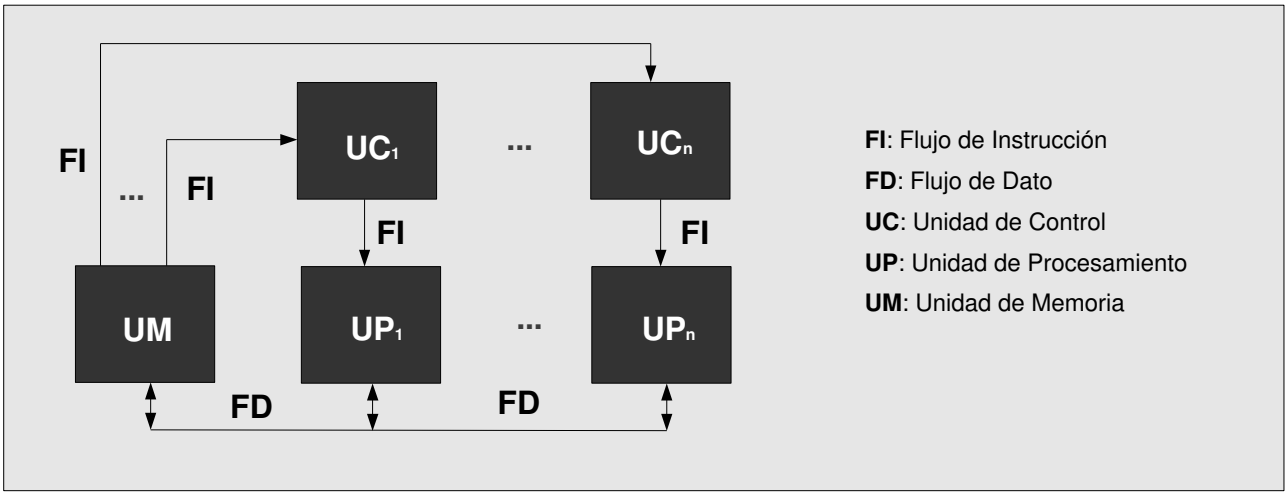


Fig. 2.2. Arquitectura MISD

Son arquitecturas paralelas de propósito específico, adaptadas a un problema en particular. Se la suele categorizar como una máquina hipotética y no práctica.

2.1.3. SIMD (Single Instruction, Multiple Data)

En esta arquitectura se cuenta con una única Unidad de Control que envía al conjunto de Elementos de Procesamiento las instrucciones decodificadas. La ejecución de las instrucciones se realiza de manera sincrónica, por causa de la existencia de una sola Unidad de Control y un único reloj en el sistema. La ejecución de las instrucciones puede realizarse en todos los Elementos de Procesamiento o no, ya que pueden ser desahabilitados alguno de ellos. Además, cada Elemento de Procesamiento puede tener su memoria local con acceso privado y exclusivo.

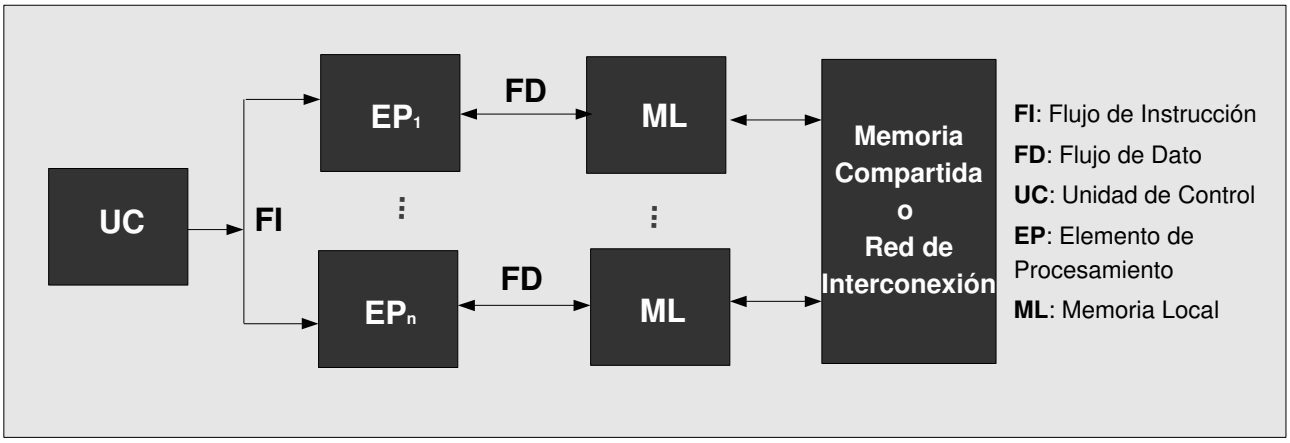


Fig. 2.3. Arquitectura SIMD con Memoria Distribuida

En la Fig. 2.3. se puede observar que los Elementos de Procesamiento pueden estar comunicados por medio de una memoria compartida o de una red de comunicaciones. Pueden identificarse en esta categoría a los procesadores vectoriales.

2.1.4. MIMD (Multiple Instruction, Multiple Data)

La arquitectura MIMD es la que representa a las máquinas paralelas y es caracterizada por ser de propósito general. Está comprendida por *n* procesadores, los cuales cuentan con una Unidad de Control propia. Por lo tanto, cada procesador ejecutará su propio flujo de instrucciones, además de procesar su propio flujo de datos. Dependiendo de la interconexión de los procesadores pueden identificarse dos subconjuntos dentro de esta categoría: los *multiprocesadores* y las *multicomputadoras.*

Los multiprocesadores se dicen que son *fuertemente acoplados* porque el grado de interconexión entre los procesadores que conforman la arquitectura es alto. Los procesadores comparten una única memoria, por lo que una dirección de memoria contiene el mismo contenido para todos los procesadores, conformando así su medio de comunicación. En este tipo de comunicación se hace imprescindible el uso de accesos sincronizados para evitar interferencias entre las áreas de memoria común.

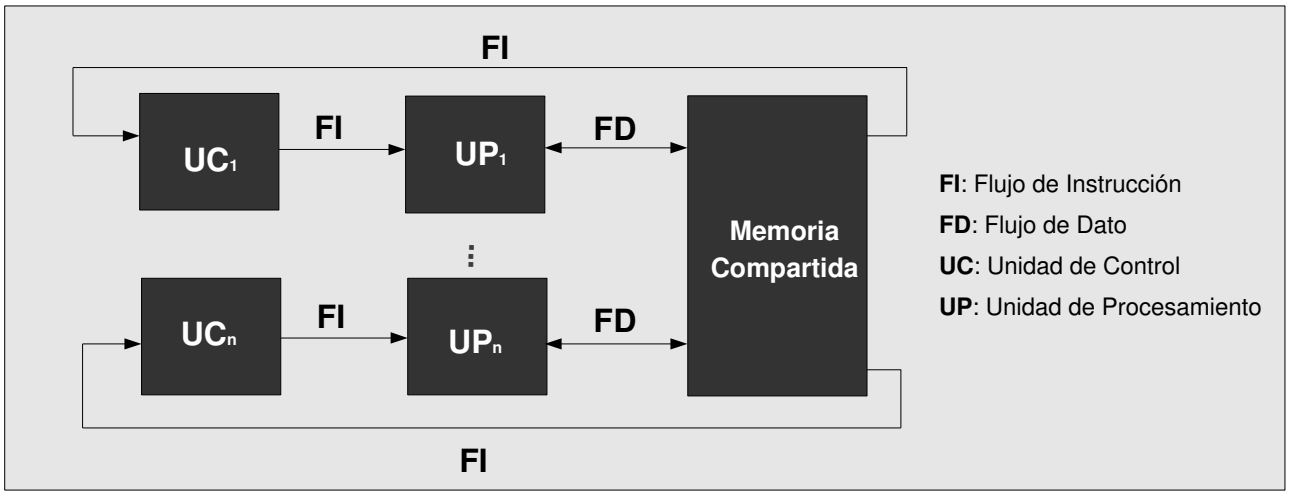


Fig. 2.4. Arquitectura MIMD con Memoria Compartida

Por su parte, las multicomputadoras se dice que son *débilmente acopladas*. Reciben varios nombres: DMPC (Distributed Memory Parallel Computers), multicomputadoras, multicomputadoras de paraje de mensajes y computadoras con arquitectura de memoria distribuida.

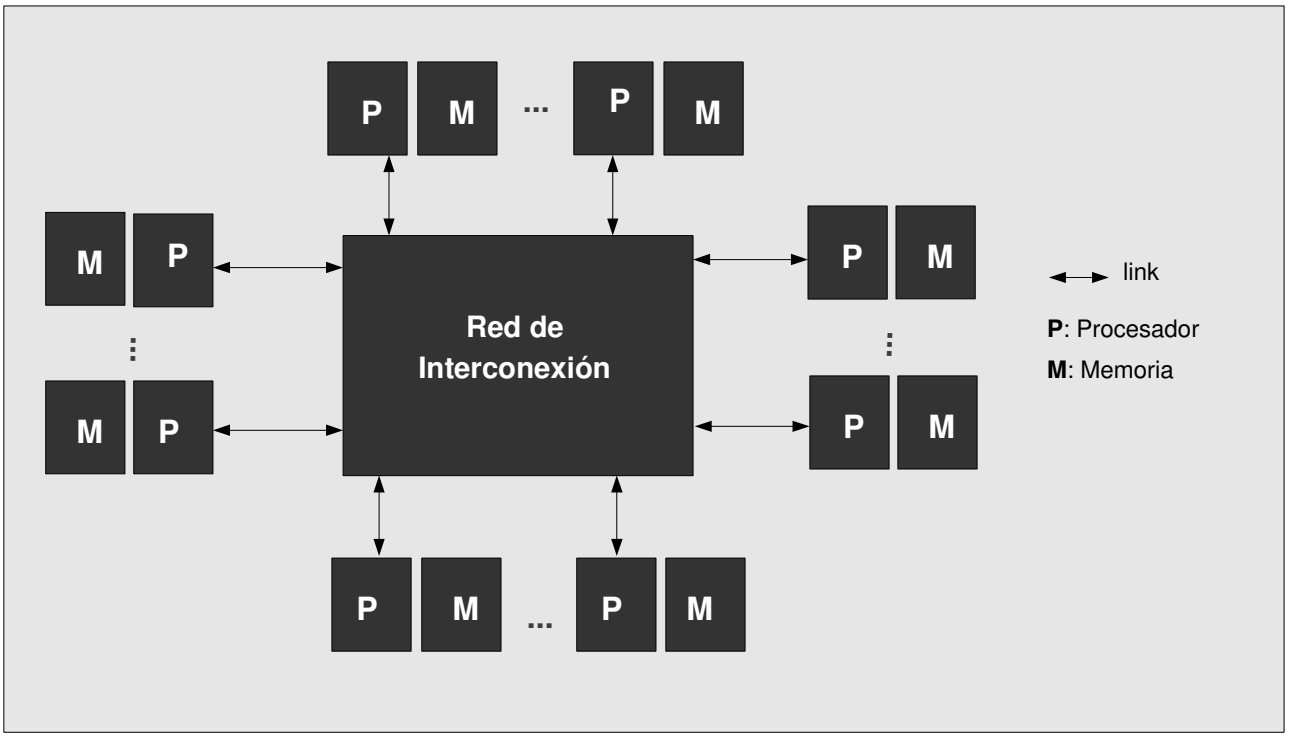


Fig. 2.5. Arquitectura MIMD con Memoria Distribuida

En la Fig. 2.5. puede apreciarse que cada **P** (Procesador) comprendería a la Unidad de Control y la Unidad de Procesamiento, en las otras arquitecturas anteriormente mencionadas. El flujo de datos y de instrucciones provienen de su propia memoria. Los links (o enlaces) junto con la red de interconexión, proveen el medio de comunicación a través del envío de mensajes entre los procesadores.

2.2. Multiprocesadores de Memoria Compartida

Los multiprocesadores de memoria compartida se caracterizan por estar conformados por procesadores y unidades de memoria conectadas por una red de interconexión, como se aprecia en la Fig. 2.6. [And00] Los procesadores comparten la memoria principal, pero cuentan con una memoria caché propia y privada.

En cuanto a la red de interconexión, si se cuenta con un número reducido de procesadores, los mismos se conectan a través de un bus de memoria o un switch crossbar. Estos multiprocesadores se los denomina UMA (Uniform-Memory-Access).

En este tipo de estructuras el espacio de direcciones de memoria es uniforme y compartido, lo que significa que el tiempo de acceso de todos los procesadores a todas

las posiciones de memoria es el mismo. Estos tipos de arquitecturas también son denominadas Multiprocesadores Simétricos (SMP, de su sigla en inglés), ya que todos los procesadores tienen igual acceso a todos lo periféricos. En el caso que solamente uno de ellos o un subconjunto de los mismos tengan control del sistema, se dice que son Multiprocesadores Asimétricos.

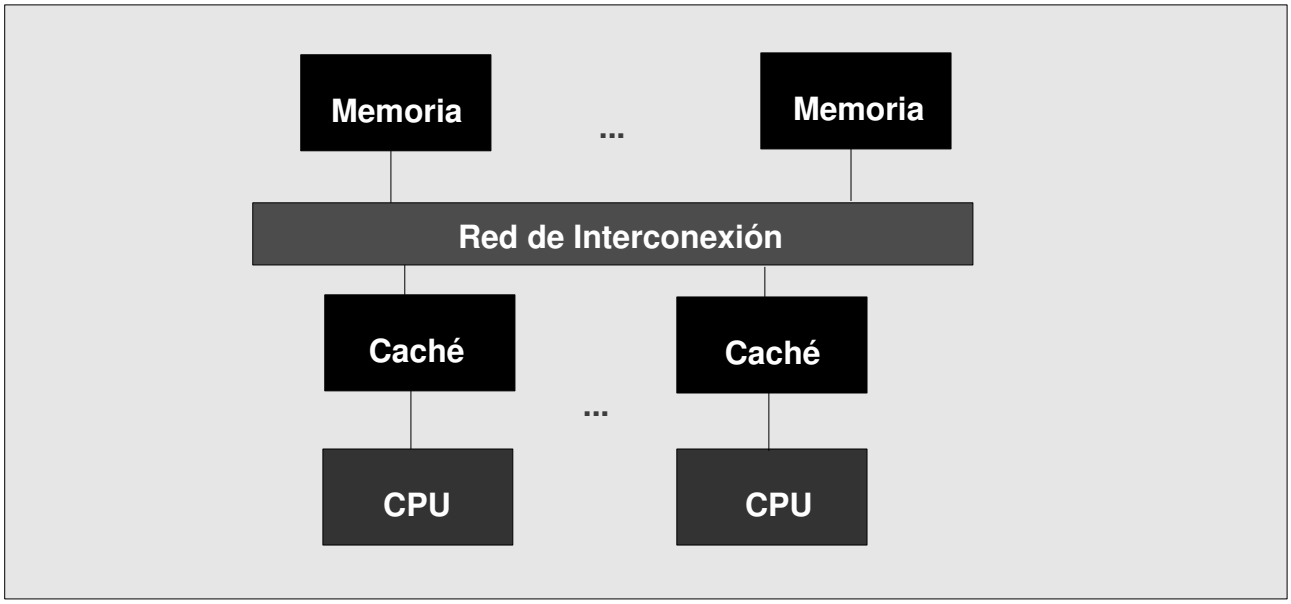


Fig. 2.6. Multiprocesador de Memoria Compartida

En las plataformas de Multiprocesadores de Memoria Compartida a gran escala, la memoria se organiza de forma jerárquica. La interconexión entre los procesadores está conformada por una colección de estructura tipo árbol de switches y memorias. Cada procesador tiene posiciones de memoria cercanas, por lo tanto de menor tiempo de acceso que aquellas posiciones más alejadas de él. Es por eso, que a estos tipos de multiprocesadores se los denomina NUMA (Nonuniform-Memory-Access), es decir, el tiempo de acceso depende de la dirección de la memoria a la que el procesador deba acceder.

2.2.1. Coherencia de caché

Tanto para la arquitectura UMA como la NUMA, cada procesador contiene su propia memoria caché. Esta característica tiene el inconveniente de la coherencia de caché. Si dos o más procesadores acceden a una posición de memoria, el contenido de la misma será almacenado en las cachés de dichos procesadores. En problema ocurre cuando los procesadores referencian la misma posición de memoria en el mismo instante de tiempo, y por ejemplo, uno de ellos modifica el contenido de la dirección de memoria

compartida, ocurre lo que se denomina coherencia (o consistencia) de caché, es decir, la caché de los procesadores que no modificaron el contenido de la posición de memoria compartida tendrá un valor que no es el correcto. Para solucionar este problema existen dos opciones actualizar las cachés con el nuevo valor, o invalidar dicha línea de caché.

Dichas soluciones pueden estar implementadas en hardware, lo que lo hace completamente transparente al software, o se puede utilizar una solución por software con un hardware específico que ayude a dicha tarea. Existen tres soluciones [DGT98]: (1) protocolos de hardware, (2) datos de escritura compartidos no asignables en caché y (3) esquemas basados en software.

Los protocolos de hardware permiten que todas las direcciones de memoria puedan ser asignadas a la memoria caché, y es el hardware del sistema que realiza el mantenimiento de la coherencia en las cachés. La complejidad de dicho hardware crece a medida que la cantidad de procesadores son agregados a la plataforma.

Tanto para la solución (2), como la (3), se requiere de la ayuda de hardware para garantizar la consistencia. Los programadores deben ser conscientes del problema de la consistencia de caché derribando totalmente la abstracción que el mismo tiene de la arquitectura. La solución (2) define que ciertos datos no puedan ser asignados a las memorias cachés y que deban permanecer todo el tiempo en memoria principal. Esta definición es indicada por el programador, y es el compilador que indica qué datos son los que no pueden ser transferidos a las cachés. Claro está que el inconveniente en esta solución es la cantidad de memoria empleada que no puede ser transferida, siendo dicha cantidad totalmente dependiente del problema específico que se esté solucionando.

En el caso (3), los datos son transferidos a las memorias caché solo si no se producirán inconsistencias. Si se sabe que se producirá una inconsistencia se escriben los datos en la memoria principal. La ejecución se encuentra dividida en dos etapas, la primera en la cual se transfieren los datos a las cachés, y la segunda cuando los datos modificados son escritos en memoria principal.

Es importante agregar, que la transferencia de datos hacia las memorias caché se realizan por conjunto de palabras. Por lo cual, se presenta un nuevo problema, lo que se denomina *false sharing*. Este problema ocurre cuando dos o más procesadores comparten una línea de caché, y por ejemplo uno de ellos modifica un dato de dicha línea, que no era utilizado por los otros procesadores, sin embargo, toda la línea de caché será invalidada. Las invalidaciones y actualizaciones de las líneas de caché hacen que todo el sistema se ejecute más lento, por lo cual, en los casos como los anteriormente mencionado, para evitar los *false sharing*, se debería tratar de que los datos se encuentre almacenados en direcciones de memoria separadas.

2.2.2. Coherencia de memoria

Si bien en las arquitecturas con memoria global compartida la programación suele ser más fácil [GGKK03] y el problema de la consistencia de memoria caché se encuentra en la mayoría de los casos implementada por hardware, el programador debe lidiar con el problema de la consistencia de memoria [And00], es decir, no saber cúando será actualizada la memoria principal. Algunos modelos de consistencia de memoria son: (a) sequential consistency, (b) processor consistency , (c) release consistency.

El caso de la sequential consistency, la memoria se actualiza en orden secuencial, y cada procesador debe ver el mismo orden. Para (b), las actualizaciones a la memoria se realizarán en el orden que los procesadores realicen sus escrituras a la memoria, el orden de dichas escrituras pueden ser vistas en diferente orden por los demás procesadores. En la realease consistency, la memoria es actualizada a partir de puntos de sincronización especificados por el programador.

El inconveniente en la consistencia de la memoria se halla en que el programador espera que la consistencia sea secuencial, es decir, que si en el programa desarrollado se realiza la asignación de un valor a una variable, se espera que dicha asignación sea vista por los demás procesos en el programa.

La actualización secuencial suele ser de las más costosas, ya que por cada escritura el hardware debe invalidar o actualizar las cachés y modificar la memoria principal.

El sistema de memoria en forma jerárquica, busca reducir los tiempo de acceso a memoria principal. Sin embargo, si el programador no conoce el sistema de memoria sobre el cual se ejecutará su aplicación no podrá aprovechar al máximos el performance de la arquitectura.

2.2.3. Multicore

Desde la demostración teórica de Alan Turing en 1936, que era posible diseñar una máquina secuencial de propósito general que ejecutara de manera eficiente cierto conjunto de problemas computables, hasta más tarde convertirse en realidad por parte de Von Neumman, la tecnología ha evolucionado de formas nunca antes imaginadas.

"La idea detrás de las computadoras digitales puede explicarse diciendo que estas máquinas están destinadas a llevar a cabo cualquier operación que pueda ser realizada por un equipo humano."

Alan Mathison Turing

La célebre máquina secuencial de Von Neumman (Fig. 2.7.), ha sido la base para todas las computadoras secuencial conocidas. El avance en la tecnología del silicio, permitió la miniaturización de los transistores en los procesadores, consiguiendo que una mayor cantidad de estos puedan ser colocados, incrementando de esta manera la velocidad de los microprocesadores, lo cuales, podían ejecutar cada vez más instrucciones por segundo, incrementándose dicha velocidad un 15% cada año.

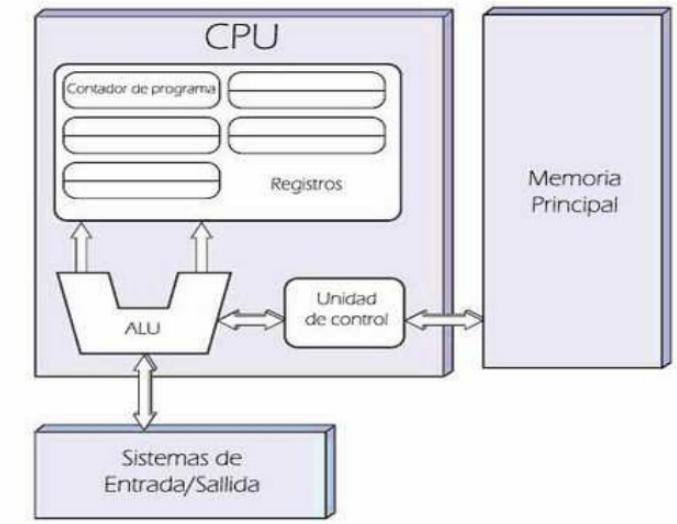


Fig. 2.7. Máquina de Von Neumman

En 1965 Gordon Moore [Moo65] [Int05], predijo que el número de transistores que podían ser colocados en un procesador se duplicaría cada año. Dicha predicción se llegó a conocer como “Ley de Moore”. En los últimos cuarenta años la cantidad de transistores ha aumentado de cincuenta (en 1965) a cientos de millones (en 2005).

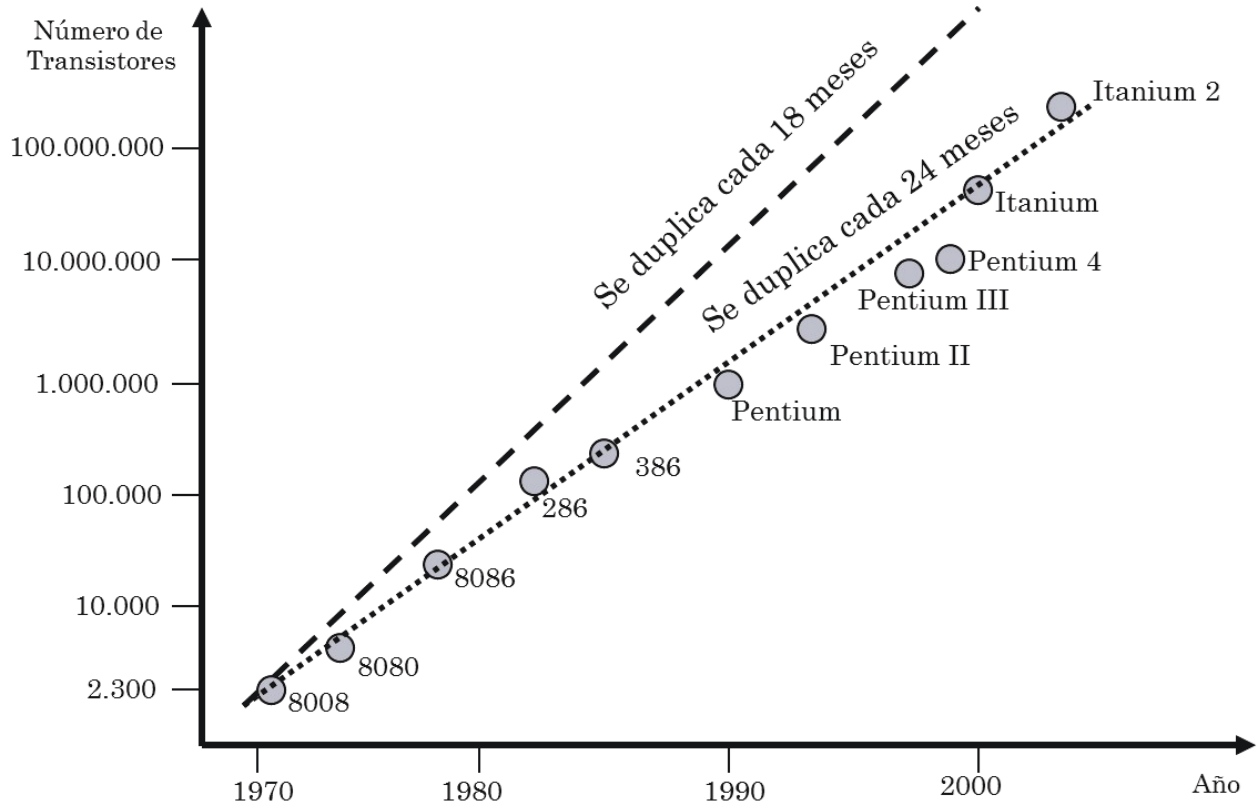


Fig. 2.8. Evolución de los transistores por chip siguiendo la ley de Moore

En la actualidad, la tecnología ha seguido poniendo a disposición de los arquitectos, un número mayor de transistores, sin embargo, el paradigma de cómputo que se había venido utilizando ya no pudo transformar este recurso tecnológico en el rendimiento con la misma tendencia seguida desde 1965. Este fenómeno es también conocido como “Moore’s Gap”, ya que hay un vacío en cuanto al rendimiento esperado como resultado del avance tecnológico. Los modelos de programación siguen siendo fundamentalmente secuenciales, lo que hace cada vez más difícil encontrar paralelismo en las aplicaciones que se ejecutan en los nuevos procesadores. Ciertas técnicas en el diseño de los procesadores se han empleado para mantener el modelo de programación secuencial, técnicas que aumentaban la complejidad de los procesadores pero que los hacían energéticamente ineficientes.

Para obtener mayor eficiencia en estos procesadores secuenciales, se incorporaron instrucciones que aprovechaban el paralelismo de datos, cuando la misma operación se debía ejecutar sobre diferentes datos (Single Instruction Multiple Data). Estas instrucciones eran específicas en cada arquitectura. En el caso de Intel, se empezó con la extensión de instrucciones MMX para, posteriormente, introducir la extensión SSE. Estas extensiones permitirían aplicar, mediante una sola instrucción, el mismo cálculo a

varios elementos en punto flotante consecutivos en memoria. Las contrapartidas del uso de este tipo de instrucciones incluyen la perdida de portabilidad del código y el aumento de la complejidad del mismo.

Con el objetivo de extraer el máximo paralelismo a nivel de instrucción, los diseñadores implementaron, dentro del procesador, sistemas de control muy complejos (segmentación, renombrado, ejecución fuera de orden, predicción de saltos, pre-búsquedas de instrucciones).

La reducción progresiva del tamaño del transistor y el aumento implícito de la frecuencia de trabajo proporcionó a la industria del software el "free-food". Es decir, un mismo software funcionaría más rápido cuanto más rápido fuera el hardware sobre el que se ejecutaba debido a que el rendimiento del cálculo computacional era proporcional a la frecuencia de trabajo. No obstante, este "free-food" no podía ser eterno. El rendimiento obtenido estaba muy por debajo de lo esperado. Esta complejidad en cuanto a la arquitectura de los procesadores se enfrentó a un nuevo problema: la disipación de calor.

Los niveles de calor alcanzados por los procesadores eran tan altos que resulta inviable (desde el punto de vista económico y funcional) seguir manteniendo el mismo paradigma computacional. Es por este motivo que los fabricantes de procesadores empezaron a optar, a finales del siglo XX, por sistemas que aumentaran el rendimiento de cálculo sin necesidad de aumentar la frecuencia de trabajo. De esta forma, la industria, prácticamente en su totalidad, decidió que su futuro estaba en el cómputo paralelo. La industria vio que su única opción viable era reemplazar su modelo de uni-procesador complejo e ineficiente, por un modelo de multiprocesador sencillo y eficiente. Esta estrategia dio entrada a lo que hoy conocemos como procesadores multinúcleo. Por un lado, se resolvía el problema de la complejidad del uniprocesador y por el otro, se utilizaría el mayor número de transistores para incrementar el número de procesadores o núcleos cada 18 meses, mientras se siguiera sosteniendo lo previsto por Gordon Moore.

En una arquitectura multinúcleo, cada procesador contiene múltiples procesadores (o núcleos), donde cada uno de éstos cuenta con su unidad de proceso independiente. El estilo de computación para este tipo de procesadores es MIMD (Multiple Instruction Multiple Data), es decir, que se tiene la capacidad de ejecutar más de una instrucción de manera concurrente y que, cada una de estas instrucciones, será ejecutada utilizando múltiples secuencias de datos, llevándonos a esquemas de paralelismo de procesos o hilos.

Esta estrategia también impactó a los sistemas de Supercómputo (o supercomputadoras) considerados como las computadoras con mayor capacidad de cómputo y diseñadas para atender necesidades computacionales complejas, que requieren tiempos de cómputo muy grandes. A diferencia de las Supercomputadoras

anteriores, este tipo de sistemas se integra con miles de procesadores económicos conectados en paralelo.

El rápido crecimiento del los sistemas multinúcleos y los diversos enfoques que estos han tomado, permiten que procesos complejos que antes solo eran posibles de ejecutar en supercomputadoras, hoy puedan ser ejecutados en soluciones de bajo costo también denominadas “hardware de comodidad”. Dichas soluciones pueden ser implementadas usando los procesadores de mayor demanda en el mercado de consumo masivo (Intel y AMD).

Esta tendencia hacia sistemas multicore requiere que dentro del desarrollo de aplicaciones se tomen en cuenta nuevos elementos en el momento de medir los niveles de desempeño ofrecidos, especialmente aquellos que afectan de manera directa el trabajo de procesos en paralelo, la distribución de la carga y la sincronización de datos entre los diferentes cores. Estos nuevos desafíos deben ser tenidos en cuenta desde el momento del análisis y diseño, evitando que los cuellos de botella impacten de manera negativa las soluciones productivas. En la evaluación de los sistemas multicore se debe prestar especial atención al tamaño y la jerarquía de los sistemas de cachés, a los algoritmos de coherencia entre ellos, la arquitectura de comunicación entre procesadores, memorias y elementos de entrada salida.

Anteriormente, la jerarquía de memoria estaba diseñada para servir peticiones a un solo núcleo, mientras que en este nuevo modelo la memoria puede servir peticiones a más de un núcleo, por lo que es necesaria una jerarquía más compleja en la que, por ejemplo, pueda haber una memoria cache para cada uno de los núcleos y otra compartida entre ellos. En las arquitecturas Intel y AMD los sistemas multicore mantienen esquemas de cache privados para los niveles L1 y L2 y dadas sus prestaciones de velocidad de acceso y cercanía con los registros del procesador pueden hacer la diferencia para lograr un rendimiento superior. Por ejemplo, en el caso de matrices de gran tamaño, se deben diseñar estrategias de subdivisión de las mismas o particionamiento tendientes a lograr que estas al ser operadas puedan entrar dentro de los tamaños de caché específicos, evitando acceder a información que está localizada en un caché compartida (como es el caso del L3) o que se encuentre dentro de una caché de un core distinto.

2.3. Multiprocesadores de Memoria Distribuida

Los Multiprocesadores de memoria distribuida (Fig. 2.9.) se encuentran conectados por una red de interconexión, al igual que los de memoria compartida con la diferencia que cada uno tiene su propia memoria principal privada. La comunicación entre los

mismos se realiza a través del pasaje de mensajes.

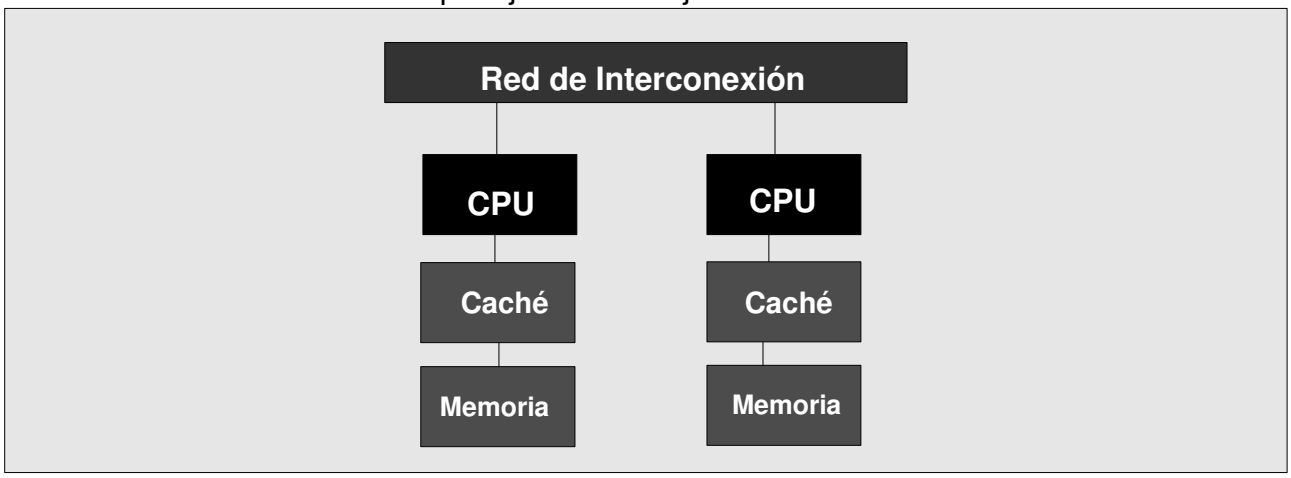


Fig. 2.9. Plataforma de Memoria Distribuida

En el caso de este tipo de plataformas, no existe la problemática de la consistencia de caché porque cada procesador tiene su propia memoria privada no compartida con los demás. Si alguno de los otros procesadores requiere escribir o leer una posición de memoria de otro procesador, debe enviar un mensaje al mismo y esperar la respuesta.

El sistema de red está conformado por una conexión física local tal como: Ethernet, por ejemplo. Dicho sistema de interconexión es utilizado por los nodos de la arquitectura para intercambiar datos, trabajo y realizar acciones de sincronización.

2.3.1. Costo de la comunicación entre nodos

Como se mencionó en el Capítulo 1, la comunicación entre los nodos de una arquitectura paralela es una de las fuentes de overhead en los algoritmos desarrollados para estas arquitecturas. El costo de comunicación depende de entre otras cosas de [GGKK03]: la semántica del modelo de programación, la topología de la red, el manejo y ruteo de los datos y los protocolos de software asociados.

El tiempo que toma enviar un mensaje entre dos nodos en una red es la suma del tiempo de preparar el mensaje y el tiempo que el mensaje tarda en atravesar la red hasta su destino. La latencia en el envío de los mensajes está dada por los siguientes parámetros:

startup time (t_s): es el tiempo requerido para preparar el mensaje que es enviado o recibido. Ocurre solo una vez por cada transferencia de un mensaje.

Per-hop time (t_h): tiempo que tarda en llegar el header de un mensaje de un nodo al siguiente. Se lo conoce también como latencia del nodo.

Per-word transfer time (t_w): tiempo para comunicar una palabra del mensaje entre dos nodos vecinos. Siendo *r* el ancho de banda de la red, por lo tanto se tiene que $t_w = 1/r$.

Si el mensaje tiene una longitud de *m* palabras y atraviesa *l* links, se tiene que el costo de comunicación está dado por la siguiente ecuación:

$$t_{com} = t_s + l\, t_h + t_w\, m \qquad (2.1)$$

Para optimizar el envío de mensajes deberíamos seguir las siguientes estrategias:

Utilización de paquetes: en lugar de enviar un mensaje grande, se envían paquetes en lugar del mensaje, reduciendo así el startup time.

Minimizar el volumen de los datos: reducir el tamaño de las palabras a enviar minimizaría el tiempo t_w.

Minimizar las distancias entre los datos a transferir: se consigue minimizando el número de hop o saltos que debe atravesar el mensaje.

Mientras, que los dos primeros objetivos son fáciles de llevar a cabo, no sucede lo mismo con la disminución de la distancia entre los nodos que se comunican. Ciertas características de las plataformas y paradigmas paralelos determinan dicho inconveniente:

- Muchas librerías de pasaje de mensaje solo le permiten al programador tener un pequeño control sobre el mapeo de los procesos a los procesadores físicos.
- Varias arquitecturas utilizan enrutamiento aleatorio, los cuales envían el mensaje a un nodo al azar desde el nodo origen y desde este nodo intermedio hasta el destino final.

El tiempo de salto (t_h) está fuertemente influenciado por la latencia de inicio (t_s) en mensajes pequeños o por la componente por palabra ($t_w m$) en mensajes grandes. Dado que el tiempo por salto (*l*) es relativamente despreciable en la mayoría de las redes, el mismo puede ser ignorado teniendo así una pérdida de precisión pequeña.

Los puntos mencionados anteriormente, nos permiten definir un modelo de costo de comunicación más simplificado, por lo tanto la ecuación 2.1 queda simplificada como la 2.2:

$$t_{com} = t_s + t_w\, m \qquad (2.2)$$

El presente modelo más simplificado influye directamente en el diseño de

algoritmos independiente de la arquitectura y en la precisión de las predicciones de la ejecución. La cantidad de tiempo que se necesita para comunicar un mensaje entre dos nodos es la misma para cualquiera de ellos, caracterizando de esta manera a una red completamente conectada. El beneficio del modelo se encuentra en que no es necesario diseñar algoritmos para cada una de las distintas topologías de red (malla, hipercubo, árbol, etc.), ya que puede ser transportado a cualquier arquitectura paralela.

La pérdida de precisión que ocurre en la predicción cuando el algoritmo es transportado desde el modelo simplificado a la arquitectura real es mínima, siempre que t_h se encuentre directamente dominado por t_s o t_w. A pesar de sus ventajas, el modelo de costo simplificado solo es válido para redes no congestionadas. Y dicha congestión depende de la tipología empleada, igual que el patrón de comunicación entre nodos. Por lo tanto, en resumen: el modelo de costo simplificado es válido mientras el patrón de comunicación no congestione la red.

2.3.2. Cluster y Multicluster

2.3.2.1. Cluster

Un ***Cluster*** es la combinación de equipos independientes que conforman un sistema unificado. Cuando utilizamos dos o más computadoras para resolver un problema estamos usando un cluster [Thi05].

La definición anteriormente presentada marca una característica importante de los cluster: cada nodo dentro del sistema es totalmente independiente, esto significa que está formado por hardware y software propio. A partir de ello, podemos deducir que dicho cluster pueden estar conformados por máquinas iguales o diferentes, lo cual, conformaría un cluster homogéneo o heterogéneo respectivamente.

El cluster a su vez puede estar conformado por multiprocesadores de memoria compartida. Conformando así lo que se denomina *clusters de multicore.* En cuanto a la red de interconexión, los nodos dentro de un cluster se encuentran conectados por una red LAN.

Dependiendo de su finalidad se los puede clasificar en:

Cluster de Alta Disponibilidad (High-Availability Cluster): tienen el objetivo de proveer seguridad contra fallas a determinados servicios. Operan a través de redundancia de nodos. Por lo cual, si uno de ellos falla, otro nodo puede sustituirlo sin perjudicar el servicio ofrecido por el HAC. Es muy utilizado en ambientes donde la disponibilidad es

crucial.

Balanciamiento de carga: en este tipo de clusters varias computadoras están unidas para balancear la carga de trabajo que comparten.

Cluster de Alto Desempeño (High Performance Cluster): estos cluster son utilizados para trabajos computacionales muy pesados como cálculos matemáticos complejos, renderizado de imágenes 3D, simulaciones y otros problemas diversos que exigen gran poder de procesamiento.

2.3.2.2. Multicluster

Un ***multicluster*** es el resultado de interconectar varios clusters. En este tipo de arquitecturas es imprescindible tener un scheduler que planifique y asigne las tareas de la aplicación paralela en los cluster que conforman el sistema. Las tareas asignadas a los distintos cluster son vistas como trabajos normales por los schedulers de los clusters [YLAKK++09].

2.4. Arquitecturas combinadas

2.4.1. Clusters de Multicore

En la actualidad, los procesadores multicore se han convertido en standard en el mercado de las computadoras. Esta realidad a impacto en los clusters, conformando así cluster de multicore [TW09]. Esto implica no solo un cambio en los sitemas utilizados para la ejecución de aplicaciones paralelas, sino también en el modelo de programación. Ahora, si se quiere explotar al máximo la potencia del hardware subyacente, la manera de programar deberá ser una combinación entre pasaje de mensajes (para la comunicación entre los nodos del cluster) y memoria compartida (comunicación entre los cores del procesador de cada nodo del cluster). Dicho modelo de programación se lo denomina: Híbrido (será tratado con mayor detalle en el Capítulo 4).

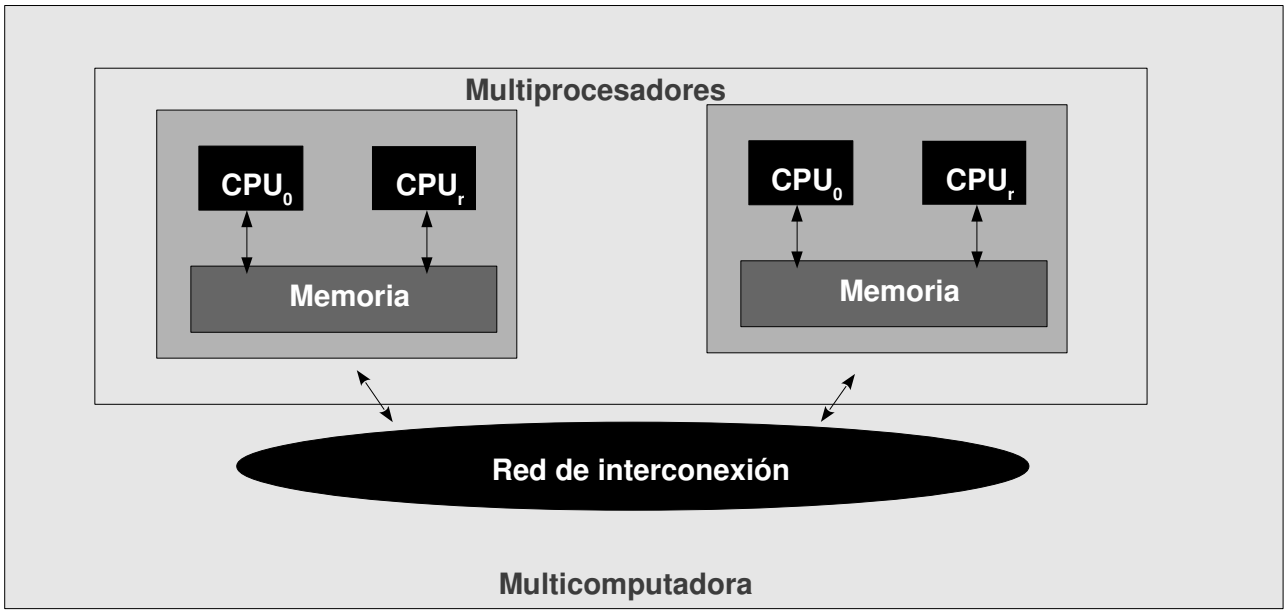


Fig. 2.10. Cluster de multicore

Otra característica importante de este tipo de plataforma es el cambio en la jerarquía de memoria. El uso de este tipo de plataformas agregan al sistema recursos críticos que deben ser cuidadosamente utilizados, tales como la caché L2 y los canales de transferencia de datos. Por lo general, los cores de un mismo procesador suelen compartir la cache L2 y L3. Los cores que pertenecen a distintos procesadores pero se encuentran en un mismo nodo comparten la memoria principal. En cambio, si los cores se encuentran en distintos nodos, no existe ningún recurso de memoria que compartan.

A la hora de mapear las tareas a lo nodos del cluster podría alocarse en un mismo nodo aquellas tareas que tienen una comunicación frecuente entre ellas. Sin embargo, dependiendo de la cantidad de cómputo y los datos a procesar, colocar múltiples tareas en un mismo nodo puede ocasionar un cuello de botella debido a los recursos compartidos entre los cores del procesador. El mapeo de las tareas en este tipo de arquitectura dependerá fuertemente de la aplicación paralela que se quiera ejecutar.

Se han propuesto muchos modelos de programación para la resolución de problemas en este tipo de arquitecturas, pero mucho de ellos no tienen en cuenta la jerarquía de memoria. El algoritmo paralelo desarrollado para esta plataforma tiene que tener en cuenta los recursos que se comparten para aprovechar al máximo el potencial de esta arquitectura [SDB10].

2.5. Arquitectura GPU-CUDA

Las GPUs (Unidades de Procesamiento Gráfico) han sido un coprocesador cuyo fin específico era el renderizado de gráficos, es decir, se encargaba de la transformaciones de escenas en 3D a 2D, mientras que la CPU podía ser usada para la ejecución de otras tareas [Nvi03] [Pic11]. Sin embargo, como se comentó en el Capítulo 1, a partir del 2003 el gran poder de cómputo de las GPUs comenzó a ser usado para la ejecución de aplicaciones de propósito general, naciendo así GPGPU.

El uso de las GPUs para el cómputo de altas prestaciones como supercomputadoras, genera nuevas oportunidades para que muchas aplicaciones puedan explotar el poder del cómputo paralelo masivo de las GPUs.

> *"Las GPU han evolucionado al punto que muchas aplicaciones del mundo real se están implementando fácilmente en ellas y se ejecutan muchísimo más rápido que en sistemas con múltiples núcleos. Las arquitecturas de computación del futuro serán sistemas híbridos con GPU de núcleos paralelos trabajando en tándem con CPU de múltiples núcleos".*
> Jack Dongarra, Profesor, Universidad de Tennessee

Por las ventajas que esta arquitectura supone (las cuales se presentarán a lo largo de la presente Tesina de Grado) el futuro del cómputo paralelo parece estar asociado a esta nueva arquitectura heterogénea basada en multicore y GPUs, permitiendo que tanto las aplicaciones secuenciales como las paralelas se vean beneficiadas.

2.5.1. Ejecución en CUDA

Las GPUs fueron diseñadas desde sus orígenes como arquitecturas paralelas con el fin de realizar el procesamiento de cada píxel de una escena en 3D siendo cada uno de ellos procesado independientemente uno de los otros. Una GPU moderna podría ser vista como un array de procesadores ejecutando en paralelo [Che09].

La arquitectura de una GPU-Nvidia CUDA [KH10] típica como muestra la Fig. 2.11.,

está organizada en Streams Multiprocessors (SMs), los cuales tienen un determinado número de Streams Processors (SPs). Los SPs comparten la lógica de control y la caché de instrucciones.

Nvidia [ND10] varia la cantidad de SMs en cada nueva generación de GPUs logrando que las mismas sean escalables en performance, además de variar el número de DRAMs para lograr escalar el ancho de banda de memoria y la capacidad.

Cada SM provee los suficientes threads, cores y memoria shared para ejecutar uno o más bloques de threads CUDA. Los que realizan verdaderamente la ejecución son los SPs, que ejecutan múltiples threads concurrentemente.

Los threads se organizan en bloques de hilos, y existen uno o más bloques ejecutándose concurrentemente en SM. La organización de los threads puede verse por nivel o jerárquicamente (Fig. 2.12) de la siguiente manera:

- *Grid*: nivel superior, está conformado por bloques de threads.
- *Bloque*: nivel medio, está conformado por threads.
- *Threads*: nivel inferior, son los que realizan el trabajo.

