INSTITUTO TECNOLÓGICO DE APIZACO

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

"PARALELIZACIÓN DE ALGORITMO GENÉTICO MODULAR MEDIANTE ARQUITECTURAS DE HARDWARE MULTINÚCLEO"

TESIS

QUE PARA OBTENER EL GRADO DE

MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:

LIC. SERGIO PALAFOX UGARTE

ASESORES:

Director: DR. JOSÉ FEDERICO RAMIREZ CRUZ

Co-Director: Dr. ROBERTO MORALES CAPORAL

APIZACO, TLAXCALA. FEBRERO 2015

RESUMEN

En este trabajo de tesis se analizan diversas técnicas y tecnologías de paralelización de código, posteriormente se propone un algoritmo genético (AG) para resolver problemas de optimización aplicando técnicas de cómputo evolutivo y se presenta una implementación secuencial y una paralelización en Cilk, la aplicación de esta tecnología de paralelización permite explotar el paralelismo inherente en el algoritmo genético y reducir el esfuerzo computacional asociado a operaciones tales como la creación de la población inicial, la evaluación de los individuos, la mutación y la selección. El algoritmo genético está diseñado para ser aplicado a problemas de optimización con maximización de 1 a N variables. Por otra parte, se analiza el efecto de la variación en el tamaño de población en términos de precisión, eficacia, tiempos de cómputo y factores de aceleración. Se reportan mejoras significativas en términos de tiempos de ejecución con respecto a tres implementaciones programadas en lenguaje C, la primera de estas implementaciones se refiere a la versión secuencial del algoritmo, mientras que la segunda se refiere a una versión en la cual se paralelizan todas las operaciones evolutivas y la implementación principal en la cual se paralelizan solamente las operaciones de evaluación selección y mutación. Finalmente se compara el rendimiento entre estas implementaciones en dos tipos de arquitecturas multinúcleo.

ABSTRACT

In this thesis various techniques and technologies parallelization of code are analyzed, then a genetic algorithm is proposed to solve optimization problems using evolutionary computation techniques and sequential implementation and parallelization in Cilk occurs, the application of this technology allows parallelization exploit the inherent parallelism in genetic algorithm and reduce the computational effort associated with operations such as creating the initial population, evaluation of individuals, mutation and selection. The genetic algorithm is designed to be applied to optimization problems with N maximizing variables. Moreover, the effect of variation in the population size in terms of accuracy analyzes, effectiveness, computing time and acceleration factors. Improvements significant are reported in terms of execution times on three implementations programmed in C language, the first of these implementations refers to the sequential version of the algorithm, while the second refers to a version in which parallelize all main evolutionary operations and the implementation in which operations are parallelized evaluation only selection and mutation. Finally compares the performance between these two types of implementations multicore architectures.

TABLA DE CONTENIDO

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO

API Del Inglés Application Programming Interface es el conjunto de funciones y procedimientos que se ofrecen en forma de biblioteca para ser utilizado por otro software.

Apple Empresa multinacional que diseña y produce equipos electrónicos y software.

ATI Empresa especializada en la fabricación y diseño de hardware para procesamiento gráfico.

C/C++ Lenguaje de programación de propósito general híbrido, ya que puede manejar objetos o programación funcional, que presenta un modelo de codificación flexible y de alto rendimiento.

Cilk Lenguaje de programación de propósito general específicamente diseñado para la programación paralela multihilo.

Cluster Conjunto de ordenadores construidos con hardware común y que están ideados para comportarse como una única máquina.

CPU Del Inglés Central Processing Unit es el componente principal de los ordenadores y demás dispositivos programables. Su función es interpretar instrucciones y procesar los datos que conforman los programas.

CUDA Siglas del Inglés Compute Unified Device Architecture son un conjunto de herramientas y un compilador creados por la compañía Nvidia que permiten programar las GPU de la misma empresa con una versión ampliada del lenguaje C.

Dato Representación simbólica, un atributo o una característica de una entidad. Es la unidad mínima de información que puede manipular un ordenador.

GPC Del Inglés Graphics Computing Cluster es la nomenclatura en las tecnologías de CUDA para referirse a las unidades de procesamiento gráfico.

GPU Acrónimo Inglés de Graphics Processing Unit hace referencia a un coprocesador especializado en procesamiento de gráficos y operaciones en punto flotante, de forma que elimina esa carga de trabajo de la CPU.

Intel Multinacional especializada en la fabricación de CPUs y creación de lenguajes o APIs de alto rendimiento.

Khronos Consorcio industrial financiado por sus miembros enfocado a la creación de APIs estándares abiertas y libres de cargo que permiten la creación y reproducción multimedia en un amplio abanico de plataformas y dispositivos.

Manycore Procesador multicore con una agregación de núcleos muy superior a la usual en los procesadores multinúcleo.

Memoria Dispositivos que retienen datos informáticos durante algún intervalo de tiempo.

MPI Estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

Multinúcleo Procesador que combina dos o más núcleos de ejecución independientes en un solo circuito integrado.

Nvidia Empresa especializada en la fabricación y diseño de GPUs y clusters de procesamiento de alto rendimiento con a su tecnología CUDA.

OpenCV Biblioteca libre de visión artificial originalmente desarrollada por la compañía Intel.

OpenGL Especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

OpenMP/OMP API diseñada para la programación multiproceso de memoria compartida en múltiples plataformas.

Open-Source En español código abierto, es una filosofía de desarrollo de software que se fundamenta en la libre creación y distribución de éste. No confundir con el termino software libre, ya que aunque similares, no enfocan el mismo concepto.

PCIe Es un nuevo desarrollo del bus PCI que usa los conceptos de programación y los estándares de comunicación existentes, pero se basa en un sistema de comunicación serie mucho más rápido.

Proceso Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados

SIMD Siglas del inglés Single Instruction, Multiple Data es una técnica empleada para conseguir paralelismo a nivel de datos.

SMX Unidad de procesamiento utilizado por NVidia en sus GPU.

TBB Tecnología creada por Intel que corresponde con el acronimo Threading Building Blocks es una biblioteca basada en plantillas para C++ para facilitar la escritura de programas que exploten las capacidades de paralelismo de los procesadores con arquitectura multinúcleo.

Thread En español hilo, es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo.

Capítulo 1: Introducción

1.1 Introducción

La tarea de optimizar es inherente a los seres humanos, nos resulta altamente motivante la idea de poder hacer más con menos. Y esto nos ha llevado a buscar formas más creativas y controladas de realizar nuestras actividades.

En general, para problemas sencillos es suficiente con la observación y la experiencia para llegar a las soluciones óptimas; pero cuando el problema se hace complejo, interviene una mayor cantidad de variables o se incrementa el espacio en el que se va a buscar la solución, se hace necesario un análisis más detallado y aumenta la cantidad de operaciones que requieren mucho tiempo de procesamiento, por lo que se vuelve necesario refinar el método para resolver el problema o hacer uso de herramientas que faciliten la tarea.

Debido a que existe una estrecha relación entre los problemas de optimización y búsqueda, en ocasiones se usa indistintamente uno u otro término (Rosete, 2000).

Se le llama métodos de optimización y búsqueda de propósito general a aquellos que solamente necesitan para su aplicación de la definición del espacio de soluciones, los operadores y la función objetivo, sin exigir el cumplimiento por ésta última de ninguna característica especial, como podría ser la continuidad, la existencia de derivadas, etc. También se les llama métodos débiles, técnicas heurísticas modernas o meta-heurísticas. La importancia vital de estos métodos está en que son la única posibilidad para resolver problemas con las condiciones siguientes:

- No existe un algoritmo eficiente para la solución particular del mismo.
- No es posible la exploración exhaustiva del espacio completo de búsqueda, debido a las dimensiones del mismo.

Ninguno de ellos garantiza la obtención del óptimo global en el marco de condiciones prácticas. Sin embargo, ellos permiten obtener buenas soluciones que en muchos casos satisfacen a los usuarios.

Existen varios tipos de algoritmos de optimización de propósito general, entre otros (Rosete, 2000):

- Búsqueda aleatoria
- Escaladores de colinas
    - Clásico
    - Estocástico con mejor acceso
    - Estocástico con primer acceso
- Métodos derivados con un solo punto
    - Escalador de colinas con reiniciación
    - Recocido o enfriamiento simulado
    - Búsqueda tabú
- Métodos con poblaciones de puntos
    - Basados en mutaciones
    - Basados en cruzamiento: Algoritmos genéticos
    - Algoritmos de estimación de distribuciones
    - Enjambres de partículas

El presente trabajo se centra en los métodos heurísticos basados en poblaciones cuyos principales exponentes son los Algoritmos Genéticos.