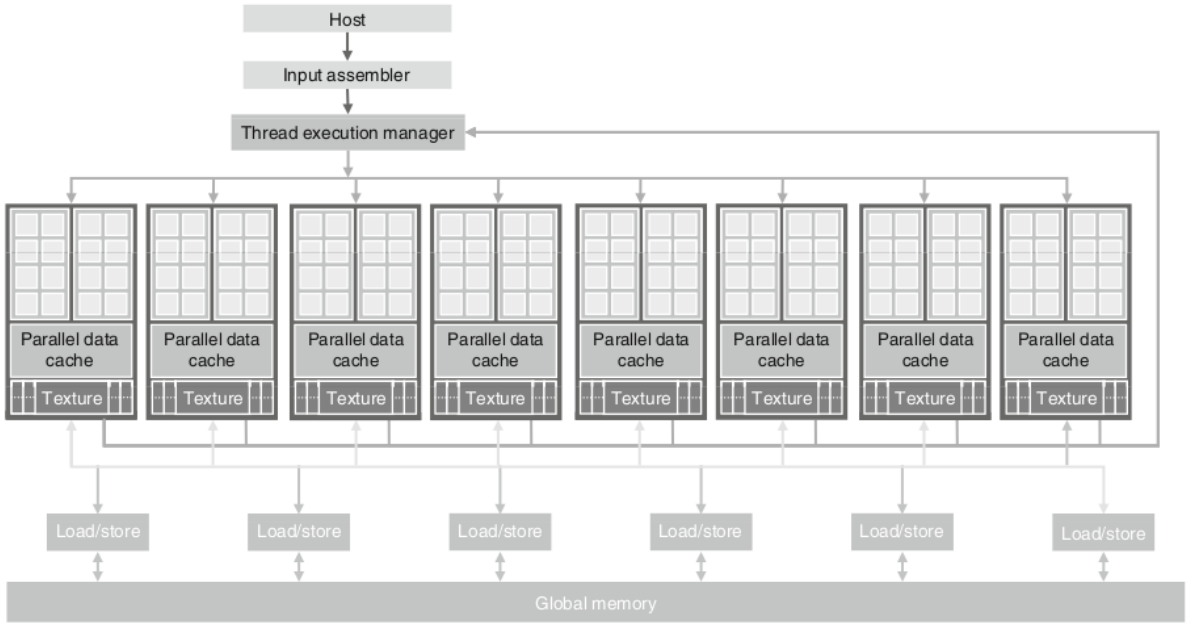


Fig. 2.11. Arquitectura de una GPU CUDA (tomado de [KH10])

Los bloques de hilos en un grid pueden ser de una dimensión, de dos dimensiones

o de tres dimensiones. A su vez, cada bloque está dividido en threads de una dimensión o dos dimensiones. En un bloque de threads, los hilos están organizados en warps, los cuales, por lo general, agrupan 32 hilos. Todos los hilos de un warp son planificados juntos durante la ejecución, y de manera independiente [Lue08].

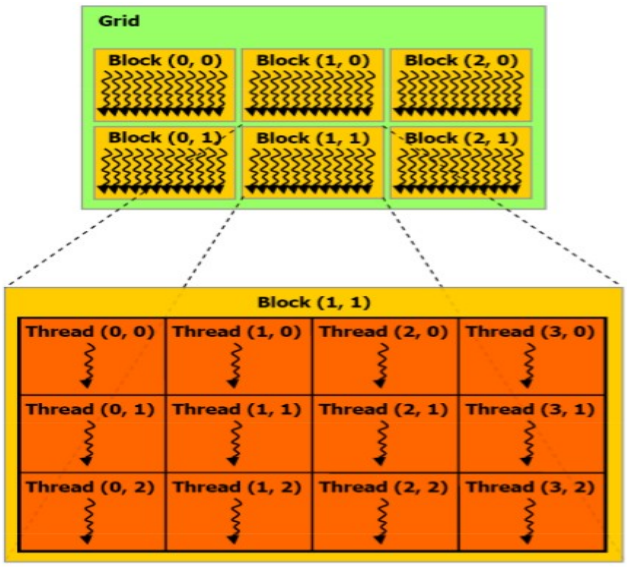


Fig. 2.12. Grid de Bloques de threads (tomado de [Nvi11])

Los hilos de un bloque pueden comunicarse con otro hilo a través de la memoria compartida. Deben ser ejecutados en el mismo multiprocesador. De esta forma, los hilos pueden sincronizarse entre ellos.

Cada thread tiene un identificador que le permite diferenciarse de los demás, direccionar memoria y tomar decisiones de control.

La cantidad de bloques de hilos por multiprocesador depende de la memoria compartida y los registros alocados para cada bloque. Por lo cual, cuanta más memoria compartida y más registros son asignados por cada bloque, menor es la cantidad de bloques de hilos por multiprocesador.

La creación de los threads, así como su ejecución y terminación, es automática y manejada por la GPU. El usuario solo debe especificar el número de hilos por bloque y el número de bloques por grid.

2.5.2. Sistema de memoria

El sistema de memoria de la GPU [Nvi11][Nvi12][Pic11][PP10], es totalmente

diferente al de la CPU. Su jerarquía de memoria está compuesta por distintos tipos de memorias que se diferencian tanto en los tipos de accesos de los hilos a estas como las limitaciones en las operaciones permitidas en ellas, al igual que su ubicación dentro del dispositivo.

La importancia de conocer la jerarquía de memoria, tanto en la GPU como en la CPU, radica en utilizarla lo más eficientemente posible para no degradar el rendimiento de la aplicación. En particular, el sistema de memoria de la GPU está compuesto por: una memoria global, una memoria de textura, una memoria constante, una memoria compartida, una memoria local a cada thread y registros.

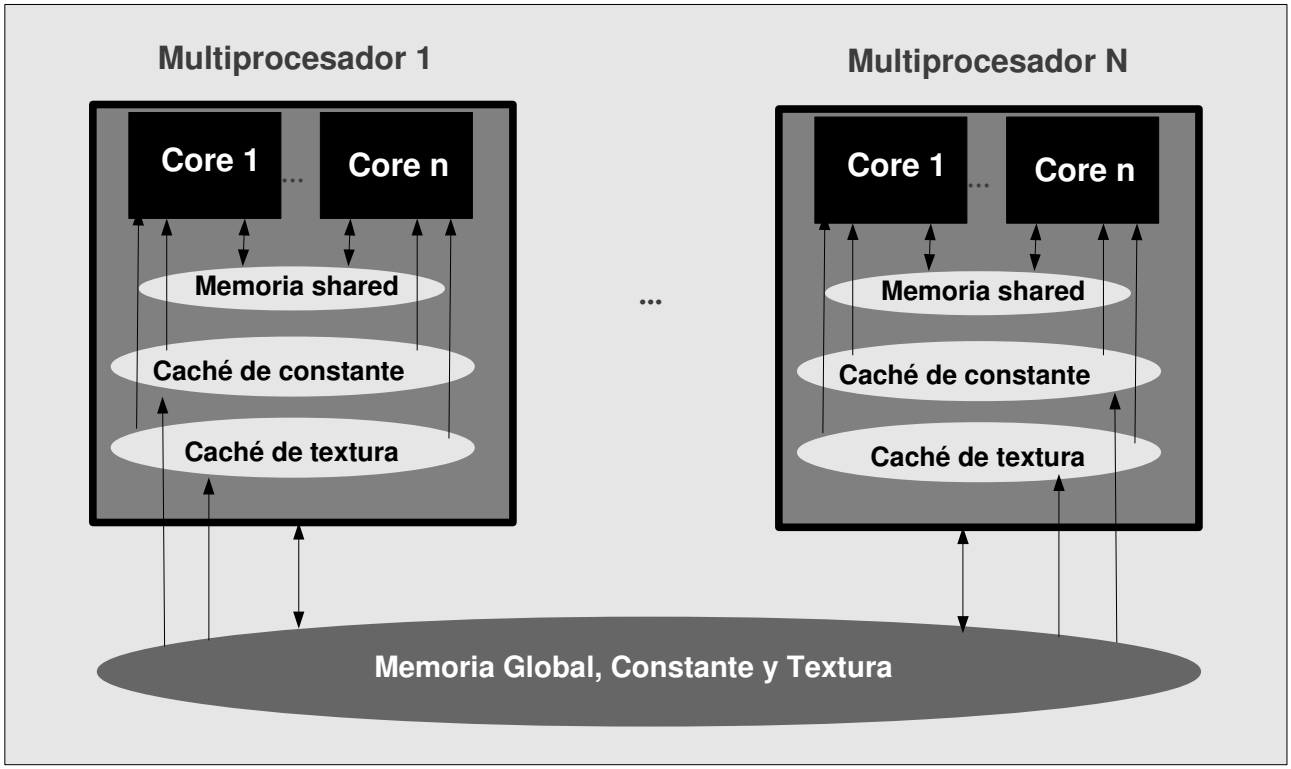


Fig. 2.13. Modelo de memoria en CUDA

2.5.2.1 Memoria Global

La memoria global es la más abundante y la de mayor latencia. Todas las GPUs tienen 4 gigabytes o más de memoria DRAM (GDDR). Es la responsable de la mayor contribución en cuanto a rendimiento de la programación CUDA, que es la coalescencia de memoria. Debido a la arquitectura SIMT (Single Instruction, Multiple Threads), los accesos a la memoria global realizados por un half warp de hilos para Capacidad de

Cómputo[2] 1.X (de ahora en más C.C.) o un warp para C.C. 2.0, se realizan a la vez, mejorando el ancho de banda de memoria en un factor igual al número de hilos del warp o half warp (16 para C.C. 1.X, 32 para C.C. 2.0). Los requisitos para que el acceso a la memoria sea coalescente se detallan a continuación:

- En los dispositivos con C.C. 1.0 o 1.1 se debe cumplir:
  - El tamaño de la palabra a la que accede cada hilo tiene que ser de 4, 8 y 16 B.
  - Para palabras de los siguientes tamaños:
    - 4 B, las 16 palabras deben estar en el mismo segmento de 64 B.
    - 8 B, las 16 palabras deben estar en el mismo segmento de 128 B.
    - 16 B, las 8 primeras palabras deben estar en un segmento de 128 B y las 8 restantes en el segmento de 128 B siguiente.
  - El acceso debe ser secuencial: el k- ésimo hilo debe acceder a la k-ésima palabra, aunque no es necesario que participen todos los hilos.
- Los dispositivos con C.C. 1.2 o 1.3 siguen el siguiente protocolo:

  1. Encontrar el segmento de memoria que contiene la dirección requerida por el hilo activo con el identificador más bajo. El tamaño del segmento depende del tamaño de las palabras a las que se accede:
  - ✔ 32 B para palabras de 1 B.
  - ✔ 64 B para palabras de 2 B.
  - ✔ 128 B para palabras de 4, 8 y 16 B.

  2. Encontrar el resto de hilos activos cuya dirección requerida está en el mismo segmento.

  3. Reducir el tamaño de la transacción si es posible:
  - Si la transacción es de 128 B y solo una mitad es utilizada, reducirla a 64 B.
  - Si la transacción es de 64 B (originalmente o tras ser reducida) y solo una mitad es utilizada, reducirla a 32 B.

  4. Realizar la transacción y marcar los hilos servidos como inactivos.

  5. Repetir hasta que todos los hilos del half warp estén servidos.

- En los dispositivos con C.C. 2.0:
  - Existe una caché que se divide en líneas de 128 B.
  - Las peticiones de memoria se resuelven copiando los segmentos de 128 B necesarios en la caché.

2 En el Apéndice C se presentan dos tablas con las características de cada C.C.

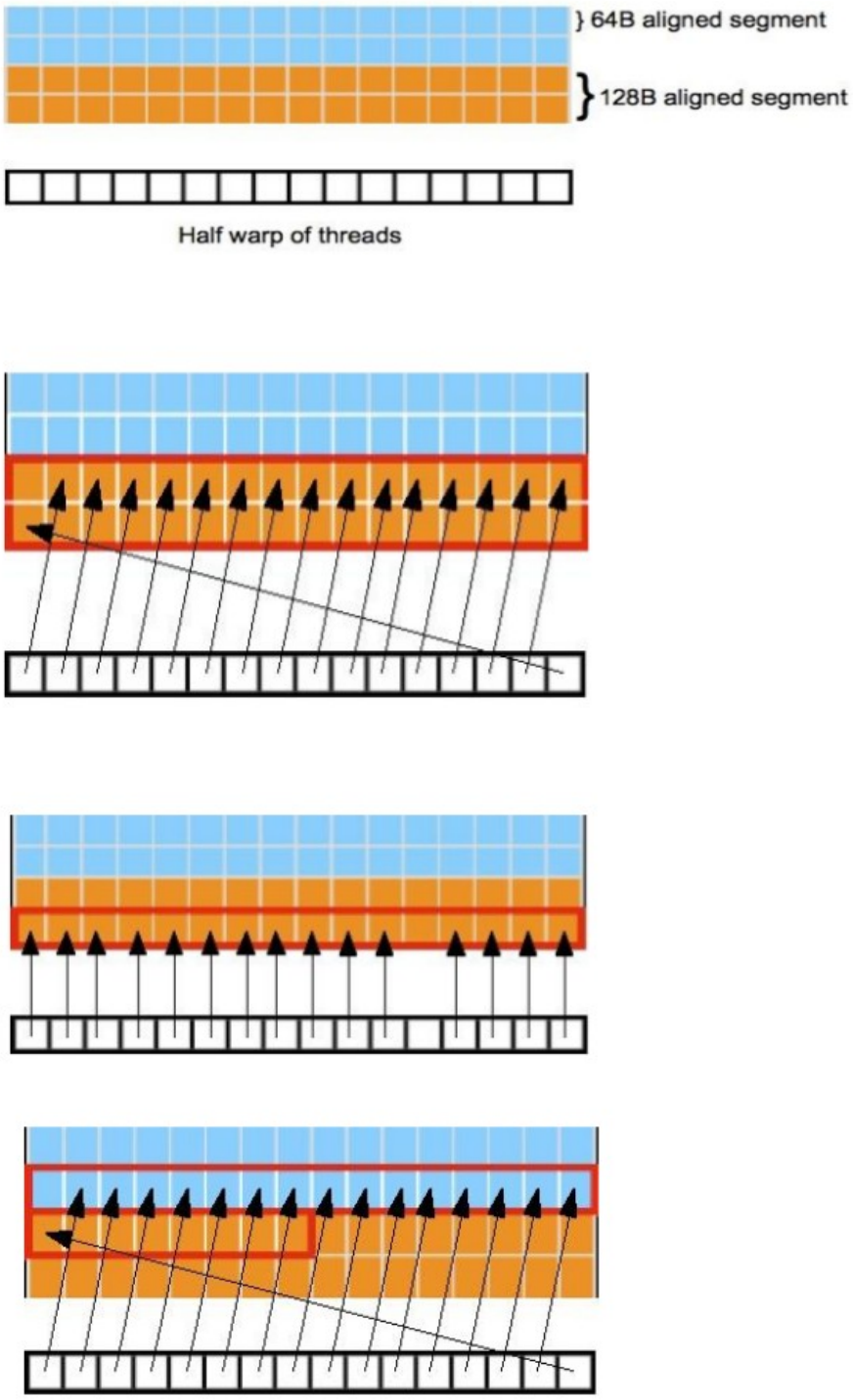


Fig. 2.14. Acceso a memoria global con medio warp (tomado de [Nvi11])

Es una memoria de lectura/escritura a la cual pueden acceder todos los thread del device. Todos los datos residen en esta al comienzo de la ejecución del kernel.

2.5.2.2. Memoria de textura

La memoria de texturas es una memoria de solo lectura con caché. Originalmente diseñada para el manejo de texturas de 1, 2 o 3 dimensiones (de ahí su nombre), como demuestran algunas de sus propiedades orientadas hacia ese campo, aunque dichas propiedades ofrecen ventajas en su utilización en otros campos. Consiste en un chip con la caché y un pequeño procesador para algunas operaciones, a través del cual los multiprocesadores acceden a la memoria global utilizando las ventajas de la memoria de texturas.

La caché de texturas está optimizada para localidad espacial bidimensional o tridimensional, así que los hilos que lean direcciones de memoria cercanas entre sí conseguirán el máximo rendimiento. También está optimizada para ofrecer lecturas con una latencia constante, con lo que la caché reduce la demanda de ancho de banda de memoria pero no la latencia de lectura. Esto puede ser aprovechado para mejorar el tiempo de lectura en algunos casos; por ejemplo, si en un caso concreto no se puede conseguir que las lecturas sean coalescentes, o si se tienen que realizar lecturas a posiciones aleatorias o pseudoaleatorias.

Las texturas se leen habitualmente utilizando el método de direccionamiento llamado de coordenadas absolutas, que son coordenadas en coma flotante en el rango [0, N), donde N es el tamaño de la dimensión correspondiente de la textura. Existe otro método de direccionamiento llamado de coordenadas normalizadas, en el cual éstas se especifican en el rango [0.0, 1.0) en lugar de [0, N). Este último método de coordenadas normalizadas es muy útil cuando se desea leer los datos de una textura independientemente de su tamaño.

Otro aspecto importante de la memoria de texturas es la gestión de los bordes. Por defecto, las lecturas a la memoria que se hacen fuera del rango de la misma se resuelven con el método clamp to edge (ajustar al borde), donde dichas lecturas se ajustarán al valor válido más cercano; por ejemplo si en una textura bidimensional de tamaño 640 × 480 se intenta acceder al valor (642, 100), la memoria de texturas devolverá el valor de la posición (639, 100), que es el valor válido más cercano. El otro método de gestión de los

bordes, el cual solo está disponible para texturas con coordenadas normalizadas, es el llamado wrap (envolver), en el cual la textura se trata como si fuera una señal periódica; por ejemplo si en una textura bidimensional normalizada se intenta acceder al valor (1.3, 0.6), la memoria de texturas devolver´ el valor de la posición (0.3, 0.6). La gestión automática de los bordes es una gran ventaja en muchos algoritmos, especialmente los relacionados con imágenes, ya que permite escribir el código de manera más general al no tener que preocuparse de escribir casos particulares cerca de los bordes, por ejemplo en un algoritmo de filtrado de una imagen.

Otra de las ventajas de la memoria de texturas es el hecho de que pueden definirse para varios tipos de datos. Aparte de los habituales como int, uchar o float, se pueden definir texturas de tipos empaquetados como float2 o uchar4, donde cada tipo tiene varias componentes (por ejemplo, float2 estará formado por dos componentes de tipo float; mientras que uchar4 estará formado por 4 componentes de tipo uchar, muy util al hacer texturas de imágenes con los 4 canales RGBA). También la memoria de texturas es capaz de normalizar los valores devueltos como valores de coma flotante tipo float en los rangos [0.0, 1.0] o [–1.0, 1.0] (para tipos sin signo y con signo respectivamente); por ejemplo se puede hacer que una textura de tipo uchar devuelva valores de tipo uchar (enteros en el rango [0, 255]) o valores tipo float en el rango [0.0, 1.0].

La última de las ventajas es el filtrado automático de las texturas. Por defecto, la memoria de texturas devuelve el valor más cercano al valor pedido; por ejemplo, si en una textura bidimensional de tamaño 640 × 480 se intenta acceder al valor (120.3, 100), la memoria de texturas devolverá el valor de la posición (120, 100), que es el valor más cercano . Otra posibilidad disponible cuando el valor devuelto sea de coma flotante (bien porque el tipo de textura sea float o porque se devuelva el valor normalizado) es que el valor devuelto sea una interpolación lineal de baja precisión de los valores más cercanos, también llamado filtrado lineal (o bilineal o trilineal en los casos de 2 y 3 dimensiones respectivamente); por ejemplo, si en una textura unidimensional de tamaño 100 se intenta acceder al valor 50.7, la memoria de texturas devolver´ la interpolación lineal de los valores en las posiciones 50 y 51 para el valor leído.

Hay dos modos de leer una estructura a través de la memoria de texturas. Uno de ellos es asociar la región de memoria lineal que ocupa dicha estructura a la textura, de este modo se pueden aprovechar todas las ventajas de la memoria de texturas para leer dicha región salvo la localidad espacial, pero solo funciona con texturas de 1 o 2 dimensiones. Si se quiere conservar la localidad espacial bidimensional o tridimensional es necesario utilizar el otro modo, en el que hay que copiar la región de memoria a un array que es rellenado siguiendo una Z-curva propia de NVIDIA. Este segundo modo es la única manera de obtener texturas tridimensionales.

2.5.2.3. Memoria constante

Como la memoria de texturas, la memoria constante es un tipo de memoria de solo lectura con caché, que se encuentra en un chip en la GPU común para todos los multiprocesadores y cada uno de ellos con una caché. Si todos los hilos en ejecución leen la misma dirección de memoria constante, solo se realiza una lectura tan rápida como la lectura de un registro, si el valor a leer se encuentra en caché. Si los hilos leen diferentes direcciones, los accesos se serializan.

En dispositivos C.C. 2.0 se puede obviar el uso de esta memoria gracias a la instrucción LDU (LoaD Uniform), que se usa cuando todos los hilos leen una misma dirección de memoria global intependientemente de su identificador.

2.5.2.4. Memoria compartida

La memoria compartida es un tipo de memoria que se encuentra en cada multiprocesador. Es mucho más rápida que la memoria global (aproximadamente una latencia 100 veces menor). En los dispositivos con C.C. 1.X, la memoria compartida es una especie de caché programable para todos los hilos de un bloque que se utiliza cuando es necesario compartir datos entre ellos, o para evitar lecturas de la memoria global innecesarias si cada hilo tiene que leer varias veces el mismo dato, por ejemplo [Lue08].

En los dispositivos con C.C. 2.0 la memoria compartida es la caché de nivel 1, y se puede utilizar como tal memoria compartida (caché programable) o como caché nivel 1 de manera automática. La memoria compartida está organizada en bancos de memoria (16 con C.C. 1.X, 32 con C.C. 2.0), de modo que palabras de 4 B consecutivas son asignadas en bancos sucesivos (entrelazado). Cuando varios hilos de un warp (C.C. 2.0) o half warp (C.C. 1.X) acceden a direcciones en el mismo banco de memoria, se puede producir un conflicto de banco, con lo que el acceso se secuenciará reduciendo la velocidad. No existe conflicto cuando varios hilos leen la misma dirección de memoria, ya que entonces se produce un broadcast. Tampoco se produce cuando, en un dispositivo C.C. 2.0, se accede a palabras de 8 B, ya que la arquitectura está diseñada para manejar palabras de ese tamaño.

2.5.2.5. Memoria local

La memoria local no es un tipo de memoria propiamente dicha, sino que es el uso de la memoria global por los hilos cuando éstos se han quedado sin registros disponibles. Se llama local por este motivo, porque pertenece a un hilo y ningún otro puede acceder a ella. Ya que se trata de memoria global actuando como local, está limitada por todas sus condiciones de acceso coalescente, lo que se traduce en un acceso secuencial y lento para dispositivos con C.C. 1.X (ya que accede un solo hilo) y un acceso algo más rápido para dispositivos con C.C. 2.0 (gracias a la caché).

2.5.2.6. Registros

El acceso a los registros del multiprocesador no consume ciclos de reloj, pero puede darse el caso de retardos por dependencia read-after-write y por conflictos de registros de memoria. La latencia por dependencias read-after-write es de aproximadamente 24 ciclos, pero es ocultado por la posibilidad de tener 192 hilos activos por multiprocesador (6 warps) para dispositivos con compute capability 1.x (8 cores CUDA por multiprocesador * 24 ciclos de latencia = 192 threads activos para ocultar la latencia). Para dispositivos con compute capability 2.x, que tiene 32 cores CUDA por multiprocesador, se requieren 768 threads activos para ocultar la latencia.

El compilador y el scheduler hardware de threads planificarán las instrucciones lo más óptimo posible para evitar conflictos en los bancos de memoria. Se consiguen mejores resultados cuando la cantidad de hilos por bloques es múltiplo de 64. Register pressure, ocurre cuando no existen suficientes registros para darle a una tarea. Para que no ocurra esto, por ejemplo, puede utilizarse el comando -maxrregcount=N, para controlar el número máximo de registros reservados para cada thread.

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x.

Fig. 2.15. Características sobresalientes del sistema de memoria de la GPU (tomado de [Nvi11])

2.6. CPU vs. GPU

La industria de la tecnología del semiconductor ha seguido dos caminos principales en la fabricación de multiprocesadores. Uno de ellos es la arquitectura *multicore*, los cuales comenzaron con dos cores, doblándose el número de cores con cada nueva generación de semiconductores. En el otro extremo se encuentran las arquitecturas *manycore* cuyo enfoque está basado en la ejecución de aplicaciones paralelas. Por eso, estas arquitecturas nacieron desde sus inicios con una gran cantidad de cores, los cuales se doblan en número con cada nueva generación.

Existe un gap entre ambos tipos de arquitecturas en cuanto a la velocidad de cálculo. El performance de las CPUs ha ido disminuyendo lentamente de manera significativa, mientras que las GPUs han continuado evolucionando incrementando la cantidad de cálculos de punto flotante por segundo.

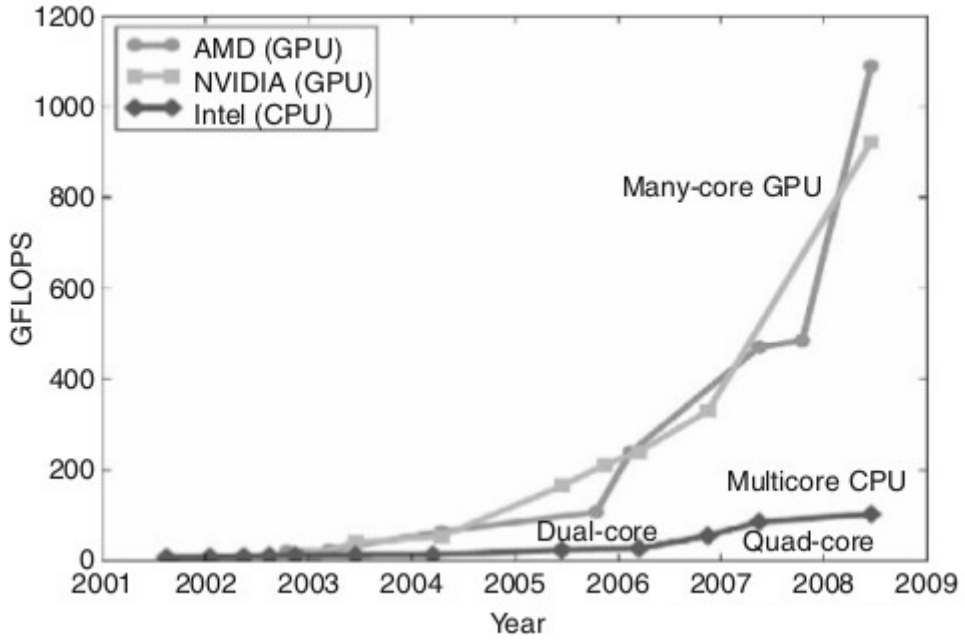


Fig. 2.11. Gap de performance entre las CPUs y las GPUs (tomado [KH10])

Este gap entre las arquitecturas se debe a la diferencias en el diseño de las mismas. La CPU está optimizada para la ejecución de algoritmos secuenciales. El diseño de la GPU fue desde el inicio pensado para realizar cálculos masivos en punto flotante. Por lo tanto, es una arquitectura paralela desde sus orígenes.

El aumento de la cantidad de transistores para aumentar el rendimiento en los procesadores dificulta el aumento proporcional de la velocidad de acceso a memoria, con lo que la brecha de rendimiento entre estos dos elementos se volvía cada vez más pronunciada. El ancho de banda de la memoria incrementa solo un 25% cada año, mientras que la latencia solo mejora un 5% por año. Al ser imposible proporcionar datos desde la memoria principal al procesador de forma más rápida, se ha optado por mitigar el problema usando una jerarquía con varias memorias caché de menor tamaño y mayor proximidad al núcleo, aprovechando la localidad de los datos. Los chips gráficos han operado en aproximadamente 10 veces el ancho de banda de los chips de CPU. El ancho de banda, en cada nueva generación de GPU ha logrado mejorar más, proporcionalmente que su antecesora.

A medida que las computadoras evolucionan, se pueden conseguir mayores prestaciones aprovechando los progresos de la tecnología. El diseño de procesadores se ha basado en la utilización de pipelines para proveer mayor velocidad en la ejecución de las instrucciones. A pesar de que se obtienen mejores prestaciones, tiene una serie de

inconvenientes tales como: el tamaño de cada etapa varia en duración de tiempo, las sentencias de bifurcación pueden ocasionar la invalidación de varias instrucciones de captación, y cuanto mayor es el número de etapas este modelo sencillo de prestaciones deja de ser útil, ya que en cada etapa del cause hay algún gasto extra de transferencia de datos, la realización de varias funciones de preparación y distribución.

La GPU también es una procesador segmentado, pero con rasgos muy particulares:

- No fluyen las instrucciones, sino los datos.
- Los operadores son unarios, lo que evita las dependencias y maximiza el paralelismo.
- Se dispone de un menor número de etapas dada la baja frecuencia.
- Se fomenta el procesamiento vectorial y superescalar, buscando aprovechar el paralelismo de datos.

Parece un cauce de segmentación más homogéneo, pero no lo es:

- Cualitativamente: Los datos no son los mismos al principio (vértices) que al final (píxeles).
- Cuantitativamente: Los datos son más numerosos conforme se aproximan al final.

La industria de los videojuegos ejerce una gran presión económica sobre los vendedores de GPU, haciendo que estos busquen aumentar sus ganancias generando arquitectura que cada vez ejecutan mayor cantidad de operaciones en punto-flotante por segundo. Consiguen un mayor rendimiento haciendo que el hardware aproveche la gran cantidad de threads que se pueden crear para solapar tiempos de espera con cómputo, es decir, mientras que algunos threads están en espera se oculta la latencia del acceso a memoria ejecutando otros threads [Par10].

La caché que se utiliza para disminuir los tiempos de acceso a memoria son pequeñas, lo cual, permite que más lugar dentro del chip sea dedicado a el cálculo de punto flotante. Para compensar la cantidad limitada de memoria caché, la jerarquía de memoria de la GPU está expuesta al programador. La latencia del acceso a la memoria global es ocultado con el uso de la memoria shared, el acceso coalescente y la ejecución la disponibilidad de varios grupos de threads.

La principal diferencia de los modelos de ejecución entre la arquitectura CPU y GPU claramente es que la última está basado en hardware multithreading. La CPU ejecuta una sola instrucción lo más rápido le sea posible, mientras que la GPU incrementa la velocidad de cómputo a través de la ejecución de grupos de instrucciones de un conjunto de threads planificados por hardware y con context switching muy eficiente teniendo disponible una gran cantidad de threads activos que superan el número de

recursos de cómputo [Par10]. Como la GPU está diseña para la resolución de cómputo numérico, es de esperar que no realice bien todas las tareas que puede hacer la CPU, por lo cual, se pretende armar una arquitectura híbrida CPU-GPU que permite ejecutar en la CPU aquellas aplicaciones para la que se encuentra optimizada y lo mismo para la GPU.

Otro punto interesante a notar, es que además del factor de la elección del procesador que mayor performance puede otorgar para la ejecución de una aplicación, también influye qué procesador es el más comercializado en un momento dado en el mercado, el costo del software a desarrollar justificado por el gran número de usuarios o posibles compradores (uno de los grandes problemas en el software paralelo). Esto ha cambiado mucho con las GPUs por la gran cantidad de ventas. Convirtiéndose así en la primera arquitectura paralela masivamente vendida. Por su costo económico hace que las GPUs sean arquitecturas paralelas muy atractivas para el desarrollo de aplicaciones con gran requerimiento de cálculo [KH10].

Capítulo 3

Diseño de Algoritmos Paralelos

"El desarrollo de algoritmos es un componente crítico para resolver problemas usando computadoras"[1]

Un algoritmo secuencial conforma una secuencia de pasos que son resueltos mediante una computadora secuencial, mientras que los algoritmos paralelos resuelven los problemas usando múltiples procesos. Por lo tanto, la definición de un algoritmo paralelo involucra más que una especificación de pasos. El diseño de algoritmos paralelos debe tener en cuenta el grado de concurrencia, la arquitectura para la que se desarrolla el algoritmo, el no determinismo de la ejecución, la comunicación y sincronización de las tareas, el particionamiento y distribución de los datos, el balance de carga, la tolerancia a fallos, la heterogeneidad, la memoria distribuida o compartida, los posibles deadlocks, race conditions.

Todas estas consideraciones son esenciales si se quiere lograr algún performance en el uso del paralelismo. Por lo tanto, la definición de algoritmos paralelos no es una tarea trivial y puede incluir algunas o todas de las siguientes dificultades [GGKK03]:

- ✗ Identificar la porción de trabajo que puede ser ejecutada concurrentemente.

1 Tomado de Capítulo 3 de [GGKK03]

- ✗ Mapear las tareas a los procesadores para ser ejecutadas en paralelo.
- ✗ Distribuir los datos de entrada, salida e intermedios asociados al programa.
- ✗ Manejar los accesos a los datos compartidos entre tareas.
- ✗ Sincronizar la ejecución de la ejecución de las tareas.

Existen varias decisiones a tomar para lograr el perfomance de los distintos pasos mencionados anteriormente. Sin embargo, el mejor performance alcanzado se obtiene para una arquitectura en particular y un paradigma de programación, siendo a su vez fuertemente dependiente de la aplicación a desarrollar.

3.1. Grafo de dependencias de tareas

Las tareas son unidades de cómputo definidas por el programador, en el cual la computación principal es subdividida en porciones de cómputo más pequeño. La ejecución simultánea de múltiples tareas pretende reducir el tiempo requerido para resolver el problema. En general, algunas tareas pueden usar datos que producen otras tareas, por lo que pueden necesitar esperar que dichas tareas terminen su ejecución. La representación de estas dependencias relativas al orden de ejecución es conocida como ***grafo de dependencias de tareas***. Este grafo es acíclico. Los nodos representan las tareas y los arcos o arístas indican la dependencias entre las mismas.

La tarea representada por un nodo podrá comenzar su ejecución solo si las tareas anteriores conectadas a ella han finalizado. Además, el grafo de dependencias de tareas puede ser desconexo y el conjunto de arcos estar vacío.

El número y el tamaño de las tareas para un problema determinan la granularidad de la descomposición. Una descomposición con un gran un número de tareas de pequeño tamaño es llamado de *grano fino* y una descomposición en una pequeño número de tareas de tamaño grande se denomina *grano grueso.*

Otro concepto relacionado a la granularidad en el grado de concurrencia. El máximo número de tareas que pueden ejecutarse simultáneamente en un programa paralelo en algún momento dado es conocida como *máximo grado de concurrencia*. En muchos casos, el máximo grado de concurrencia es menor al número total de tareas debido a la dependencias que existen entre ellas. El grado de concurrencia determina la cantidad máxima de tareas que pueden ser utilizadas.

Un indicador aún más útil del performance en un programa paralelo es el *grado promedio de concurrencia* el cual es el número promedio de tareas que pueden ejecutarse concurrentemente durante la ejecución del programa. El grado promedio de concurrencia

indica la cantidad máxima de procesadores que pueden ser utilizados.

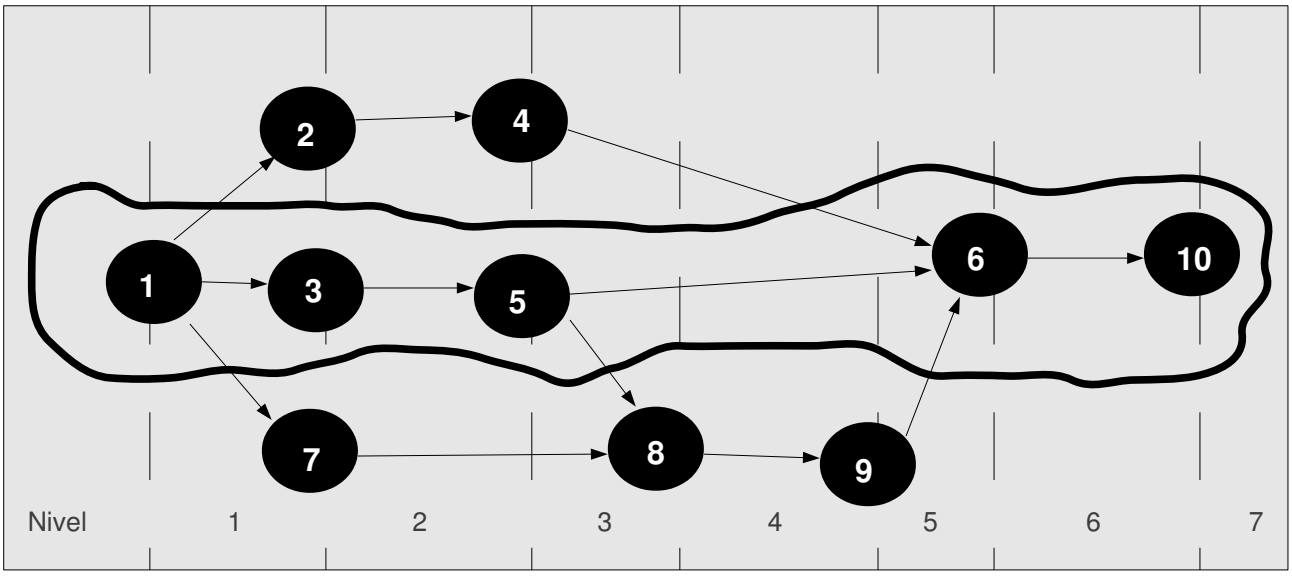


Fig. 3.1. Ejemplo de un grafo de dependencias de tareas

En la Fig. 3.1. se aprecia un ejemplo de un grafo de dependencias de tareas. El conjunto de nodos(1,3,5,6,10) conforman el *camino crítico,* que es la longitud del camino directo entre el nodo de comienzo y el nodo final. El camino crítico representa el tiempo mínimo en que el programa podrá ejecutarse en paralelo. La suma de los pesos de los nodos a lo largo del camino crítico se denomina *longitud del camino crítico*, siendo el peso del nodo el tamaño o la cantidad de trabajo asociada a dicha tarea.

3.2. Grafo de Interacción de tareas

Mientras que el grafo de dependencias de tareas representa las dependencias de control, el grafo de interacción de tareas representa dependencias de datos. Las tareas y subtareas necesitan intercambiar datos. Estas tareas y subtareas se encuentran representadas por nodos y la interacción entre las tareas por arcos. Los nodos y las aristas del grafo pueden tener asignados pesos proporcionales a la cantidad de computación ejecutada por la tarea y la cantidad de interacción que ocurre a lo largo del arco, si se conoce dicha información. Los arcos en este grafo suelen ser no dirigidos, aunque la dirección de los arcos podría ser usada para indicar hacia dónde se dirige el flujo de los datos, si se trata de un grafo de interacción de tareas unidireccional. El conjunto de arcos es un superconjunto del conjunto de aristas del grafo de dependencia de tareas.

Ambos grafos por separado no tienen utilidad. Se los utiliza combinados para obtener un mejor tiempo aproximado de la ejecución del algoritmo paralelo [GGKK03].

3.3. Etapas del diseño de algoritmos paralelos

El diseño de algoritmos paralelos comprende cuatro etapas: particionamiento, comunicación, aglomeración y mapeo.

3.3.1. Etapa: Particionamiento

La etapa de particionamiento se refiere a la descomposición en tareas del problema a resolver. Lo que se intenta es definir un número grande de tareas pequeñas para obtener una descomposición de grano fino. Sin embargo, en etapas posteriores, puede ocurrir que varias tareas sean unidas, para por ejemplo reducir comunicación. A continuación, se describirán de forma no exhaustivas, algunas técnicas de descomposición que puede ser usadas en muchos problemas de manera combinada o no.

3.3.1.1. Descomposición Recursiva

La descomposición recursiva es una técnica que se utiliza para problemas del tipo *"divide y vencerás"*. El problema a resolver es dividido en un conjunto de subproblemas independientes. Cada uno de estos problemas se resuelve recursivamente aplicando una división similar a los subproblemas obteniendo subproblemas más pequeños y combinando sus resultados.

3.3.1.2. Descomposición basada en los datos

Para aplicar la descomposición de datos se debe identificar el conjunto de datos sobre los que se realizarán los cálculos, y luego dividir los datos entre las tareas. La partición correcta de los datos está ligada al balance de carga y al performance alcanzado.

Las operaciones que las tareas ejecutan sobre las diferentes porciones de datos son por lo general similares o se eligen desde un pequeño conjunto de operaciones.

A partir de los datos se pueden realizar varias posibles formas de descomposición:

Descomposición de datos de salida: cada elemento de salida puede ser computado independiente de los demás en función de los datos de entrada. Las descomposición no son únicas, lo que significa que para un mismo problema podemos tener dos descomposiciones distintas.

Descomposición de datos de entrada: para muchos problemas es la única solución por desconocer cuáles o cuántas son las salidas. Los datos de entrada son divididos entre las distintas tareas. Las tareas son creadas para resolver una dicha porción de los datos de entrada. Los procesadores pueden ser asignados a las tareas a un mismo nivel o reusarse en sucesivas divisiones. Los resultados parciales generados por cada tarea luego deben ser unificados en un resultado total.

Descomposición de datos de entrada y salida: es usada en los casos en los cuales pueden realizarse particiones de los datos de las dos maneras anteriores, lo que permitiría lograr mayor concurrencia.

Descomposición de datos intermedios: los algoritmos que pueden ser estructuras como computaciones multi-nivel, donde la salida de un nivel es la entrada de otro nivel subsiguiente, puede aprovechar una partición de datos de entrada o salida de un nivel intermedio.

3.3.1.3. Descomposición exploratoria

Es utilizada para problemas en los cuales los cómputos que se realizan corresponden a búsquedas en un espacio de solución. El espacio de búsqueda se particiona en pequeñas partes, y se realiza la búsqueda concurrentemente en cada una de las partes hasta encontrar la solución deseada. En algunos casos, la ejecución del problema evoluciona en tiempo real. Además, puede darse el caso de obtener un speedup superlineal, es decir, el algoritmo paralelo realiza menos trabajo que el algoritmo secuencial.

3.3.1.4. Descomposición especulativa

Es útil para cando un programa puede tomar una o varias posibles alternativas computacionales dependiendo de la salida de la computación anterior. Se aplica a problemas en los que no se conoce a priori la dependencia de tareas, y además es difícil asegurar que se realiza una descomposición independiente.

Mientras unas tareas están realizando computaciones para decidir que computación se ejecutará, otras tareas podrían estar ejecutando otras computaciones.

Cuando se decide cual es la alternativa que se ejecutará en el próximo paso, las demás computaciones realizadas son descartadas. Por tanto, existe un compromiso entre la concurrencia que se puede alcanzar y el porcentaje de procesos que hay que abortar y reiniciar.

La descomposición especulativa se diferencia de la exploratoria en:

- La entrada de la bifurcación a múltiples tareas es desconocida para la descomposición especulativa, en tanto que para la descomposición exploratoria lo es la salida.
- En la descomposición especulativa el algoritmo paralelo realiza más trabajo que el secuencial, mientras que el la descomposición exploratoria puede ocurrir que el speedup sea superlineal.

3.3.1.5. Descomposición híbrida

Se trata de una descomposición que resulta de una combinación de otras técnicas de particionamento. Cada técnica distinta es utilizada en diferentes niveles de la descomposición.

3.3.2. Etapa: Comunicación

Pretende determinar el intercambio de datos entre las tareas. Puede subdividirse en dos etapas: definir un canal que enlace a las tareas que cooperan y especificar los mensajes que viajarán por ellos.

La interacción entre las tareas de un algoritmo paralelo puede ser totalmente diferente a la utilizada para otro algoritmo. Los tipos de interacción pueden ser descriptos dependiendo de diferentes características [GGKK03]:

Interacción estática o dinámica: un patrón de interacción estática se caracteriza por la ocurrencia de comunicación en un tiempo predeterminado, y las tareas que interactúan en ese momento son conocidas a priori en el momento de la ejecución. Por lo tanto, no solo se conoce la interacción entre tareas sino que también el cuándo dicha comunicación ocurre.

En la interacción dinámica tanto el tiempo de interacción como las tareas que se comunican no es conocido a priori en la ejecución del algoritmo.

La interacción estática es fácil de programar en el paradigma de pasaje de

mensajes, no así la dinámica, esto es porque ambas tareas que interaccionan necesitan conocer información una de la otra. En este paradigma, para que este tipo de interacción sea posible requiere de la utilización de sincronizaciones

Sin embargo, en el caso del uso del paradigma de memoria compartida ambos tipos de interacciones son igualmente de fáciles de programar.

Interacción regular o irregular: otra forma de interacción está basada en la estructura espacial. Un patrón de interacción es regular si la estructura puede ser explorada con una implementación eficiente. Por lo cual, una interacción irregular no tiene un patrón regular. Tanto la comunicación dinámica como la irregular son difíciles de manejar, particularmente en el paradigma de pasaje de mensajes.

Interacción de solo lectura o de lectura-escritura: está relacionado con el acceso a los datos compartidos.

3.3.3. Etapa: Aglomeración

A partir de las etapas anteriores se obtiene como resultado un algoritmo abstracto que no puede ser ejecutado de manera eficiente en una arquitectura particular. Por lo tanto, en esta etapa se busca obtener un algoritmo que se ejecute de forma eficiente en una máquina real.

En esta etapa se debe tener en cuenta un conjunto de criterios:

- Dos tareas que son independientes son dos tareas distintas.
- Si existe demasiada interacción entre las tareas, pueden unirse.

Como puede apreciarse, la aglomeración de tareas comprende el pasaje de la aplicación a un grano más grueso. Pero, si el grano resultante es muy grueso se puede perder escalabilidad y balance de carga. Sin embargo, las decisiones a tomar con respecto a la uniones entre tareas no son las mismas para todos los algoritmos paralelos, no existe un patrón de actividades a llevar a cabo, depende mucho de la aplicación y la arquitectura.

3.3.4. Etapa: Mapeo

Puede decirse que las tres etapas anteriores son independientes de la arquitectura, y conforman un modelo genérico que representa la aplicación. La etapa de mapeo está relacionada directamente con la arquitectura a utilizar, y es la que determina en qué

procesador físico se ejecutará cada tarea. Claramente, esta especificación de tarea-procesador no existe en máquinas monoprocesador o con espacio de memoria compartida donde la asignación de la tarea a un procesador se realiza de manera automática.

El objetivo final es reducir el tiempo de cómputo. Sin embargo, las decisiones a tomar pueden entrar en conflicto entre sí: ubicar tareas que se ejecutan concurrentemente en procesadores distintos para incrementar la concurrencia o colocar tareas que realizan mucha comunicación en los mismos procesadores para incrementar la localidad.

Dicha problemática es NP-Completo, es decir, no existe un algoritmos tratable computacionalmente en tiempo polinomial que pueda resolver todas las posibles combinaciones entre tareas y procesadores, por lo cual, se suelen usar heurísticas para resolver esta clase de problemas.

Las técnicas de mapping, pueden dividirse en [GGKK03]:

- **Estática**: las tareas se mapean a los procesadores a priori. Se deben tener en cuenta el tamaño de la tarea. Los algoritmos resultantes suelen ser más fáciles de diseñar y programar.
- **Dinámica**: las tareas se crean dinámicamente en ejecución, no se conoce a priori la cantidad de tareas ni tampoco su tamaño. Los algoritmos suelen ser más complicados. La migración de datos que se produce en muchos problemas suele agregar un overhead extra.
- **Jerárquico**: algunos problemas tienen una expresión que naturalmente se corresponde con un grafo de dependencia de tareas. Sin embargo, esta manera de mapear las tareas produce muchas veces desbalance de carga. Por lo cual, lo que se suele utilizar es una combinación de técnicas en diferentes niveles.

Es muy importante tener en cuenta la escalabilidad y el balance de carga en esta etapa. Un mapeo estático, por ejemplo, es más propenso a balancear mal el trabajo. En cambio, el mapeo dinámico obtiene un mejor balance de carga, pero produce una mayor cantidad de comunicación, que el mapeo estático no genera.

En el caso de la escalabilidad, la elección de un mapping dinámico puede aumentar el overhead de comunicación a medida que la arquitectura o el tamaño del problema se incrementa.

Para lograr balancear la carga entre los procesos existen varias maneras de distribuir el trabajo:

- ***Distribución estática directa***: es eficiente en arquitecturas homogéneas y

regulares. Se basa en una distribución equitativa del trabajo entre todos los procesadores, suponiendo igual tiempo para todos los procesadores. Si la arquitectura es irregular pero si se sabe a priori la cantidad de cálculo que se va a realizar con cada dato, se podría lograr un buen balance de carga. Si la arquitectura es heterogénea, este tipo de distribución no es útil, ya que se debe predecir la cantidad de cálculo por dato y a partir de ello acumularlos.

- ***Distribución estática predictiva***: trata de resolver los inconvenientes de la distribución anterior. Los datos se distribuyen respecto de la potencia de cómputo de los procesadores. La problemática que se presenta es calcular la potencia de cómputo relativo de cada procesador. Una manera de calcular el computo relativo es calcular el tiempo promedio a partir de pruebas de la aplicación con un tamaño de entrada pequeño. El inconveniente es que se debe conocer el tiempo exacto que se va a calcular cada dato en cada máquina, y que además siempre se tiene un margen de error que será perjudicial dependiendo de la aplicación.
- ***Distribución dinámica por demanda***: en este tipo de distribución hay que tener en cuenta la cantidad de datos que se le va a dar a cada procesador. Una manera de implementar esta distribución, es realizar una distribución estática al inicio, y balancear dinámicamente al final. En este caso, se debe considerar el porcentaje de datos a repartir estáticamente y si la distribución se repartirá dependiendo la potencia de cómputo de cada procesador. Esto es, si no se conoce la cantidad de cómputo por dato, entonces la distribución estática al inicio no sería útil. En el caso del uso de un paradigma Master-Worker, en este caso, el master no puede realizar trabajo porque debe estar atento a la peticiones.
- ***Distribución basa en la estimación en el trabajo inicial***: este tipo de distribución puede usarse tanto para arquitecturas homogéneas como heterogéneas. Distribuye un porcentaje del trabajo al inicio estáticamente. Los worker trabajan y calculan el tiempo que les va ha llevar trabajar (esto se hace a partir de un porcentaje del trabajo inicial) y a partir de esto se determina la cantidad de trabajo que le toca a cada uno. Los workers siguen trabajando mientras el master a partir de la estimación hecha por cada worker calcula la cantidad de datos a enviarles.

El algoritmo puede estar bien balanceado en cuanto a cantidad de tareas pero puede originar ociosidad y desbalance en ejecución.

Capítulo 4

Modelos de Algoritmos paralelos

"El paralelismo es el futuro de la computación"[1]

4.1. Modelo de paralelismo de datos

Es uno de los modelos de algoritmos más simples. En este modelo las tareas son estática o semiestáticamente mapeadas a los procesadores y cada tarea ejecuta una operación similar sobre los diferentes datos. El trabajo puede ser hecho en fases y los datos que son computados en cada fase pueden ser diferentes. Por lo general, las tareas llevan a cabo sincronizaciones entre fases. La descomposición del problema en tareas está basado en la partición de datos, y como suelen mapearse estáticamente a los procesadores garantiza balance de carga.

Los algoritmos con paralelismo de datos pueden ser implementados tanto en memoria compartida como en paraje de mensajes. En el caso del paradigma de memoria compartida la ventaja se halla en la facilidad de programación, especialmente si la distribución de los datos es diferente en cada fase del algoritmo.

1 Tomado de: "GPU Computing" por John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips , 2008, IEEE

Una manera de reducir la interacción de las tareas es solapar cómputo con interacción y hacer uso de las comunicaciones colectivas. Una característica de los problemas que pueden ser paralelizados a partir de los datos es que al incrementar el tamaño del problema crece en grado de paralelismo de datos, pudiéndose usar más procesos.

4.2. Modelo de grafo de tareas

Para ciertos algoritmos paralelos puede usarse explícitamente el grafo de dependencias de tareas para mapear las tareas a los procesadores. Es muy utilizado para resolver problemas en los cuales la cantidad de datos asociados con las tareas es relativamente grande comparado con la cantidad de cómputo con ella. Las tareas son mapeadas estáticamente para ayudar a optimizar el costos de interrelación entre tareas. Aunque un mapeo dinámico también puede ser utilizado.

Las técnicas para reducir la interacción incluyen disminuir el volumen y la frecuencia de la interacción promoviendo la localidad a través de un mapeo de tareas basado en el patrón de interacción de las mismas, y además usando métodos de interacción asincrónicos que permitan solapar comunicación con cómputo.

Por estar expresado a partir de tareas independientes se lo suele denominar paralelismo de tareas.

4.3. Modelo Master-Workers

En este modelo existe un proceso master o maestro que es el encargado de repartir el trabajo entre los procesos trabajadores o workers. La asignación de tareas puede ser estática o dinámica. En el caso de la asignación estática, dependiendo del problema en particular, incluso el master puede actuar como un worker después de haber repartido el trabajo. En un escenario con asignación dinámica es preferible que el master no trabaje para estar atento a los pedidos de los workers.

Pude ser implementado tanto en memoria compartida y memoria distribuida. Además, pueden existir más de un nivel de master, formando así una estructura multinivel. Lo que resulta muy útil para reducir los cuello de botella cuando la cantidad de procesos se incrementa.

Es necesario tener muy en cuenta la granularidad que van a tener las tareas. Debe

ser elegido de tal forma que el costo de realizar trabajo sea mayor que el costo de sincronización y comunicación. Una de las maneras de reducir overhead por comunicación es el solapamiento cómputo utilizando comunicaciones asincrónicas.

4.4. Modelo Híbrido

Un modelo híbrido puede estar compuesto de la combinación de múltiples modelos aplicados jerárquicamente o secuencialmente en diferentes fases del algoritmo paralelo. Esto es porque, algunas veces un algoritmo puede tener características de más de un modelo de algoritmo. En la presente Tesina de grado, se utilizará un modelo híbrido compuesto por la combinación de programación de memoria compartida con memoria distribuida.

4.5. Modelo GPU-CUDA

Existe una gran diferencia entre la programación en la GPU y la CPU. Por lo cual, sus modelos de programación son diferentes.

La principal distinción es que la GPU [Nvi03] no es un procesador serial, pero si un procesador de stream. En el modelo de programación de stream, todos los datos son representados como stream, definiéndolos como un conjunto de datos de algún tipo de dato. El tipo de dato puede ser simple (streams de enteros o punto flotante) o complejos (streams de puntos o triángulos o matrices de transformación). Los streams pueden tener cualquier longitud, pero cuanto mayor es la longitud del mismo más eficientes son las operaciones que se realizan sobre ellos. Los procesadores de stream trabajan de manera diferente a los seriales, ya que ejecutan una función (fragmento de programa) sobre un conjunto de registros de entrada (fragmentos), produciendo un conjunto de registros de salida (pixels) en paralelo. Los procesadores de stream se refieren a esta función como kernel y al conjunto de registros como stream. Los datos fluyen en el procesador, y este opera sobre los mismos a través de la función kernel, enviando los resultados a la memoria.

El kernel opera sobre entidades stream de elementos no sobre elementos individuales. Además, de evaluar una función sobre cada elemento del stream de entrada, el kernel, también realiza operaciones tales como expansión (más de un elemento de salida es producido por un elemento de entrada), reducción (más de un elemento de entrada es combinado para producir un elemento de salida), o filtrado (un subconjunto de

los elementos de entrada se convierten en elementos de salida). Se dice que se tiene un uniform streaming cuando se aplica algún kernel a todos los elementos del stream.

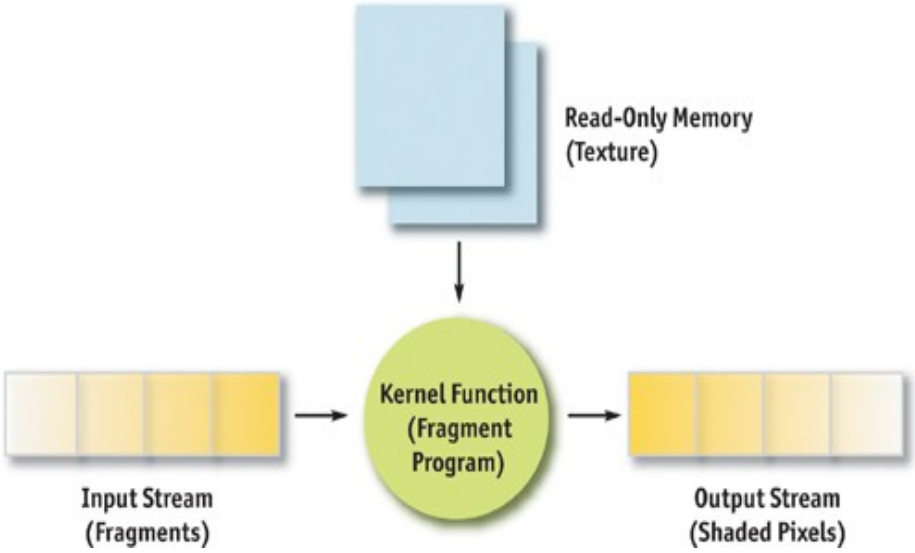


Fig. 4.1. Modelo de programación GPU (tomado de [Nvi03])

Cada elemento es procesado independientemente, es decir, sin existir ninguna dependencia entre los elementos. Esto le permite ejecutar el programa en paralelo sin la necesidad de expresar las unidades paralelas o ninguna instrucción paralela por parte del programador. Esta restricción de independencia entre los datos, tiene dos ventajas: primero, los datos requeridos por el kernel para la ejecución son completamente conocidos cuando el kernel es escrito (o compilado). El kernel puede ser muy eficiente con los elementos de entrada y los datos intermedios computados que son almacenados localmente o son cuidadosamente controlados con referencias globales. Segundo, la independencia de los elementos permite que el kernel sea una función de cómputo serial sobre un hardware de datos paralelos.

En el modelo de programación de streams, las aplicaciones son construidas como una tubería de múltiples kernel juntos. Por ejemplo, en un pipeline gráfico se podría escribir un kernel que compute vértices, un kernel en assembly de triángulos, un kernel clipping, y uno que conecte la salida de un kernel al otro.

Este modelo hace que la comunicación entre los kernels sea explícita, tomando las ventajas de la localidad de los datos entre los kernels en el pipeline gráfico.

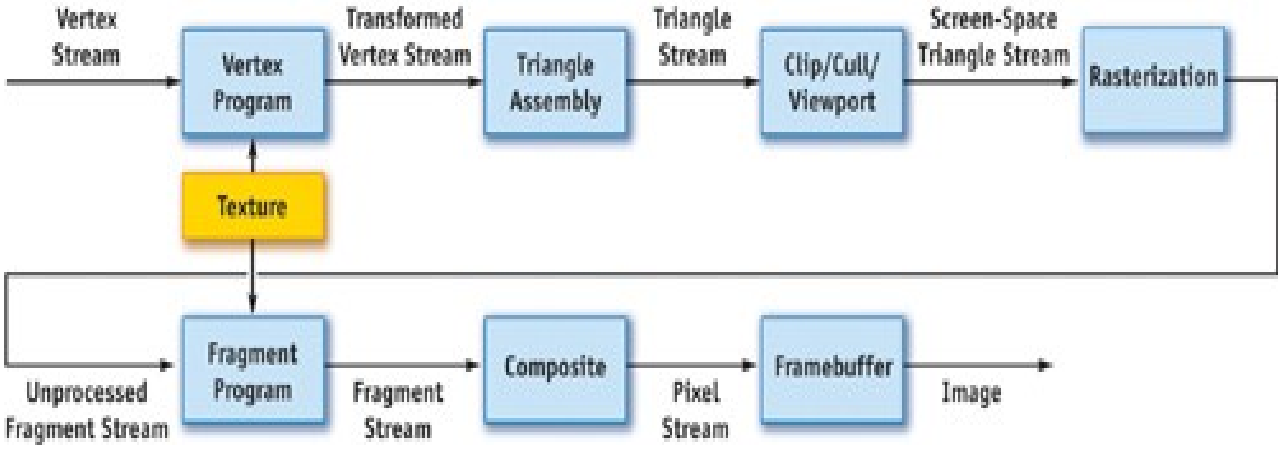


Fig. 4.2. Mapeo de un pipeline gráfico al modelo de streams (tomado de [Nvi03])

El pipeline gráficos [OHLG++08] es una buena forma de correspondencia con el modelo de stream por varias razones. Los pipeline gráficos están estructurados como escenarios conectados por el flujo de datos que se está computando. Esta estructura es análoga a los streams y kernels del modelo de programación de streams. Así, como en el pipeline gráfico los datos pasan de un escenario a otro, en el modelo de streams estos pasan de un kernel a otro, por lo que se puede notar que su comportamiento es similar.

El modelo de programación de streams, realiza cómputo de manera eficiente por las siguientes razones:

- Las entidades streams pueden ser procesadas paralelamente gracias al paralelismo hardware de datos. Los streams con una gran cantidad de elementos utilizan de manera más eficiente el paralelismo a nivel de datos.
- Las aplicaciones son construidas a partir de múltiples kernels, los cuales, forman un pipeline y procesan en paralelo haciendo uso del paralelismo a nivel de tarea.

La combinación del paralelismo a nivel de datos y tareas permite que la GPU use docenas de unidades funcinales simultáneamente.

Existen herramientas que ayudan a aprovechar las ventajas de la GPU:

- “Reduce” es alguna operación que computa valores simples desde un conjuntos de datos. Por ejemplo: max/min, avg, promedio, etc.
- “Sort and Search” provee mecanismos para construir y usar estructuras de datos en la GPU.

Usando estas operaciones, se puede ejecutar muchos algoritmos de propósito general en la GPU.

Capítulo 5

Herramientas para la Programación Paralela

"La meta más agresiva del paralelismo de hoy es hacer que los programas sean eficientes, portables y escalables, pero también es deseble que sea fácil la programación de los mismos como lo es actualmente la tarea de escribir programas para computadoras secuenciales."[1]

1 Anónimo

5.1. Programación paralela en memoria compartida

La programación en memoria compartida [GGKK03] puede tener muchos mecanismos para compartir los datos, modelos de concurrencia y soporte de sincronización.

Varias tareas se ejecutan en paralelo, pudiendo tener una memoria local "exclusiva" y todas o por grupos acceder a un espacio de direcciones de memoria en común. La sincronización se lleva a cabo mediante la escritura y lectura de áreas de la memoria compartida. El problema de la sincronización es responsabilidad del programador, utilizando las herramientas que provea el lenguaje. Sin embargo, hay que tener en cuenta, que estos mecanismos reducen la eficiencia y pueden originar efectos indeseables tales como los deadlocks.

Por lo general, el programador no maneja la distribución de los datos ni lo relacionado a la comunicación de los mismos. La localidad es un factor determinante sobre todo en arquitecturas NUMA. En algunos lenguajes, el programador tendrá la posibilidad de actuar sobre dicha localidad o deberá re-estructurar el código.

La ventaja de la programación en memoria compartida es la transparencia, ya que la ubicación de los datos, su replicación y migración no deben ser tratados por el programador. Sin embargo, la desventaja se encuentra en la dificultad de predecir el performance a partir de la lectura del algoritmo.

La programación en memoria compartida está soportada por diferentes modelos de programación que tienen su manera de expresar la concurrencia y la sincronización:

- Modelos basados en procesos suponiendo datos locales/privados a cada proceso.
- Modelos basados en threads o procesos "livianos" suponene memoria global (Pthread).
- Modelos basados en directivas que extienden el modelo basado en threads para facilitar su creación y sincronización (OpenMP).

El presente trabajo se concentra principal en los modelos basados en procesos livianos. A continuación, se explicarán las ventajas del uso de threads:

Un ***thread*** es un hilo de control simple dentro del flujo de un programa. La principal ventaja del uso de hilos es la reducción del tiempo de context-switching. Sin embargo, existen otras ventajas:

- ***Portabilidad de software***: las aplicaciones que utilizan threads pueden ejecutarse tanto en máquinas secuenciales como paralelas sin la necesidad de cambio.
- ***Ocultamiento de latencia***: uno de los overhead de los programas es la latencia en el acceso a memoria, E/S y comunicación. La ejecución de múltiples threads en un mismo procesador puede ser usada para solapar cómputo con tiempo de espera ocultando de esta manera la latencia.
- ***Scheduling y balance de carga***: al no existir comunicación remota, se reduce el overhead por comunicación y la ociosidad. La posibilidad de creación de un gran número de threads y el mapeo dinámico a los procesadores permiten la reducción de ociosidad.
- ***Facilidad de programación y uso***: suele ser más fácil de programar que la programación con pasaje de mensajes. Además no requiere un manejo explíto de la comunicación.

5.1.1. Pthread

Existen varias APIs para el manejo de threads. En la presente Tesina de Grado se ha utilizado la API Pthread para presentar la solución en plataformas de memoria compartida. Debido a esto, solo se introducirá sobre las características más destacantes de esta API en particular.

Pthread [And00] [GGKK03] es un standard especificado por la IEEE (estandard 1003.1c-1995), y está soportado por muchos vendedores. Sin embargo, el concepto en si mismo de las características de la programación en memoria compartida son independientes de dicha API, son válidos para ser trasladados a: JAVA Threads, NT Threads, Solaris Threads, etc.

A continuación, se explican las funciones básicas utilizadas en los programas paralelos:

5.1.1.1. Creación y terminación de threads

La función básica de toda API de manejo de threads es la creación y la espera de la terminación la ejecución de todos los threads. En Pthreads los hilos son creados usando la función `pthread_create` cuyo prototipo es:

```
int pthread_create(pthread_t *thread_handle,
                   const pthread_attr_t *atribute,
                   void* (*thread_function)(void *),void *arg);
```

La función `pthread_create` crea un thread que invoca a la función `thread_function`. Si la creación es exitosa, la función `pthread_create` retorna el valor cero, sino un código de error, además se asocia un identificador único al thread creado y se le asigna un puntero a la dirección de `thread_handle`. Los atributos del thread se encuentran descriptos en el argumento `atribute`. Cuando dicho argumento es NULL, el thread es creado con los atributos por defecto. El argumento `arg` especifica un puntero al argumento función `thread_function`. Este argumento es usado para pasar el workspace y otros datos específicos a dicho thread.

El *main*, programa principal, debe esperar a que todos los threads terminen su ejecución. Dicha espera se logra a través de la función `pthread_join` el cual suspende la ejecución hasta que todos los threads terminen. El prototipo de dicha función es el siguiente:

```
int pthread_join(pthread_t thread, void **ptr);
```

La función esperará la terminación de todos los threads con id `thread`. Si el llamado a `pthread_join` es exitoso, el valor pasado a `pthread_exit` es retornado en la dirección del puntero `ptr`. La terminación con éxito de `pthread_join` retorna 0 (cero), o un código de error en otro caso.

5.1.1.2. Primitivas de sincronización

A pesar de que la comunicación está implícita en la programación en memoria compartida, el esfuerzo más grande se encuentra en escribir programas correctos los cuales hacen uso de la sincronización para evitar las posibles interferencias entre los distintos threads que conforman el programa paralelo.

Cuando múltiples threads comparten datos el resultado puede ser inconsistente si no se hace un uso correcto de los mecanismos de sincronización. Pthread provee soporte para implementar secciones críticas y operaciones atómicas usando ***mutex-locks*** (locks de exclusión mutua). Los mutex-locks tienen dos estado: locked y unlocked. Solo un thread puede tener acceso a un mutex-locks a la vez. Un lock es una operación atómica generalmente asociada con una pieza de código que manipula los datos compartidos. Para acceder a los datos compartidos, el thread debe primero tratar de adquirir el mutex-lock. Si el mutex-lock está siendo usado por otro thread deberá esperar a que se desbloquee. Que el mutex-lock esté bloqueado significa que otro thread está en la sección de datos compartida o sección crítica. Cuando un thread avandona la sección crítica debe desbloquear el mutex-lock. Todos los mutex-lock deben ser inicializados como unlocked al iniciar el programa.

La API Pthread provee funciones para manejar los mutex-locks. La función `pthread_mutex_lock` es usadas para que el thread adquiera el mutex-lock. El prototipo de la función es el siguiente:

```
int pthread_mutex_lock(pthread_mutex_t *mutex_lock);
```

Si el mutex-lock está siendo usado por otro hilo, el thread es bloqueado. Si el llamado a la función es exitosa retorna 0 (cero) o un código de error en otro caso.

Para abandonar la sección crítica el thread debe desbloquear el mutex-lock. Pthread permite realizar dicha operación utilizando la función `pthread_mutex_unlock`, cuyo prototipo es el siguiente:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex_lock);
```

Cuendo un thread llama a esta función, el mutex-lock es desbloqueado. Si existen otros threads esperando para entrar a la sección crítica, el scheduler eligirá uno para que pueda acceder a los datos compartidos.

Existe otra función importante relacionada a los mutex-locks, la función de inicialización de los mutex: `pthread_mutex_init`. El prototipo de la función es el siguiente:

```
init pthread_mutex_init(pthread_mutex_t *mutex_lock,
                        const pthread_mutexattr_t *lock_attr);
```

La inicialización por defecto de los mutex-lock es desbloqueados, y es especificado por el argumento `lock_attr`. Si el argumento es NULL, el mutex-lock es inicializado con el valor por defecto.

Los locks representan punto de serialización, si dentro de una sección crítica los segmentos del programa son largos se degradará la performance de la aplicación total. Se puede reducir el overhead por espera ociosa utilizando la función `pthread_mutex_trylock`, la cual, retorna el control informando si el thread pudo o no hacer el lock. Su prototipo es el siguiente:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex_lock);
```

Las ventajas de su uso están relacionadas a la reducción de los tiempos ociosos y menor costo por no tener que usar colas de espera.

Existen otras variables denominadas de condición que sirven para que un thread se autobloquee hasta que se alcance un estado determinado del programa. Cada varible condición estará asociada con un predicado. Cuando el predicado se convierte en verdadero la variable de condición da una señal para el/los threads que están esperando por el cambio de estado de la condición. Una única varible de condición puede asociarse a varios predicados. Además siempre tiene un mutex asociada a ella. Cada thread bloquea este mutex y testea el predicado definido sobre la variable compartida. Si el predicado es falso el thread espera en la varibale condición utilizando la función `pthread_cond_wait` (no requiere el uso de la CPU). Los siguientes son los prototipos de las funciones para manejar variables condición:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
```

```
int pthread-cond_destroy(pthread_cond_t *cond);
```

5.1.1.3. Objetos atributos

La API Pthread permite que puedan ser cambiados los atributos por defecto de las entidades usando attributes objects. Un attribute object es una estructura de datos que describe las propiedades de: threads, mutex, variables condición.

Una vez establecidas las propiedades el attribute object es pasado al método que inicaliza la entidad. Las ventajas que provee el uso de estos objetos atributos son: mejora la moduralaridad y facilita la modificación de código.

Los prototipos de dichas funciones son:

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);
```

Las propiedades asociadas con el attribute object pueden ser cambiadas con las siguientes funciones:

```
pthread_attr_setdetachstate
pthread_attr_setguardsize_np
pthread_attr_setstacksize
pthread_attr_setinheritsched
pthread_attr_setschedpolicy
pthread_attr_setschedparam
```

Para el manejo de atributos de los mutex, la API Pthread provee las siguientes funciones:

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);

int pthread_mutexattr_settype_np ( pthread_mutexattr_t *attr,
                                   int type);
```

El argumento `type` especifica el tipo de mutex y puede tomar los valores:

```
PTHREAD_MUTEX_NORMAL_NP
PTHREAD_MUTEX_RECURSIVE_NP
PTHREAD_MUTEX_ERRORCHECK_NP
```

Se pueden construir operaciones de sincronización de más alto nivel, a partir de los constructores básicos. Como bloqueos para lecturas-escrituras y barreras.

En muchas aplicaciones una estructura de datos es leída con mayor frecuencia que escrita (modificada). En tales casos puede convenir diferenciar los bloqueos de lectura y de escritura. Un bloqueo de lectura es habilitado aún cuando haya otros threads realizando un bloqueo de lectura. Si hubiera un bloqueo de escritura sobre el dato o en cola, el thread deberá realizar un condition wait. Múltiples threads pidiendo un bloqueo de escritura se encolarán con alguna política (normalmente FIFO) . Con esta descripción, se han definido funciones para:

- Bloqueo de lectura (`mylib_rwlock_rlock`)
- Bloqueo de escritura (`mylib_rwlock_wlock`)
- Desbloqueo (`mylib_rwlock_unlock`)

También pueden construirse barreras. Para ello se necesita: un contador, un mutex y una varibale condición. El contador es un entero que se utiliza para saber cuántos threads han alcanzado la barrera. Si la cuenta es menor al total de threads involucrados el thread ejecutará un wait condicional. El último thread que alcanza la barrera despierta a todos los demás threads utilizando un broadcast de la variable condición.

Los tipos de datos y la inicialización se realiza como sigue:

```
typedef struct { pthread_mutex_t count_lock;
pthread_cond_t ok_to_proceed;
int count;
} mylib_barrier_t;

void mylib_init_barrier (mylib_barrier_t *b)
{ b -> count = 0;
pthread_mutex_init(&(b -> count_lock), NULL);
pthread_cond_init(&(b -> ok_to_proceed), NULL);
```

```
}
```

5.1.1.4. Cancelación de threads

Pthread provee una primitiva de cancelación, cuyo prototipo es el siguiente:

```
int pthread_cancel(pthread_t thread);
```

El argumento `thread` es el manejador del thread que se quiere cancelar. Un thread puede cancelarse a sí mismo o cancelar a otros threads. Cuando se realiza un llamado a esta función, la misma envía una señal de cancelación al thread especifiicado. Sin embargo, no garantiza que el thread recibirá dicha cancelación o actuará realizando su cancelación. Las acciones tomadas cuando un thread se cancela son similares a las llevadas a cabo en la invocación a la función `pthread_exit`. En el caso de una cancelación exitosa la función retorna cero, o un código de error en otro caso.

5.2.Programación paralela en memoria distribuida

Existen dos atributos que caracterizan a la programación en memoria distribuida por pasaje de mensajes [And00] [GGKK03], los cuales son:

- El espacio de direcciones se encuentra particionado.
  - Cada elemento pertenece a una partición.
  - Toda interacción requiere de la cooperación de los procesos que se comunican.
- Requiere de un paralelismo explícito.

Lógicamente una máquina que soporta pasaje de mensaje consiste de *p* procesos, cada uno con su propio espacio de direcciones privado. Existen dos implicaciones relacionadas directamente a la memoria distribuida: primero, los datos deben ser particionados explícitamente y además distribuidos. Segundo, se requiere la cooperación de los procesos que se comunican, ya sea para realizar operaciones de solo lectura o de lectura-escritura. Estas razones hacen que sea más compleja la tareas de escribir programas en este tipo de plataformas.

El programador es el responsable de descomponer el problema en tareas, especificar la concurrencia, distribuir los datos, realizar scheduling de los procesos, así como también la comunicación entre las tareas. Si bien, la programación, el dubugging y el mantenimiento en programs de pasaje de mensajes suele clasificarse como dificultosa, puede ofrecer un alto performance y escalabilidad a un gran número de procesos. Puede alcanzarse una mayor eficiencia, ya que el programador puede manejar el balance de carga, la distribución de los datos y procesos (dinámicamente), replicar datos. Provee portabilidad, existen muchas bibliotecas para facilitar la ejecución de código en diferentes arquitecturas.

Los programas con pasaje de mensaje a menudo se escriben utilizando paradigmas asincrónicos (todos los procesos se ejecutan asicrónicamente) o débilmente sincrónicos (un subconjunto de procesos deben sincronizar para interactuar, en los tiempos entre itereacciones se ejecutan asincrónicamente).

La mayoría de los programas en pasaje de mensajes se escriben para el modelo SIMD (Single Program, Multiple Data). Los procesos no están sincronizados en la ejecución de casa sentencia, y no todos realizan las mimas sentencias.

Las ventajas del uso de pasaje de mensaje son:

- ✔ El programador tiene el control total para logarar sistemas eficientes y escalables.
- ✔ Puede implementarse eficientemente en muchas arquitecturas paralelas.
- ✔ Más fácil de predecir el rendimiento.

Como desventajas se pueden nombrar:

- ✗ Mayor complejidad al implementar los algoritmos para lograr buen performance.

Para la interacción entre procesos existen una serie de comunicaciones que pueden llevarse a cabo. Las dos operaciones básicas son el send y el receive.

*Send (void *sendbuf, int nelems, int dest)*

*Receive (void *recvbuf, int nelems, int source)*

Existen diferentes protocolos para el send y el receive, los cuales, se explicarán a continuación:

➔ **send y receive bloqueante**: no devuelven el control hasta que el dato que se transmite esté seguro. Puede ocasionar ociosidad en los procesos. Hay dos posibilidades.

- Send/Receive bloqueantes sin buffering

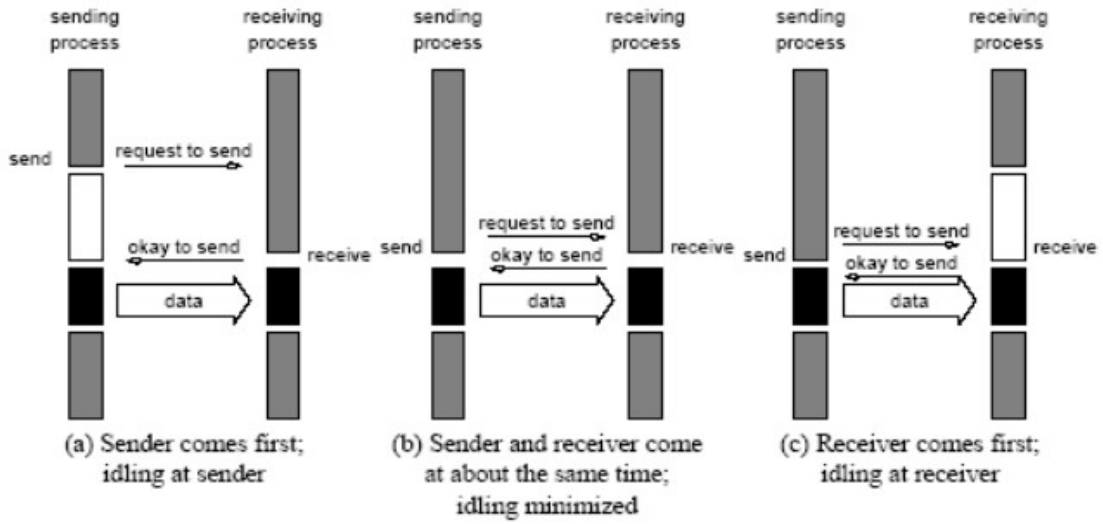


Fig. 5.1. Send y receive bloqueante sin buffering (tomado de [GGKK03])

En el caso del send, por ejemplo, este se bloquea hasta que el receptor no termine el receive del mensaje. Ocasiona ociosidad, y deadlocks si las sentencias de comunicación no hacen matching.

- Send/Receive bloqueantes con buffering

Por ejemplo en el caso del send, se bloquea hasta que el mensaje llega a un buffer prealocado. La transmisión del mensaje puede realizarse mediante hardware para realizar comunicación asincrónica, o sin hardware especial, en este caso el que envía se desbloquea cuando el mensaje llega al buffer del receptor.

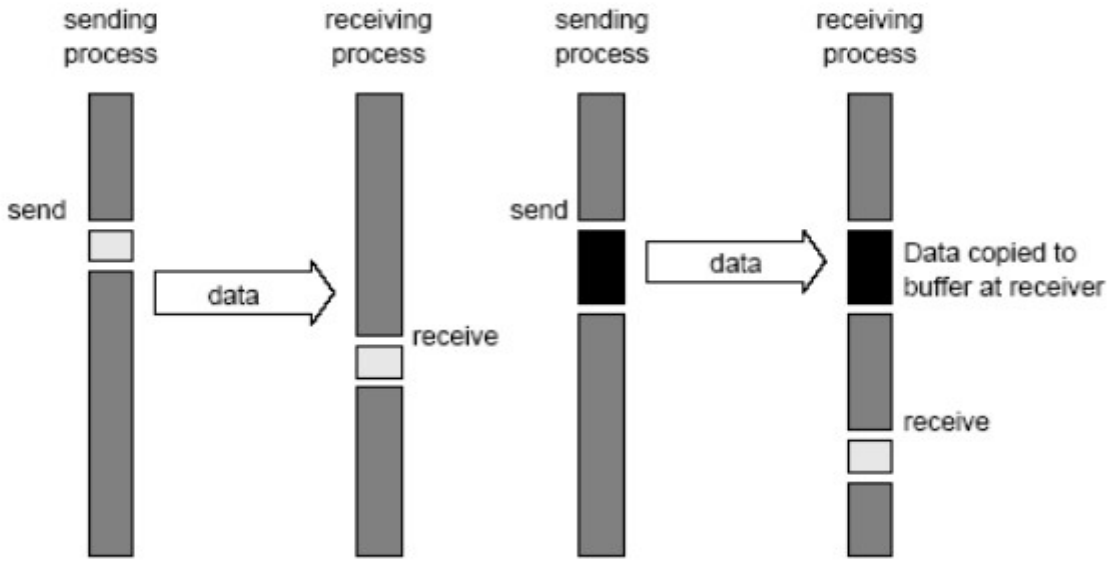


Fig. 5.2. Send y receive bloqueante con buffering (tomado de [GGKK03])

Este tipo de comunicación reduce el tiempo en que los procesadores se encuentran ociosos, pero aumenta el costo por el manejo de buffers. Existe otro inconveniente relacionado al tamaño del buffer, como este es limitado, puede producir el bloqueo del proceso hasta que haya lugar en el buffer. En el caso del receive bloqueante pueden producirse deadlocks.

➔ **send y receive no bloqueante**: para evitar overhead por ociosidad o manejo de buffer, se devuelve el control de la operación inmediatamente. Requiere de un chequeo posterior para asegurar la finalización de la comunicación. Existen dos posibilidades:

- Send/Receive no bloqueante sin buffering

  Para el caso del send, la comunicación se inicia al llegar al receive.

- Send/Receive no bloqueante con buffering

  El emisor utiliza acceso directo a memoria (DMA) para copiar los datos a un buffer prealocado mientras el proceso continúa se cómputo. El uso del buffer reduce el tiempo en el que el dato está no seguro.

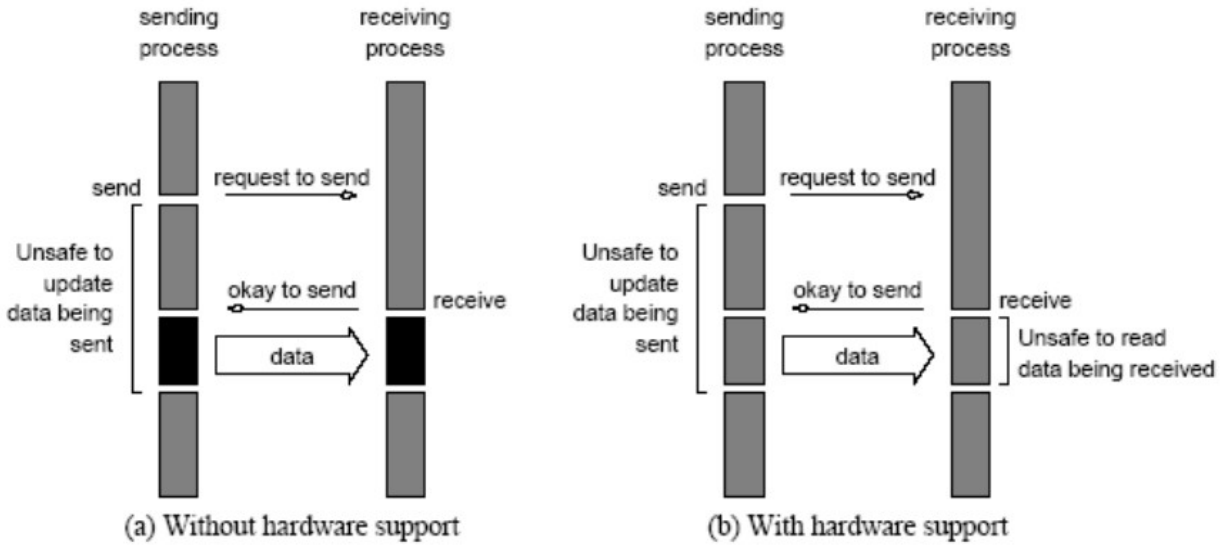


Fig 5.3. Send y receive no bloqueante (tomado de [GGKK03])

Los protocolos de las operaciones de pasaje de mensaje podrían ser resumidas de la siguiente manera:

	Operación bloqueante	**Operación no bloqueante**
Con buffer	**El proceso que envía retorna luego que los datos han copiados en el buffer de comunicación**	**El proceso que envía retorna luego de iniciar la transferencia con el DMA al buffer. La operación puede no haberse completado cuando se retorna el control**
Sin buffer	**El proceso que envía se bloquea hasta que se realice la operación receive que matchea con él**	

Fig. 5.4. Posibles protocolos para las operaciones send y receive

5.2.1. MPI

MPI es una especificación de pasaje de mensajes, diseñada para ser el estándar de la computación paralela en memoria distribuida usando mensajes. Intenta establecer un estándar práctico, eficiente, portable y flexible para el uso de mensajes.

MPI es un conjunto de funciones y macros que conforman una biblioteca estándar de C y C++, y subrutinas en Fortran. Incluye operaciones punto a punto y colectivas, destinadas a un grupo específico de procesos [MFB05].

MPI define la sintaxis y la semántica de más de 125 rutinas. Hay implementaciones de MPI de la mayoría de los proveedores de hardware.

5.2.1.1. Inicialización y finalización

La función de inicialización del entorno MPI es `MPI_Init` se invoca en todos los procesos antes que cualquier llamado a rutinas de MPI. Su prototipo es el siguiente:

```
MPI_Init (int *argc, char **argv) ;
```

Algunas implementaciones de MPI requieren los parámetros `argc` y `argv` para inicializar el entorno.

Para finalizar la ejecución de los procesos se utiliza `MPI_Finalize`, esta función es invocada por todos los procesos como último llamado a las rutinas MPI. Sirve para cerrar el entorno MPI. Su prototipo es:

```
MPI_Finalize () ;
```

5.2.1.2. Comunicadores

Un comunicador define el dominio de comunicación de un proceso. Cada proceso puede pertenecer a uno o varios dominios. Existen un comunicador que incluye a todos los procesos de la aplicación denominado: `MPI_COMM_WORLD`. Los comunicadores son variables del tipo `MPI_Comm` y almacenan la información sobre que procesos pertenecen a él. En cada operación de transferencia se debe indicar el comunicador sobre el cual se va a realizar dicha operación.

5.2.1.3. Adquisición de información

MPI define una serie de funciones que sirven para obtener información del entorno MPI:

`MPI_Comm_size` indica la cantidad de procesos en el comunicador. Su prototipo es el el siguiente:

```
MPI_Comm_size (MPI_Comm comunicador, int *cantidad);
```

`MPI_Comm_rank` indica el “rank” o identificador del proceso dentro de ese comunicador. Su prototipo es:

```
MPI_Comm_rank (MPI_Comm comunicador, int *rank) ;
```

El identificador de un proceso es un valor entero entre cero y la cantidad total de procesos de la aplicación. Dicho rank puede ser diferente en cada comunicador.

5.2.1.4.Tipos de datos

MPI define sus propios tipos de datos, que se corresponden con los tipos de datos primitivos definidos en C.

Tipo de dato MPI	Tipo de dato C
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	Long double

Existen dos tipos de datos no definidos en C, que son: MPI_BYTE y MPI_PACKED.

5.2.1.5. Operaciones send y receive

Las operaciones básicas para la comunicación entre procesos se realiza a partir de sends y receives entre los mismos. En MPI, el prototipo de dichas comunicaciones es:

```
MPI_Send (void *buf, int cantidad,
          MPI_Datatype tipoDato,
          int destino, int tag, MPI_Comm comunicador) ;
```

El valor del Tag se encuentra entre [0..MPI_TAG_UB] . `MPI_Send` es la rutina básica para enviar un mensaje a otro proceso.

`MPI_Send` puede ser:

- `MPI_Send` send con buffer
- `MPI_Rsend` envía cuando el receptor está listo para recibir.
- `MPI_Ssend` send sincrónico bloqueante sin buffer
- `MPI_Bsend` send bloqueante con buffer

```
MPI_Recv (void *buf, int cantidad,
          MPI_Datatype tipoDato,
          int origen, int tag, MPI_Comm comunicador,
          MPI_Status *estado) ;
```

`MPI_Recv` es la operación básica de recepción de un mensaje.

MPI permite definir “comodines” para el argumento del identificador de receptor/emisor y para el canal por donde se recibe/envía, los cuales son: `MPI_ANY_SOURCE` y `MPI_ANY_TAG`.

`MPI_Status` es una estructura que contiene información del estado de la comunicación. Su protoipo es el siguiente:

```
typedef struct MPI_Status { int MPI_SOURCE;
                            int MPI_TAG;
```

```
                                int MPI_ERROR; }
```

Si se quiere por alguna razón conocer la cantidad de elementos recibidos, se puede utilizar la rutina:

```
MPI_Get_count(MPI_Status *estado,
              MPI_Datatype tipoDato, int *cantidad) ;
```

`MPI_Sendrecv`: permite enviar un mensaje a un proceso y recibir otro mensaje del mismo u otro proceso. Su prototipo es:

```
MPI_Sendrecv (void *bufEnvio, int cantEnvio,
              MPI_Datatype tipoDatoEnvio, int destino,
              int tagEnvio, void *bufRecepcion,
               int cantRecepcion, MPI_Datatype tipoDatoRecepcion,
               int origen, int tagRecepcion,
               MPI_Comm comunicador, MPI_Status *estado) ;
```

`MPI_Sendrecv_replace`: igual al anterior pero utiliza un solo buffer.

```
MPI_Sendrecv_replace (void *buf, int cant, MPI_Datatype tipoDato,
                      int destino, int tagEnvio, int origen,
                      int tagRecepcion, MPI_Comm comunicador,
                      MPI_Status *estado) ;
```

Las operaciones de comunicación anteriormente mencionadas son bloqueantes. MPI ofrece comunicaciones no bloqueantes:

```
MPI_Isend (void *buf, int cantidad, MPI_Datatype tipoDato,
           int destino, int tag, MPI_Comm comunicador,
           MPI_Request *solicitud) ;

MPI_Irecv (void *buf, int cantidad, MPI_Datatype tipoDato,
           int origen, int tag, MPI_Comm comunicador,
           MPI_Request *solicitud) ;
```

Las operaciones no bloqueantes requieren que se consulte la finalización exitosa de la comunicación. Esta operación es posible realizarla a través de la rutina:

```
MPI_Test (MPI_Request *solicitud, int *flag, MPI_Status *estado) ;
```

Además, se puede requerir que el proceso se bloquee hasta que finalice una operación, para ello se utiliza:

```
MPI_Wait (MPI_Request *solicitud, MPI_Status *estado) ;
```

Este tipo de comunicación permite solapar cómputo con comunicación. Evita overhead de manejo de buffer, pero deja en manos del programador asegurar que la comunicación se realice correctamente.

Es posible obtener información de un mensaje antes de recibirlo, utilizando:

```
MPI_Probe (int origen, int tag, MPI_Comm comunicador,
           MPI_Status *estado) ;

MPI_Iprobe (int origen, int tag, MPI_Comm comunicador, int *flag,
             MPI_Status *estado) ;
```

`MPI_Iprobe` chequea el arribo de un mensaje que cumpla con el origen y el tag. Pueden usarse los comodines mencionados anterirormente.

Todas las operaciones de comunicación vistas hasta este punto son punto a punto, es decir, solo intervienen un emisor y un receptor. A continuación, se presentarán las comunicaciones colectivas más importantes que provee MPI.

5.2.1.6. Comunicación colectiva

MPI provee un conjunto de funciones para realizar operaciones colectivas, sobre un grupo de procesos asociados a un comunicador. Todos los procesos del comunicador deben llamar a la rutina colectiva.

Una de estas operaciones es la sincronización por barrera:

```
MPI_Barrier(MPI_Comm comunicador) ;
```

Es posible desde un proceso enviar el mismo mensaje a todos los demás procesos que están en un comunicador usando **broadcast**:

```
MPI_Bcast (void *buf, int cantidad, MPI_Datatype tipoDato,
           int origen, MPI_Comm comunicador) ;
```

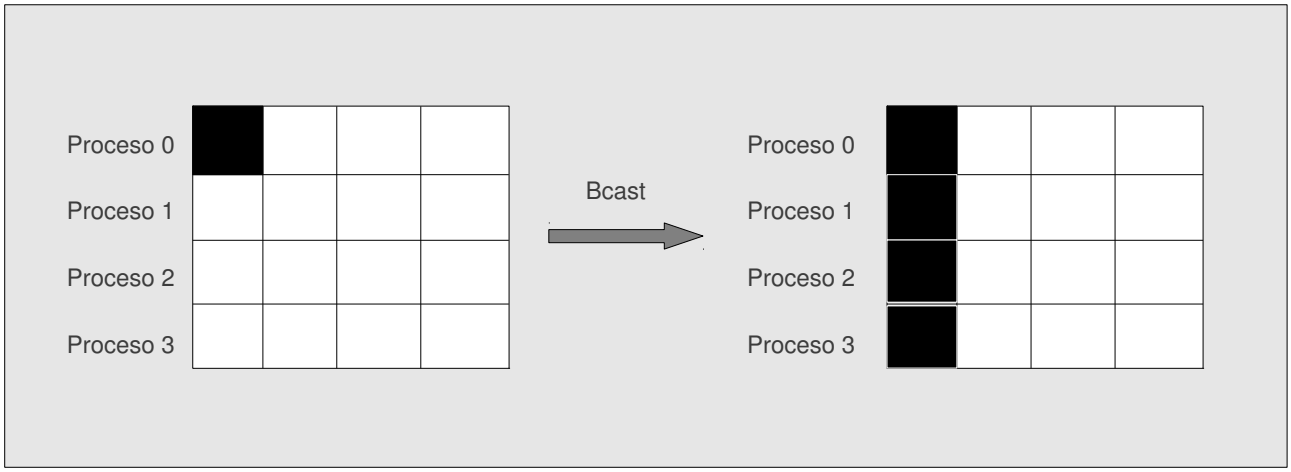


Fig. 5.5. Comunicación colectiva broadcast

Como se observa en la Fig. 5.5. el proceso emisor envía el mensaje a todos los procesos, incluso a él mismo.

Otro, tipo de comunicación colectiva es combinar los elementos enviados por cada proceso (inclusive el destino):

```
MPI_Reduce (void *sendbuf, void *recvbuf, int cantidad,
            MPI_Datatype tipoDato, MPI_Op operación,
            int destino , MPI_Comm comunicador) ;
```

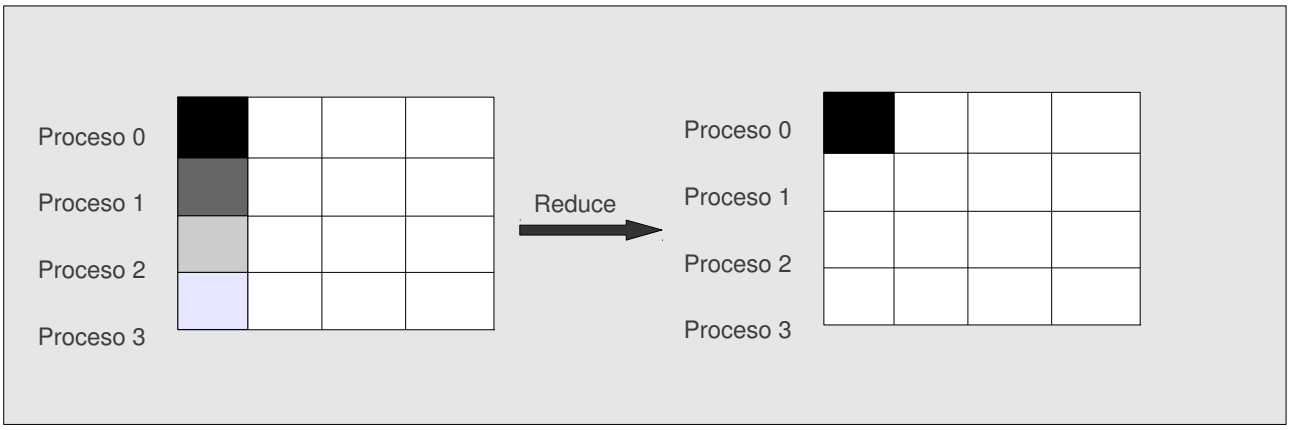


Fig. 5.6. Comunicación colectiva Reduce

Existe otro tipo de reducción en el cual el resultado de una operación es reducido en todos los procesos.