Los algoritmos genéticos, propuestos por Holland (Vose, 1999) en su versión original, basan su comportamiento en la evolución de las especies, en donde sobreviven los más adaptados al entorno.

Por otra parte, las nuevas herramientas computacionales nos han permitido disminuir el tiempo necesario para encontrar la solución a los problemas de optimización y han hecho posible que abordemos nuevos problemas, tan complejos, que sin ellas simplemente no podríamos resolver.

En este sentido se han explorado distintas maneras de explotar las nuevas capacidades de las herramientas computacionales, por ejemplo: incrementando la capacidad de los procesadores, agregando procesadores a una misma computadora, haciendo trabajar cooperativamente a

varias computadoras o construyendo súper computadoras diseñadas con propósitos específicos.

Inicialmente la programación paralela implicaba que dos o más procesadores trabajaran en colaboración para la solución de un problema y se tenía que desarrollar en equipos específicamente diseñados para este propósito, sumamente costosos y poco accesibles para la mayoría de los desarrolladores.

Los grupos de computadoras conectadas en red trabajando de manera colaborativa (cluster), son una alternativa para el diseño de aplicaciones en paralelo y son menos costosos que una súper computadora.

Conforme los procesadores fueron evolucionando y se fueron reduciendo sus costos, las computadoras de uso común se han ido equipando con procesadores más poderosos que incluyen más de un núcleo.

Por otro lado, en el área de los videojuegos, para responder a la necesidad de mostrar imágenes con apariencia más realista, las tarjetas gráficas también han evolucionado y han sido dotadas de una mayor capacidad computacional, agregándoles su propia memoria, procesadores y unidades aritmético lógicas. No es de extrañar que hubiese quien se planteara, ¿por qué no aprovechar esta capacidad de cómputo para algo distinto que el juego?, así es que empezaron a desarrollarse aplicaciones que, mediante el uso de instrucciones para el manejo de pixeles, resolvían problemas matemáticos complejos; pero había que tomarse el trabajo de traducir el resultado expresado en pixeles, líneas o colores, al contexto del problema que se estuviese resolviendo; en cierto modo, era como "engañar" a la Unidad de Procesamiento Gráfico (del inglés Graphics Processing Units, abreviado GPU).

Al prever esta necesidad de usar para otros fines la capacidad de cómputo de las GPU, la compañía NVIDIA se dio a la tarea de hacer esto posible diseñando tarjetas capaces de procesar datos de propósitos diferentes al procesamiento de gráficos, así como el lenguaje de programación CUDA (del inglés Compute Unified Device Architecture), el cual permite unificar la arquitectura del GPU tanto para gráficos como para cálculos matemáticos. Con esto

se tiene otra alternativa para desarrollar aplicaciones en paralelo que por supuesto resulta ser más accesible que la adquisición de una súper computadora y que el montaje de un clúster.

Actualmente existen varias alternativas de arquitecturas para implementar aplicaciones en paralelo permitiendo optimizar problemas complejos, de las cuales durante el desarrollo de este trabajo se tuvo la oportunidad de acceder a un clúster de computadoras de 16 nodos, durante la estancia de investigación realizada en la Facultad de Ciencias de la Computación (FCC) de la Benemérita Universidad Autónoma de Puebla (BUAP).

Con el presente trabajo, se ha explorado la alternativa de paralelización de código con diversas tecnologías y arquitecturas multinúcleo para la solución de problemas de optimización de propósito general que han sido ampliamente estudiados en la última década (Chow y Rad 2002) y tienen aplicación en diversas áreas como la oceanografía (Carmona y Castro 2009), los bioprocesos (Herrera y Martínez 2003), el procesamiento digital de imágenes (Mussi et al., 2009) y las condiciones climáticas (Hernández y Herrera 2005).