```
MPI_Allreduce (void *sendbuf, void *recvbuf, int cantidad,
               MPI_Datatype tipoDato, MPI_Op operación,
               MPI_Comm comunicador) ;
```

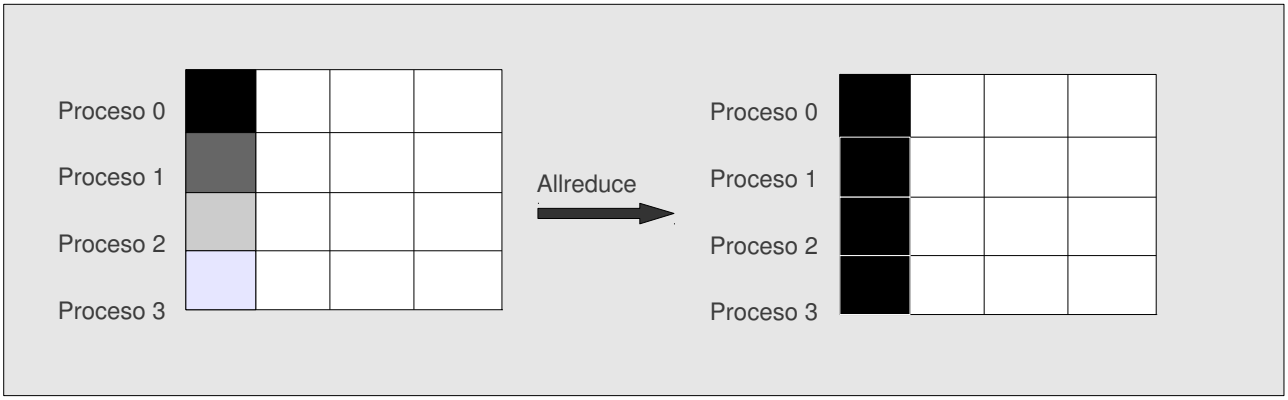


Fig. 5.7. Comunicación colectiva AllReduce

Para recolectar los datos de todos los procesos y concatenarlos en un resultado único se utiliza la operación Gather:

Los vectores de datos son iguales:

```
MPI_Gather (void *sendbuf, int cantEnvio,
            MPI_Datatype tipoDatoEnvio, void*recvbuf, int cantRec,
            MPI_Datatype tipoDatoRec, int destino,
            MPI_Comm comunicador) ;
```

Vectores de datos de distinto tamaño:

```
MPI_Gatherv (void *sendbuf, int cantEnvio,
             MPI_Datatype tipoDatoEnvio, void*recvbuf,
             int *cantsRec, int *desplazamientos,
             MPI_Datatype tipoDatoRec, int destino,
             MPI_Comm comunicador) ;
```

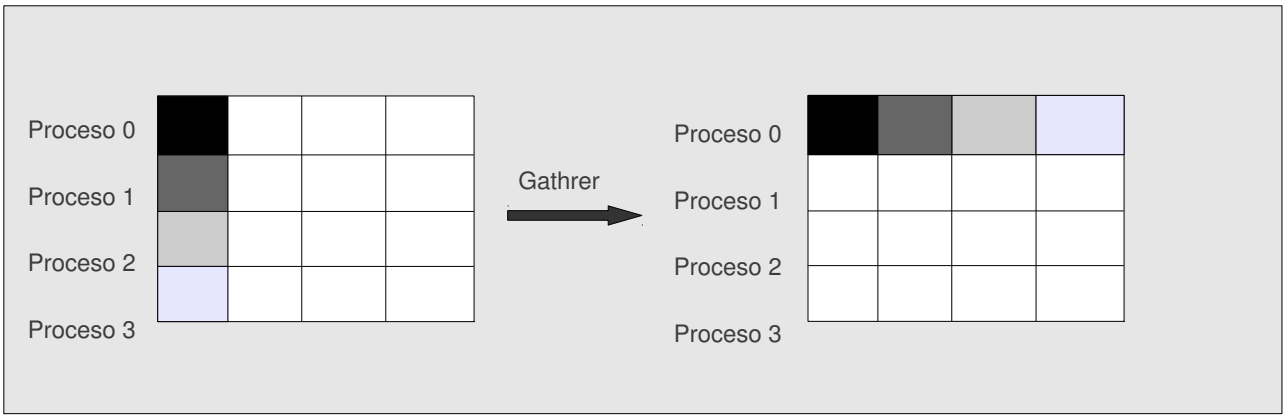


Fig. 5.8. Comunicación colectiva Gather

Al igual que el reduce, también existe la posibilidad de que todos los procesos puedan tener los datos recolectados y concatenados en orden. Para ello se utiliza:

Todos los vectores de datos de igual tamaño:

```
MPI_AllGather (void *sendbuf, int cantEnvio,
               MPI_Datatype tipoDatoEnvio, void*recvbuf,
               int cantRec, MPI_Datatype tipoDatoRec,
               MPI_Comm comunicador) ;
```

Con vectores que pueden ser de diferente tamaño:

```
MPI_Allgatherv (void *sendbuf, int cantEnvio,
                MPI_Datatype tipoDatoEnvio, void*recvbuf,
                int *cantsRec, int *desplazamientos,
                MPI_Datatype tipoDatoRec, MPI_Comm comunicador) ;
```

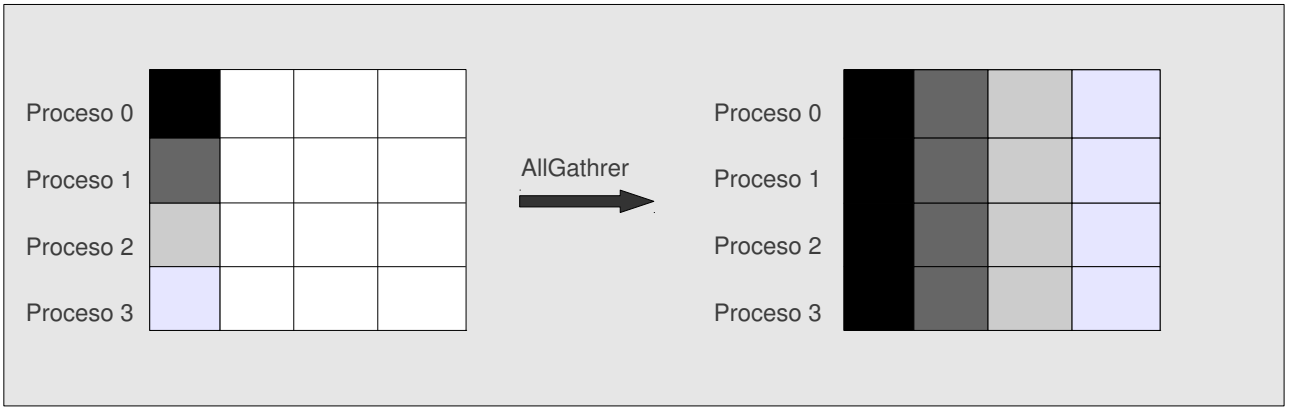


Fig. 5.9. Comunicación colectiva AllGather

La operación *scatter* reparte un vector de datos entre todos los procesos (inclusive al mismo dueño del vector).

Repartir en forma equitativa:

```
MPI_Scatter (void *sendbuf, int cantEnvio,
             MPI_Datatype tipoDatoEnvio, void*recvbuf,
             int cantRec, MPI_Datatype tipoDatoRec,
             int origen, MPI_Comm comunicador) ;
```

Darle a cada proceso diferentes cantidades de elementos:

```
MPI_Scatterv (void *sendbuf, int *cantsEnvio,
              int *desplazamientos, MPI_Datatype tipoDatoEnvio,
              void*recvbuf, int cantRec,
             MPI_Datatype tipoDatoRec,
              int origen, MPI_Comm comunicador) ;
```

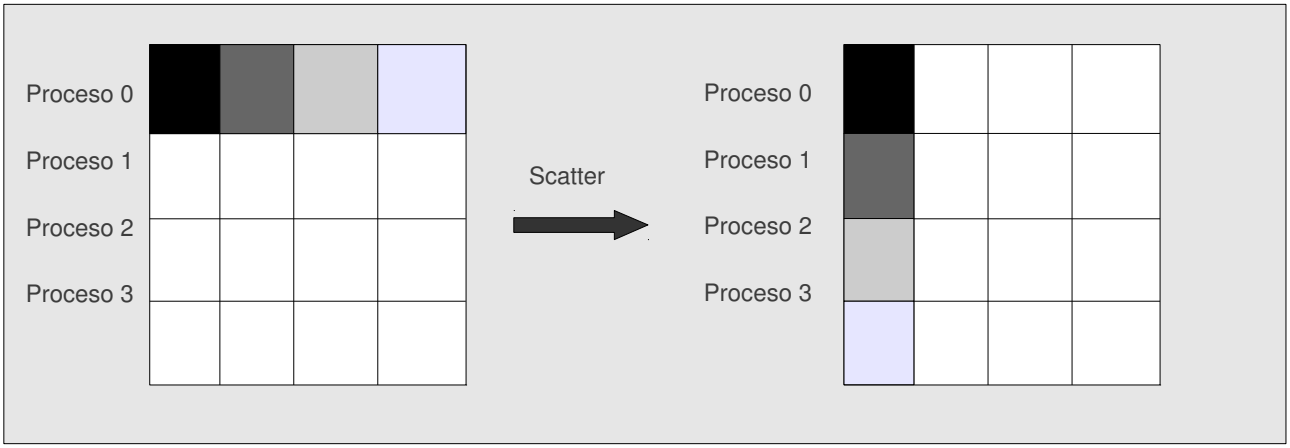


Fig. 5.10. Comunicación colectiva Scatter

5.3. Programación paralela en GPU

5.3.1. CUDA

El desafío actual, se encuentra en desarrollar aplicaciones de software paralelas que se adapten de manera transparente a la cantidad creciente de núcleos. El modelo de programación paralela CUDA fue diseñado para cumplir este desafío, y al mismo tiempo mantener una curva de aprendizaje baja para los programadores familiarizados con lenguajes de programación estándar como C [Nvi03] [Lue08] [CCCT10].

CUDA [ND10] comprende el hardware y el software para el cómputo paralelo sobre las GPUs de Nvidia para ejecutar programas con C, C++, Fortran, OpenCL, DirectCompute y otros lenguajes. CUDA preserva el modelo de los lenguajes y los extiende agregándoles una mínima lista de abstracciones para expresar paralelismo. Por lo tanto, el programador se enfoca en las características importantes del paralelismo utilizando un lenguaje familiar.

El entorno de desarrollo de CUDA crece gracias al aporte de compañías que ofrecen servicios y soluciones para las GPUs de NVIDIA. Las tarjetas gráficas que soportan CUDA son la serie GeForce 8, Quadro y Tesla. Estas pueden ser usadas en Pcs, notebooks y servidores. Las tarjetas gráficas NVIDIA son una nueva tecnología con una arquitectura de computación extremadamente multithreaded.

El lenguaje CUDA es una extensión del lenguaje C [Par10] que permite al programador definir funciones C, denominadas kernels, las cuales al ser llamadas se

ejecutan paralelizadas *n* veces en *n* threads diferentes. Esto contrasta con una función C clásica, que se ejecuta en uno solo thread cada vez que es llamada.

Nvidia introduce el concepto de thread CUDA, lo cuales, son independientes entre sí y capaces de ejecutar en paralelo, por lo que pueden ser vistos como unidades de paralelismo [Par10]. La computación intensiva o partes del problema con paralelismo de datos son implementadas a través de kernels o núcleos CUDA que es ejecutado por todos los threads que componen un bloque. Múltiples bloques pueden ser ejecutados en un mismos Procesador de Streams (SM), pero solo un bloque puede ser ejecutado a través de los diferentes SMs [FX10].

El núcleo CUDA encierra tres abstracciones clave: la jerarquía de grupos de threads, la memoria compartida y las barreras de sincronización (que el programador las ve como simples extensiones del lenguaje). Los problemas pueden dividirse en sub-problemas que se resuelven independientemente en paralelo en bloques de threads y cada sub-problema se resuelve también en paralelo de manera cooperativa entre los threads asignados. La escalabilidad es directa ya que el mismo código compilado puede correr en GPUs con diferente número de núcleos, beneficiándose del administrador o thread scheduler.

En subsiguientes secciones, se pretende introducir las operaciones básicas referentes a la programación en CUDA.

5.3.1.1. Flujo de ejecución en un programa CUDA

La ejecución paralela está expresada en la función kernel que se ejecuta en la GPU o device [Pic11] [Nvi10] [Nvi12]. El código de la función kernel se ejecuta en un solo hilo, y cada hilo ejecuta esta misma función. Por lo cual, se trata de una arquitectura de tipo SIMD (Single Instruction Multiple Data).

La función kernel puede invocar código secuencial para ser ejecutado en la CPU. El kernel debe ser configurado con el número de hilos en cada bloque y el número de hilos por grid. Para declarar un grid y un bloque de hilos en CUDA se utiliza un tipo de dato predefinido dim3, un vector de enteros que especifica las dimensiones de la grid y el bloque de hilos. En la función kernel el llamado a las variables grid y bloque son escritas de la siguiente manera: <<< grid, block >>> . A partir de esta invocación el grid y el bloque de hilos es creado dinámicamente. Los threads son planificados a nivel hardware. La función kernel retorna siempre void, y con el __global__ se indica que dicha función se ejecuta en la GPU.

__global__	es una función kernel. Se ejecuta en el dispositivo y se puede llamar sólo desde un programa que corre en la CPU o host.
__device__	es una función que se ejecuta en la placa gráfica y se puede llamar sólo desde la placa.
__host__	es una función que se ejecuta en la CPU y que sólo se puede invocar desde código que corre en la CPU.

Fig. 5.11. Calificadores de función CUDA

El paradigma CUDA provee variables construidas cuyas estructuras son muy eficientes, permitiendo acceder al identificador de un bloque de threads con la variable blockIdx que puede tomar un valor desde cero a la dimensión del grid menos uno. Para acceder a la dimensión del bloque se utiliza la variable blockDim, y gridDim para la dimensión de un grid. Cada hilo individualmente es identificado por la variable threadIdx, la cual puede tomar un valor entre cero y la dimensión del bloque menos uno. WarpSize especifica el tamaño de warp. Todas estas variables son definidas en la función kernel. El tamaño máximo permitido en al dimensión para cada grid es de 65535, las dimensiones *x*, *y* , y *z* de un bloque de hilos son 512, 512 y 64 respectivamente.

__device__	es una variable que reside en el espacio global de memoria de la placa. Es accesible desde todos los threads de la grilla y desde el host.
__shared__	es una variable que se almacena en el espacio compartido de memoria de un bloque y es accesible únicamente desde todos los threads del bloque.

Fig. 5.12. Calificadores de variables CUDA

CUDA provee de una función que sincroniza a los hilos mediante una barrera: __syncthreads() (comunicación intra-SM). Sin embargo, no existe una forma de comunicación explícita entre los threads de diferentes bloques (comunicación inter-SM) debido a la carencia de soporte de comunicación entre los SMs. Por lo cual, dicha comunicación ocurre a través de la memoria global por lo que son necesarias barreras de sincronización implementadas en el host o CPU para la terminación del kernel en ejecución y el lanzamiento de un nuevo kernel [FX10]. Como los threads son planificados por hardware, está función está implementada en hardware. Los threads esperarán en el punto de sincronización hasta que todos hayan llegando a dicho punto. La sincronización

entre threads solo es posible a nivel de bloque.

Como el kernel se ejecuta en el device, la memoria debe ser alocada en el device antes de que la función kernel sea invocada y si el kernel utiliza algún dato, este debe ser copiado desde la memoria del host a la memoria del device. La memoria del device puede ser alocada como una memoria lineal o un arreglo CUDA. La API CUDA provee funciones para alocar y desalocar memoria en el device en tiempo de ejecución: cudaMalloc(), cudaFree(), etc. Una vez ejecutada la función kernel, los datos de la memoria del divice deben ser copiados a la memoria del host. Algunas funciones provistas para ello son: cudaMemCpyToSymbol(), cudaMemCpyFromSymbol(), cudaMemCpy(), etc.

cudaMalloc(...)	ubica una cantidad especificada de bytes de memoria consecutiva y retorna un puntero a ese espacio que puede utilizarse para cualquier tipo de variable.
cudaMemcpy(...)	copia bytes de un área de memoria a otra. Pudiendo copiar desde la memoria principal de la CPU a la memoria de la placa, a la inversa y entre diferentes espacios de la memoria de la GPU.

Fig. 5.13. Manejo de memoria en CUDA

La estructura básica de un programa en CUDA seguiría los siguientes pasos:

1. Alocar memoria en el device y en el host por separado.
2. Copiar los datos desde el host al device a través de las funciones provistas por la API.
3. Ejecutar la función kernel en paralelo por cada hilo.
4. Copiar los datos resultantes desde el device al host con las funciones provistas por la API.

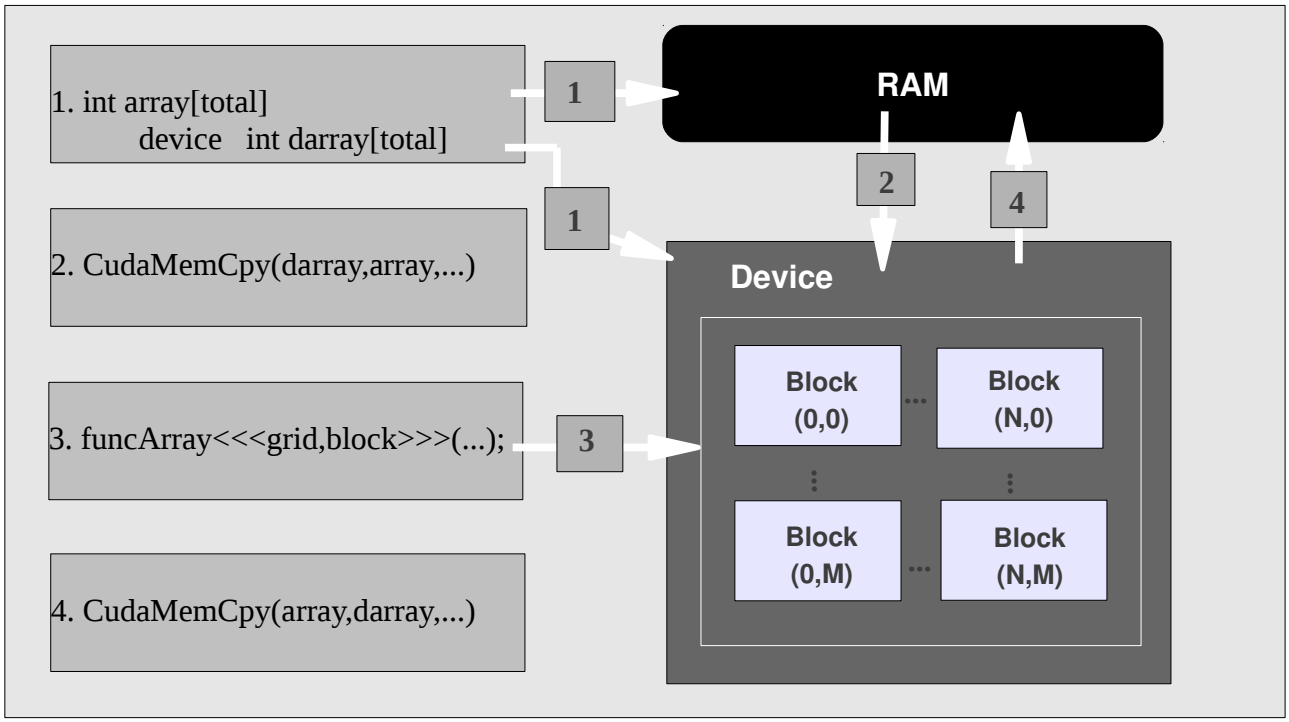


Fig. 5.14. Flujo de ejecución en un programa CUDA

En la figura anterior, se muestra un ejemplo del flujo de control de programa con CUDA. En el primer paso se declara un arreglo en el host y otro en el device. Luego, los datos son copiados desde el host al device a través de la función cudaMemCpy(). La función kernel se ejecuta en paralelo en el device y en el último paso se copian los resultados desde el device al host.

Capítulo 6

Green Computing

"Dios siempre perdona, el hombre a veces, la naturaleza no perdona jamás.."[1]

6.1. Calentamiento global

Cuando se suele hablar de Gases de Efecto Invernadero (greenhouse gas, en inglés), se lo suele asociar con terribles cambio ambientales y enfermedades. Sin embargo, estos gases son imprescindibles para la vida en la Tierra, ya que permiten que parte del calor que irradia el sol quede en la superficie terrestre. La proporción correcta de los distintos gases que conforman la atmósfera hacen que la Tierra tenga un equilibrio energético que permita el desarrollo de la vida. Los Gases de Efecto Invernadero (GEI) conforman una pequeña porción de la atmósfera, por lo que una variación en los mismos puede ocasionar grandes cambios. Cuanto más concentrados están estos gases, la temperatura promedio del planeta es mayor.

Los cambios en la composición original de la atmósfera terrestre comenzaron a producirse a partir de la Revolución Industrial, al rededor del siglo XVIII. El hombre con

1 Félix Rodriguez de la Fuente, famoso naturalista y divulgador ambientalista español, pionero en su país en la defensa de la naturaleza

todo su adelanto tecnológico permitió el avance científico e industrial por un lado, y por el otro inició la degradación ambiental del planeta. Por lo cual, el problema no son en sí mismos los GEI, sino una sobre cantidad de greenhouse gas, que causan el incremento de la temperatura global, lo que produce un desajuste en la naturaleza, convirtiendo así, lo que una vez fue beneficioso para la vida, en algo perjudicial [CS07][Guh08].

El calentamiento global es un hecho que ha comenzado a tener mayor impacto a partir del siglo XIX. En el año 2001, el Tercer Reporte de Evolución de la IPCC (Intergovernmental Panel on Climate Change), grupo formado por 1300 científicos expertos independientes de todos los países del mundo auspiciado por las Naciones Unidas, confirmó la relación entre el calentamiento global y la actividad humana, ya que hasta entonces, no había forma de establecer una influencia de la intervención humana en este fenómeno. Paralelamente nace el protocolo de Kioto el convenio que busca reducir y regular las emisiones de gases de efecto invernadero a la atmósfera. Entre estos gases de efecto invernadero se encuentra el CO_2 (Dióxido de Carbono), que se ha incrementado en un 31% desde 1750 [NU98] [NA11].

Las emisiones de CO_2, son uno de los principales problemas que enfrenta nuestro medio ambiente en la actualidad. Billones de toneladas de CO_2 son absorbidas por los océanos y los biomas. Se espera que la temperatura global del planeta aumente unos 6 ºC en el próximo siglo.

> "*Calentamiento global no solo significa que habrá un leve aumento en el promedio de las temperaturas. Cambiaría radicalmente el sistema en el que la tierra se maneja*"
> (Mark Lynas, autor de "Six Degrees").

La temperatura promedio ha aumentado entre un 0.5 a 1 ºC en estos últimos años. El aumento de 6 ºC, a la temperatura actual del planeta sería un evento totalmente catastrófico, que cambiaría el mundo tal y como lo conocemos: islas desaparecerían por el aumento del nivel de agua en los océanos, se producirían tormentas más fuertes y violentas, los terremotos y los tornados serían moneda corriente en zonas no propensas a ellos en el pasado, los desiertos serían el paisaje dominante...[NG08].

Los cinco sectores que constituyen los mayores generadores de emisiones de CO_2 a través del consumo de combustibles fósiles son: la generación de electricidad, la industria, el transporte, el comercio y las residencias [US11]. Según un informe de la Energy Information Administration, el 98% de todas las emisiones de CO_2 son causadas por el consumo energético [EI07]. Muchas organizaciones hablan sobre su deseo de cooperar de una manera "green", y a muchas otras se les aplica impuestos sobre su producción en busca de la reducción del consumo de energía. Estos son los llamados

impuestos ambientales, definidos como un instrumento de mercado para las políticas ambientales. Estudios realizados [BP02], demuestran que en el año 2007, la emisión de gases a la atmósfera aumentó un 16% con respecto a 1990. Para conseguir reducir significativamente dichas emisiones, se precisa una reducción del 80% durante los próximos 30 años.

Sin lugar a duda, la reducción del consumo de energía incide de manera directa sobre el estilo de vida de la población y sobre el estado financiero de las organizaciones. Claramente, son estas los mayores centros de consumo de energía contribuyendo a la emisión de greenhouse gas. Por lo cual, las mismas están utilizando la siguiente ecuación:

Reducir el consumo de la energía =
reducir la emisión de greenhouse gas =
reducir el costo operacional de los centros de datos y negocios [Cur08]

Sin embargo, es cierto que son pocos los que están dispuesto a sacrificar su calidad de vida presente para mejorar la calidad de vida de las generaciones futuras. Los estudiosos del tema, afirman que los efectos del cambio climático son dependientes de los intereses políticos y económicos. Lo cual explica, por qué algunos países no quieren comprometerse con acuerdos tales como el Protocolo de Kioto, ya que si se reduce la producción debido la disminución de actividades para reducir las emisiones de GEI, las economías de estos se verían fuertemente afectadas. Las empresas como una forma de igualar o superar a sus rivales, y cumplir con la demanda del mercado, reemplaza el trabajo vivo por máquinas más productivas, esto es y ha sido, el capitalismo. Sin embargo, salvar el clima necesita de la reducción de la sobreproducción y el sobreconsumo de bienes materiales [Tan09]. Lo que estaría íntimamente relacionado a un cambio de la calidad de vida actual de la población.

> "*El mundo es un lugar peligroso. No por causa de los que hacen el mal, sino por aquellos que no hacen nada por evitarlo.*"
> (Albert Einstein)

La BBC en Español reportó en el 2007, las afirmaciones de la organización británica Christian Aid, por medio de su portavoz Rachel Baird: "*el cambio climático agudizará la crisis de la migración global y se creará inestabilidad política e incluso potenciales guerras ya que millones de personas tendrían que encontrar otros lugares*

para vivir y trabajar ".

6.2. Green Computing

Sin duda alguna, la tecnología ha permitido mejorar la calidad de vida mundial. Sin embargo, el uso inadecuado de la misma puede ser mortal. Desde nuestra posición, es nuestra responsabilidad proveer no solo un avance tecnológico sino un uso responsable del mismo. Ya sea desde el hardware o el software, debemos comenzar a pensar en reducir gastos innecesarios en el consumo energético.

Green Computing comprende el *uso eficiente de los recursos*. Su objetivo es reducir el uso de materiales peligrosos, maximizar la eficiencia de la energía durante el ciclo de vida de producción, y promover el reciclado o biodegrabilidad de los productos y desechos fabriles. En promedio una PC, requiere 85 watts aun en estado ocioso con el monitor apagado. Produciendo 1500 pounds (libras) de CO_2 anualmente a la atmósfera. Un árbol absorbe entre 3 y 15 pounds por año, lo que significaría que se necesitan más de 500 árboles para absorber la cantidad de CO_2 que produce una computadora en un año. Aun cuando los dispositivos están apagados, siguen consumiendo energía en estado standby, por lo que se los suele denominar "vampire energy loss" . Estos representan entre el 5% y el 8% de la electricidad consumida durante un año por una residencia. A escala mundial, la energía consumida en estado standby representa el 1% de las emisiones de CO_2 del mundo [SE08].

Los consumidores no toman en cuenta el impacto ecológico a la hora de comprar sus computadoras, solo prestan atención a la velocidad de sus prestaciones y al precio. Sin embargo, sabemos que a mayor velocidad de procesamiento se requiere mayor poder energético, lo cual nos pone frente al problema de la disipación del calor, que por su lado, necesita más energía eléctrica para mantener al procesador en la temperatura normal de trabajo. Los diseñadores de hardware ya han planeado varias estrategias para colaborar con el medio ambiente desde la fabricación del equipo hasta su reciclado, como por ejemplo, remplazar el plástico por bioplástico (con polímeros a base de compuestos vegetales), ya que requieren menos petróleo y menos energía para su producción, o el uso de memorias flash, en lugar de discos rígidos cuyas partes mecánicas consumen mayor cantidad de energía, etc [Ver07].

Un gran avance se ha realizado con el paso del monoprocesador a los procesadores multicore, ya que estos últimos suelen ser más simples, por lo que hacen un uso más eficiente de la energía. Las grandes compañías de procesadores (Intel y AMD), han tomado conciencia sobre esta necesidad del uso eficiente de los recursos y la producción de los mismos disminuyendo no solo la producción de CO_2, sino también,

otras toxinas como el plomo, el cadmio, el mercurio, entre otras, que concentrados en sangre pueden causar varias enfermedades en el corazón, riñones, intestinos, y órganos reproductores [Nic10] [Amd11]. La evolución de las CPUs a las APUs (Unidades de Procesamiento de Aplicaciones), que integran en un mismo chip la CPU y la GPU (Unidad de Procesamiento Gráfico), permite reducir el consumo energético permitiendo la ejecución de aplicaciones de alta calidad gráfica 2D y 3D. Todos estos son ejemplos de los esfuerzos que se están realizando desde el hardware para contribuir con el medio ambiente.

Green Computing no solo debe comprender al hardware sino también al software. Dentro de los factores influyentes en el desarrollo de aplicaciones, se suma la eficiencia energética. Estudiar el consumo energético de los algoritmos que desarrollamos, debe formar parte de los punto considerados a la hora de ejecutar nuestras aplicaciones. En el campo del Computación de Altas Prestaciones (HPC, High Performance Computing), se están realizando investigaciones orientadas no solo a disminuir la energía consumida, sino también en un sistema de gestión de energía que dada una aplicación y un plataforma HPC presente alternativas de ejecución dependientes de: el consumo energético, la potencia máxima (capacidad de la infraestructura eléctrica y el equipo de refrigeración) y el rendimiento [BUSRL11]. En HPC la eficiencia energética es un factor limitante. La cantidad de energía necesaria para procesar las grandes cantidades de datos de las aplicaciones que hoy se desarrollan se ha convertido en un problema importante al cual cada vez se le presta más atención.

A la hora de implementar nuestras aplicaciones, tenemos en cuenta varios factores tales como: fiabilidad, escalabilidad, portabilidad, eficiencia, etc. Sin embargo, el factor ambiental o la eficiencia del consumo energético no se toma en cuenta. Contar con equipos que realizan un menor consumo de energía, pero utilizarlos mal, nos lleva nuevamente al problema inicial de una cantidad de emisiones injustificadas de dióxido de carbono a la atmósfera. Un ejemplo son aquellas aplicaciones que se ejecutan sobre procesadores multicore pero que solo hacen uso eficiente de un solo núcleo. Este es el caso de muchas aplicaciones que fueron desarrolladas para monoprocesadores y que ahora se ejecutan en computadoras multicore. Aunque también es típico el caso de aplicaciones desarrolladas para arquitecturas multicore que no hacen uso eficiente de los recursos ni las capacidades del hardware [FG08].

Dos consideraciones más se suman al consumo energético, además de la eficiencia energética. Una es la escalabilidad de los sistemas y otra el precio de la energía eléctrica. La gran cantidad de energía consumida reduce la escalabilidad de los sistemas de cómputo. Los sistemas no escalables no son sistemas útiles a largo plazo. Ahora no solo se debe prever la escalabilidad del problema y la arquitectura, sino también si el aumento de cualquiera de los anteriores influirá de tal manera que el consumo final

supere la cantidad de electricidad que se nos puede suministrar, así como también el presupuesto económico disponible para tales gastos.

El agotamiento de los combustibles fósiles nos ha sumido en una crisis energética en la cual el precio de la misma a venido aumentado a lo largo de los años. Si bien en la actualidad se utilizan una variedad de fuentes de producción eléctrica, la más utilizada a lo largo del tiempo han sido los combustibles fósiles. En el caso del petróleo, el precio del mismo está sujeto a condiciones geopolíticas. Por ejemplo, en 1980, se produjo una paralización de la actividad de muchos pozos petrolíferos como consecuencia del inicio de la guerra entre Irán e Iraq. En 2008, se alcanzó el máximo de consumo debido a la demanda energética de China e India. La producción petrolíferas no puede satisfacer toda la demanda, sin el descubrimiento de nuevos yacimientos las reservas mundiales se agotarán en 35 años aproximadamente.

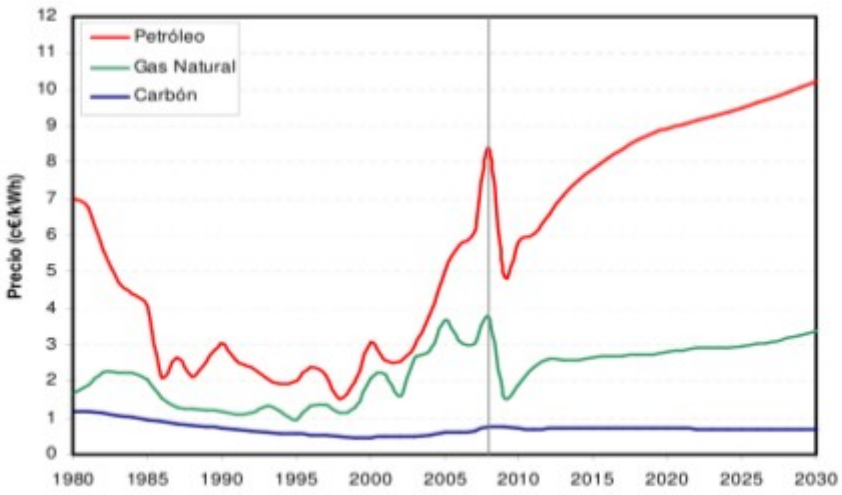


Fig. 6.1. Evolución de la historia del precio del combustible y previsiones del futuro

En la Fig. 6.1. se observa la evolución del precio del petróleo desde 1980, y las previsiones de los próximos 20 años. El precio de la electricidad mediante dichas fuentes, el precio de la electricidad aumentará a nivel mundial en el futuro. Por lo cual se han propuesto fuentes alternativas como lo son la eólica, hidráulica, y la fotovoltaica. A pesar de ello, estas requieren de zonas específicas y suelen ser más caras de producir [Lor10]. Es por ello, que se sigue utilizando mayormente los combustibles fósiles. Sin embargo, como se mencionó anteriormente, el uso de combustibles fósiles es uno de los principales contaminantes.

"*Para enfrentarnos a esta crisis global es preciso partir de una comprensión correcta de nuestras actuales formas de utilizar y afectar a la naturaleza y conocer con más detalle los efectos que nuestras acciones ejercen sobre los ecosistemas y los mecanismos de funcionamiento global de la naturaleza".*

José Antonio Pascual Trillo [Tri07].

Ya sea por reducir la contaminación ambiental, como conseguir escalabilidad de los sistemas de cómputo, o disminuir los costos económicos en el consumo energético, Green Computing está instalado en nuestro ámbito, y ha de marcar una nueva era en la manufactura hardware como en la producción del software.

"Pero las estadísticas confiesan. Los datos ocultos bajo el palabrerío revelan que el veinte por ciento de la humanidad comete el ochenta por ciento de las agresiones contra la naturaleza, crimen que los asesinos llaman suicidio, y es la humanidad entera quien paga las consecuencias de la degradación de la tierra, la intoxicación del aire, el envenenamiento del agua, el enloquecimiento del clima y la dilapidación de los recursos naturales no renovables."

Eduardo Galeano

6.3. Medición del consumo energético

Para satisfacer completamente cualquier experimento científico, o para comparar resultados de un experimento con respecto a otros, deben utilizarse formas estándares de medición. En el caso, de la medición del consumo energético, se debe medir la cantidad de watts por hora que consume la aplicación que se está ejecutando sobre una arquitectura en particular. Para entender claramente qué es lo que se está midiendo, se deben introducir algunos términos y conceptos básicos de física, electricidad y electrónica, los cuales, serán desarrollados en las siguientes secciones.

6.3.1. Principios fundamentales de la física

La Física es una de las ciencias que más ha contribuido al conocimiento de la electricidad y la electrónica. Gracias a la física hoy podemos conocer principios fundamentales de la estructura atómica de la materia.

La estructura básica de todos los materiales que se ven o usan diariamente concuerdan con un modelo conocido como estructura atómica de la materia. La *materia* es cualquier cosa que tiene masa y ocupa un lugar en el espacio, pudiéndose presentar en diversos estados: líquido, gaseoso, o sólido.

La partícula más pequeña que puede reducirse de un compuesto (sustancia que no puede ser cambiada por procedimientos químicos y no se encuentra combinada o mezclada con otros compuestos) sin perder sus características originales es un átomo. De forma similar al sistema solar, el átomo consiste de un cuerpo central relativamente grande con pequeños cuerpos girando en órbitas alrededor de él. El cuerpo central se lo denomina núcleo, y las partículas que giran alrededor se los llama electrones. El núcleo está formado de partículas con carga positiva denominadas protones, mientras que los electrones tienen carga negativa. La palabra carga implica una fuerza potencial. Cuando un átomo determinado es neutro, o balanceado, las cargas negativas y positivas se encuentran equilibradas, y la carga neta del átomo es cero.

carga de electrones + carga de protones = carga del átomo (1)

Si por medio de una fuerza externa se expulsa un electrón de la órbita al exterior del átomo, el átomo deja de ser neutro. Por ejemplo, si se le extrae un electrón al átomo de cobre, el átomo de cobre se transforma en un *cuerpo positivo* con carga neta +1, debido a la falta de un electrón para hacerlo neutro.

Si la carga del átomo de cobre era: (-29) + (29) = 0, según (1). Con un electrón menos se tendría: (-28) + (29) = +1. De igual manera, sucede si por una fuerza exterior se obliga a un electrón a entrar a la órbita del átomo, por lo cual, se dice que el átomo se convierte en un *cuerpo negativo* con carga neta -1. Siguiendo el ejemplo del átomo de cobre: (-30) + (29) = -1.

Cuando un átomo se convierte en un cuerpo positivo, significa que un electrón fue expulsado de su órbita por un fuerza externa que lo convirtió de esta forma en un *electrón libre*. Los electrones libres se encuentran moviéndose entre los átomos del material. La fuerza externa que origina que el electrón sea liberado, da al electrón libre movimiento y por lo tanto velocidad. Entonces, la aplicación de una fuerza externa que hace que los electrones de un material se desplacen de un átomo a otro produce un flujo de cargas

eléctricas negativas, es decir, *corriente eléctrica*.

6.3.2. Principios fundamentales de la Corriente Continua

La corriente eléctrica se clasifica en dos tipos generales [PF70]: la corriente alterna (C.A.) y la corriente continua (C.C.). Ambas corrientes mantienen su importancia en todas las clases de artefactos eléctricos y electrónicos. Como su nombre lo indica, la corriente alterna es un tipo de corriente eléctrica que fluye primeramente en una dirección durante cierto tiempo, y luego invierte su dirección y fluye en sentido contrario durante un período igual de tiempo. La corriente continua, cuando alcanza su valor máximo en un corto período de tiempo, se mantiene en dicha magnitud mientras el circuito se halle cerrado.

De ambos tipos, fue la C.C. la que primero fue utilizada en forma amplia, ya que fue la primera de la que se conocieron sus características. Sin embargo, la C.C. tiene un número limitado de aplicaciones: máquinas eléctricas, baterías de acumuladores de C.C., y circuitos electrónicos.

Cuando las máquinas eléctricas se aplicaron en la práctica por primera vez, se pensó que la C.C. se podría aprovechar más fácilmente que la C.A.. Sin embargo, pronto se observó que la C.C. tiene ciertas desventajas en relación con la C.A., entre otras, que no se podía transmitir a largas distancias porque se experimentaban pérdidas elevadas de energía. Si se transmite desde el punto de origen al consumo, la resistencia de los conductores entre ambos puntos ocasionan la pérdida, bajo forma de calor, de gran parte de la energía de la C.C. antes de llegar al destino. La corriente que circula por un alambre está limitada por el diámetro del mismo, por lo tanto, si la corriente sobrepasa ese límite, se produce un calentamiento excesivo que origina la pérdida antes mencionada. La C.A., puede transmitirse a grandes distancias sin pérdidas apreciables. La C.A. se transmite por líneas de alta tensión y larga distancia.

La corriente C.C. es el flujo o movimiento de electrones por un circuito en una sola dirección. La dirección del flujo de la corriente es la que va desde el terminal negativo al positivo.

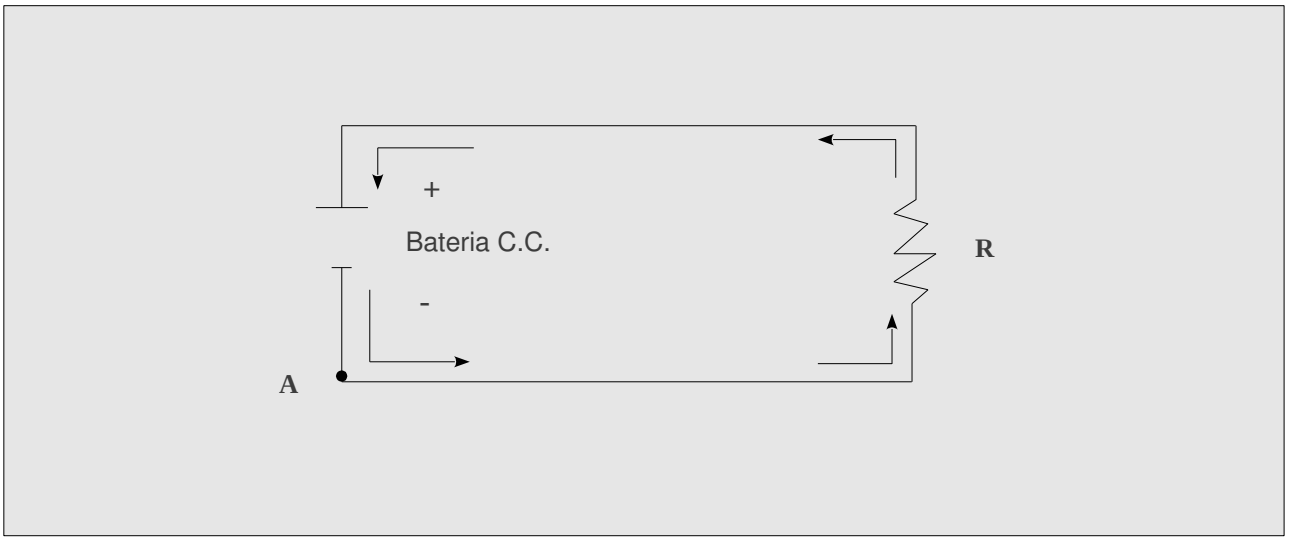


Fig. 6.1. Circuito simple C.C.

La representación gráfica de la C.C. es una curva como se observa en la Fig. 6.2., en la que la tensión y la corriente en el circuito de C.C. permanecen constantes mientras haya flujo de electrones, es decir, que cuando se conecta una carga resistiva a los terminales de una fuente C.C., y se mide la corriente y la tensión el circuito a intervalos regulares de tiempo, se encontrará que sus valores permanecen constantes.

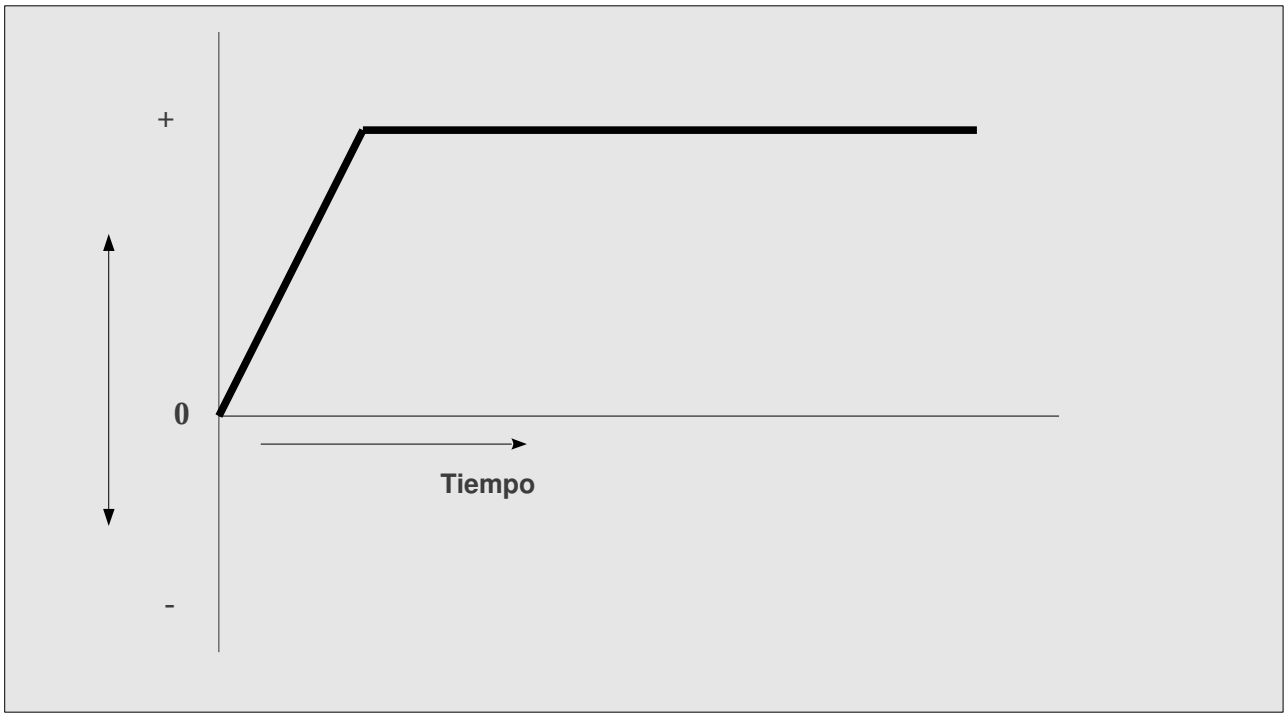


Fig. 6.2. Representación gráfica de la Corriente Continua

6.3.3. Unidades eléctricas fundamentales

Tres magnitudes invisibles y fundamentales están presentes en todo circuito eléctrico: la tensión, la corriente y la resistencia. A continuación, se presentan las unidades fundamentales que permiten la medición de estas magnitudes.

6.3.3.1. Coulomb

Es la unidad de carga eléctrica. Esta unidad, que no puede derivarse de las unidades de la mecánica, fue originalmente denominada Coulomb (término castellanizado a culombio, cuyo símbolo es C) en honor a Charles-Augustin de Coulomb, primero que midió directamente la fuerza entre cargas eléctricas. Debido a la gran dificultad de medir directamente las cargas eléctricas con precisión, se ha tomado como unidad básica la unidad de corriente eléctrica, que en el Sistema Internacional de Unidades es el amperio. La unidad de carga resulta entonces una unidad derivada, que se define como la cantidad de carga eléctrica que fluye durante 1 segundo a través de la sección de un conductor que transporta una intensidad constante de corriente eléctrica de 1 amperio:

$$C = A \cdot s \qquad (2)$$

Además, un coulomb = $6{,}28 \times 10^{18}$ electrones.

6.3.3.2. Joule

Un joule o julio es el trabajo necesario para mover una carga eléctrica de un coulomb a través de una tesión (diferencia de potencial) de un voltio. Es decir, el producto de un voltio por un coulomb. El julio es una unidad de energía muy pequeña para la vida corriente. Aproximadamente, un julio es la cantidad de energía necesaria para levantar 1 kg una altura de 10 cm en la superficie terrestre. Una patada de un deportista puede tener una energía de unos 200 J; una lamparita de bajo consumo de 20 W durante 8 horas gasta unos 600.000 J; y el consumo eléctrico de una familia media durante un mes puede ser de 1.000.000.000 J (unos 278 kWh). Por eso es más frecuente utilizar la unidad kWh (kilovatio hora), en lugar del MJ (megajoule) o el GJ (gigajoule), como debería hacerse.

6.3.3.3. Volt

Es la unidad de medida de potencial eléctrico. Se dice que entre dos puntos existe una diferencia de potencial de un volt cuando, al circular un coulomb entre dos puntos, produce un movimiento de electrones de un joule a través de un conductor entre ambos puntos.

$$1 \text{ volt} = \frac{1 \text{ joule}}{1 \text{ coulomb}} \qquad (3)$$

La diferencia de potencial, la carga eléctrica y el volt, son modos de expresar la fuerza producida entre cuerpos cargados eléctricamente. Esta fuerza puede llamarse también *fuerza electromotriz* o *tensión*. Hay muchos métodos para producir tensión. Algunos de estos métodos son: fricción, energía calorífica, energía luminosa, energía química y energía mecánica. La energía química y la mecánica es la que se utilizan como principales medios para producir energía eléctrica en forma constante para mantener un flujo continuo de electrones para el funcionamiento de aparatos eléctricos.

6.3.3.4. Ampere

El ampere es la unidad de medida eléctrica que mide la cantidad de flujo de electrones que fluye a través de un material o medio. Se dice que la intensidad de flujo de corriente en un conductor es de un ampere, cuando por un punto del conductor, fluye un coulomb por segundo. Expresando la intensidad de flujo la ecuación de la misma es:

$$I = \frac{Q}{T} \qquad (4)$$

donde

I = corriente en ampere
Q = cantidad de carga eléctrica
T = tiempo en segundos

En condiciones normales los electrones libres en un conductor no fluyen en ninguna dirección especial; esta actividad de los electrones es una acción casual, desordenada, donde los electrones son liberados por efecto de la temperatura ambiente, y

simplemente saltan de un átomo a otro, sin rumbo determinado. Sin embargo, cuando un conductor esta conectado a los terminales de una fuente de tensión, esta fuerza producirá un movimiento de los electrones libres, desde el terminal negativo de la fuente hasta el terminal positivo.

El movimiento de los electrones es el resultado del movimiento al azar más el movimiento producido por la tensión aplicada. Como resultado, todos los electrones simultáneamente son guiados dentro del conductor por acción de una tensión sostenida. Las colisiones entre ellos en el conductor contrarresta el incremento de velocidad de los mismos. Se ha determinado que la velocidad de desplazamiento de la corriente de electrones correspondiente al ampere alcanza aproximadamente 1 cm por segundo, según las dimensiones del conductor, el material y otros factores.

Para lograr que la energía eléctrica pueda recorrer muchos kilómetros de distancia para suministrarla, debe producirse los siguientes movimientos de los electrones: cuando se cierra la llave del circuito, aparece una tensión. El terminal positivo atrae electrones dejando con déficit de electrones al punto A (como se observa en la Fig. 6.3.). entonces, el punto A atrae electrones del punto B, y este mismo efecto se repite a los largo del conductor, C a D y E a F. Casi en el mismo instante en que el terminal positivo atrae un electrón del punto A, el terminal negativo suministra un electrón al punto F. De esta manera, pese a que los electrones se mueven a una pequeña velocidad, el efecto de los cambios de posiciones de los electrones se propaga a lo largo del conductor casi instantáneamente. La velocidad de propagación es aproximadamente la de la luz (300.000 km por segundo).

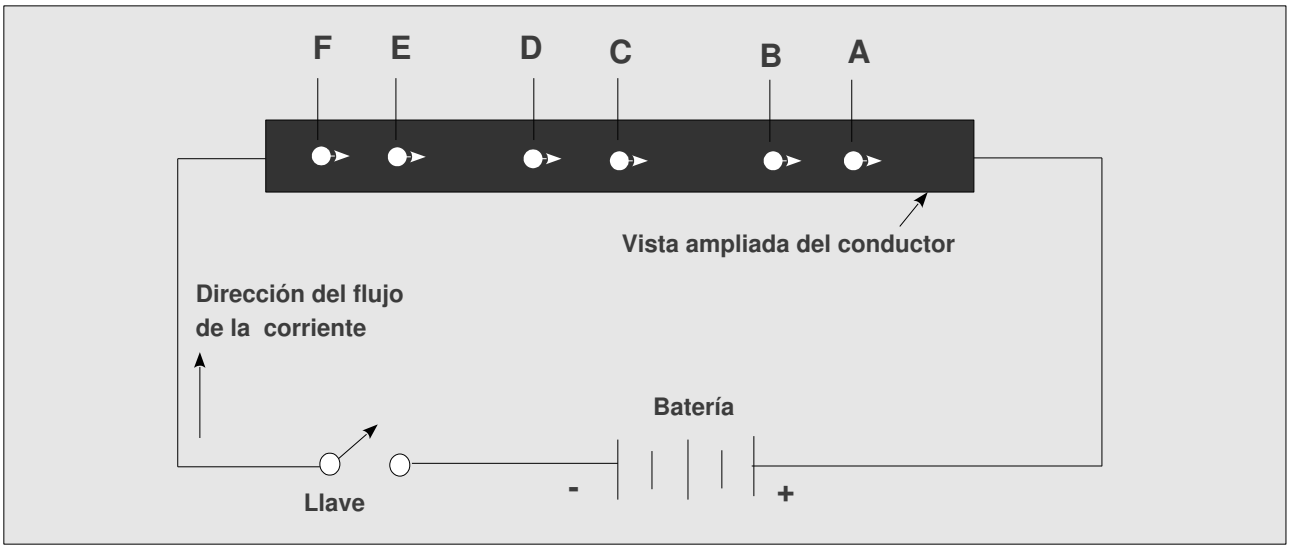


Fig. 6.3. Movimiento resultante de electrones a través de un conductor

6.3.3.5. Watt o Vatio

El vatio o watt [Sap53] [RAE12], es la unidad de potencia de energía. Se puede decir que la potencia es el ritmo al que se usa (o genera) la energía o se realiza trabajo. la diferencia de energía potencial entre dos puntos se le denomina voltaje. Esta tensión puede ser vista como si fuera una "presión eléctrica" debido a que cuando la presión es uniforme no existe circulación de cargas y cuando dicha "presión" varía se crea un campo eléctrico que a su vez genera fuerzas en las cargas eléctricas. Matemáticamente, la diferencia de potencial eléctrico entre dos puntos *A* y *B* es la integral de línea del campo eléctrico:

$$V(A) - V(B) = - \int_{B}^{A} \vec{E} \quad \vec{dt} \qquad (5)$$

Siendo $\vec{E}$ el campo eléctrico.

Generalmente se definen los potenciales referidos a un punto inicial dado. A veces se escoge uno situado infinitamente lejos de cualquier carga eléctrica. Cuando no hay campos magnéticos variables, el valor del potencial no depende de la trayectoria usada para calcularlo, sino únicamente de sus puntos inicial y final. Se dice entonces que el campo eléctrico es *conservativo*.

Otra de las formas de expresar la tensión entre dos puntos, es en función de la intensidad de corriente y la resistencia existentes entre ellos. Así se obtiene uno de los enunciados de la ley de Ohm:

$$V = I \cdot R \qquad (6)$$

donde

I = corriente eléctrica

R = resistencia

En el caso de campos no estacionarios el campo eléctrico no es conservativo y la integral de línea del campo eléctrico contiene efectos provenientes de los campos magnéticos variables inducidos o aplicados, que corresponden a una fuerza electromotriz inducida, que también se mide en voltios. Entonces:

$$Watt = I \cdot V \qquad (7)$$

siendo

I = corriente eléctrica

V = volts

La fuerza electromotriz, cuyo origen es la inyección de energía externa al circuito, permite mantener una diferencia de potencial entre dos puntos de un circuito abierto o de producir una corriente eléctrica en un circuito cerrado. Esta energía puede representarse por un campo de origen externo cuya circulación (integral de línea sobre una trayectoria cerrada *C*) define la fuerza electromotriz del generador. Esta expresión corresponde al trabajo que el generador realiza para forzar el paso por su interior de una carga, del polo negativo al positivo (es decir, en contra de las fuerzas eléctricas), dividido por el valor de dicha carga. El trabajo así realizado puede tener origen mecánico (dínamo), químico (batería), térmico (efecto termoeléctrico) o de otro tipo.

Finalmente, también se puede definir al *vatio* como un julio por segundo. Por ejemplo, si una lámpara de 100 vatios está encendida durante una hora, la energía consumida es de 100 vatios-hora (W·h) o 0,1 kilovatio-hora (*kW·h*) o (60 × 60 × 100) 360.000 julios (*J*). La capacidad o potencia de una central energética se mide en vatios, pero la energía generada anualmente se medirá en vatios-hora (W·h), kilovatios-hora (kW·h), megavatios-hora (MW·h), gigavatios-año (GW·año).

6.3.4. Instrumentos de medición

6.3.4.1. Amperímetro

Un amperímetro es un instrumento que sirve para medir la intensidad de corriente que está circulando por un circuito eléctrico. Para efectuar la medida de la intensidad de la corriente circulante el amperímetro ha de colocarse *en serie*, para que sea atravesado por dicha corriente. Esto lleva a que el amperímetro debe poseer una resistencia interna lo más pequeña posible, a fin de que no produzca una caída de tensión apreciable. Para ello, en el caso de instrumentos basados en los efectos electromagnéticos de la corriente eléctrica, están dotados de bobinas de hilo grueso y con pocas espiras.

6.3.4.2. Osciloscopio

Se denomina osciloscopio a un instrumento de medición electrónico para la representación gráfica de señales eléctricas que pueden variar en el tiempo, que permite visualizar fenómenos transitorios así como formas de ondas en circuitos eléctricos y electrónicos y mediante su análisis se puede diagnosticar con facilidad cuáles son los problemas del funcionamiento de un determinado circuito. Es uno de los instrumentos de medida y verificación eléctrica más versátiles que existen y se utiliza en una gran cantidad de aplicaciones técnicas. Un osciloscopio puede medir un gran número de fenómenos.

El osciloscopio presenta los valores de las señales eléctricas en forma de coordenadas en una pantalla, en la que normalmente el eje X (horizontal) representa tiempos y el eje Y (vertical) representa tensiones. La imagen así obtenida se denomina oscilograma. Suelen incluir otra entrada, llamada "eje Z" que controla la luminosidad del haz, permitiendo resaltar o apagar algunos segmentos de la traza. El funcionamiento del osciloscopio está basado en la posibilidad de desviar un haz de electrones por medio de la creación de campos eléctricos y magnéticos.

Capítulo 7

Problema de aplicación: N Body

"Una inteligencia que en un momento dado conociera todas las fuerzas que actúan en la naturaleza y la posición de cada objeto en el universo -si estuviera dotada de un cerebro lo suficientemente capaz para todos los cálculos necesarios - podría describir con una sola fórmula los movimientos de los grandes cuerpos celestes y la de los átomos más pequeños. A tal inteligencia nada le sería incierto, tanto el futuro como el pasado, sería como un libro abierto." [1]

7.1. Descripción del problema N Body

El problema N Body es importante en cosmología y astronomía, permitiéndole a los científicos ser capaces de visualizar y comprender el comportamiento de las galaxias, el nacimiento de sistemas planetarios, y la evolución del Universo, entre otros [CGSP04].

1 Pierre-Simon Laplace, matemático, físico, estadístico y astrónomo francés

Sin embargo el problema de los N Body caracteriza una variedad de problemas, tales como la atracción electrostática de las partículas cargadas electricamente, entre otros. Carente de una solución analítica, es uno de los problemas más estudiados desde que aparecieron las primeras computadoras [Bru11].

Por la similitud que existe entre la atracción entre los cuerpos astronómicos y las partículas electrostáticas, el problema de los N body, suele ser usado para simular la evolución de cualquiera de estos dos casos de estudio. A continuación se realizará una breve introducción a ambas problemáticas para entender lo que se pretende simular con el problema de los N Body.

7.1.1. Fuerza de atracción electrostática

Desde la antigüedad [PF70] se conoce el comportamiento de los cuerpos cargados eléctricamente. Los antiguos griegos observaron que se producía una fuerza invisible cuando se frotaba un material amarillento y resinoso llamado ámbar con un trozo de piel. Esta fuerza hacía que porciones pequeñas de papel y raspaduras de madera fueran atraídas por el ámbar. Más tarde se descubrió que muchas otras sustancias, tales como el vidrio frotado con seda, o la goma frotada con piel o la lana, producían el mismo efecto de atracción que el ámbar sobre objetos no metálicos livianos.

La electricidad estática trata las acciones y reacciones entre cuerpos cargados eléctricamente. La ley de electrostática dice que: *las cargas distintas se atraen y las cargas iguales se repelen.* Esta atracción o repulsión de los cuerpos cargados eléctricamente se debe a una fuerza invisible denominada campo electrostático que rodea a los cuerpos cargados.

El físico francés Charles Coulomb, estableció a través de experimentos lo que hoy se conoce como *Ley de fuerza*[2], que dice: *"la fuerza con la que se atraen o repelen dos cuerpos eléctricamente cargados es proporcional a la magnitud de las cargas e inversamente proporcional al cuadrado de la distancia existente entre ellas."* . La expresión matemática para dicha ley es la siguiente:

$$F = \frac{Q_1 \cdot Q_2}{d^2} \qquad (7.1)$$

2 O Ley de Coulomb

donde

F = fuerza eléctrica

Q = magnitud de carga

d = distancia entre los cuerpos

7.1.2. Fuerza de atracción gravitacional

Newton descubrió que la gravitación es un fenómeno universal que no se restringe a nuestro planeta.

Antes de Newton, nadie había sospechado que la gravitación es un fenómeno inherente a todos los cuerpos del Universo. Muy por el contrario, durante la Edad Media y aun hasta tiempos de Newton, se aceptaba el dogma de que los fenómenos terrestres y los fenómenos celestes son de naturaleza completamente distinta. La gravitación se interpretaba como una tendencia de los cuerpos a ocupar su "lugar natural", que es el centro de la Tierra.

Es justo mencionar que, antes de Newton, el intento más serio que hubo para explicar el movimiento de los planetas se debe al científico inglés Robert Hooke, contemporáneo de Newton. En 1674, Hooke ya había escrito:

...todos los cuerpos celestes ejercen una atracción o poder gravitacional hacia sus centros, por lo que atraen, no sólo, sus propias partes evitando que se escapen de ellos, como vemos que lo hace la Tierra, sino también atraen todos los cuerpos celestes que se encuentran dentro de sus esferas de actividad.

Sin esa atracción, prosigue Hooke:

...los cuerpos celestes se moverían en línea recta, pero ese poder gravitacional modifica sus trayectorias y los fuerza a moverse en círculos, elipses o alguna otra curva.

Todos los cuerpos en el Universo se atraen entre sí gracias a la fuerza de atracción gravitacional. Newton descubrió que la fuerza de atracción entre dos cuerpos es proporcional a sus masas e inversamente proporcional al cuadrado de la distancia que los separa.

Cada cuerpo cuenta con una masa, una posición inicial y una velocidad. La

gravedad hace que los cuerpos se aceleren y se muevan, provocando que los cuerpos se atraigan unos con otro. Este tipo de fuerza, es denominada de *acción a distancia*.

El peso y la masa son cantidades íntimamente relacionadas entre sí. Cualquier cosa que tiene peso y ocupa un lugar en el espacio tiene masa. La masa es una cantidad de medida más importante que el peso. La masa de un cuerpo no puede cambiar. Sin embargo, el peso depende de la atracción de la gravedad. Si el cuerpo fuera movido a un punto en el espacio donde no existiera fuerza de gravedad, el peso del mismo sería cero pero tendría la misma masa.

La relación entre masa y peso se puede deducir del Principio de Masa de Newton[3] que dice: *Fuerza es el producto de la masa por la aceleración*. Este principio expresado en términos matemáticos sería:

$$F = m \cdot a \qquad (7.2)$$

donde

F = cantidad de fuerza

m = masa

a = aceleración

La aceleración debido a la gravedad es el porcentaje de variación de cambio de la velocidad del cuerpo. Se puede calcular dicha aceleración como:

$$a = \frac{F}{m} \qquad (7.3)$$

Durante un pequeño intervalo de tiempo, llamémosle *dt*, la aceleración del cuerpo *i* denominada a_i es aproximadamente constante, entonces el cambio de velocidad del cuerpos será:

$$dv_i = a_i \cdot dt \qquad (7.4)$$

El cambio de la posición del cuerpo puede ser calculada como la integral de la

3 O también conocida como Lay del Movimiento

velocidad y la aceleración en el intervalo de tiempo *dt*, que es aproximadamente:

$$dp_i = v_i \cdot dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) \cdot dt \qquad (7.5)$$

También, puede calcularse la magnitud de la fuerza de gravedad entre dos cuerpos *i* y *j* a través de la siguiente fórmula expresada por Newton:

$$F = \frac{G\ m_i\ m_j}{r^2} \qquad (7.6)$$

siendo

m = masa

r = distancia

G = constante gravitacional (cuyo valor es 6,67 x 10^{-11})

Si los cuerpos se hallan en un plano espacial de dos dimensiones, las posiciones de los cuerpos pueden ser representadas como puntos coordenados en el plano tal como: (p_i . x, p_i . y) y (p_j . x, p_j . y). El plano espacial donde se hallan los cuerpos se lo denomina *espacio euclídeo.* La ecuación que permite medir la distancia euclídea entre dos puntos es la siguiente:

$$\sqrt{(p_i . x - p_j . x)^2 + (p_i . y - p_j . y)^2} \qquad (7.7)$$

Si la distancia de dos cuerpos es pequeña significa que los mismos están muy cerca de colisionar. Se considerará, que se producen choques entre los cuerpos, que es lo que sucede naturalmente con los cuerpos en el espacio. Los meteoritos que se mueven por el espacio son atraídos por la fuerza de gravedad de estrellas y planetas, produciendo finalmente una colisión si la fuerza de gravedad de la estrella o el planeta es mayor que la del meteorito, sino, el mismo se desviará o será repelido continuado su trayectoria.

7.2. Soluciones al problema N Body

Al problema de los N Body se lo identifica como un problema de alta demanda computacional. El movimiento que hace un cuerpo es simulado a través de pasos en instantes discretos de tiempo. En cada paso se calcula la fuerza de atracción de cada cuerpo y se modifican las posiciones y velocidades de los mismos.

Este problema ha sido estudiado a lo largo de la historia de la computación. En 1980 fueron desarrollados varios algoritmos tales como el Treecode y el Fast Multipole Method (FMM), que han reducido exitosamente la complejidad de $O(N^2)$ a O(N log N) para el primero mencionado y O(N) el segundo [HYNN++09] [YB10] [BGP12] .

Una de las soluciones propuestas, y en la cual se basa esta Tesina de Grado, es la denominada *all-pair o simulación directa* [And00] [Béd07], en la cual, todos los cuerpos interactúan con todos.

```
#calcular las fuerzas totales para cada par de cuerpos
Procedure calcularFuerzas(){

     for(i = 0 to n-1; j = i + 1 to n){
          Distancia = sqrt((posicion_i.x - posicion_j.x)^2 +
                           (posicion_i.y - posicion_j.y)^2);
          Magnitud = (G * m_i * m_j) / distancia^2;
          Dirección = (posicion_j.x - posicion_i.x),
                      (posicion_j.y - posicion_i.y);
          Fuerza_i = (fuerza_i.x + magnitud * direccion_i.x /
                     distancia),(fuerza_i.y + magnitud *
                     direccion_i.y / distancia);
          Fuerza_i = (fuerza_j.x + magnitud * direccion_j.x /
                     distancia),(fuerza_j.y + magnitud *
                     direccion_j.y / distancia);
     }

}
```

```
#calcular las velocidades y posiciones de cada cuerpo
Procedure moverCuerpos(){

     for(i = 0 to n){
          Deltav = (fuerza_i.x / masa_i * dt),
                   (fuerza_i.y / masa_i * dt);

          Deltap = (velocidad_i.x + deltav.x / 2) * dt,
                   (velocidad_i.y + deltav.y /2) * dt;

          Velocidad_i = (velocidad_i.x + deltav.x),
                        (velocidad_i.y + deltav.y);

          Posicion_i = (posicion_i.x + deltap.x),
                       (posicion_i.y + deltap.y);

          Fuerza_i.x = Fuerza_i.y = 0.0f;
     }
}


#ejecutar la simulación steps pasos por dt tiempo
for(i = 0 to steps){
     CalcularFuerzas();
     MoverCuerpos();
}
```

Fig. 7.1. Seudocódigo de la solución secuencial para resolver el problema N Body

En la Fig. 7.1. se presenta el pseudocódigo del algoritmo secuencial, donde se puede observar cómo calcular la fuerza de atracción gravitacional, en el caso de una simulación en la que se quiere estudiar la evolución de una galaxia.

Para realizar la simulación, se utilizar un vector de fuerza, un vector de velocidades, un vector de posiciones, y un vector de masas. Cada uno de los vectores con un tamaño N.

El algoritmo realiza las siguientes acciones:

1. Calcula la fuerza del cuerpo *i*, la cual se encuentra determinada por los N – *i* cuerpos. A su vez, todos los cuerpos N – *i* al estar influenciados por el cuerpos *i* se les modificará su fuerza de gravedad.
2. Se modifica la posición y la velocidad del cuerpo *i*, dependiendo de su fuerza de gravedad.

La solución *all-pair* se la podría describir como una simulación por fuerza bruta, en la que todos los cuerpos interactúan con todos. Es uno de los métodos más simples, pero no muy usado por su demanda computacional. La complejidad de dicho algoritmo es $O(N^2)$, lo que lo hace útil sólo para sistemas de tamaño moderado, además de demandar gran cantidad de memoria.

Muchas aplicaciones utilizan el algoritmo de los N Body en sus versiones jerárquicas. Para el caso gravitacional solo las partículas más próximas contribuyen significativamente a la dinámica del sistema, las más lejanas hacen aportes menores. La idea básica de los métodos de tipo árbol es establecer una relación jerárquica que permita definir “cercano” y “lejano” para poder calcular una solución eficiente [VR08].

Uno de ellos es Treecode, que divide recursivamente el espacio en sub-regiones hasta que se alcanza un criterio dado, por ejemplo, hasta que dicho espacio contenga *k* cuerpos. Cada espacio conforma una celda de cuerpos que interactúan. Cada celda ejecuta el método *all-pair*, en una o más capas de celdas vecinas. El resultado es que muchos cuerpos en una celda, así como una celda con otra ejecutan la solución *all-pair* para calcular las fuerzas. El problema que tiene esta aproximación es que cuanto más profundo se llega en la jerarquía más trabajo hay que realizar en cuerpos que pertenecen a celdas muy alejadas [Béd07] [VR08] [Bar12].

Otro algoritmo jerárquico utilizado es el Fast Multipole Method (FMM), el cual, usa también una estructura tipo árbol y el calculo de la fuerza gravitacional se realiza celda a celda, en lugar de partícula a celda como en el algoritmo mencionado en el párrafo anterior. El FMM reduce la complejidad de la simulación de los N Body a O(N) [HYNN+ +09].

El algoritmo directo para la resolución de problemas de N Body se suele utilizar y tiene buenos resultados en tamaños pequeños del problema. A gran escala los algoritmos mencionados anteriormente son claramente más óptimos por su capacidad de reducir la complejidad del problema significativamente [Nak09].

Capítulo 8

Solución al problema N Body

"Divide las dificultades que examinas en tantas partes como sea posible para su mejor solución"[1]

8.1. Solución secuencial

En el Capítulo anterior se explicó el algoritmo all-pair o solución directa. Sin embargo, existe una solución alternativa al algoritmo all-pair, con la cual, pueden obtenerse menores tiempos de ejecución. Este algoritmo, realiza básicamente las siguientes acciones:

1. Calcula la fuerza de atracción del cuerpo *i*, con respecto a los demás N – *i* cuerpos.
2. Calcula la nueva posición y la velocidad del cuerpo *i*.

1 René Descartes, fue un filósofo, matemático, científico francés.

Como puede observarse, a diferencia del algoritmo all-pair, el cálculo de la fuerza de gravedad de un cuerpo no modifica las fuerzas de gravedad de los demás. Cada uno actualiza su fuerza de gravedad para un momento dado en el tiempo bajo un escenario dado. Las fórmulas empleadas para realizar la simulación son las mencionadas en el Capítulo 7.

En la solución se utilizan estructuras de datos de tipo vector de una dimensión para almacenar: las fuerzas de atracción, las posiciones, las masas y las velocidades. Las estructuras son de tamaño N.

El procesamiento de la fuerza de atracción de los cuerpos se realiza dividiendo los vectores de fuerzas, posiciones y masas en bloques para aprovechar la localidad espacial y temporal de la caché, y de este modo, reducir su tasa de fallo. El procesamiento se realiza de la siguiente manera:

Se produce la carga en la caché del primer bloque del vector de fuerzas. Por cada elemento que comprende el bloque se subdividirá en bloques del mismo tamaño los vectores de posiciones y masas. Una vez realizado el cálculo de la fuerza de atracción gravitacional para todos los cuerpos del bloque, se carga un nuevo bloque de fuerzas, con el que se repetirá el procesamiento explicado anteriormente, y así siguiendo por cada bloque hasta haber realizado la ejecución de todos los cuerpos.

```
#Calcular la fuerza de atracción gravitacional de los N cuerpos
procedure calcularFuerzaAtracción(){

for 1 to cantidadBloques
begin
    for 1 to cantidadBloques
    begin
        for desplazaInicioF to desplazaFinF
        begin
            for desplazaInicio to desplazaFin
            begin
                if(si no es el mismo cuerpo)
                begin
                  Actualizar la fuerza de atracción como se mostró en el Capítulo 7 (Fig. 7.1)
                end
            end
        end
     end
end
Actualizar las velocidades y posiciones de los N cuerpos (como en Capítulo 7, Fig. 7.1)
}
```

Fig. 8.1. Pseudocódigo de la solución secuencial implementada para el problema de los N cuerpos

El algoritmo se implementó para la arquitectura convencional de CPU y para la arquitectura GPU. Para el caso de la CPU se presentan además de la versión secuencial, la solución paralela en memoria compartida, la solución paralela en memoria distribuida, y la solución híbrida en memoria distribuida compartida.

En el Apéndice A pueden encontrarse los algoritmos de cada una de las versiones del problema.

8.2. Solución paralela en memoria compartida

En el caso de la solución en memoria compartida se utilizó como herramienta para la expresión de paralelismo Pthread. La independencia entre tareas se encuentra al calcular la fuerza de atracción, la posición y la velocidad de cada cuerpo. Dicho cálculo no requiere de, por ejemplo, un resultado previo obtenido por otro cuerpo, sino que en cada paso de ejecución cada cuerpo calculará su fuerza de atracción independiente de la de los demás. A cada cuerpo *i* le interesa el valor de la posición de sus N-*i* cuerpos vecinos la cual no cambiará hasta que todos hallan calculado su fuerza de atracción.

Partiendo del algoritmo secuencial descripto en la sección anterior, el algoritmo paralelo realiza las mismas actividades solo que sobre porciones más pequeñas de datos, esto significa que la división de los datos por bloques se realiza sobre una porción de N/*t* datos, siendo N la cantidad de cuerpos y *t* la cantidad de threads. La cantidad de bloques totales en los cuales se dividen los vectores de fuerza, posición y masa es igual a la del secuencial. Por ejemplo, si en el algoritmo secuencial se realizó una división de 200 bloques, en el algoritmo paralelo supongamos con 8 threads se realizará una división de 200 bloques con 25 bloques del vector de fuerza para procesar a cada thread. Mientras tanto, los vectores de posiciones y masa serán dividido en 200 bloques para todos los hilos, ya que se lo necesita completo para el cálculo de la fuerza de atracción de cada cuerpo.

Una vez que las fuerzas de atracción de todos los cuerpos fueron actualizadas, se procede a realizar el calculo de las velocidades y la posición. La velocidad de cada cuerpo dependerá de su fuerza de atracción y es independiente de las de los N-*i* cuerpos vecinos. La posición se calcula a partir de la velocidad del cuerpo y es independiente de los demás. Realizados dichos cálculos, los threads deben esperar en una barrera que todos los demás hallan terminado, antes de realizar otro paso de simulación. Esta condición de espera es necesaria porque para calcular la fuerza del cuerpo *i* el thread *x* necesita conocer las posiciones de los N – *i* cuerpos vecinos, por lo tanto, es necesario

que todas las posiciones del paso previo hallan sido calculadas.

```
#Calcular la fuerza de atracción gravitacional de los N cuerpos
procedure calcularFuerzaAtracción(){
For i to cantPasosSimulación
begin
   For 1  to cantidadBloques
   begin
       For 1 to cantidadBloques
       begin
           For (idThread*(N/T)) to desplazaFinF
           begin
               For ((idThread*(N/T)) + TamañoBloque)  to desplazaFin
               begin
                     if(si no es el mismo cuerpo)
                     begin
                        Actualizar la fuerza de atracción como se mostró en el Capítulo 7 (Fig. 7.1)
                     end
               end
           end
       end
   end
   Actualizar las velocidades y posiciones de los N/T cuerpos (como en Capítulo 7, Fig. 7.1)
   Esperar en la barrera de sincronización antes de realizar otro paso de simulación.
end
}
```

Fig. 8.2. Pseudocódigo de la solución paralela en memoria compartida implementada para el problema de los N cuerpos

La solución se ha probado con hasta 8 threads por las características de la arquitectura utilizada para la experimentación, que será descripta en el próximo Capítulo. El usuario indica la cantidad de threads. La creación final de los threads es de T-1 threads, siendo T la cantidad de hilos indicados por el usuario, ya que el thread principal también realiza trabajo.

8.3. Solución paralela en memoria distribuida

La solución en memoria distribuida es similar a la anterior en lo referido al cálculo de los datos. Solo que en lugar de tener threads ejecutándose, se tiene procesos. Para expresar paralelismo se utilizó MPI.

El proceso con rank cero es el que realiza la inicialización de las estructuras de datos. Una vez realizada, reparte los datos entre todos los demás procesos. Cada proceso realizará las actualizaciones sobre los vectores de fuerza, posición y velocidad sobre una porción de los datos que es igual a N/*p*, donde N es la cantidad de cuerpos y *p* es igual a la cantidad de procesos. La división de los datos en bloques se realiza como se explicó en el apartado anterior.

Una vez que cada proceso ha terminado de realizar todos los cálculos correspondientes a la fuerza de atracción, cada uno transmite los valores obtenidos a la porción de datos que le correspondía. Luego, actualizarán los vectores de velocidad y posición, y al terminar dicho cálculo transmitirán los resultados a los demás. Por lo tanto, todos los procesos tendrán al finalizar la aplicación los datos resultantes.

```
#Calcular la fuerza de atracción gravitacional de los N cuerpos
procedure calcularFuerzaAtracción(){
For i to cantPasosSimulación
begin
   For 1  to cantidadBloques
   begin
      For 1 to cantidadBloques
      begin
         For (idProceso*(N/P)) to desplazaFinF
         begin
            For ((idProceso*(N/P)) + TamañoBloque)  to desplazaFin
            begin
                  if(si no es el mismo cuerpo)
                  begin
                     Actualizar la fuerza de atracción como se mostró en el Capítulo 7 (Fig. 7.1)
                  end
            end
         end
      end
   end
   Actualizar las velocidades y posiciones de los N/P cuerpos (como en Capítulo 7, Fig. 7.1)
   Realizar la comunicación de las nuevas velocidades y posiciones de los cuerpos procesados.
end
}
```

Fig. 8.3. Pseudocódigo de la solución paralela en memoria distribuida implementada para el problema de los N cuerpos

El mapeo de los procesos se realizó en dos hojas del blade (arquitectura utilizada descripta en el Capítulo 9). Se intercaló el mapeo de los procesos en las dos hojas. Por ejemplo, si se tienen 4 procesos, el proceso 0 y el 2 se mapean a la hoja *x*, mientras que el 1 y 3 a la hoja *y*.

8.4. Solución paralela híbrida

La solución híbrida es la combinación de las últimas dos soluciones presentadas, y se utilizaron como herramientas MPI para la creación de los procesos y Pthread para la utilización de threads por cada proceso. Se tiene un conjunto de *p* procesos que interactúan entre sí para resolver el problema. A su vez, cada proceso crea un conjunto de *t* threads.

Al igual que las demás soluciones, cada proceso recibirá N/*p* cuerpos (N = cantidad de cuerpos y *p* = cantidad de procesos). Dicha porción es repartida entre los threads que pertenecen al proceso, por lo tanto, cada thread recibe *(N/p)/t* cuerpos, siendo *t* la cantidad de threads por proceso. La división en bloques sigue la misma metodología de los dos apartados anteriores, sólo que en este caso, siguiendo el ejemplo planteado en la sección 8.2., los 25 bloques que le corresponden a cada proceso son repartidos entre el thread principal (proceso MPI) y el resto de los threads (threads Pthreads creados dentro del proceso MPI). Por lo tanto si se crean 5 threads cada uno recibirá 5 bloques del vector de fuerza para procesar.

El proceso con rank igual a cero es el encargado de inicializar los datos, y más tarde repartirlos entre los demás procesos. Una vez, que todos los procesos tienen los datos a computar, cada uno de ellos crea T-1 threads. Cada thread, incluyendo al proceso o thread principal, calculan la fuerza de atracción gravitacional de para los cuerpos correspondientes a los bloques que le fueron asignado. Luego todos los threads computan las nuevas velocidades y posiciones de la porción de cuerpos que les corresponde. Al terminar, dichas actualizaciones deben ser comunicadas entre los demás procesos, por lo tanto, antes de comenzar un nuevo paso de simulación, los threads del proceso deben sincronizar en una barrera.

```
#Calcular la fuerza de atracción gravitacional de los N cuerpos
procedure calcularFuerzaAtracción(){
For i to cantPasosSimulación
begin
   For 1  to cantidadBloques
   begin
      For 1 to cantidadBloques
      begin
         For (((N/P)/T) / TamañoBloque) to desplazaFinF
         begin
            For (N / TamañoBloque)  to desplazaFin
            begin
               if(si no es el mismo cuerpo)
               begin
                  Actualizar la fuerza de atracción como se mostró en el Capítulo 7 (Fig. 7.1)
               end
            end
         end
      end
   end
   Actualizar las velocidades y posiciones de los (N/P)/T cuerpos (como en Capítulo 7, Fig. 7.1)
   Esperar en una barrera de sincronización a que los threads y el thread principal hayan procesado los
cuerpos que le corresponden
   Esperar  la comunicación de las nuevas velocidades y posiciones de los cuerpos procesados (para los
threads) y comunicar dichos datos para el thread del proceso principal.
end
}
```

Fig. 8.4. Pseudocódigo de la solución paralela híbrida implementada para el problema de los N cuerpos

8.5. Solución paralela en GPU

La solución planteada es apta para ser ejecutada en la GPU por la independencia de cómputo en el cálculo de la fuerza de atracción gravitacional de los cuerpos. Cada cuerpo debe resolver su fuerza gravitacional dependiendo de las posiciones de los demás sin modificar otro dato que no sea su propia fuerza. Del mismo modo, la actualización de las posiciones y velocidades se realiza de manera independiente. En la Fig.8.1. Se muestra el modelo de bloques de threads utilizado para resolver el problema de los N Body en el presente trabajo.

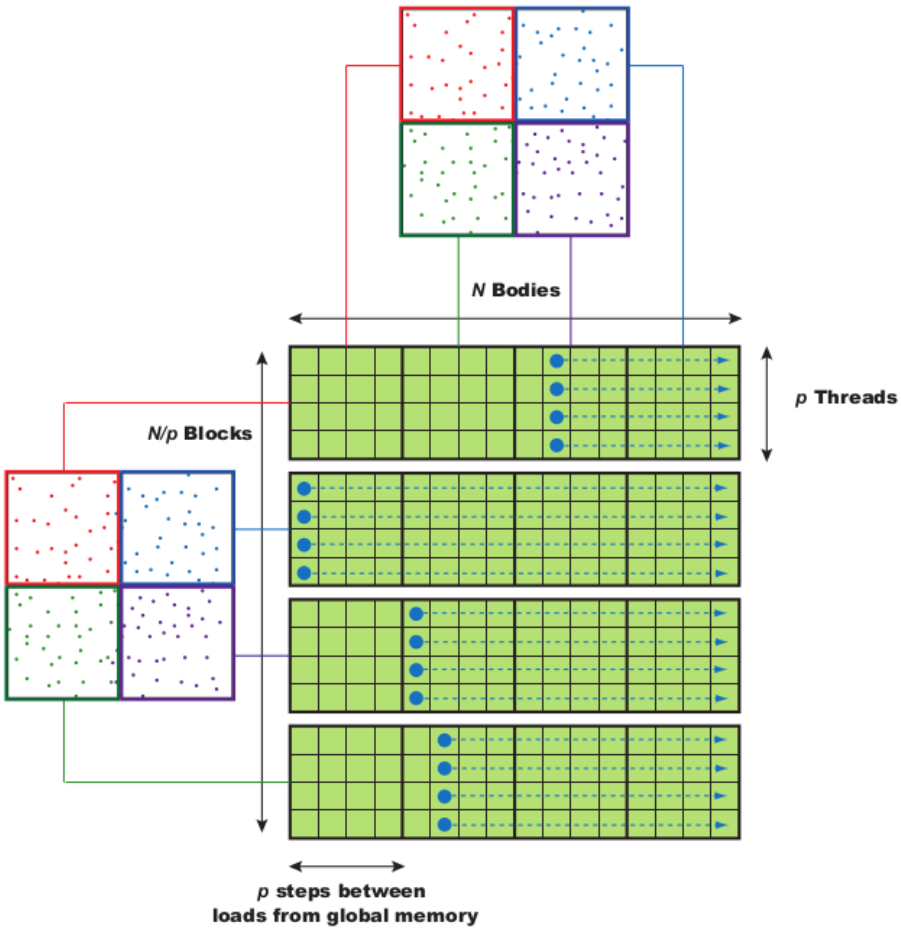


Fig. 8.1. Modelo de bloque de thread problema N Body en GPU (tomado de [Nvi03])

Como se mencionó en el Capítulo 5, el flujo de un programa CUDA, comprende una reserva de memoria en la GPU, una comunicación desde la CPU a la GPU, la ejecución de una función kernel sobre los datos comunicados, y finalmente una comunicación de los datos calculados desde la GPU a la CPU. Por lo tanto, se almacena en la memoria global de la GPU las mimas estructuras utilizadas por la CPU: los vectores de fuerza, velocidad, posición y masa. Por convención se suele denominar a las variables igual a las usadas en la CPU pero con una letra *d* (indicando *device*) al inicio. Por ejemplo, si se declara la estructura fuerza en la CPU, para la GPU se denominará d_fuerza.

El paso siguiente es la comunicación de los datos previamente inicializados por la CPU a través del PCI-E. A continuación, la CPU delegará la ejecución a la GPU, y quedará en espera a la respuesta de esta.

En el kernel o función que calcula la fuerza de atracción gravitacional cada thread calculará su posición en la memoria global que le permitirá acceder al cuerpo para el que

tendrá que calcular dicha fuerza. Se declararán tres vectores con dimensión igual a la cantidad de threads por bloque, dos de ellos se utilizarán para almacenar una porción del vector de posiciones, y el tercero para las masas. Estos vectores permiten computar por tile o bloque, ya que como se mencionó en el Capítulo 2, el acceso a memoria global es muy costoso, por lo tanto es necesario disminuir los accesos haciendo uso de la memoria compartida.

Además de su identificador en la memoria global, cada thread calculará su identificador dentro del vector en memoria compartida. Cada thread traerá de memoria global la posición del cuerpo que le corresponde, por lo tanto, esto significa que el vector almacenado en memoria compartida es cargado por todos los threads, y la posición del cuerpo que le corresponde será almacenada en la dirección correspondiente a su identificador dentro de la memoria shared. Antes de comenzar a calcular la fuerza de atracción, los thread son sincronizados, para garantizar que todas las posiciones de los vectores de la memoria compartida hayan sido cargadas [Lue08].

Luego, cada thread realizará el cálculo correspondiente a la fuerza de atracción para su cuerpo con las posiciones que se encuentran en los vectores de posiciones de la memoria compartida. Cuando todos los threads del bloque hayan utilizado todas las posiciones de los vectores de la memoria shared, deben sincronizarse para volver a traer una porción más de las posiciones y de las masas de los cuerpos de la memoria global a la memoria compartida. Este ciclo se repetirá hasta que todo el vector de posiciones haya sido utilizado (Fig. 8.2.). Es necesario destacar que en el proceso de calculo de la fuerza de atracción se utilizó otra de las optimizaciones de código recomendadas en la literatura [Pic11][Nvi12] de la arquitectura: desenrollo de código. A partir de varias pruebas realizadas se obtuvo que un desenrollo del loop interno del cálculo de la fuerza de atracción gravitacional de cuatro (4) instrucciones es el óptimo para la presente solución.

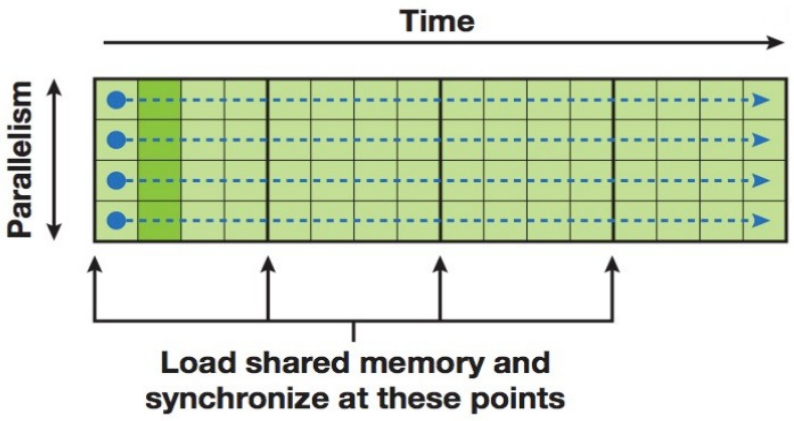


Fig. 8.2. Carga de los cuerpos de memoria global a memoria shared en tile

(tomado de [Coo11])

Como se puede apreciar en la Fig. 8.3. el cálculo de la fuerza de atracción gravitacional de un cuerpo con el resto es inevitablemente un proceso secuencial que no puede ser paralelizado.

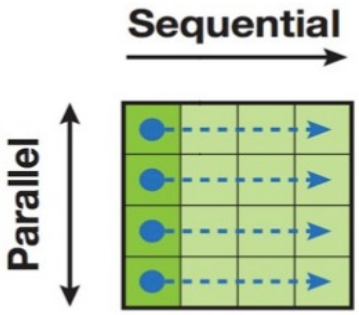


Fig. 8.3. Ejecución del cálculo de atracción gravitacional (tomado de [Coo11])

El paralelismo lo obtenemos al ejecutar N threads que se corresponden con los N cuerpos que conforman la simulación.

Finalmente, cada thread calculará la velocidad y la posición del cuerpo que le corresponde. El control del flujo de ejecución es devuelto a la CPU una vez completados todos los pasos de simulación en el kernel de la GPU. Por último, la GPU envía los resultados calculados a la CPU por PCI-E.

Capítulo 9

Resultados

"Tan pronto como exista una Máquina Analítica,
esta necesariamente guiará el curso futuro de la ciencia.
Cuando se busque un resultado por su ayuda,
en el acto se planteará la pregunta —¿Por cuál trayectoria de
cálculo se pueden obtener estos resultados con la
máquina en el menor tiempo?"[1]

9.1. Entorno experimental

Para realizar la experimentación se ha utilizado un Blade de 8 hojas, con 2 procesadores Quad core Intel Xeón e5405 de 2.0 GHz en cada una de ellas. Cada hoja tiene 2 Gb de memoria RAM (compartido entre ambos procesadores) y cache L2 de 2x6Mb entre par de núcleos. Sistema operativo Fedora 12 de 64bits.

La GPU utilizada es una Nvidia GPU Geforce TX 560TI, con 384 procesadores con una cantidad máxima de thread de 768 y 1 GB de memoria RAM.

9.2. Resultados experimentales

Parte de los resultados de la presente sección, fueron publicados en el XVIII Congreso Argentino de Ciencias de la Computación (CACIC 2012). El Apéndice E, contiene el artículo referenciado.

1 Charles Babbge, fue un matemático británico y científico de la computación.

Para las pruebas realizadas se utilizaron tamaños de problema de 256000 y 512000 cuerpos para cada uno de los vectores. La cantidad de pasos de simulación se variaron entre 5, 6 y 7. A continuación se presentan las tablas con los tiempos de ejecución medidos en segundos. La Tabla 1 muestra los resultados de las soluciones secuenciales y paralela en MPI. La Tabla 2 presenta los tiempos de ejecución para los algoritmos secuenciales y paralelo en Pthread. La Tabla 3 contiene los tiempos en segundos resultantes de la ejecución de la versión híbrida (MPI-Pthread).

Tamaño del vector de cuerpos	Cantidad de pasos de simulación	Secuencial	MPI (P = 2)	MPI (P = 4)	MPI (P = 8)
256000	5	19357,36	9711,28	4856,14	2429,29
	6	23227,00	11653,19	5827,61	2914,85
	7	27098,72	13596,05	6798,60	3400,58
512000	5	77426,49	38848,18	19430,01	9716,82
	6	92918,43	46620,05	23316,08	11659,93
	7	108358,99	54384,43	27200,91	13602,60

Tabla 1. Tiempos de ejecución (en segundos) para el algoritmo MPI en CPU. Con P = cantidad de procesos.

Tamaño del vector de cuerpos	Cantidad de pasos de simulación	Secuencial	Pthread (T = 2)	Pthread (T = 4)	Pthread (T = 8)
256000	5	19357,36	9680,89	4839,94	2419,80
	6	23227,00	11617,94	5809,11	2903,56
	7	27098,72	13553,43	6776,39	3387,41
512000	5	77426,49	38727,09	19364,25	9679,42
	6	92918,43	46472,62	23230,21	11613,76
	7	108358,99	54221,36	27101,45	13551,05

Tabla 2. Tiempos de ejecución (en segundos) para el algoritmo Pthread en CPU. Con T = cantidad de threads.

Tamaño del vector de cuerpos	Cantidad de pasos de simulación	Secuencial	MPI-Pthread (P = 2, T = 2)	MPI-Pthread (P = 2, T = 4)	MPI-Pthread (P = 4, T = 2)
256000	5	19357,36	9679,7	9674,12	4840,68
	6	23227,00	23226,83	11609,07	5809,57
	7	27098,72	13545,11	13543,22	6776,8
512000	5	77426,49	38697,55	38697,92	19363,59
	6	92918,43	46620,05	23316,08	11659,93
	7	108358,99	54179,95	54175,52	27109,68

Tabla 3. Tiempos de ejecución (en segundos) para el algoritmo Híbrido MPI-Pthread en CPU. Con P = cantidad de procesos y T = cantidad de threads por proceso.

Las pruebas experimentales en GPU se realizaron con bloques de 256 threads que es el óptimo para la arquitectura utilizada en la experimentación. Los resultados se presentan en la Tabla 4.

Tamaño del vector de cuerpos	Cantidad de pasos de simulación	GPU (T = 256)
256000	5	37,90
	6	45,47
	7	53,05
512000	5	151,37
	6	181,64
	7	211,90

Tabla 4. Tiempos de ejecución (en segundos) para el algoritmo CUDA en GPU. Con T = cantidad de threads.

Con el fin de evaluar las soluciones desarrolladas se analiza la aceleración obtenida en las arquitecturas basadas en CPU y GPU. Las Fig. 1 y 2 muestran el Speedup (calculado como la relación entre el tiempo de la solución secuencial y el tiempo paralelo) para las soluciones MPI y Pthread con 256000 y 512000 cuerpos para 2 procesos/threads. Las Fig. 3 y 4 presentan los Speedup para 4 y 8 procesos/ threads para MPI, Pthread y la versión híbrida del problema usando la combinación MPI-Pthread.

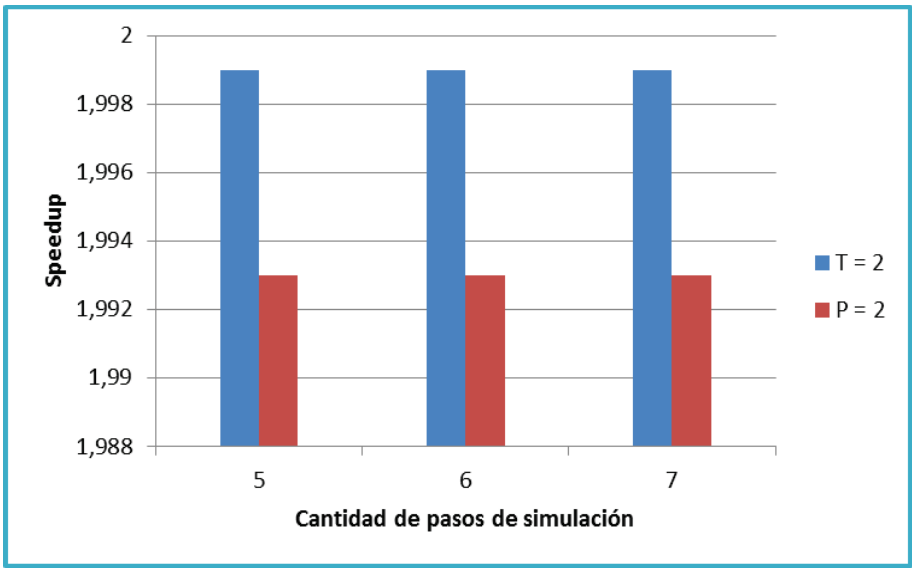


Fig. 1. Speedup de los algoritmos paralelos MPI y Pthread para 256000 cuerpos.

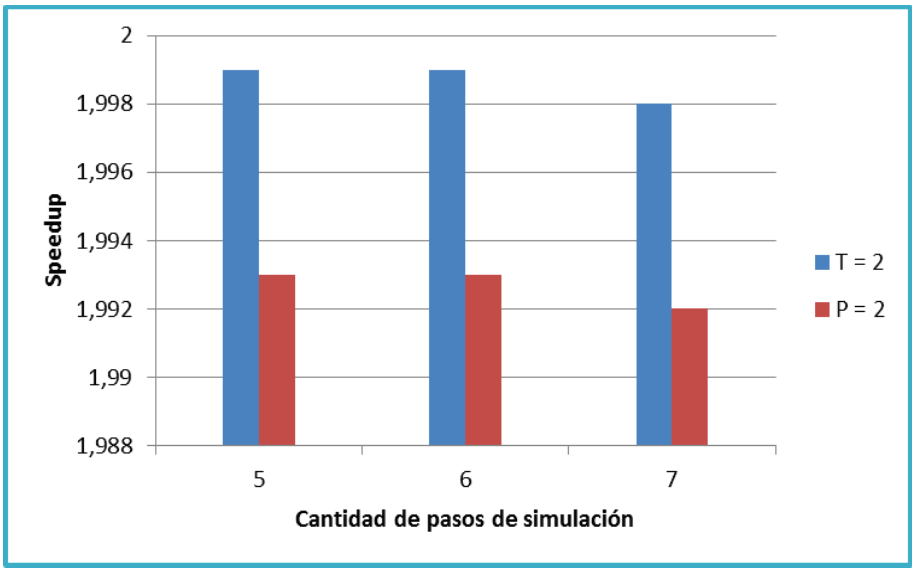


Fig. 2. Speedup de los algoritmos paralelos MPI y Pthread para 512000 cuerpos.

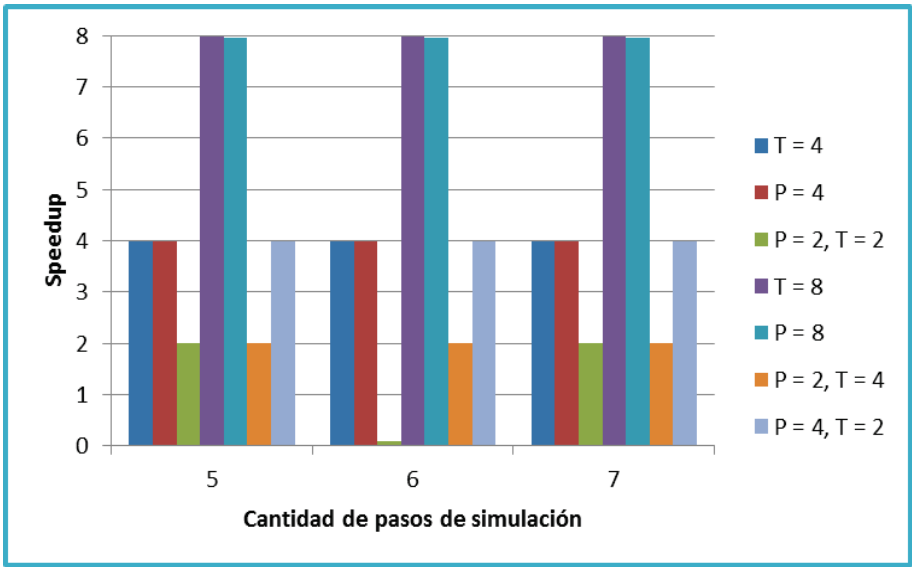


Fig. 3. Speedup de los algoritmos paralelos MPI, Pthread e Híbrido para 256000 cuerpos (Con T = thread, P = proceso).

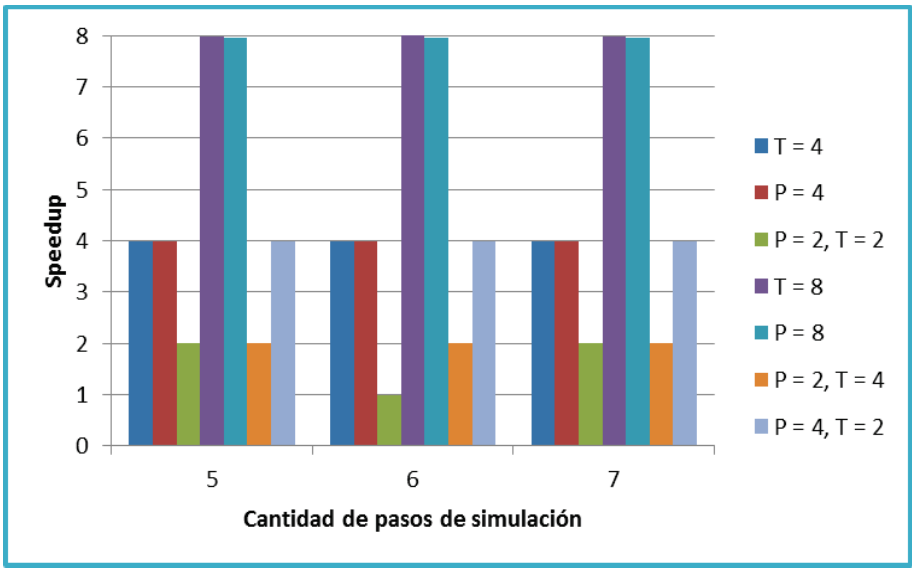


Fig. 4. Speedup de los algoritmos paralelos MPI, Pthread e Híbrido para 512000 cuerpos (Con T = thread, P = proceso).

A partir de Fig. 3 y 4, puede observarse que para el caso de las versiones híbridas de la solución del problema se obtiene una pobre aceleración. La combinación P = 4, T = 2 resulta mejor que la combinación P = 2, T = 4 al tener una menor cantidad de barreras de sincronización. Los threads dentro de un proceso se deben sincronizar entre sí durante el cálculo de la fuerza de atracción gravitacional y con el thread principal (el proceso) para esperar que se realice la comunicación entre los procesos que conforman la solución. Por lo tanto, si se compara dicha solución con la versión en memoria compartida y distribuida se estaría agregando dos barreras de sincronización extra que no existen en la solución de Pthread más la espera por la comunicación entre los procesos, que en este caso se utiliza la misma que en la solución MPI. Cuanto mayor es la cantidad de procesos y menor la de thread por proceso el overhead por espera en las barreras de sincronización claramente disminuye.

Por tratarse de dos arquitecturas diferentes, las métricas utilizadas para los algoritmos paralelos en CPU no son posibles de ser utilizados para comparar la performance obtenida con la GPU. Por lo tanto, suele referirse a la ganancia obtenida por el uso de GPU a una aceleración alcanzada, la misma se calcula (de manera similar que el *Speedup*) como el tiempo secuencial (en la CPU) respecto del tiempo paralelo (en la GPU). Esto se muestra en la Fig. 5.

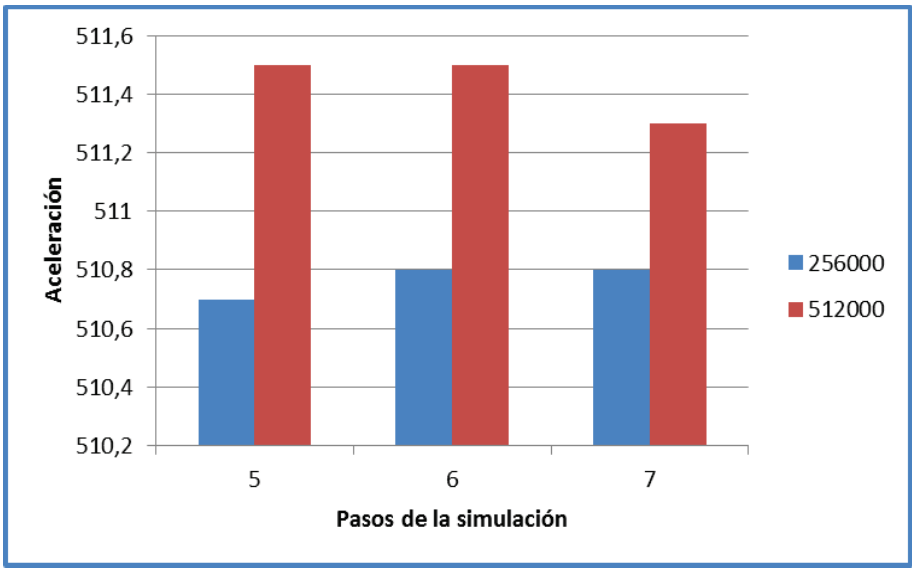


Fig. 5. Aceleración obtenida para el algoritmo N-Body en GPU.

Con respecto a los resultados obtenidos puede observarse que las soluciones Pthread y MPI presentan características similares en tiempo, no habiendo una diferencia significativa entre ellas en este aspecto. Esto se debe a que la etapa de sincronización entre todas las tareas es más significativa a la de comunicación.

En cambio, para la implementación GPU-CUDA se logran tiempos sensiblemente inferiores con respecto a los mencionados, denotando una alta adecuación de la arquitectura (algoritmo, solución) al problema. Un análisis equivalente puede realizarse a partir de los valores de Speedup y aceleración mostrados.

En el presente Capítulo se ha realizado un análisis del speed up y la aceleración obtenida de las diferentes soluciones con el fin de lograr el mejor aprovechamiento de la arquitectura de base. En el Apéndice D, se realiza un análisis desde el punto de vista del consumo energético para el algoritmo secuencial y el algoritmo en GPU-CUDA.

Conclusiones y Trabajos futuros

La reducción de los tiempos de ejecución en algunas aplicaciones, requiere que su solución deba realizarse por medio del procesamiento paralelo para la obtención de tiempos de respuesta aceptables, que no pueden ser cumplidos a través del procesamiento secuencial. Una diversidad de arquitecturas multiprocesador están disponibles con este fin, al igual que varias herramientas de programación que permiten explotar el paralelismo en las mismas.

En la última década, ha surgido una arquitectura de propósito específico como una alternativa para el cómputo de altas prestaciones: las GPUs han evolucionado al punto de ofrecer un poder de cómputo que permite alcanzar una gran performance, reduciendo los tiempos de ejecución de las aplicaciones adaptables a dicha arquitectura como se ha mostrado en el presente trabajo para problemas que comparten las mismas características que el problema de los N Body.

Los resultados para la arquitectura CPU depende de cuán bien opera el algoritmo en paralelo teniendo en cuenta factores tales como el balance de carga, la comunicación, la sincronización, el mapeo de los procesos/threads a los procesadores, etc. Por lo cual, la paralelización no es automática y requiere de un análisis y estudio de la arquitectura para el desarrollo de un algoritmo eficiente para dicho problema [Par10]. Claramente la solución obtenida a partir de lo anteriormente mencionado alcanzará un grado de eficiencia para la arquitectura en particular para la cual fue desarrollada. Por lo tanto, la programación está ligada profundamente a la plataforma hardware subyacente.

La computación en GPU comenzó a ser muy popular en el ámbito industrial, comercial y académico, y aplicable a problemas que exhiben un gran paralelismo de datos [Par10]. Pero de una forma similar, la programación en GPU requiere un cierto esfuerzo por parte del programador. Si bien con las plataformas de programación como CUDA, el desarrollo de los algoritmos se facilita [OHLG++008], las consideraciones a tener en cuenta para obtener el máximo rendimiento posible a partir de la arquitectura es alto. La programación y el hardware es totalmente distinto al de la CPU, lo que obliga al programador a replantear su manera de pensar los problemas partiendo directamente de una solución paralela ya que la arquitectura es paralela en sí misma. El conocimiento del hardware es indispensable en el desarrollo de las aplicaciones, así como las técnicas de optimización recomendadas para alcanzar la mayor performance posible (uso de memoria shared, desenrollo de código, garantizar coalescencia, evitar divergencias y conflictos de bancos de memoria, realizar transmisiones grandes en lugar de pequeñas por PCI-e de los datos, utilizar tamaños de bloques de threads múltiplo de 32) [Pic11].

Otras características a tener en mente a la hora de implementar aplicaciones usando el modelo de ejecución CUDA es el tamaño de los bloques. El tamaño de los bloques está relacionado a la cantidad de warps que pueden efectivamente ocultar las latencias producidas por la ejecución de instrucciones costosas o los accesos a memoria. Si la cantidad de warps por SM es pequeña las unidades de cómputo estarán ociosas cuando los threads del bloque accedan a memoria. La cantidad de bloques también es un factor importante a considerar. El tamaño de los bloques está íntimamente relacionada a la porción de datos a procesar. Mientras que los bloques pequeños usan pocos recursos propiciando la ociosidad y mal uso de los recursos de cómputo, los bloques grandes permiten ejecutar las instrucciones de manera más eficiente si existen una gran cantidad de bloques por SM. Lo ideal es que el número de bloque en un grid sea mucho mayor a la cantidad de SMs en la GPU [WXXY+10].

Un programador habituado a la programación en C, se adapta rápidamente a la plataforma CUDA. Las dos características principales que el programador debe tener en cuenta en el uso de la GPU son el mapeo de los threads y el manejo de la jerarquía de memoria.

Cabe agregar además, que el costo de una placa GPU es sensiblemente inferior al de la arquitectura blade utilizada para la comparación, reforzando de esta forma una de las ventajas de su empleo en problemas de este tipo.

El presente trabajo de Grado, muestra claramente los beneficios del uso de la arquitectura GPU a problemas adaptables a dicha plataforma. Los tiempos de ejecución obtenidos son considerablemente inferiores comparados a los tiempos obtenidos para las distintas soluciones del caso de estudio teniendo como base la arquitectura CPU.

Las experimentaciones realizadas desde el punto de vista del consumo energético favorecen el uso de la GPU para problemas de este estilo. De igual manera, como en el trabajo [RPS+12], pueden apreciarse los beneficios del uso de dicha en este sentido.

Como trabajos futuros pueden mencionarse el uso de clusters de GPUs y el estudio del paradigma de programación híbrida utilizando MPI-CUDA para dicha arquitectura, así como avanzar en el análisis de consumo energético en el uso de GPU. Asimismo, se plantea avanzar en el análisis, estudio y uso de otros algoritmos, tales como Treecode y FMM (Fast Multipole Method). Por último, interesa estudiar las nuevas arquitecturas MIC de próxima aparición.

Apéndice A

1. Código del algoritmo secuencial

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>



int DT, steps, N, inicio, fin, TamBlock;

float *velocidadx,*velocidady, *posicionx,*posiciony, *fuerzax,*fuerzay, *masa;

//Para calcular tiempo
double dwalltime(){
        double sec;
        struct timeval tv;

        gettimeofday(&tv,NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
}


int main(int argc,char*argv[]){

if ((argc != 7) || ((N = atoi(argv[1])) <= 0) || ((TamBlock=atoi(argv[2]))<=0)
|| ((steps = atoi(argv[3])) <= 0) || ((DT = atoi(argv[4])) <= 0) || ((inicio =
atoi(argv[5]))< 0) || ((fin=atoi(argv[6]))<= 0))){
      printf("%s","Error faltan parámetros");
      return 1;
}else{
      //Alocar espacio para las posiciones, velocidades, fuerzas y masas
      posicionx = (float*)malloc(sizeof(float)*N);
      velocidadx = (float*)malloc(sizeof(float)*N);
      fuerzax = (float*)malloc(sizeof(float)*N);
      posiciony = (float*)malloc(sizeof(float)*N);
      velocidady = (float*)malloc(sizeof(float)*N);
      fuerzay = (float*)malloc(sizeof(float)*N);
      masa = (float*)malloc(sizeof(float)*N);

      int i;

      int cantBlock = N / TamBlock;

      printf("\nCantidad de cuerpos: %i\n",N);
      printf("\nTamaño de bloques: %i\n",TamBlock);
```

```
      printf("\nCantidad de bloques: %i\n",cantBlock);
        printf("\nPasos de la simulacion: %i\n\n",steps);


      //inicializar posiciones, velocidad, fuerza y masa
      for(i=0 ; i < N ; i++){
            fuerzax[i] = rand() % (fin -inicio +1) + inicio;
            fuerzay[i] = rand() % (fin - inicio +1) + inicio;
            posicionx[i] =  rand() % (fin-inicio+1) + inicio;
            posiciony[i] = rand() % (fin-inicio+1) + inicio;
            velocidadx[i] = rand() % (fin-inicio+1) + inicio;
            velocidady[i] =  rand() % (fin-inicio+1) + inicio;
            masa[i] = rand() % (fin-inicio+1) + inicio;
      }

      float distancia, magnitud;
      float direccionx,direcciony;
      int h,j,k,l,despK,despL,despH,despJ;
      float deltavx,deltavy,deltapx,deltapy;

      double time = dwalltime();

      //comienzo de la simulación
      for(i = 0; i < steps; i++){

            //Calcula la fuerza de atracción entre cada uno de los cuerpos

            despK = TamBlock;
            despL = TamBlock;
            despH=0;
            despJ=0;
            for(h=0;h<cantBlock;h++){
             for(j= 0; j < cantBlock; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] +=  magnitud * direccionx / distancia;
                          fuerzay[k] +=  magnitud * direcciony / distancia;
                        }
                  }
             }
             despJ =despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+= TamBlock;
            }


            //Calcula las nuevas velocidades y posiciones de cada cuerpos
            for(k = 0; k < N; k++){
```

```
                    deltavx = fuerzax[k] / masa[k] * DT;
                    deltavy = fuerzay[k] / masa[k] * DT;

                    fuerzax[k] = fuerzay[k] = 0.0; //se resetea el vector de
fuerzas

                    deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                    deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                    posicionx[k] = posicionx[k] + deltapx;
                    posiciony[k] = posiciony[k] + deltapy;

                    velocidadx[k] = velocidadx[k] + deltavx;
                    velocidady[k] = velocidady[k] + deltavy;
                }

      }

      //Tiempo final de la ejecución del algoritmo
      printf("Tiempo en segundos %f \n", dwalltime() - time);

//Liberar la memoria utilizada
free(masa);
free(fuerzax);
free(posicionx);
free(velocidadx);
free(fuerzay);
free(posiciony);
free(velocidady);

      return 0;
}

}
```

2. Código del algoritmos paralelo en memoria compartida

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
#include<pthread.h>

typedef struct{
    int id, inicio, fin;
} identificador;

int DT, steps, N, inicio, fin, T,TamBlock;

float *velocidadx, *posicionx, *fuerzax, *velocidady, *posiciony, *fuerzay;
float *masa;

pthread_barrier_t barrera;

void *calcularAtraccion(void *param){

      float distancia, magnitud,calculo;
      float direccionx,direcciony;
      int i,k,l,h,j,despK,despL,despH,despJ;
```

```
      float deltavx,deltavy,deltapx, deltapy;

      int cores = 8;
      int num = (((((identificador *)(param))->id))%cores);
      unsigned long mask = pow(2,num);

      if (sched_setaffinity(0, sizeof(mask), &mask) <0)  {
          perror("sched_setaffinity");
      }

      int cantBlock1 = (N/T) / TamBlock;
      int cantBlock2 = (N) / TamBlock;
      //comienzo de la simulación
      for(i = 0; i < steps; i++){

            despH=(((identificador *)(param))->id)*(N/T);
            despK = despH+TamBlock;
            despJ=0;
            despL = TamBlock;
       for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] += magnitud * direccionx / distancia;
                          fuerzay[k] += magnitud * direcciony / distancia;
                        }
                  }
             }
             despJ=despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+= TamBlock;
            }


            //Calcula las nuevas velocidades y posiciones de cada cuerpo
            for(k = (((identificador *)(param))->inicio); k < (((identificador
*)(param))->fin); k++){

                  deltavx = fuerzax[k] / masa[k] *DT;
                    deltavy = fuerzay[k] / masa[k] * DT;

                  fuerzax[k] = fuerzay[k] = 0.0;

                  deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                  deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                  posicionx[k] = posicionx[k] + deltapx;
                  posiciony[k] = posiciony[k] + deltapy;
```

```
                velocidadx[k] = velocidadx[k] + deltavx;
                velocidady[k] = velocidady[k] + deltavy;
            }

            pthread_barrier_wait(&barrera);
      }
      pthread_exit(0);
}


//Para calcular tiempo
double dwalltime(){
        double sec;
        struct timeval tv;

        gettimeofday(&tv,NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
}


int main(int argc,char*argv[]){

if ((argc != 8) || ((N = atoi(argv[1])) <= 0) || ((TamBlock=atoi(argv[2]))<=0)
|| ((T = atoi(argv[3])) <= 0) || ((steps = atoi(argv[4])) <= 0) || ((DT =
atoi(argv[5])) <= 0) || ((inicio = atoi(argv[6]))< 0) || ((fin=atoi(argv[7]))<=
0))){
      printf("%s\n","Error faltan parámetros");
      return 1;
}else{
      int cantBlock1 = (N/T) / TamBlock;
      int cantBlock2 = (N) / TamBlock;
      //Alocar espacio para las posiciones, velocidades, fuerzas y masas
      masa = (float*)malloc(sizeof(float)*N);
      posiciony = (float*)malloc(sizeof(float)*N);
      velocidady = (float*)malloc(sizeof(float)*N);
      fuerzay = (float*)malloc(sizeof(float)*N);
      posicionx = (float*)malloc(sizeof(float)*N);
      velocidadx = (float*)malloc(sizeof(float)*N);
      fuerzax = (float*)malloc(sizeof(float)*N);

      int i, j;

      printf("\n\nCantidad de threads: %i\n",T);
      printf("\nCantidad de cuerpos: %i\n",N);
      printf("\nTamaño de bloques: %i\n",TamBlock);
      printf("\nCantidad de bloques: %i\n",cantBlock2);
      printf("\nCantidad de bloques por thread: %i\n",cantBlock1);
      printf("\nPasos de la simulacion: %i\n\n",steps);


      for(i=0; i< N; i++){
            fuerzax[i] = (rand() % (fin-inicio+1) + inicio);
            fuerzay[i] = (rand() % (fin-inicio+1) + inicio);
            posicionx[i] = (rand() % (fin-inicio+1) + inicio);
            posiciony[i] = (rand() % (fin-inicio+1) + inicio);
            velocidadx[i] = (rand() % (fin-inicio+1) + inicio);
            velocidady[i] = (rand() % (fin-inicio+1) + inicio);
            masa[i] = (rand() % (fin-inicio+1) + inicio);
      }
```

```
      pthread_t calculan[T];
      identificador a_arg[T];
      int resultado;

      //Inicializa la barrera
      resultado = pthread_barrier_init(&barrera, NULL, T);
      if(resultado){
            printf("ERROR crendo barrera %d\n",resultado);
            exit(0);
      }

      float distancia, magnitud,calculo;
      float direccionx,direcciony;
      int k,l,h,despK,despL,despH,despJ;
      float deltavx, deltapy,deltavy,deltapx;
      int inicio = (T-1) * (N/T);
      int fin = ((T-1) * (N/T)) + (N/T);
      double time = dwalltime();

      //Comienzo de la simulacion
      for(i = 0; i < T-1; i++){
            a_arg[i].id = i;
            a_arg[i].inicio = i * (N/T);
            a_arg[i].fin = (i * (N/T)) + (N/T);
            resultado = pthread_create(&calculan[i],NULL,calcularAtraccion,(void
*) &a_arg[i]);
            if(resultado){
                  printf("ERROR creando thread codigo: %d",resultado);
                  exit(-1);
            }
      }


      int cores = 8;
      int num = ((T-1)%cores);
      unsigned long mask = pow(2,num);

      if (sched_setaffinity(0, sizeof(mask), &mask) <0)  {
          perror("sched_setaffinity");
      }


      //comienzo de la simulación
      for(i = 0; i < steps; i++){
            despH=(T-1)*(N/T);
            despK = despH+TamBlock;
            despJ=0;
            despL = TamBlock;
       for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] += magnitud * direccionx / distancia;
```

```
                        fuerzay[k] += magnitud * direcciony / distancia;
                    }
                }
            }
            despJ=despL;
            despL+=TamBlock;
            }
            despJ = 0;
            despL = TamBlock;
            despH=despK;
            despK+= TamBlock;
           }


           //Calcula las nuevas velocidades y posiciones de cada cuerpo
           for(k = inicio; k < fin; k++){

                 deltavx = fuerzax[k] / masa[k] *DT;
                   deltavy = fuerzay[k] / masa[k] * DT;

                 fuerzax[k] = fuerzay[k] = 0.0;

                 deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                 deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                 posicionx[k] = posicionx[k] + deltapx;
                 posiciony[k] = posiciony[k] + deltapy;

                 velocidadx[k] = velocidadx[k] + deltavx;
                 velocidady[k] = velocidady[k] + deltavy;
           }

           pthread_barrier_wait(&barrera);
     }



     for(i=0; i<T-1; i++){
                 pthread_join(calculan[i],NULL);
     }

     //Tiempo final de la ejecución del algoritmo
     printf("Tiempo en segundos %f \n\n", dwalltime() - time);

     return 0;
}
}
```

3. Código del algoritmos paralelo en memoria distribuida

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
#include<mpi.h>
```

```
int nProc, N, steps, DT, inicio, fin,TamBlock;


//Para calcular tiempo
double dwalltime(){
        double sec;
        struct timeval tv;

        gettimeofday(&tv,NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
}


int main(int argc, char *argv[]){
if ((argc != 8) || ((N = atoi(argv[1])) <= 0) || ((TamBlock=atoi(argv[2]))<=0)
|| ((nProc=atoi(argv[3]))<= 0) || ((steps = atoi(argv[4])) <= 0) || ((DT =
atoi(argv[5])) <= 0) || ((inicio = atoi(argv[6]))< 0) || ((fin=atoi(argv[7]))<=
0))){
      printf("%s","\nERROR...\n");
      printf("%s","\nmpirun -np cantProcesos ejecutable_mpi cantCuerpos
cantBloques cantProcesos cantPasos cantDT rangoInicio rangoFin\n\n");
      return 1;
}else{

int h,i,j,k,l,despK,despL,despH,despJ,myRank;
float calculo, distancia, magnitud;
float direccionx,direcciony;
static double time;
float *fuerzax, *fuerzay,*posicionx,*posiciony,*velocidadx,*velocidady;
float *masa, deltavx,deltavy,deltapx, deltapy;
MPI_Status estado;


masa = (float*)malloc(sizeof(float)*N);
posicionx = (float*)malloc(sizeof(float)*N);
velocidadx = (float*)malloc(sizeof(float)*N);
fuerzax = (float*)malloc(sizeof(float)*N);
posiciony = (float*)malloc(sizeof(float)*N);
velocidady = (float*)malloc(sizeof(float)*N);
fuerzay = (float*)malloc(sizeof(float)*N);


int res = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nProc);
MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

      int cantBlock1 = (N/nProc) / TamBlock;
      int cantBlock2 = (N) / TamBlock;

//Master
if(myRank == 0){

      printf("\n\nCantidad de procesos: %i\n",nProc);
      printf("\nCantidad de cuerpos: %i\n",N);
      printf("\nTamaño de bloques: %i\n",TamBlock);
      printf("\nCantidad de bloques: %i\n",cantBlock2);
      printf("\nCantidad de bloques por proceso: %i\n",cantBlock1);
```

```
      printf("\nPasos de la simulacion: %i\n\n",steps);

      inicio = N;
      fin = N * 100;

      //Inicializar los vectores de fuerza, velocidad, posicion y masa
      for(i=0; i< N; i++){
            fuerzax[i] = (rand() % (fin-inicio+1) + inicio);
            fuerzay[i] = (rand() % (fin-inicio+1) + inicio);
            posicionx[i] = (rand() % (fin-inicio+1) + inicio);
            posiciony[i] = (rand() % (fin-inicio+1) + inicio);
            velocidadx[i] = (rand() % (fin-inicio+1) + inicio);
            velocidady[i] = (rand() % (fin-inicio+1) + inicio);
            masa[i] = (rand() % (fin-inicio+1) + inicio);
      }
}

  time = dwalltime();


       res= MPI_Bcast(fuerzax,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(fuerzay,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(posicionx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(posiciony,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(velocidadx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(velocidady,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(masa,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);


      //Realizar la simulacion steps veces

      for(i=0; i < steps; i++){

            //Calcular las fuerza de atracción gravitacional para los cuerpos
correspondientes a myRank

            despH=myRank*(N/nProc);
            despK = despH+TamBlock;
            despJ=0;
            despL = TamBlock;
            for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
```

```
                        distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                        magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                        fuerzax[k] += magnitud * direccionx / distancia;
                        fuerzay[k] += magnitud * direcciony / distancia;
                      }
                }
             }
             despJ=despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+=TamBlock;
            }


            //Calcula las nuevas velocidades y posiciones de cada cuerpo en el
bloque de mi rank
               for(k = myRank*(N/nProc); k < (myRank+1)*(N/nProc); k++){

                   deltavx = fuerzax[k] / masa[k] *DT;
                   deltavy = fuerzay[k] / masa[k] * DT;

                 fuerzax[k] = fuerzay[k] = 0.0;

                 deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                 deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                 posicionx[k] = posicionx[k] + deltapx;
                 posiciony[k] = posiciony[k] + deltapy;

                 velocidadx[k] = velocidadx[k] + deltavx;
                 velocidady[k] = velocidady[k] + deltavy;
               }


              MPI_Allgather(&velocidadx[myRank*(N/nProc)], (N/nProc),
MPI_FLOAT, &velocidadx[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
              MPI_Allgather(&posiciony[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&posiciony[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
              MPI_Allgather(&velocidady[myRank*(N/nProc)], (N/nProc),
MPI_FLOAT, &velocidady[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
              MPI_Allgather(&posicionx[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&posicionx[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
      }

if(myRank == 0){
    //Imprimir el tiempo total de la ejecucion del algoritmo
    printf("El tiempo en segundos es: %f \n\n\n", dwalltime() - time);
}

//Liberar la memoria utilizada
free(masa);
free(fuerzax);
free(posicionx);
free(velocidadx);
free(fuerzay);
```

```
free(posiciony);
free(velocidady);

MPI_Finalize();

return(0);
}
}
```

4. Código del algoritmos paralelo híbrido

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>
#include<sched.h>
#include<mpi.h>
#include<pthread.h>

typedef struct{
    int id;
    int inicio;
    int fin;
} identificador;


float *fuerzax, *fuerzay, *velocidadx,*velocidady, *posicionx,*posiciony, *masa;

int N, T, nProc, steps, DT, inicio, fin, TamBlock;
pthread_barrier_t barrera;

//Para calcular tiempo
double dwalltime(){
        double sec;
        struct timeval tv;

        gettimeofday(&tv,NULL);
        sec = tv.tv_sec + tv.tv_usec/1000000.0;
        return sec;
}

void *calcularAtraccion(void *param){

      float distancia, magnitud,calculo;
      float direccionx,direcciony;
      int i,k,l,h,j,despK,despL,despH,despJ;
      float deltavx,deltavy,deltapx, deltapy;

      int cores = 7;
      int num = (((((identificador *)(param))->id))%cores);
      unsigned long mask = pow(2,num);

      if (sched_setaffinity(0, sizeof(mask), &mask) <0)  {
          perror("sched_setaffinity");
      }
```

```
      int cantBlock1 = ((N/nProc)/T) / TamBlock;
      int cantBlock2 = (N) / TamBlock;

      //comienzo de la simulación
      for(i = 0; i < steps; i++){

            despH=(((identificador *)(param))->id)*(N/T);
            despK = despH+TamBlock;
            despJ=0;
            despL = TamBlock;
       for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] += magnitud * direccionx / distancia;
                          fuerzay[k] += magnitud * direccion / distancia;
                        }
                  }
             }
             despJ=despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+= TamBlock;
            }


            //Calcula las nuevas velocidades y posiciones de cada cuerpo
            for(k = (((identificador *)(param))->inicio); k < (((identificador
*)(param))->fin); k++){

                  deltavx = fuerzax[k] / masa[k] *DT;
                    deltavy = fuerzay[k] / masa[k] * DT;

                  fuerzax[k] = fuerzay[k] = 0.0;

                  deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                  deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                  posicionx[k] = posicionx[k] + deltapx;
                  posiciony[k] = posiciony[k] + deltapy;

                  velocidadx[k] = velocidadx[k] + deltavx;
                  velocidady[k] = velocidady[k] + deltavy;
            }

            pthread_barrier_wait(&barrera);
            pthread_barrier_wait(&barrera);
      }
      pthread_exit(0);
}
```

```
int main(int argc, char *argv[]){


if((argc != 9)||((N= atoi(argv[1])) <= 0)||(((TamBlock = atoi(argv[2])))<=0)||
((nProc = atoi(argv[3]))<=0)||((T = atoi(argv[4]))<=0)||((steps =
atoi(argv[5]))<=0)||((DT = atoi(argv[6]))<0) || ((inicio = atoi(argv[7]))< 0) ||
((fin=atoi(argv[8]))<= 0))){
  printf("\nERROR!!! faltan parametros\n");
  return(1);
}else{

MPI_Status estado;
int h,i,j,k,l,despK,despL,despH,despJ,myRank;
float calculo, distancia, magnitud;
float direccionx,direcciony, deltavx,deltavy,deltapx, deltapy;

fuerzax = (float*)malloc(sizeof(float)*N*T);
fuerzay = (float*)malloc(sizeof(float)*N*T);
velocidadx = (float*)malloc(sizeof(float)*N);
velocidady = (float*)malloc(sizeof(float)*N);
posicionx = (float*)malloc(sizeof(float)*N);
posiciony = (float*)malloc(sizeof(float)*N);
masa = (float*)malloc(sizeof(float)*N);

int res = 0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nProc);
MPI_Comm_rank(MPI_COMM_WORLD,&myRank);

pthread_t calculan[T];
identificador a_arg[T];
int resultado;

//Inicializa la barrera
resultado = pthread_barrier_init(&barrera, NULL, T);
if(resultado){
  printf("ERROR creando barrera %d\n",resultado);
  exit(0);
}

int cantBlock1 = (N/nProc) / TamBlock;
int cantBlock2 = (N) / TamBlock;

//Master
if(myRank == 0){

      printf("\n\nCantidad de procesos: %i\n",nProc);
      printf("\nCantidad de threads: %i\n",T);
      printf("\nCantidad de cuerpos: %i\n",N);
      printf("\nTamaño de bloques: %i\n",TamBlock);
      printf("\nCantidad de bloques: %i\n",cantBlock2);
      printf("\nCantidad de bloques por proceso: %i\n",(N/nProc)/TamBlock);
      printf("\nCantidad de bloques por threads: %i\n",((N/nProc)/T)/TamBlock);
      printf("\nPasos de la simulacion: %i\n\n",steps);


      //Inicializar los vectores de fuerza, velocidad, posicion y masa
      for(i=0; i< N; i++){
```

```
            fuerzax[i] = (rand() % (fin-inicio+1) + inicio);
            fuerzay[i] = (rand() % (fin-inicio+1) + inicio);
            posicionx[i] = (rand() % (fin-inicio+1) + inicio);
            posiciony[i] = (rand() % (fin-inicio+1) + inicio);
            velocidadx[i] = (rand() % (fin-inicio+1) + inicio);
            velocidady[i] = (rand() % (fin-inicio+1) + inicio);
            masa[i] = (rand() % (fin-inicio+1) + inicio);
      }


double time = dwalltime();

res= MPI_Bcast(fuerzax,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(fuerzay,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(posicionx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(posiciony,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(velocidadx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(velocidady,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(masa,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);


//Realizar la simulacion steps veces

    int ant = 0;
    for(i = 0; i < T-1; i++){
      a_arg[i].id = i;
      a_arg[i].inicio = (i*(((myRank*(N/nProc)))/T))+ant;
      a_arg[i].fin = (i*((((myRank+1)*(N/nProc)))/T))+
(((((myRank+1)*(N/nProc))))/T);
      ant+= a_arg[i].fin;
      resultado = pthread_create(&calculan[i],NULL,calcularAtraccion,(void *)
&a_arg[i]);
      if(resultado){
      printf("ERROR creando thread codigo: %d",resultado);
      exit(-1);
      }
    }

//Realizar la simulacion steps veces

      for(i=0; i < steps; i++){

            //Calcular las fuerza de atracción gravitacional para los cuerpos
correspondientes a myRank

            despH = a_arg[T-2].fin;
            despK = despH+TamBlock;
```

```
            despJ=0;
            despL = TamBlock;
            for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] += magnitud * direccionx / distancia;
                          fuerzay[k] += magnitud * direcciony / distancia;
                        }
                  }
             }
             despJ=despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+=TamBlock;
            }


            //Calcula las nuevas velocidades y posiciones de cada cuerpo en el
bloque de mi rank
                for(k = a_arg[T-2].fin; k < (myRank+1)*(N/nProc); k++){

                    deltavx = fuerzax[k] / masa[k] *DT;
                    deltavy = fuerzay[k] / masa[k] * DT;

                  fuerzax[k] = fuerzay[k] = 0.0;

                  deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                  deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                  posicionx[k] = posicionx[k] + deltapx;
                  posiciony[k] = posiciony[k] + deltapy;

                  velocidadx[k] = velocidadx[k] + deltavx;
                  velocidady[k] = velocidady[k] + deltavy;
                }

             pthread_barrier_wait(&barrera);

MPI_Allgather(&velocidadx[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&velocidadx[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&posiciony[myRank*(N/nProc)], (N/nProc), MPI_FLOAT, &posiciony[0],
(N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&velocidady[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&velocidady[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&posicionx[myRank*(N/nProc)], (N/nProc), MPI_FLOAT, &posicionx[0],
(N/nProc), MPI_FLOAT, MPI_COMM_WORLD);

             pthread_barrier_wait(&barrera);
```

```
      }

      for(i=0; i<T-1; i++){
        pthread_join(calculan[i],NULL);

      }

      printf("El tiempo en segundos es: %f \n", dwalltime() - time);

}else{ //Workers

      res= MPI_Bcast(fuerzax,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(fuerzay,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(posicionx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(posiciony,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
        res= MPI_Bcast(velocidadx,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(velocidady,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);
          res= MPI_Bcast(masa,N,MPI_FLOAT,0,MPI_COMM_WORLD);
            if (res != 0)
                        printf ("Error en bcast! %d\n", res);

  int ant = myRank*(N/nProc);
          for(i = 0; i < T-1; i++){
            a_arg[i].id = i;
            a_arg[i].inicio = (i*(((myRank*(N/nProc)))/T))+ant;
            a_arg[i].fin = (i*((((myRank)*(N/nProc)))/T))+
(((myRank+1)*(N/nProc))/T)+(ant/T);
            resultado = pthread_create(&calculan[i],NULL,calcularAtraccion,(void
*) &a_arg[i]);
            if(resultado){
                printf("ERROR creando thread codigo: %d",resultado);
                exit(-1);
            }
       }
//Realizar la simulacion steps veces

      for(i=0; i < steps; i++){

            //Calcular las fuerza de atracción gravitacional para los cuerpos
correspondientes a myRank

            despH = a_arg[T-2].fin;
            despK = despH+TamBlock;
            despJ=0;
            despL = TamBlock;
            for(h=0;h<cantBlock1;h++){
             for(j= 0; j < cantBlock2; j++){
             for(k= despH; k < despK; k++){
                  for(l= despJ;l< despL;l++){
```

```
                        if(l != k){
                          direccionx = posicionx[l] - posicionx[k];
                          direcciony = posiciony[l] - posiciony[k];
                          distancia = sqrt((posicionx[k] -
posicionx[l])*(posicionx[k] - posicionx[l])+(posiciony[k] -
posiciony[l])*(posiciony[k] - posiciony[l]));
                          magnitud = ((6.67e-11) * masa[k] * masa[l]) /
(distancia*distancia);
                          fuerzax[k] += magnitud * direccionx / distancia;
                          fuerzay[k] += magnitud * direcciony / distancia;
                        }
                    }
             }
             despJ=despL;
             despL+=TamBlock;
             }
             despJ = 0;
             despL = TamBlock;
             despH=despK;
             despK+=TamBlock;
            }


//Calcula las nuevas velocidades y posiciones de cada cuerpo en el bloque de mi
rank
                for(k = a_arg[T-2].fin; k < (myRank+1)*(N/nProc); k++){

                    deltavx = fuerzax[k] / masa[k] *DT;
                    deltavy = fuerzay[k] / masa[k] * DT;

                  fuerzax[k] = fuerzay[k] = 0.0;

                  deltapx = (velocidadx[k] + deltavx / 2 ) * DT;
                  deltapy = (velocidady[k] + deltavy / 2 ) * DT;

                  posicionx[k] = posicionx[k] + deltapx;
                  posiciony[k] = posiciony[k] + deltapy;

                  velocidadx[k] = velocidadx[k] + deltavx;
                  velocidady[k] = velocidady[k] + deltavy;
                }

             pthread_barrier_wait(&barrera);

MPI_Allgather(&velocidadx[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&velocidadx[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&posiciony[myRank*(N/nProc)], (N/nProc), MPI_FLOAT, &posiciony[0],
(N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&velocidady[myRank*(N/nProc)], (N/nProc), MPI_FLOAT,
&velocidady[0], (N/nProc), MPI_FLOAT, MPI_COMM_WORLD);
MPI_Allgather(&posicionx[myRank*(N/nProc)], (N/nProc), MPI_FLOAT, &posicionx[0],
(N/nProc), MPI_FLOAT, MPI_COMM_WORLD);

             pthread_barrier_wait(&barrera);
      }

      for(i=0; i<T-1; i++){
          pthread_join(calculan[i],NULL);
      }

}
```

```
//Liberar la memoria utilizada
free(fuerzax);
free(fuerzay);
free(velocidadx);
free(velocidady);
free(posicionx);
free(posiciony);
free(masa);

MPI_Finalize();

return(0);
}
}
```

5. Código del algoritmos paralelo en GPU-CUDA

```
#include<stdio.h>
 #include<stdlib.h>
 #include<math.h>
 #include<sys/time.h>

int DT, steps, N, inicio, fin, T;
float *velocidadx, *fuerzax, *posicionx, *d_velocidadx, *d_fuerzax,
*d_posicionx,*velocidady,*posiciony,*fuerzay,*d_velocidady,*d_posiciony,*d_fuerz
ay;
float *masa, *d_masa;

  //Para calcular tiempo
 double dwalltime(){
          double sec;
          struct timeval tv;

          gettimeofday(&tv,NULL);
          sec = tv.tv_sec + tv.tv_usec/1000000.0;
          return sec;
 }

  __global__ void calcularAtraccionGravitacional(float *fuerzax,float
*fuerzay,float *posicionx,float *posiciony, float *velocidadx, float
*velocidady,float *masa,int N, int DT, int steps){

    //Indice dentro de la memoria global
    int tx = blockDim.x * blockIdx.x + threadIdx.x;
int p;
__shared__ float posicionesx[256];
__shared__ float posicionesy[256];
__shared__ float masas[256];

 float distancia, magnitud;
  int  j, tile,i;
 float myPosicionx, myPosiciony;
float forcex, forcey,myMasa;

for(p=0;p<steps;p++){
```

```
forcex = fuerzax[tx];
forcey = fuerzay[tx];

myPosicionx =   posicionx[tx];
myPosiciony = posiciony[tx];
myMasa = masa[tx];

float direcciony, direccionx;

float calculo;
int idtile;
tile=0;
for(j=0; j < N; j+=blockDim.x){
      idtile = tile * blockDim.x + threadIdx.x;
      posicionesx[threadIdx.x] = posicionx[idtile];
      posicionesy[threadIdx.x] = posiciony[idtile];
      masas[threadIdx.x] = masa[idtile];
      __syncthreads();

      for(i=0; i < blockDim.x;i+=4){
            if(i != threadIdx.x){
            calculo =(((myPosicionx - posicionesx[i])*(myPosicionx -
posicionesx[i]))+((myPosiciony - posicionesy[i])*(myPosiciony -
posicionesy[i])));
            distancia = sqrtf(calculo);
            magnitud = ((6.67e-11)*myMasa*masas[i])/(distancia*distancia);
            direccionx = posicionesx[i] - myPosicionx;
            direcciony = posicionesy[i] - myPosiciony;
            forcex = forcex + magnitud * direccionx / distancia;
            forcey = forcey + magnitud * direcciony / distancia;
            }
             if((i+1) != threadIdx.x){
                  calculo =(((myPosicionx - posicionesx[(i+1)])*(myPosicionx -
posicionesx[(i+1)]))+((myPosiciony - posicionesy[(i+1)])*(myPosiciony -
posicionesy[(i+1)])));
                  distancia = sqrtf(calculo);
                  magnitud = ((6.67e-11)*myMasa*masas[(i+1)])/
(distancia*distancia);
                  direccionx = posicionesx[(i+1)] - myPosicionx;
                  direcciony = posicionesy[(i+1)] - myPosiciony;
                  forcex = forcex + magnitud * direccionx / distancia;
                  forcey = forcey + magnitud * direcciony / distancia;
                  }
             if((i+2) != threadIdx.x){
                  calculo =(((myPosicionx - posicionesx[(i+2)])*(myPosicionx -
posicionesx[(i+2)]))+((myPosiciony - posicionesy[(i+2)])*(myPosiciony -
posicionesy[(i+2)])));
                  distancia = sqrtf(calculo);
                  magnitud = ((6.67e-11)*myMasa*masas[(i+2)])/
(distancia*distancia);
                  direccionx = posicionesx[(i+2)] - myPosicionx;
                  direcciony = posicionesy[(i+2)] - myPosiciony;
                  forcex = forcex + magnitud * direccionx / distancia;
                  forcey = forcey + magnitud * direcciony / distancia;
                 }
             if((i+3) != threadIdx.x){
                  calculo =(((myPosicionx - posicionesx[(i+3)])*(myPosicionx -
posicionesx[(i+3)]))+((myPosiciony - posicionesy[(i+3)])*(myPosiciony -
posicionesy[(i+3)])));
                  distancia = sqrtf(calculo);
                  magnitud = ((6.67e-11)*myMasa*masas[(i+3)])/
```

```
(distancia*distancia);
                direccionx = posicionesx[(i+3)] - myPosicionx;
                direcciony = posicionesy[(i+3)] - myPosiciony;
                forcex = forcex + magnitud * direccionx / distancia;
                forcey = forcey + magnitud * direcciony / distancia;
                }
        }
      __syncthreads();
      tile++;
}
float deltavx,deltavy,deltapx, deltapy;

deltavx = forcex / myMasa * DT;
                 deltavy = forcey / myMasa * DT;

                 fuerzax[tx] = fuerzay[tx] = 0.0; //se resetea el vector de
fuerzas

                 deltapx = (velocidadx[tx] + deltavx / 2 ) * DT;
                 deltapy = (velocidady[tx] + deltavy / 2 ) * DT;

                 posicionx[tx] = posicionx[tx] + deltapx;
                 posiciony[tx] = posiciony[tx] + deltapy;

                 velocidadx[tx] = velocidadx[tx] + deltavx;
                 velocidady[tx] = velocidady[tx] + deltavy;

}

}

int main(int argc, char *argv[]){

if ((argc != 6) || ((N = atoi(argv[1])) <= 0) || ((steps = atoi(argv[2])) <= 0)
|| ((DT = atoi(argv[3])) <= 0) || ((inicio = atoi(argv[4]))< 0) ||
((fin=atoi(argv[5]))<= 0))){
      printf("%s\n","Error faltan parámetros");
      return 1;

 }else{
   double timetick;

   //Reserva memoria en el host
   velocidadx = (float*)malloc(sizeof(float)*N);
   fuerzax = (float*)malloc(sizeof(float)*N);
   posicionx = (float*)malloc(sizeof(float)*N);
   velocidady = (float*)malloc(sizeof(float)*N);
   fuerzay = (float*)malloc(sizeof(float)*N);
   posiciony = (float*)malloc(sizeof(float)*N);
   masa = (float*)malloc(sizeof(float)*N);

   //Reserva memoria en el device
   cudaMalloc(&d_fuerzax,sizeof(float)*N);
   cudaMalloc(&d_fuerzay,sizeof(float)*N);
   cudaMalloc(&d_posicionx,sizeof(float)*N);
   cudaMalloc(&d_posiciony,sizeof(float)*N);
   cudaMalloc(&d_masa,sizeof(float)*N);
   cudaMalloc(&d_velocidady,sizeof(float)*N);
   cudaMalloc(&d_velocidadx, sizeof(float)*N);

printf("\nCantidad de threads: %i",256);
```

```
printf("\nPasos de la simulacion: %i\n",steps);
printf("Tamaño del problema %i\n\n",N);

  int i;
   //inicializar posiciones, velocidad, fuerza y masa
         for(i=0 ; i < N; i++){
                fuerzax[i] = (rand() % (fin-inicio+1) + inicio);
                fuerzay[i] = (rand() % (fin-inicio+1) + inicio);
                posicionx[i] = rand() % (fin-inicio+1) + inicio;
                posiciony[i] = rand() % (fin-inicio+1) + inicio;
                velocidadx[i] = rand() % (fin-inicio+1) + inicio;
                velocidady[i] = rand() % (fin-inicio+1) + inicio;
                masa[i] = rand() % (fin-inicio+1) + inicio;
         }

timetick= dwalltime();


//Transferencia de datos del host al device
   cudaMemcpy(d_fuerzax,fuerzax, sizeof(float)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(d_velocidadx,velocidadx, sizeof(float)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(d_posicionx,posicionx, sizeof(float)*N, cudaMemcpyHostToDevice);
   cudaMemcpy(d_masa,masa, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_fuerzay,fuerzay, sizeof(float)*N, cudaMemcpyHostToDevice);
  cudaMemcpy(d_velocidady,velocidady, sizeof(float)*N,
cudaMemcpyHostToDevice);
  cudaMemcpy(d_posiciony,posiciony, sizeof(float)*N, cudaMemcpyHostToDevice);


//Calcular la atraccion gravitacional de los N-bodys en el paso i
calcularAtraccionGravitacional<<<N/256,256>>>(d_fuerzax,d_fuerzay,d_posicionx,d_
posiciony,d_velocidadx,d_velocidady,d_masa,N,DT,steps);

 cudaThreadSynchronize();

//Transferencia de datos desde el device al host
cudaMemcpy(velocidady,d_velocidady,sizeof(float)*N,cudaMemcpyDeviceToHost);
cudaMemcpy(velocidadx,d_velocidadx,sizeof(float)*N,cudaMemcpyDeviceToHost);
cudaMemcpy(posiciony,d_posiciony,sizeof(float)*N,cudaMemcpyDeviceToHost);
cudaMemcpy(posicionx,d_posicionx,sizeof(float)*N,cudaMemcpyDeviceToHost);

printf("\nEl tiempo en segundos es: %f\n",dwalltime()-timetick);

free(fuerzax);
free(fuerzay);
free(posicionx);
free(posiciony);
free(velocidadx);
free(velocidady);
free(masa);

cudaFree(d_fuerzax);
cudaFree(d_posicionx);
cudaFree(d_velocidadx);
cudaFree(d_fuerzay);
cudaFree(d_velocidady);
cudaFree(d_posiciony);
cudaFree(d_masa);
}
}
```

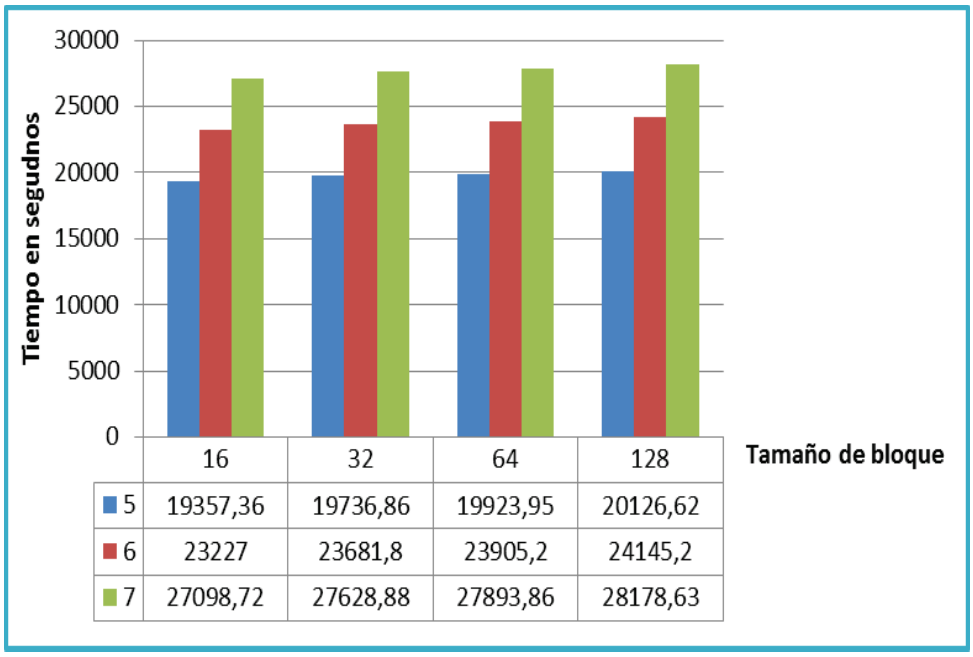
Apéndice B

Búsqueda del tamaño de bloque óptimo

Se realizaron pruebas para los dos tamaños de problema y los distintos tipos de configuraciones en cada versión del algoritmo para hallar el tamaño de bloque óptimo. A continuación, se presentan los resultados obtenidos:

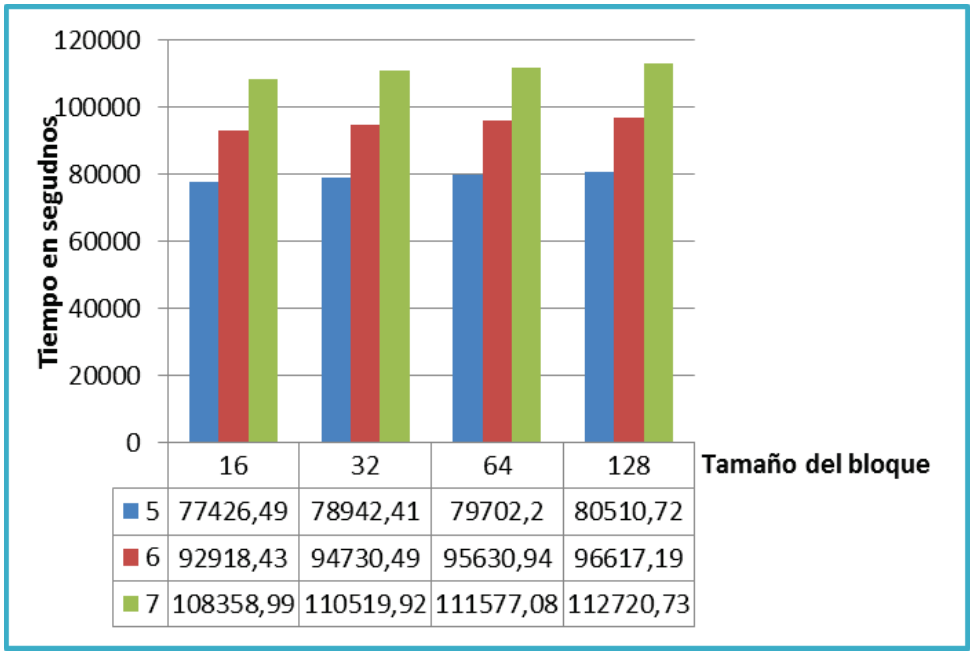
B.1. Algoritmo secuencial

La solución secuencial del problema se hicieron pruebas para las cantidades de cuerpos 256000 y 512000, para 5, 6 y 7 pasos de simulación con tamaños de bloque: 16, 32, 64 y 128.



	16	32	64	128
5	19357,36	19736,86	19923,95	20126,62
6	23227	23681,8	23905,2	24145,2
7	27098,72	27628,88	27893,86	28178,63

Fig. B.1. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000)

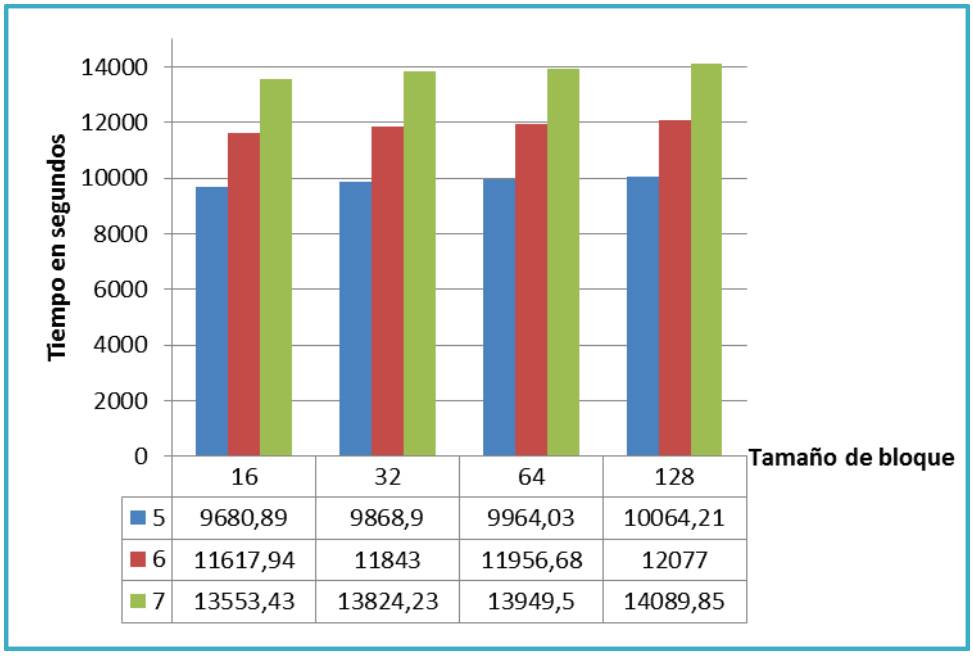


	16	32	64	128
5	77426,49	78942,41	79702,2	80510,72
6	92918,43	94730,49	95630,94	96617,19
7	108358,99	110519,92	111577,08	112720,73

Fig. B.2. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000)

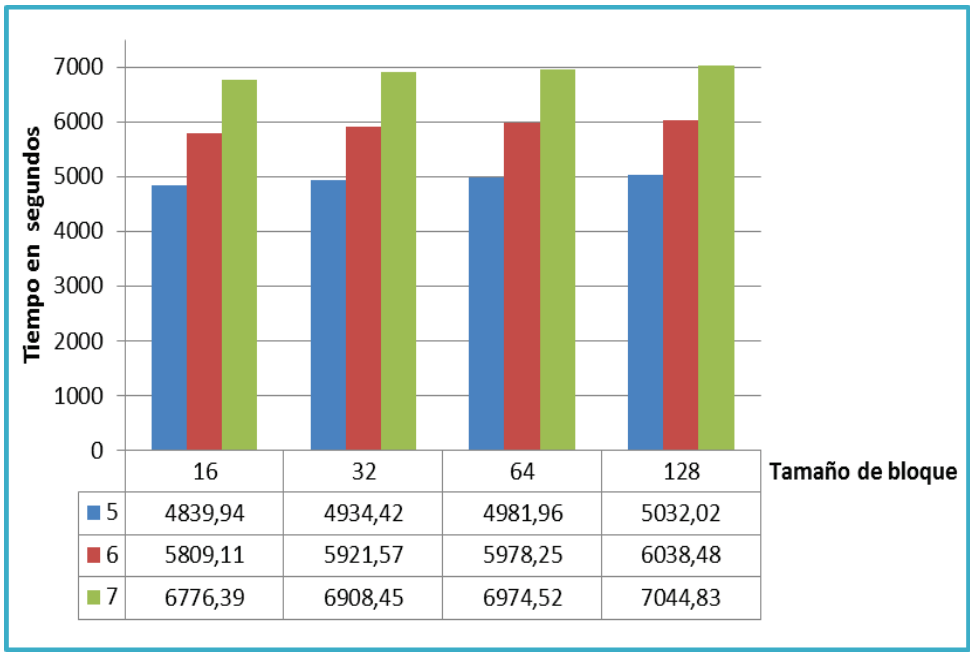
B.2. Algoritmo paralelo en memoria compartida

Para el algoritmo paralelo en Pthread se hicieron pruebas para las cantidades de cuerpos 256000 y 512000, con 2, 4 y 8 threads de ejecución, en 5, 6 y 7 pasos con tamaños de bloque: 16, 32, 64 y 128.



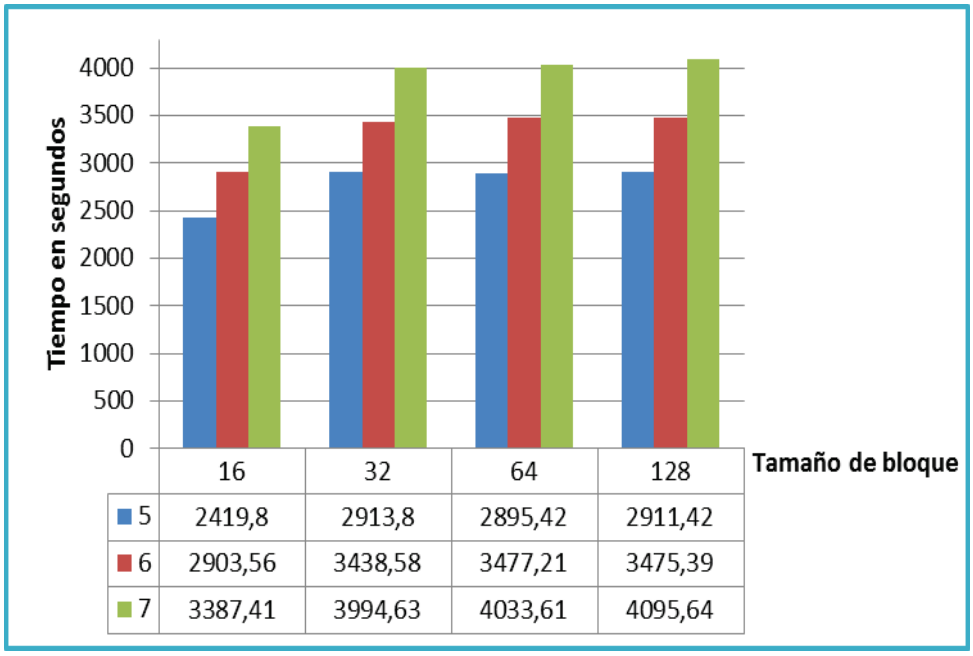
	16	32	64	128
5	9680,89	9868,9	9964,03	10064,21
6	11617,94	11843	11956,68	12077
7	13553,43	13824,23	13949,5	14089,85

Fig. B.3. Comparación de los tiempos de ejecución obtenidos para l las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de threads = 2)



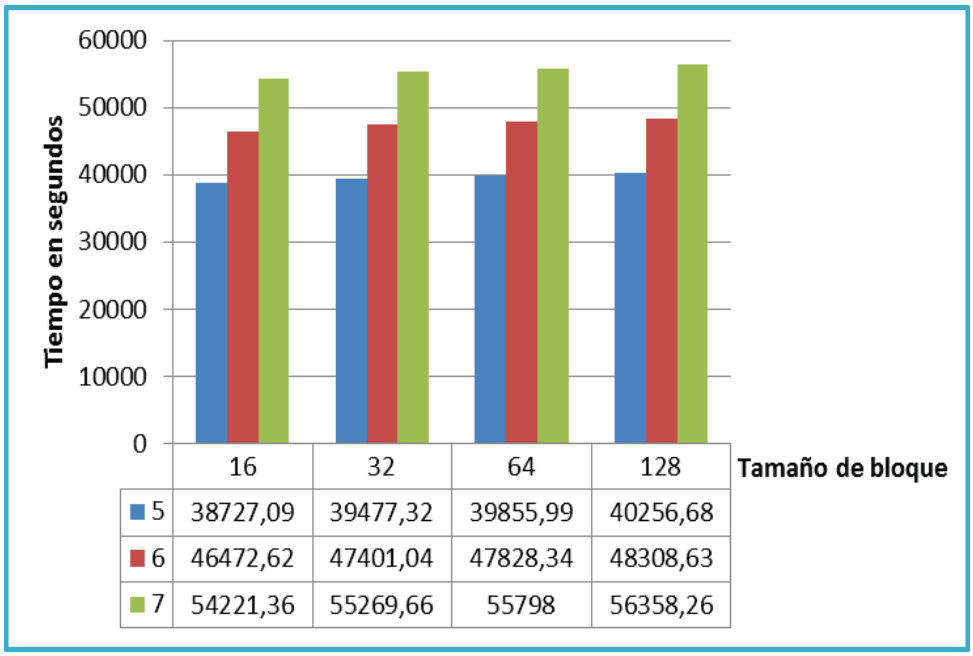
	16	32	64	128
5	4839,94	4934,42	4981,96	5032,02
6	5809,11	5921,57	5978,25	6038,48
7	6776,39	6908,45	6974,52	7044,83

Fig. B.4. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de threads = 4)



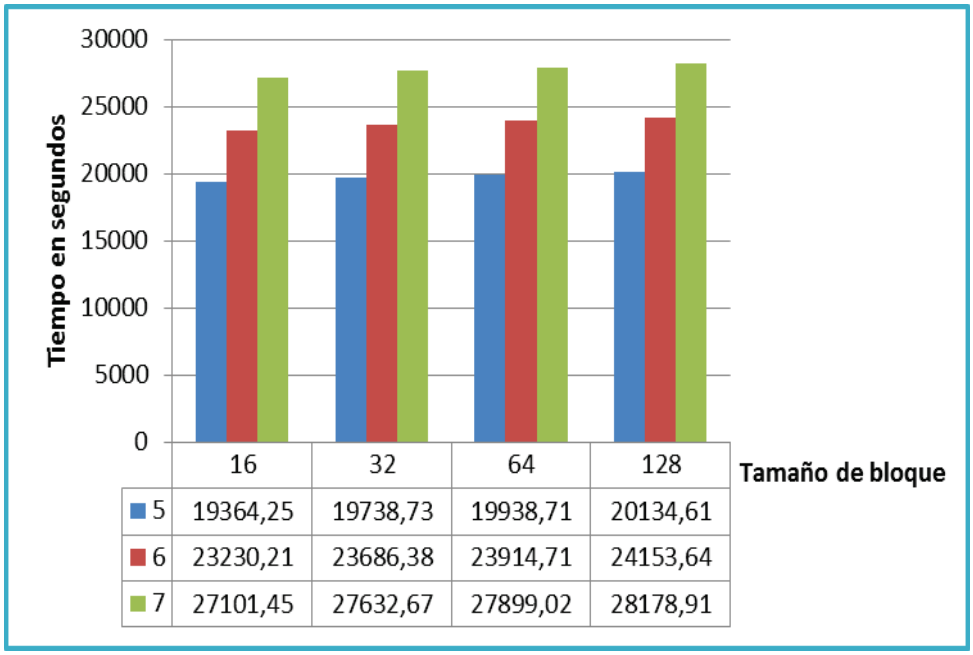
	16	32	64	128
5	2419,8	2913,8	2895,42	2911,42
6	2903,56	3438,58	3477,21	3475,39
7	3387,41	3994,63	4033,61	4095,64

Fig. B.5. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de threads = 8)



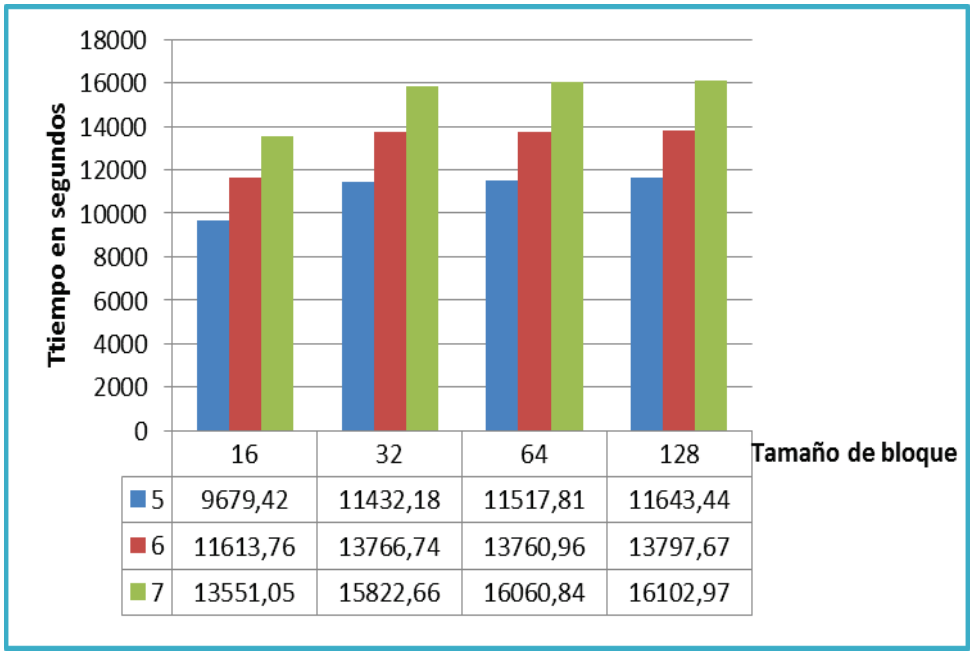
	16	32	64	128
5	38727,09	39477,32	39855,99	40256,68
6	46472,62	47401,04	47828,34	48308,63
7	54221,36	55269,66	55798	56358,26

Fig. B.6. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de threads = 2)



	16	32	64	128
5	19364,25	19738,73	19938,71	20134,61
6	23230,21	23686,38	23914,71	24153,64
7	27101,45	27632,67	27899,02	28178,91

Fig. B.7. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de threads = 4)

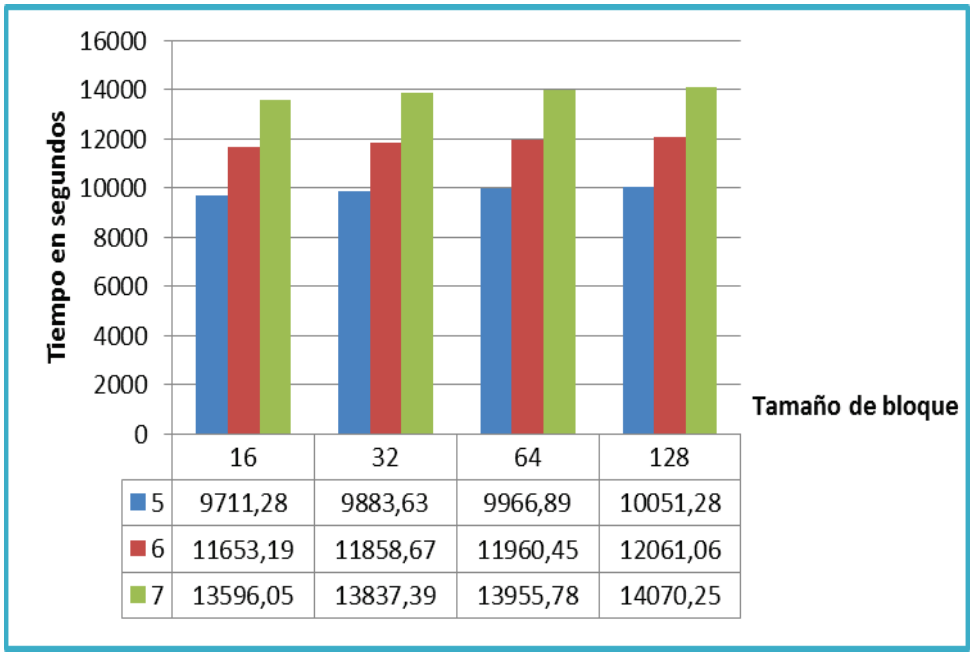


	16	32	64	128
5	9679,42	11432,18	11517,81	11643,44
6	11613,76	13766,74	13760,96	13797,67
7	13551,05	15822,66	16060,84	16102,97

Fig. B.8. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de threads = 8)

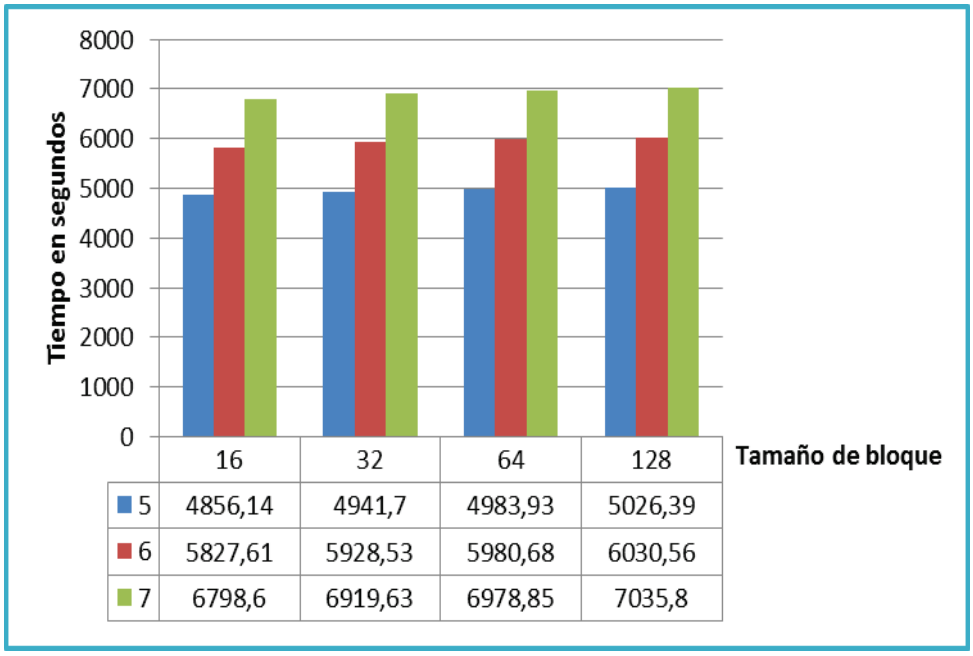
B.3. Algoritmo paralelo en memoria distribuida

Se realizaron pruebas para el algoritmo paralelo en memoria distribuida con MPI para 256000 y 512000 cuerpos, con 2, 4 y 8 procesos, en 5, 6 y 7 pasos con tamaños de bloque: 16, 32, 64 y 128.



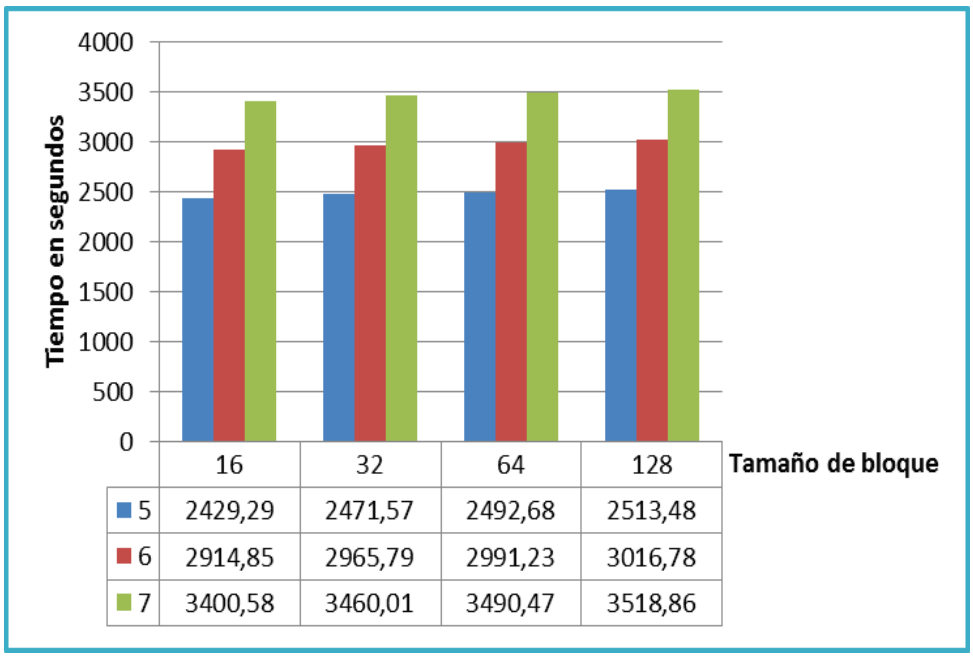
	16	32	64	128
5	9711,28	9883,63	9966,89	10051,28
6	11653,19	11858,67	11960,45	12061,06
7	13596,05	13837,39	13955,78	14070,25

Fig. B.9. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 2)



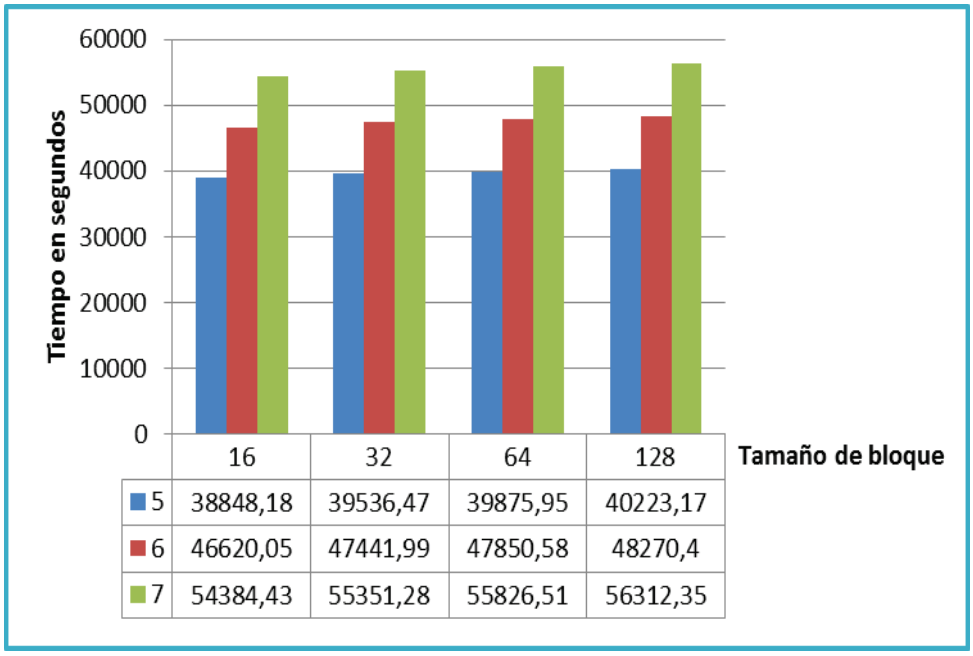
	16	32	64	128
5	4856,14	4941,7	4983,93	5026,39
6	5827,61	5928,53	5980,68	6030,56
7	6798,6	6919,63	6978,85	7035,8

Fig. B.10. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 4)



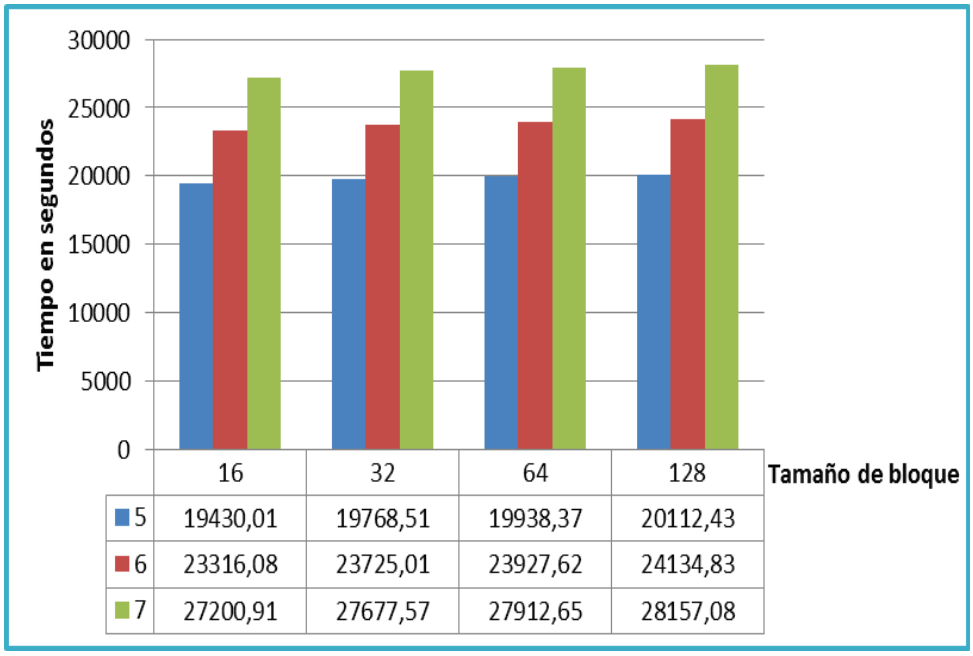
	16	32	64	128
5	2429,29	2471,57	2492,68	2513,48
6	2914,85	2965,79	2991,23	3016,78
7	3400,58	3460,01	3490,47	3518,86

Fig. B.11. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 8)



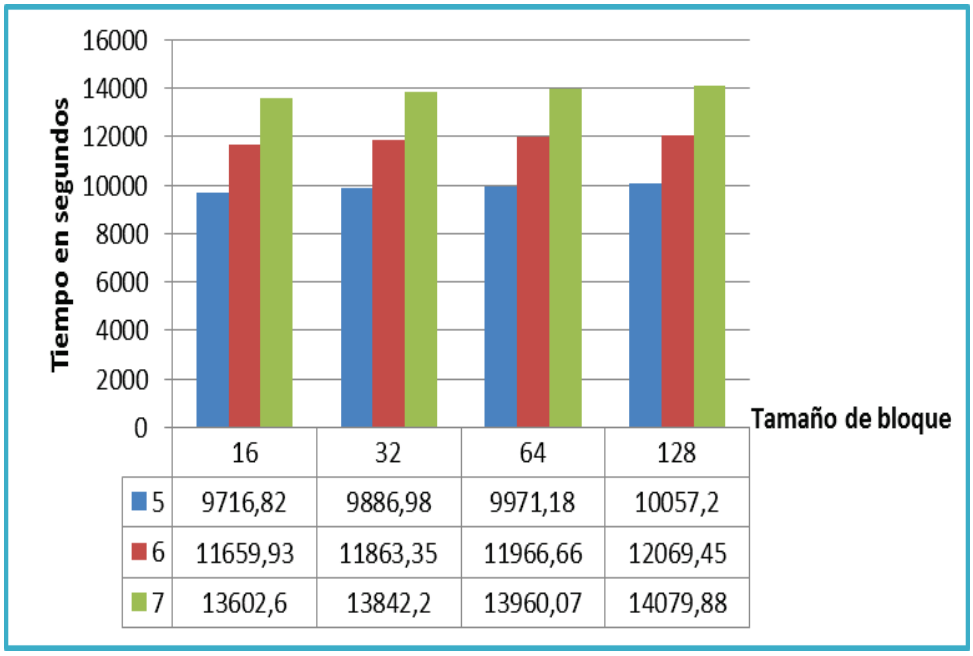
	16	32	64	128
5	38848,18	39536,47	39875,95	40223,17
6	46620,05	47441,99	47850,58	48270,4
7	54384,43	55351,28	55826,51	56312,35

Fig. B.12. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 2)



	16	32	64	128
5	19430,01	19768,51	19938,37	20112,43
6	23316,08	23725,01	23927,62	24134,83
7	27200,91	27677,57	27912,65	28157,08

Fig. B.13. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 4)

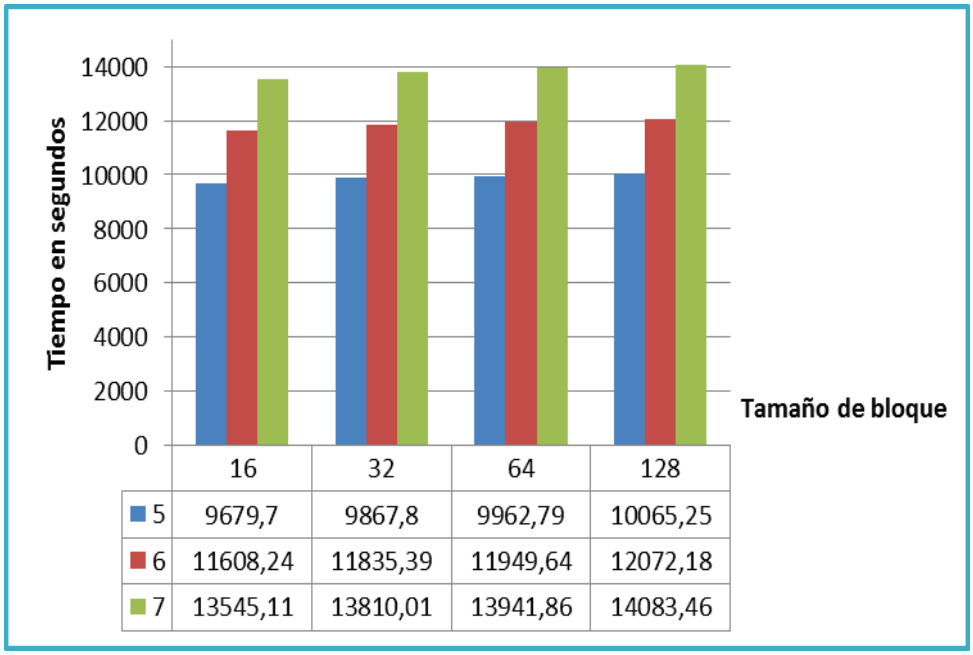


	16	32	64	128
5	9716,82	9886,98	9971,18	10057,2
6	11659,93	11863,35	11966,66	12069,45
7	13602,6	13842,2	13960,07	14079,88

Fig. B.14. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 8)

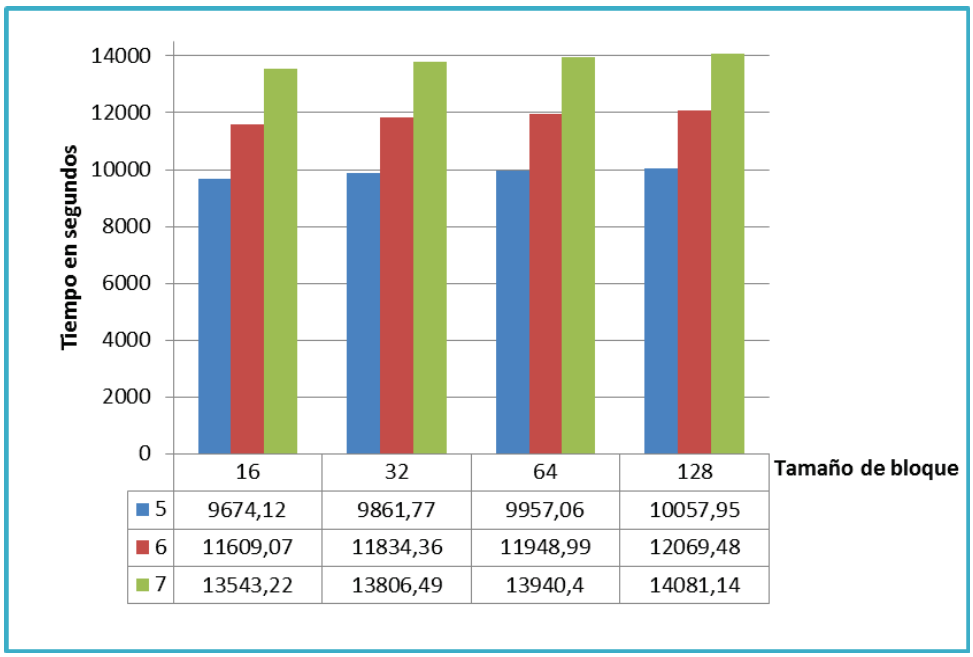
B.4. Algoritmo paralelo híbrido

Para el algoritmo híbrido MPI-Pthread se hicieron pruebas para las cantidades de cuerpos 256000 y 512000, combinando 2 proceso con 2 threads, 2 proceso con 4 threads y 4 procesos con 2 threads, en 5, 6 y 7 pasos con tamaños de bloque: 16, 32, 64 y 128.



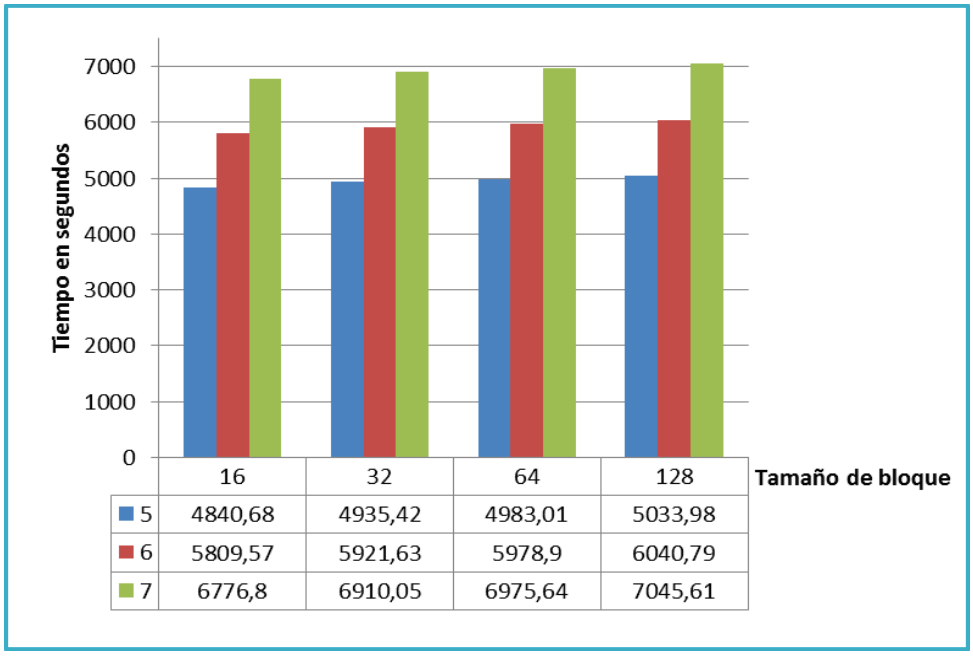
	16	32	64	128
5	9679,7	9867,8	9962,79	10065,25
6	11608,24	11835,39	11949,64	12072,18
7	13545,11	13810,01	13941,86	14083,46

Fig. B.15. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 2, cantidad de threads = 2)



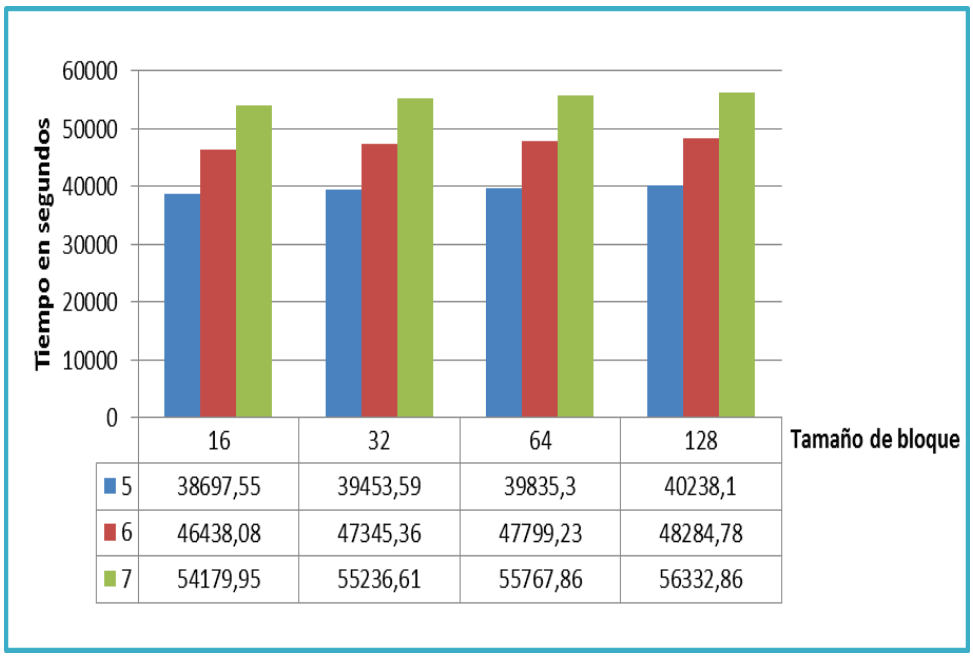
	16	32	64	128
5	9674,12	9861,77	9957,06	10057,95
6	11609,07	11834,36	11948,99	12069,48
7	13543,22	13806,49	13940,4	14081,14

Fig. B.16. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 2, cantidad de threads = 4)



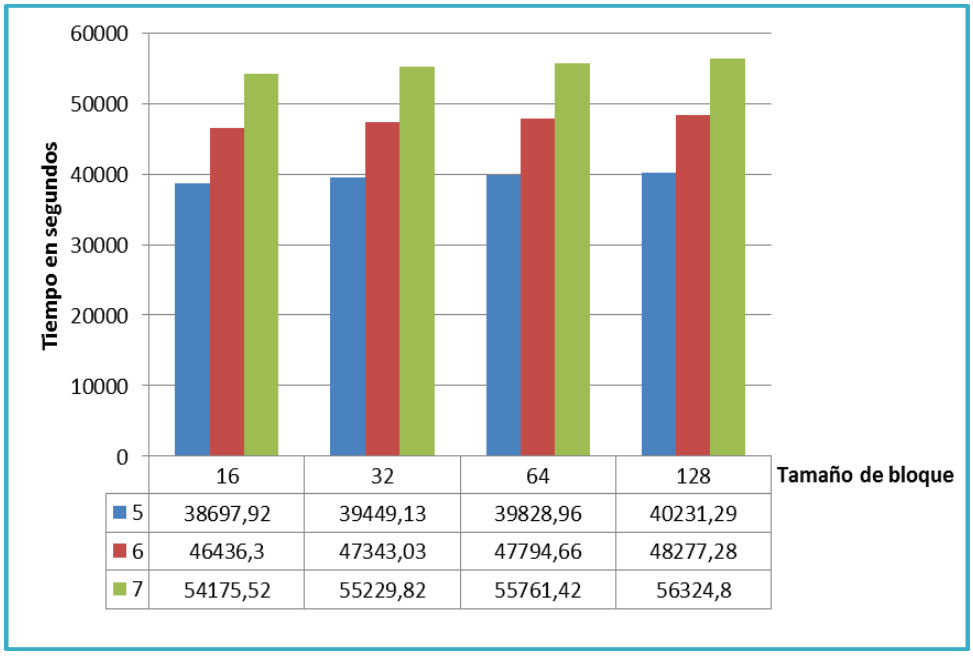
	16	32	64	128
5	4840,68	4935,42	4983,01	5033,98
6	5809,57	5921,63	5978,9	6040,79
7	6776,8	6910,05	6975,64	7045,61

Fig. B.17. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 256000, cantidad de procesos = 4, cantidad de threads = 2)



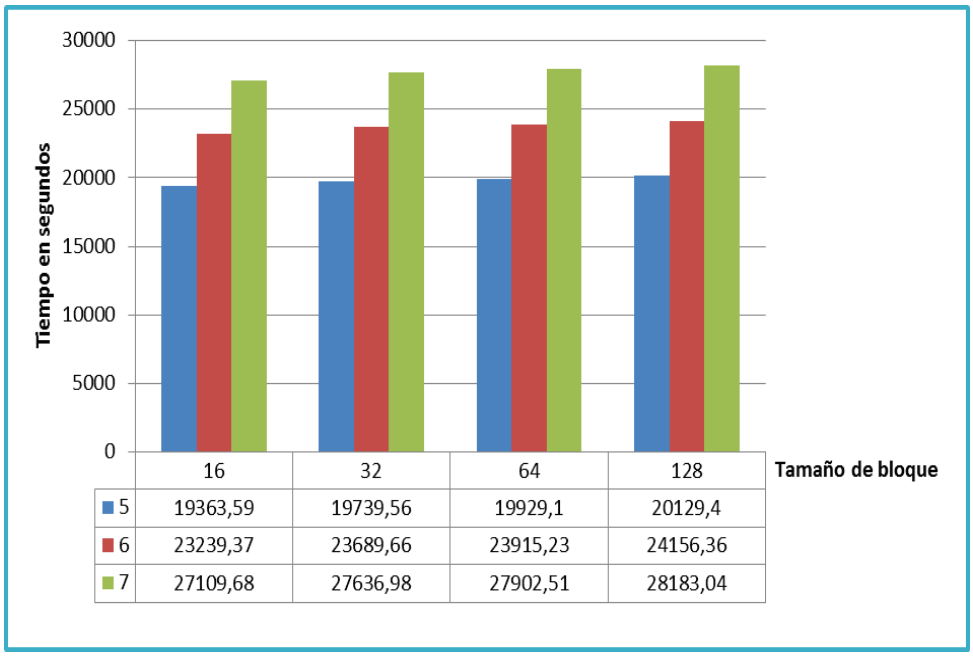
	16	32	64	128
5	38697,55	39453,59	39835,3	40238,1
6	46438,08	47345,36	47799,23	48284,78
7	54179,95	55236,61	55767,86	56332,86

Fig. B.18. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 2, cantidad de threads = 2)



	16	32	64	128
5	38697,92	39449,13	39828,96	40231,29
6	46436,3	47343,03	47794,66	48277,28
7	54175,52	55229,82	55761,42	56324,8

Fig. B.19. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 2, cantidad de threads = 4)



	16	32	64	128
5	19363,59	19739,56	19929,1	20129,4
6	23239,37	23689,66	23915,23	24156,36
7	27109,68	27636,98	27902,51	28183,04

Fig. B.20. Comparación de los tiempos de ejecución obtenidos para las distintas configuraciones del problemas (cantidad de cuerpos = 512000, cantidad de procesos = 4, cantidad de threads = 2)

B.5. Algoritmo paralelo en GPU

CUDA define una jerarquía de abstracción de los threads como: Grid de bloques, Bloques de threads y Threads (como se explica con más detalle en el Capítulo 2 del presente trabajo). Dependiendo de la Capacidad de Cómputo de la tarjeta gráfica utilizada, las cantidades de threads por dimensiones en los bloques de la grid están limitadas a 512 para la arquitectura Tesla y 1024 para la arquitectura Fermi (Para mayor detalle ver Apéndice C). La GeForce TX 560 Ti pertenece a la arquitectura Fermi, la máxima cantidad de threads a definir en la dimensión x e y es 1024. Sin embargo, generar bloques con la máxima cantidad de thread trae como consecuencia una pérdida de la performance. Las pruebas para las cantidades de cuerpos 256000 y 512000, se realizaron para bloques de 256 y 512 threads, para los 5, 6 y 7 pasos de simulación.

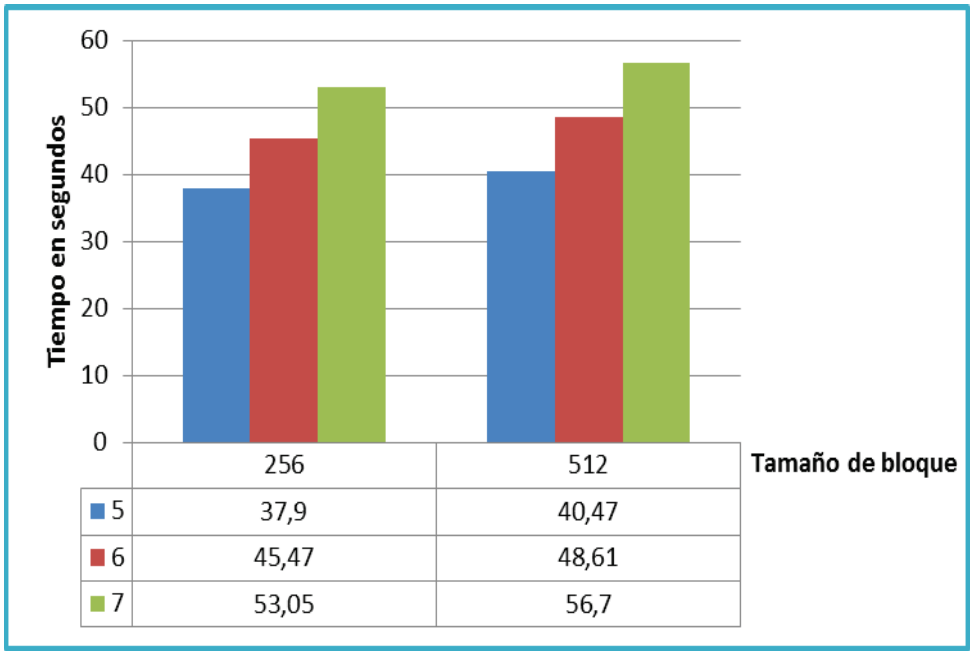


Fig. B.21. Comparación de los tiempos de ejecución obtenidos para los distintos tamaños de bloques (cantidad de cuerpos = 256000)

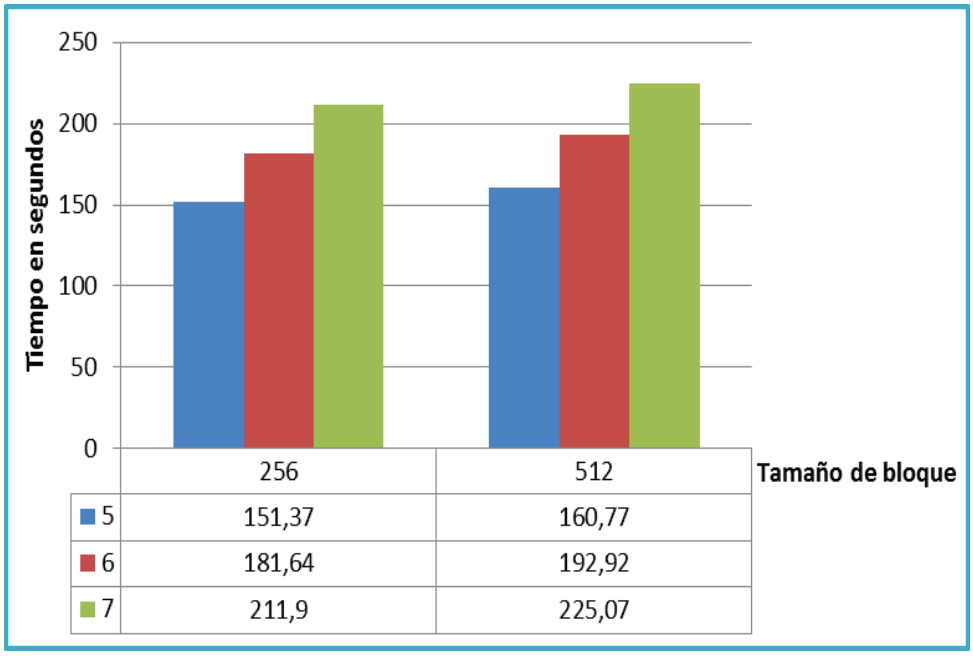


Fig. B.22. Comparación de los tiempos de ejecución obtenidos para los distintos tamaños de bloques (cantidad de cuerpos = 512000)

Apéndice C

Capacidad de Cómputo: GPUs Nvidia

Resumen de las características de las distintas Capacidades de Cómputo (C.C.) de las GPUs Nvidia. Se debe tener en cuenta que las C.C. últimas incluyen a las anteriores.

<table>
<tr><th rowspan="2">Características soportadas</th><th colspan="5">Capacidad de Cómputo</th></tr>
<tr><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th></tr>
<tr><td>Grid 3D</td><td colspan="4">No</td><td>Si</td></tr>
<tr><td>Funciones atómicas operando sobre la memoria global con palabras de 32 bit.</td><td>No</td><td colspan="4">Si</td></tr>
<tr><td>Funciones atómicas de enteros sobre la memoria global con palabras de 64 bits.</td><td colspan="2" rowspan="3">No</td><td colspan="3" rowspan="3">Si</td></tr>
<tr><td>Funciones atómicas de enteros sobre la memoria compartida con palabras de 32 bits.</td></tr>
<tr><td>Funciones de voto de warp.</td></tr>
<tr><td>Números de punto flotante de doble precisión.</td><td colspan="3">No</td><td colspan="2">Si</td></tr>
<tr><td>Operaciones de suma atómica de números de punto flotante sobre la memoria compartida y global para palabras de 32 bits.</td><td colspan="4" rowspan="5">No</td><td rowspan="5">Si</td></tr>
<tr><td>__ballot()</td></tr>
<tr><td>__threadfence_system()</td></tr>
<tr><td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or()</td></tr>
<tr><td>Funciones de Superficie</td></tr>
</table>

En la siguiente tabla se muestran las especificaciones técnicas de cada una de las C.C.

	Capacidad de Cómputo				
Especificaciones Técnicas	1.0	1.1	1.2	1.3	2.x
Máxima dimensión de un *grid*.	2				3
Máximo de la dimensión *x*, *y* y *z* de un *grid*.	65535				
Máxima cantidad de dimensiones de los bloques de *threads*.	3				
Máximo de la dimensión x e y de un bloque de *threads*.	512				1024
Máximo de la dimensión z en un bloque.	64				
Máximo número de *threads* por bloques.	512				1024
Tamaño de un *warp*	32				
Máximo número de bloques residentes en un SM.	8				
Máximo número de *warps* residentes por SM.	24		32		48
Máximo número de *threads* residentes por SM.	768		1024		1536
Número de registros de 32-bit por SM.	8 K		16 K		32 K
Máxima cantidad de memoria compartida por SM.	16 KB				48 KB
Número de bancos de memoria compartida.	16				32
Cantidad de memoria local por *thread*.	16 KB				512 KB
Tamaño de la memoria constante.	64 KB				
Cache por SM para memoria constante.	8 KB				
Cache por SM para memoria de textura.	Depende del Dispositivo, entre 6KB y 8KB				
Máximo ancho para una referencia de textura de 1D limitado a un arreglo CUDA.	8192				32768
Máximo ancho para una referencia de textura 1D limitado a memoria lineal.	227				
Máximo ancho y número de capaz para una para una referencia de textura 1D por capas.	8192 x 512				16384 x 2048
Máximo ancho y alto para una referencia de textura 2D limitado a memoria lineal o a un arreglo CUDA.	65536 x 32768				65536 x 65535
Máximo ancho, alto y número de capas para una referencia de textura 2D por capas.	8192 x 8192 x 512				16384 x 16384 x 2048
Máximo ancho, alto y profundidad para referencias de textura 3D limitado a un arreglo CUDA.	2048 x 2048 x 2048				
Máximo número de texturas, las cuales pueden ser limitado a un kernel.	128				
Máximo ancho para una referencia de superficie 1D limitado a un arreglo CUDA.	N/A				8192
Máximo ancho y alto para una referencia de superficie 2D limitado a un arreglo CUDA.					8192 x 8192
Máximo número de superficies, las cuales pueden ser limitadas a un kernel					8
Máximo número de instrucciones por kernel	2 millones				

Apéndice D

Análisis del consumo energético

D.1. Entorno de prueba

Para la ejecución del algoritmo secuencial se utilizó una máquina con un procesador Intel Pentium 4 de 3.20GHz, con una caché de 2 MB y 1009 MB de memoria RAM. Por otro lado, el algoritmo en CUDA fue ejecutado sobre una placa gráfica NVIDIA Geforce GTX 560Ti con 1GB de RAM que posee 384 SPs, distribuidos en 8 SMs.

Para la medición de consumo energético se empleó un osciloscopio digital, con una resolución de 8 bits con dos entradas, una utilizada para capturar la información de la tensión y la otra para la corriente. Esta última proviene de una pinza transductora, cuya sensibilidad puede ajustarse a los siguientes valores: 1A/100mV, 1A/10mV y 1A/1mV.

D.2. Resultados de pruebas de consumo energético

En la presente sección se muestran los resultados obtenidos de las pruebas realizadas sobre consumo energético utilizando los algoritmos expuestos en esta Tesina de Grado.

En el Capítulo 6, se introdujeron las unidades eléctricas fundamentales con las cuales se realizaron las mediciones. En este caso las mismas se realizaron de manera indirecta; obteniendo por un lado la corriente y por otro la tensión. El producto de ambos valores da como resultado la potencia eléctrica.

Fig. D.1. Configuración del Entorno de prueba
(**A**. Equipo en el que se ejecutan las pruebas. **B**. Equipo en el que se recogen los datos. **C**. Osciloscopio digital. **D**. Pinza Transductora. **E**. GPU Geforce TX 560Ti)

La Fig. D.1. muestra la configuración del entorno de prueba utilizado para realizar las mediciones. Como se observa en la imagen la tensión se midió directamente de la línea eléctrica a la cual se encuentra conectado el equipo **A** en el cual se ejecutan los algoritmos. Dicha información es recogida por un osciloscopio digital (**C**), que la envía a otro equipo **B** para ser analizada posteriormente. Para obtener la información de la corriente consumida se utilizó una pinza transductora (**D**) colocada al cable de entrada de la fuente de energía del equipo **A** que envía la información a **C**.

El osciloscopio digital (**C**) coloca los datos capturados en buffers de 10240 muestras (10 KB). Cada buffer representa un tiempo aproximado de 40 milisegundos, lo que da un intervalo de muestreo de 40 ms / 10 KB = 3,9 µs. De dichos buffers se calcula la potencia eléctrica como se mencionó anteriormente. A partir de dicha potencia se obtiene el valor eficaz calculado con al ecuación (1), donde T2 – T1 es igual a 40 ms.

$$F_{rms} = \sqrt{\frac{1}{T2 - T1}\int_{T1}^{T2} f(t)^2 dt} \quad (1)$$

La suma de los valores eficaces da como resultado el valor aproximado de la cantidad de Joules consumidos por la aplicación.

A continuación, en la Tabla 1 se muestran los Joules totales consumidos por la aplicación ejecutados sobre las arquitecturas mencionadas en la sección D.1. Para los tamaños de vector de cuerpos de entrada: 4096, 8192, 16384, 32768 y 65535.

Cantidad de cuerpos	Algoritmo secuencial	Algoritmo CUDA
4096	32,8778	22,0744
8192	117,3122	23,2539
16384	456,3074	29,6185
32768	1695,7236	33,9317
65535	7349,4453	59,0715

Tabla 1. Joules consumidos por los algoritmos ejecutados

Cada buffer de corriente y tensión capturado representa aproximadamente 40ms, por lo cual, si se multiplica dicho valor por la cantidad de buffers totales se obtiene como resultado un tiempo estimativo de lo que tarda la aplicación en ejecutarse. La Tabla 2 muestra los tiempos en segundos.

Cantidad de cuerpos	Algoritmo secuencial	Algoritmo CUDA
4096	0,24	0,16
8192	0,84	0,16
16384	3,28	0,2
32768	12,44	0,2
65535	51,72	0,28

Tabla 2. Tiempo de muestras, en segundos, de la duración de la aplicación

El cociente de los valores de la Tabla 1 y 2 da como resultado la cantidad de Watts que consume cada arquitectura durante un segundo, dichos resultados se presenta en la

Tabla 3.

Cantidad de cuerpos	Algoritmo secuencial	Algoritmo CUDA
4096	136,991	137,9653
8192	139,6574	145,3329
16384	139,1181	148,0929
32768	136,3121	169,6589
65535	142,1006	210,9669
Promedio	138,83584	162,40338

Tabla 3. Watts por segundo de cada arquitectura

De los resultados obtenidos puede observarse que, para el caso de la arquitectura CPU, la variación del consumo de energía es poco significativa. En el caso de la GPU, la variación se produce como consecuencia de las copias de memoria CPU-GPU y GPU-CPU. Por el contrario, en el caso del algoritmo secuencial, al no producirse movimientos de datos la variabilidad es menor.

El consumo total del algoritmo depende de la cantidad de Watts por segundos que insume la aplicación y del tiempo de ejecución del mismo. Si bien, el consumo promedio del algoritmo secuencial por segundo (en Watts) es menor que el caso del algoritmo CUDA, al ser mayor su tiempo de ejecución produce un consumo total mayor.

Apéndice E

El presente apéndice contiene el artículo publicado en el XVIII Congreso Argentino de Ciencias de la Computación (CACIC 2012).

El Congreso Argentino de Ciencias de la Computación (CACIC) es organizado por la Red de Universidades Nacionales con Carreras en Informática (RedUNCI). CACIC reúne desde 1995 a investigadores, docentes, profesionales, alumnos de grado y estudiantes de postgrado vinculados con la disciplina Informática.

El Congreso cubre temas de importancia en Ciencias de la Computación a través de la organización de diferentes Workshops, coordinados por expertos en la disciplina. En estos Workshops se presentan trabajos científicos evaluados por investigadores del país y del exterior.

El artículo publicado fue expuesto en el XII Workshop de Procesamiento Distribuido y Paralelo (CACIC 2012).

Comparación del uso de GPU y cluster de multicore en problemas con alta demanda computacional

Erica Montes de Oca[1], Laura De Giusti[1], Armando De Giusti[1 2], Marcelo Naiouf [1]

[1]Instituto de Investigación en Informática LIDI (III-LIDI)
Facultad de Informática, Universidad Nacional de La Plata.
La Plata, Buenos Aires, Argentina.
[2] CONICET
{emontesdeoca,ldgiusti,degiusti,mnaiouf}@lidi.info.unlp.edu.ar

Resumen. Este trabajo realiza una comparación del uso de dos arquitecturas multiprocesador, tomando como caso de aplicación un problema con alta demanda computacional como el de N-body. Se presentan las implementaciones paralelas con memoria compartida (usando Pthreads) y pasaje de mensajes (con MPI) en cluster de multicore, y una solución sobre GPU (con CUDA). Se describen y analizan los resultados experimentales obtenidos, que muestran la buena performance lograda con el uso de GPU.

Palabras claves: multicore, cluster de multicore, GPU, GPGPU, CUDA, N-Body.

1 Introducción

Los intentos por acelerar el procesamiento secuencial a través del mejoramiento del hardware alcanzó su máxima capacidad cuando ya no se pudo sostenerse el incremento de la frecuencia de reloj. [1]. El gap producido entre el rendimiento esperado y el obtenido de las arquitecturas fabricadas hasta entonces motivó a la industria del hardware a orientar su producción hacia la construcción de máquinas paralelas. La tecnología del semiconductor comenzó a tomar dos caminos principales en la fabricación de las nuevas arquitecturas de cómputo: los multicores y los manycore [2].

Los multicores resolvieron el problema al que se enfrentaban los arquitectos de hardware, reduciendo la complejidad de los procesadores además de doblar la cantidad de núcleos de procesamiento con cada nueva generación [3]. El impacto producido en el mercado desplazando a las supercomputadoras permitió reducir los costos y la infraestructura garantizando la ejecución de problemas complejos con alta demanda computacional [4].

Los cluster son un conjunto de equipos independientes que pueden ser vistos como un sola máquina unificada [5]. Cada cluster puede estar conformado por máquinas iguales (homogéneo) o diferentes (heterogéneo). El cluster de multicores nace como una consecuencia del avance tecnológico y la necesidad de un mayor poder de cómputo.

La GPU (Unidad de Procesamiento Gráfico) ha emergido en los últimos años como una plataforma alternativa para el procesamiento paralelo a costo accesible. Es

una arquitectura manycore caracterizada por un número masivo de procesadores simples, cuyo uso avanzó en otros campos además del procesamiento de imágenes. Las GPUs modernas pueden ser descriptas como un vector de procesadores que ejecutan la misma instrucción sobre diferentes datos de forma paralela [6][7][8][9].

Un problema con alta demanda computacional ampliamente estudiado es el de los N-body, caracterizando una variedad de casos que van desde la interacción de las partículas electrostáticas hasta el movimiento de los cuerpos astronómicos en el espacio.

Este trabajo presenta una comparación de soluciones al problema de la atracción gravitacional de los cuerpos celestes utilizando cluster de multicore y GPU. La Sección 2 plantea el problema de los N-body. En la Sección 3 se describen las soluciones desarrolladas, mientras que en la Sección 4 se muestran los resultados experimentales obtenidos. La Sección 5 presenta las conclusiones y trabajos futuros.

2 El problema de los N-body

Caracteriza una variedad de problemas que van desde la interacción de las partículas electrostáticas hasta el movimiento de los cuerpos astronómicos en el espacio [10]. El presente artículo se basa en una solución al problema de la atracción gravitacional de los cuerpos celestes, apoyada en la teoría desarrollada por Newton, quien descubrió que la fuerza de atracción entre dos cuerpos es proporcional a sus masas e inversamente proporcional al cuadrado de la distancia que los separa.

Cada cuerpo cuenta con una masa (m), una posición inicial (p) y una velocidad (v). La gravedad hace que los cuerpos se aceleren y se muevan, provocando la atracción entre ellos. La relación entre masa y peso se puede obtener del Principio de Masa de Newton (Ecuación 0):

$F = m \times a$ donde F = cantidad de fuerza, m = masa, a = aceleración **(0)**

Durante un pequeño intervalo de tiempo (dt), la aceleración del cuerpo i (a_i) es aproximadamente constante, por lo que el cambio de velocidad del cuerpo (dv_i) es el indicado en la Ecuación 1:

$$dv_i = a_i \times dt \quad \textbf{(1)}$$

El cambio de la posición del cuerpo (dp_i) puede ser calculado como la integral de la velocidad y la aceleración en el intervalo de tiempo dt, que es aproximadamente el indicado en la Ecuación 2:

$$dp_i = v_i \times dt + (a_i/2) \times dt^2 \quad \textbf{(2)}$$

También, puede calcularse la magnitud de la fuerza de gravedad entre dos cuerpos i y j a través de la Ecuación 3 expresada por Newton:

$F = (G \times m_i \times m_j) / r^2$ *con r = distancia ,G = cte gravitacional (6,67 x 10-11)* **(3)**

Si los cuerpos se hallan en un plano espacial de dos dimensiones, las posiciones pueden ser representadas como puntos coordenados en el plano tal como: ($p_i.x$, $p_i.y$) y ($p_j.x$, $p_j.y$). El plano espacial donde se hallan los cuerpos se lo denomina espacio euclídeo. La Ecuación 4 mide la distancia euclídea entre dos puntos:

$$\sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2} \quad \textbf{(4)}$$

Si la distancia de dos cuerpos es pequeña significa que los mismos están muy cerca de colisionar. En esos casos se considera que se producen choques entre los cuerpos, que es lo que sucede naturalmente en el espacio [10][11].

3 Descripción de las soluciones

A continuación se describen los algoritmos desarrollados en este trabajo para resolver el problema.

3.1 Solución secuencial

La implementación de la versión secuencial se realizó mediante el algoritmo Fast N-Body Simulation [12], que realiza básicamente las siguientes acciones:

1. Calcula la fuerza de atracción del cuerpo *i*, con respecto a los demás cuerpos.
2. Calcula la nueva posición y la velocidad del cuerpo *i*.

El cálculo de la fuerza de gravedad de un cuerpo no modifica las fuerzas de gravedad de los demás. Cada uno actualiza la suya para un momento dado en el tiempo bajo un escenario determinado. Se utilizan estructuras de datos de tipo arreglo unidimensional de tamaño N para almacenar: las fuerzas de atracción (*vf*), las posiciones (*vp*), las masas (*vm*) y las velocidades (*vv*).

El procesamiento de la fuerza de atracción de los cuerpos se realiza dividiendo los vectores *vf*, *vp*, y *vm* en bloques para aprovechar la localidad espacial y temporal de la caché, y de este modo, reducir su tasa de fallo. Para cada bloque de *vf* se realizan los cálculos necesarios utilizando los vectores *vp* y *vm* por bloques. Es decir que para calcular el valor de los elementos de un bloque de *vf* se toma el primer bloque de *vp* y *vm*, se realizan todos los cálculos necesarios entre ellos para actualizar el bloque de *vf*, luego se toma el segundo bloque de *vp* y *vm* para realizar las mismas acciones que con los primeros, y así sucesivamente hasta que se actualiza con los vectores completos *vp* y *vm*. Una vez realizado el cálculo de la fuerza de atracción gravitacional para todos los cuerpos del ese bloque de *vf*, se repite el cómputo con el siguiente bloque de *vf* hasta terminar con el vector completo.

3.2 Solución paralela en memoria compartida

En el caso de la solución en memoria compartida se utilizó Pthread [13] como herramienta para el manejo paralelo de los threads. El hilo principal crea T-1 threads, y entre todos realizan el trabajo.

En un paso de ejecución cada cuerpo calculará su fuerza de atracción independiente de la de los demás. A cada cuerpo *i* le interesa el valor de la posición

de los demás cuerpos, la cual no cambiará hasta que todos hayan calculado su fuerza de atracción.

Partiendo del algoritmo secuencial descripto en 3.1, esta solución paralela realiza las mismas acciones sólo que sobre porciones más pequeñas de datos (cada porción de *N/t* datos, siendo *N* la cantidad de cuerpos y *t* la cantidad de threads). La cantidad de bloques totales en los cuales se dividen los vectores *vf*, y *vm* es igual a la del secuencial. Por ejemplo, si en el algoritmo secuencial se realizó una división de 200 bloques, en el algoritmo paralelo con 8 threads se tienen 25 bloques del vector de fuerza para procesar por cada thread. En tanto, los vectores de posiciones y masa serán divididos en 200 bloques para todos los hilos, ya que se lo necesita completo para el cálculo de la fuerza de atracción de cada cuerpo.

Una vez que las fuerzas de atracción de todos los cuerpos fueron actualizadas, se realiza el cálculo de las velocidades y posiciones. La velocidad de cada cuerpo dependerá de su fuerza de atracción y es independiente de la de los demás cuerpos. La posición se calcula a partir de la velocidad del cuerpo y es independiente de los demás. Realizados dichos cálculos, los threads deben esperar en una barrera a que todos los demás hayan terminado antes de realizar otro paso de simulación, ya que para calcular la fuerza del cuerpo *i* el thread correspondiente necesita conocer las posiciones de los N cuerpos restantes.

3.3 Solución paralela en memoria distribuida

La solución en memoria distribuida es similar a la anterior en lo referido al cálculo de los datos. Solo que en lugar de tener threads ejecutándose, se tienen procesos que utilizan pasaje de mensajes de la librería MPI [14].

El proceso con rank cero es el que inicializa las estructuras de datos, y una vez realizada, reparte los datos entre todos los demás procesos. Luego cada proceso actualiza los vectores de fuerza, posición y velocidad sobre una porción de datos (*N/p* , donde N es la cantidad de cuerpos y *p* es la cantidad de procesos). La división de los datos en bloques se lleva a cabo como se explicó en la Sección 3.2.

Una vez que cada proceso ha terminado de realizar todos los cálculos correspondientes a la fuerza de atracción, actualiza el vector de velocidad y posición; al terminar dicho cálculo transmite los resultados a los demás. Por lo tanto, todos los procesos tendrán los datos resultantes al finalizar la aplicación.

3.4 Solución paralela en GPU-CUDA

En 2007 Nvidia lanzó la implementación de CUDA, plataforma hardware y software que extiende al lenguaje C con un reducido conjunto de abstracciones para la programación paralela de propósito general en GPU. La arquitectura de una GPU-Nvidia CUDA está organizada en Streams Multiprocessors (SMs), los cuales tienen un determinado número de Streams Processors (SPs) que comparten la lógica de control y la caché de instrucciones [9][15][16].

La organización de los threads puede verse por nivel o jerárquicamente de la siguiente manera: grilla o *grid* (formado por bloques de threads), bloque (conformado por threads) y threads. En un bloque, los hilos están organizados en warps (en general de 32 hilos). Todos los hilos de un warp son planificados juntos durante la ejecución.

El sistema de memoria de la GPU está compuesto por distintos tipos de memorias que se diferencian tanto en la forma de acceso de los hilos, las limitaciones en las operaciones permitidas y su ubicación dentro del dispositivo. Está compuesto por: *Memoria Global, Memoria de Textura, Memoria Constante, Memoria Shared o compartida , Memoria Local y Registro*.

CUDA permite definir una función denominada *kernel* que es ejecutada *n* veces, siendo *n* la cantidad de threads de la aplicación. Dicha función solo corre en la GPU o Device, y puede invocar código secuencial para ser ejecutado en la CPU. El kernel debe ser configurado con el número de hilos por bloque y por grid [15][16]. Como el kernel se ejecuta en el device, la memoria debe ser alocada antes de que las función sea invocada y los datos que requiera utilizar copiados desde la memoria del host a la memoria del device. Una vez ejecutada la función kernel, los datos de la memoria del device deben ser copiados a la memoria del host.

El algoritmo Fast N-Body Simulation, es apto para ser ejecutado en la GPU por la independencia de cómputo en el cálculo de la fuerza de atracción gravitacional de los cuerpos. Cada cuerpo debe resolver su fuerza gravitacional dependiendo de las posiciones de los demás sin modificar otro dato que no sea su propia fuerza. Del mismo modo, la actualización de posiciones y velocidades se realiza de manera independiente.

Se almacenan en la memoria global de la GPU las mismas estructuras utilizadas por la CPU (*vf, vv, vp* y *vm*). El paso siguiente es la comunicación de los datos previamente inicializados por la CPU a través del PCI-E. A continuación, la CPU delega la ejecución a la GPU, y queda en espera a la respuesta de esta.

En el kernel o función que calcula la fuerza de atracción gravitacional, cada thread calcula su posición en la memoria global que le permite acceder al cuerpo para el que tiene que calcular dicha fuerza. Se utilizan tres vectores con dimensión igual a la cantidad de threads por bloque, dos de los cuales son para almacenar una porción del vector de posiciones (una para las coordenadas en *x* y otro en *y*), y el tercero para una porción del vector de masas. Eso permite computar por bloque reduciendo el acceso a memoria global.

Además de su identificador en la memoria global, cada thread calcula su identificador dentro del vector en memoria compartida. Cada uno trae de memoria global la posición y la masa del cuerpo que le corresponde, por lo que el vector almacenado en memoria compartida es cargado por todos los threads, y la posición del cuerpo que le corresponde será almacenada en la dirección correspondiente a su identificador dentro de la memoria shared. Antes de comenzar a calcular la fuerza de atracción, los thread son sincronizados para garantizar que todas las posiciones de los vectores de la memoria compartida hayan sido cargadas.

Luego, cada thread realizará el cálculo correspondiente a la fuerza de atracción para su cuerpo con las posiciones que se encuentran en los vectores de posiciones de la memoria compartida. Cuando todos los threads del bloque hayan utilizado todas las posiciones de los vectores de la memoria shared, deben sincronizarse para volver a traer una porción más de las posiciones y de las masas de los cuerpos de la memoria global a la memoria compartida. Este ciclo se repetirá hasta que todo el vector de posiciones y masas haya sido utilizado. Finalmente, cada thread calcula la velocidad y posición del cuerpo que le corresponde. Cuando los pasos de simulación ha realizar se

han completado, la GPU enviará a la CPU los resultados calculados, y le devolverá el control de la ejecución.

4 Resultados experimentales

Las arquitecturas utilizadas para la experimentación fueron:

- un Blade de 16 servidores (hojas). Cada hoja posee 2 procesadores Quad Core Intel Xeón e5405 de 2,0 GHz, con cachés L2 2x6MB compartida de a par de procesadores, y sistema operativo Fedora 12 de 64bits.
- una GPU Geforce TX 560TI, con 384 procesadores con una cantidad máxima de thread de 768 y 1 GB de memoria RAM.

En las pruebas realizadas se varía la cantidad de cuerpos (256000 y 512000) y los pasos de simulación (5,6 y7). Los resultados se obtuvieron a partir de un promedio de ejecuciones.

En las Tabla 1 se muestra el tiempo de ejecución (en seg) del algoritmo secuencial y del algoritmo en MPI para 2, 4 y 8 procesos. En la Tabla 2 se presentan los tiempos de ejecución para los algoritmos secuenciales y Pthread con 2, 4 y 8 threads.

Tabla 1. Tiempos de ejecución (en segundos) para el algoritmo MPI en CPU. Con P = cantidad de procesos.

Tamaño del vector de señales	Cantidad de pasos de simulación	Secuencial	MPI (P = 2)	MPI (P = 4)	MPI (P = 8)
256000	5	19357,36	9711,28	4856,14	2429,29
	6	23227,00	11653,19	5827,61	2914,85
	7	27098,72	13596,05	6798,60	3400,58
512000	5	77426,49	38848,18	19430,01	9716,82
	6	92918,43	46620,05	23316,08	11659,93
	7	108358,99	54384,43	27200,91	13602,60

Tabla 2. Tiempos de ejecución (en segundos) para el algoritmo Pthread en CPU. Con T = cantidad de threads.

Tamaño del vector de señales	Cantidad de pasos de simulación	Secuencial	Pthread (T = 2)	Pthread (T = 4)	Pthread (T = 8)
256000	5	19357,36	9680,89	4839,94	2419,80
	6	23227,00	11617,94	5809,11	2903,56
	7	27098,72	13553,43	6776,39	3387,41
512000	5	77426,49	38727,09	19364,25	9679,42
	6	92918,43	46472,62	23230,21	11613,76
	7	108358,99	54221,36	27101,45	13551,05

Las pruebas experimentales en GPU se realizaron con bloques de 256 threads que es el óptimo para la arquitectura utilizada en la experimentación. Los resultados se presentan en la Tabla 3.

Tabla 3. Tiempos de ejecución (en segundos) para el algoritmo CUDA en GPU. Con T = cantidad de threads.

Tamaño del vector de señales	Cantidad de pasos de simulación	GPU (T = 256)
256000	5	37,90
	6	45,47
	7	53,05
512000	5	151,37
	6	181,64
	7	211,90

Las Fig. 1 y 2 muestran el Speedup (calculado como la relación entre el tiempo de la solución secuencial y el tiempo paralelo) para las soluciones MPI y Pthread con 256000 y 512000 cuerpos.

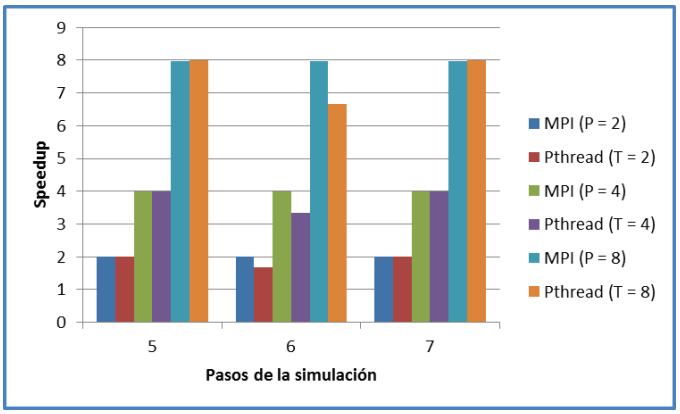


Fig. 1. Speedup de los algoritmos paralelos MPI y Pthread para 256000 cuerpos.

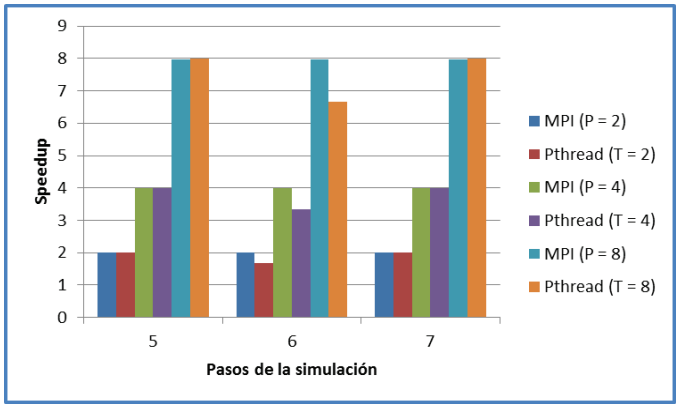


Fig. 2. Speedup de los algoritmos paralelos MPI y Pthread para 512000 cuerpos.

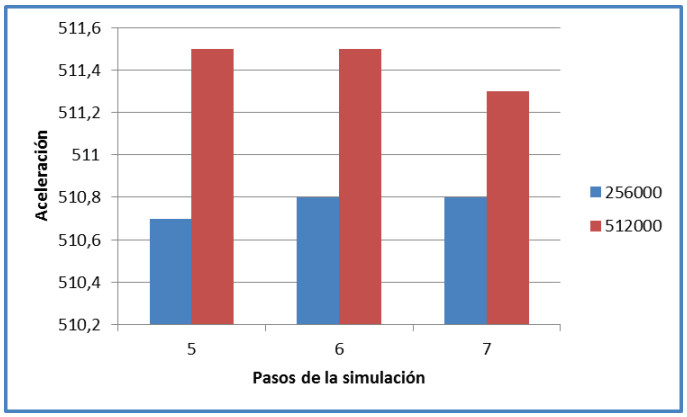


Fig. 3. Aceleración obtenida para el algoritmo N-Body en GPU.

A partir de los resultados obtenidos puede observarse que las soluciones Pthread y MPI presentan características similares en tiempo, no habiendo una diferencia significativa entre ellas en este aspecto. Esto se debe a que la etapa de sincronización entre todas las tareas es más significativa a la de comunicación.

En cambio, para la implementación GPU-CUDA se logran tiempos sensiblemente inferiores con respecto a los mencionados, denotando una alta adecuación de la arquitectura (algoritmo, solución) al problema. Un análisis equivalente puede realizarse a partir de los valores de speedup y aceleración mostrados.

5. Conclusiones y trabajos futuros

La reducción de los tiempos de ejecución en algunas aplicaciones requiere que su solución deba realizarse por medio del procesamiento paralelo para la obtención de tiempos de respuesta aceptables que no pueden ser cumplidos a través del procesamiento secuencial. Una diversidad de arquitecturas multiprocesador están disponibles con este fin, al igual que varias herramientas de programación que permiten explotar el paralelismo en las mismas.

En la última década, ha surgido una arquitectura de propósito específico como una alternativa para el cómputo de altas prestaciones: las GPUs han evolucionado al punto de ofrecer un poder de cómputo que permite alcanzar una gran performance, reduciendo los tiempos de ejecución de las aplicaciones adaptables a dicha arquitectura como se ha mostrado en el presente trabajo para problemas que comparten las mismas características que el problema de los N-Body.

Cabe agregar además, que el costo de una placa GPU es sensiblemente inferior al de la arquitectura blade utilizada para la comparación, reforzando de esta forma una de las ventajas de su empleo en problemas de este tipo.

Como trabajos futuros pueden mencionarse el uso de clusters de GPUs y el estudio del paradigma de programación híbrida utilizando MPI-CUDA para dicha arquitectura, así como avanzar en el análisis de consumo energético en el uso de GPU. Por último, interesa estudiar las nuevas arquitecturas MIC de próxima aparición.

Referencias

1. Moore Gordon E.: Cramming more components onto integrated circuits. Electronics, vol. 38, Número 8 (1965)
2. Kirk David B., Hwu Wen-mei W.: Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
3. AMD, "Evolución de la tecnología de múltiple núcleo". 2009. http://multicore.amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution.
4. Balasubramonian R., Jouppi N., Mularimanohar N.: Multi-Core Caché Hierarchies, Morgan & Claypoll (2011)
5. Thiruvathukal G. K.: Cluster Computing. Copublicado por la IEEE CS y la API. (2005)
6. Chen J. Y.: GPU Technology Trends and Future Requirements. IEEE (2009)
7. Nvidia Corporation: GPU Gems, Pearson Education (2003)
8. Nickolls J., Dally W. J.: The GPU Computing Era. IEEE (2010)
9. Perez C., Piccoli M. F.: Estimación de los parámetros de rendimiento de una GPU. Mecánica Computacional, vol. XXIX, pp. 3155-3167 (2010)
10. Bruzzone Sebastian, "LFN10, LFN10-OMP y el Método de Leapfrog en el Problema de N Cuerpos ", Instituto de Física, Departamento de Astronomia, Universidad de la República y Observatorio Astronómico los Molinos, Uruguay. (2011)
11. Andrews Gregory R., "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison-Wesley. (2000).
12. Lars Nyland, Mark Harris, and Jan Prins. Fast N-body simulation with CUDA. In GPU Gems 3, chap-ter 31, pages 677{695. Addison-Wesley Professional, 2007.
13. https://computing.llnl.gov/tutorials/pthreads.
14. Snir M., Otto S., Huss-Lederman S., Walker D., Dongarra J., "MPI: The Complete Reference". The MIT Press, Cambridge, Massachusetts. 1996.
15. Nvidia Corporation: NVIDIA CUDA C Programming Guide. (2011)
16. Nvidia Corporation: CUDA C Best Practices Guide. (2012)

Referencias

[And00] Andrews Gregory R., “Foundations of Multithreaded, Parallel, and Distributed Programming”, Addison-Wesley, 2000.

[Amd11] amd.com/public, “Meeting the challenges of the future with innovative solutions for public sector IT needs ”, 2011.

[Bar12] Lorena A Barba , “Hierarchical N-body algorithms: A pattern likely to lead at extreme scales ”, Universidad de Boston, 2012.

[Béd07] Jeroen Bédorf , “High Performance Direct Gravitational N -body Simulations on Graphics Processing Units ”, Universiteit van Amsterdam , 2007.

[BGP12] Bódorf J., Gaburov E., Portegies Zwart S., “Bonsai: A GPU Tree-Code ”, Leiden Observatory, Leiden University and Northwestern University , ASP Conference Series , Astronomical Society of the Pacific , 2012.

[BP02] Blodgett John y Parker Larry, “Global Climate Change: U.S. Greenhouse Gas Emissions — Status, Trends, and Projections”, CRS Report for Congress . Received through the CRS Web, 2002.

[BUSRL11] J. Balladini, F. Uribe, R. Suppi, D. Rexachs, E. Luque. “Factores influyentes en el consumo energético de los Sistemas de Cómputo de Altas Prestaciones basado en CPUs y GPUs”. Facultad de Informática, Universidad Nacional del Comahue, Argentina, y Departamento de Arquitectura de Computadores y Sistemas Operativos, Universidad Autónoma de Barcelona, España. XVII Congreso Argentino de Ciencias de la Computación, 2011.

[CCCT10] Chao-Tung Yang , Chih-Lin Huang , Cheng-Fang Lin , Tzu-Chieh Chang , “Hybrid Parallel Programming on GPU Clusters ”, Department of Computer Science,

Tunghai University , Taiwan, International Symposium on Parallel and Distributed Processing with Applications , 2010.

[CGSP04] Francisco Chinchilla, Todd Gamblin, Morten Sommervoll, Jan F. Prins , “Parallel N-Body Simulation using GPUs ”, Department of Computer Science , University of North Carolina at Chapel Hill , http://gamma.cs.unc.edu/GPGP , Technical Report TR04-032 , 2004.

[Che09] Chen John Y., “GPU Technology Trends and Future Requirements ”, publicado en IEEE, 2009.

[Coo11] Christopher Cooper , “GPU Computing with CUDA . Lecture 10 - Applications - N body problem ”, Universidad de Boston, 2011.

[CS07] Colque Pinelo MaTeresa y Sánchez Campos Víctor E., “Los Gases de Efecto Invernadero: ¿Por qué se produce el Calentamiento Global? ”, Asociación Civil Labor / Amigos de la Tierra – Perú . 2007.

[Cur08] Curtis Lewis, “Environmentally Sustainable Infrastructure Design”, The Architecture Journal , págs. 2-8, 2008.

[Das89] Dasgupta Subrata, “Computer Architecture. A Modern Systhesis”, volumen 2: tópicos avanzados, John Wiley & Sons, Inc., 1989.

[DGT98] De Giusti Armando y Tinetti Fernando G., “Procesamiento Paralelo. Conceptos de Arquitecturas y Algoritmos”, Primera Edición, 1998.

[EI07] Energy Information Administration , “Emissions of Greenhouse Gases in the United States 2006 ”, año 2007.

[FG08] Francis Kevin y Richardson Peter , Green Maturity Model for Virtualization , The Architecture Journal , págs. 9-15. 2008.

[FX10] Wu-chun Feng, Shucai Xiao, “To GPU synchroniza or not GPU synchroniza?”, publicado en IEEE, 2010.

[GGKK03] Grama Ananth, Gupta Anshul, Karypis George y Kumar Vipin, “Introduction to Parallel Computing”, Segunda Edición, Adison-Wesley, 2003.

[Guh08] Guhl Corpas Andrés, “Aspectos éticos del calentamiento climático global”. Revista Latinoamericana de Bioética, ISSN 1657-4702, Volumen 8, Número 2, Edición 15, Páginas 20-29, 2008.

[HB84] Hwang Kai y Briggs Fayé A., “Computer Architecture and Parallel Proccessing”, MacGraw-Hill Book Company, 1984.

[Hwu11] Hwu Wen-mei W., “GPU Computing Gems Emerald Edition”, publicado por Morgan Kaufmann e impreso Elsevier, 2011.

[HYNN++09] Tsuyoshi Hamada , Rio Yokota, Keigo Nitadori , Tetsu Narumi , Kenji Yasuoka, Makoto Taiji , “42 TFlops Hierarchical N-body Simulations on GPUs with Applications in both Astrophysics and Turbulence ” , www.stats.bristol.ac.uk, 2009.

[Int05] Intel, “Excerpts from A Conversation with Gordon Moore: Moore’s Law”, transcripción de video, 2005.

[KH10] Kirk David B. y Hwu Wen-mei W., “Programming Massively Parallel Processors: A Hands-on Approach ”, Morgan Kaufmann , 2010.

[Lor10] Lor Carlos Gracia, “El futuro de la energía solar fotovoltaica”. Grup de Dispositius Fotovoltaics y Optoelectrònics. Departamento de Física, Universitat Jaume I . Mayo de 2010 .

[Lue08] David Luebke , “CUDA: SCALABLE PARALLEL PROGRAMMING FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING ”, NVIDIA Corporation , 2008.

[MFB05] Mejía N. David, Fernández A. Diego y Bernal C. Iván, “Desarrollo de aplicaciones paralelas para clusters utilizando MPI (Massage Passing Interface)”, XIX Jornadas en Ingeniería Eléctrica y Electrónica, vol. 19, Ecuador, 2005.

[Moo65] Moore Gordon E., “Cramming more components onto integrated circuits”, Revista Electronics, Volumen 38, Número 8, 1965 .

[NA11] NASA. “Global Climate Change”. http://climate.nasa.gov/causes/

[ND10] Nickolls John y Dally William J., “The GPU Computing Era” , publicado por IEEE, 2010.

[NG08] National Geographic Channel, “Seis grados que podrían cambiar el mundo. Las

consecuencias del calentamiento global", http://www.natgeo.tv/la/especiales/seis-grados/

[Nic10] Nicolaisen Nancy, "Green Computing with Intel® AtomTM Processor-Based Devices",http://software.intel.com/en-us/articles/green-computing-with-intel-atom-processor-based-devices/, 2010.

[NU98] Naciones Unidas , "PROTOCOLO DE KYOTO DE LA CONVENCIÓN MARCO DE LAS NACIONES UNIDAS SOBRE EL CAMBIO CLIMÁTICO ". 1998.

[Nak09] Naohito Nakasato , "Oct-tree Method on GPU: $42/Gflops Cosmological Simulation ", Department of Computer Science and Engineering University of Aizu , 2009.

[Nvi03] Nvidia Corporation, "GPU gems", Pearson Education, 2003.

[Nvi08] Nvidia Corporation, "GPU gems 3", Pearson Education, 2008.

[Nvi11] Nvidia, "NVIDIA CUDA C Programming Guide ", 2011.

[Nvi12] Nvidia, "CUDA C BEST PRACTICES GUIDE ", 2012.

[OHLG++008] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, y James C. Phillips , "GPU Computing", publicado en IEEE Vol. 96, No. 5, Mayo 2008.

[Par10] Song Jun Park, "An Analysis of GPU Parallel Computing", 2009 DoD Hogh Performance Computing Modernization Program Users Group Conference, publicado en IEEE, 2010.

[PF70] Philco y Ford, "Fundamentos de Electricidad y Electrónica", volumen I y II, 1970.

[Pic11] Piccoli María Fabiana, "Computación de Alto Desempeño en GPU", XV Escuela Internacional de Informática del XVII Congreso Argentino de Ciencia de la Computación, Editorial de la Universidad Nacional de La Plata, 2011.

[PP10] Perez Cristian y Piccoli M. Fabiana , "Estimación de los parámetros de rendimiento de una GPU", Mecánica Computacional Vol XXIX, págs. 3155-3167 , 2010.

[Qui94] Quinn Michael J., "Parallel Computing. Theory and Practice", Segunda Edición, McGraw-Hill, Inc., 1994.

[RAE12] Real Academia Española, Diccionario de la Real Academia Española, 2012.

[RPS+12] Fernando Romero, Adrian Pousa, Victoria Sanz, Armando De Gisuti, “Consumo energético en Artquitecturas Multicore. Análisis sobre un algoritmo de Criptografía Simétrica”, CACIC 2012 – Congreso Argentino de las Ciencias de la Computación, Bahía Blanca, Buenos Aires, Argentina, 2012.

[Sap53] Sapiens, Enciclopedia ilustrada de la lengua castellana, Editoral Sopena Argentina, 1953.

[SE08] Schneider Electric, “Go Green, Save Green . The Benefits of Eco-Friendly Computing ”, 2008.

[SDB10] Silva Juliana M. N., Drummond Lúcia, y Boeres Cristina, “On Modelling Multicore Clusters”, 22nd International Symposium on Computer Architecture and High Performance Computing Workshops , 2010.

[Tan09] Tanuro Daniel, “Capitalismo, decrecimiento y ecosocialismo”, VIENTO SUR, Número 100, págs. 231-238, 2009.

[Thi05] Thiruvathukal George K., “ Cluster Computing ” , Copublicado por la IEEE CS y la AIP, 2005.

[Tri07] Trillo José Antonio Pascual, “Más allá del cambio climático: desarrollo sostenible”. Catedrático de Biología y Geología (IES El Escorial). Ciclo de Conferencias “Los jueves de la Ciencia”, España, 2007.

[TW09] Tinetti Fernando G. y Wolfmann Gustavo, “Parallelization Analysis on Clusters of Multicore Nodes Using Shared and Distributed Memory Parallel Computing Models”, World Congress on Computer Science and Information Engineering, 2009.

[US11] U.S. Environmental Protection Agency , “INVENTORY OF U.S. GREENHOUSE GAS EMISSIONS AND SINKS: 1990 –2009 ”, año 2011.

[Ver07] S.S. Verma, “GREEN COMPUTING ”. Departamento de física, SLIET, 2007.

[VR08] Carlos A. Vera , Boris A. Rodríguez , “Evolución Numérica de Sistemas de N Cuerpos Usando Métodos Jerárquicos de Árbol de Integración Simpléctica ”, Revista Colombiana de Física, 2008.

[WXXY+10] Wang Z., Xu X., Xiong N., Yang L. T., Zaho W., “Analysis of Parallel Algorithms for Energy Conservation with GPU”, 2010 IEE/ACM International Conference on Green Computing and Comunicactions & 2010 IEEE/ACM International Conference on Cyber, Physical and Social Computing, 2010.

[YB10] Rio Yokota , Lorena A. Barba , “Treecode and fast multipole method for N-body simulation with CUDA ”, Boston University , http:// www.bu.edu, 2010.

[YLAKK++09] Yun Zhifeng, Lei Zhou , Allen Gabrielle, Katz Daniel S., Jha Shantenu y Ramanujam Jagannathan, “An Innovative Application Execution Toolkit for Multicluster Grids ”, publicado por IEEE, 2009.