1.2 Descripción del problema

La precisión de las soluciones proporcionadas por los algoritmos genéticos ha sido una desventaja que a su vez comparte un serio problema con los métodos de optimización convencionales, el cual está asociado a que la capacidad de encontrar el óptimo de una función objetivo disminuye conforme aumenta la dimensión del problema. Para el caso particular de los algoritmos genéticos, encontrar la solución a problemas de grandes dimensiones implica aumentar significativamente el número de individuos dentro de la población en proporción al número de dimensiones y tamaño del espacio de búsqueda dentro de cada una de estas dimensiones. Es necesario considerar además, que los tiempos de ejecución crecen en proporción al tamaño de población, lo cual afecta directamente el desempeño del algoritmo. El problema asociado a los tiempos de ejecución requeridos por parte de los algoritmos genéticos es otro importante inconveniente y ha sido la razón de muchas críticas. La cantidad de evaluaciones en un algoritmo genético está definida por el tamaño de la población y el número de iteraciones totales, por lo que al incrementar ambos o alguno de estos dos parámetros aumenta significativamente los tiempos de ejecución, los cuales están sujetos a determinados

límites teóricos (Amdahl 1967) y, por supuesto, de la máquina en donde se realicen los estudios. Sin embargo, algunos algoritmos disponen de una ventaja sobre otros, y es que tienen la propiedad de ser paralelizables como lo es el caso de los algoritmos genéticos.

1.3 Justificación

Debido a la naturaleza paralela que poseen los algoritmos genéticos es posible realizar implementaciones paralelas en arquitecturas multinúcleo, lo cual beneficia directamente el desempeño de estos procedimientos, disminuyendo su tiempo de ejecución en función del número de procesadores existentes en el sistema de cómputo que los ejecuta. La aceleración en el proceso de ejecución ha sido la razón más citada en la literatura para el uso del cómputo paralelo en el diseño algoritmos genéticos. Dado que las evaluaciones y operaciones evolutivas a las que son sometidas las poblaciones pueden ser procesadas de manera simultánea para cada individuo, es común distribuir estas tareas entre múltiples procesadores o núcleos de procesamiento. El paralelismo en algoritmos genéticos se ha vuelto importante, dado el incremento en el uso de sistemas de cómputo paralelo y las mejoras permanentes de las tecnologías computacionales asociadas al desarrollo de multi-procesadores cada vez más rápidos y eficientes.

1.4 Objetivo

El objetivo principal de esta tesis es realizar el diseño de un algoritmo genético sobre arquitecturas de hardware multinúcleo, utilizando el lenguaje Cilk. Este algoritmo deberá ser capaz de resolver problemas de optimización para maximización de N variables.

1.5 Pregunta de investigación

¿Cómo realizar de forma correcta la combinación del algoritmo genético y el cómputo paralelo para resolver problemas de optimización?

1.6 Estado del arte

Hao Li (2011) realiza una implementación en Cilk de un programa escrito originalmente para una estación base transceptora (BTS) la cual es una pieza de equipamiento que facilita la comunicación inalámbrica entre equipos de usuarios y una red. Migrando la implementación a un procesador multinúcleo TILEPro64 desarrollado por la empresa Tilera, el cual consiste en una red en malla de 64 "tiles", donde cada tile aloja un procesador de propósito general y memoria cache. De igual forma migra la implementación realizada en Cilk a una plataforma x86. Finalmente evalúa y compara el desempeño de la implementación en ambas plataformas.

Olson y Peloquin (2010) describen el potencial de las GPU para lograr alta aceleración sobre los sistemas multinúcleo tradicionales lo que las ha convertido en objetivo popular para los programadores que realizan implementaciones paralelas. Sin embargo, las GPU resultan difíciles de programar y depurar. Mientras tanto, hay varios modelos de programación de sistemas multinúcleo que están diseñados para ser fáciles de diseñar y programar, como Cilk. Los autores proponen una plataforma llamada OpenCLunk en la que intentan combinar la facilidad de programación de Cilk con el buen desempeño de la GPU. Implementando un framework de tareas paralelas como Cilk para las GPU, evaluaron su desempeño y los resultados indicaron que mientras que es posible implementar un sistema de este tipo, no es posible alcanzar un buen rendimiento.

García y Pérez (2012) muestran cómo generar paralelismo en el lenguaje C utilizando Cilk, donde describen a este último como un lenguaje algorítmico basado en múltiples hilos (threads). La idea de Cilk es que un programador debe concentrarse en estructurar su programa en forma paralela sin tenerse que preocupar por cómo será la corrida en el sistema para mejorar su eficiencia en la plataforma. Los autores mencionan que en la corrida de un programa Cilk se encarga de detalles como el balanceo de carga y comunicación entre los procesadores. Concluyen que Cilk básicamente asegura que se asignen las cargas de trabajo de forma eficiente para explotar el mayor nivel de paralelismo de la implementación en cuestión.

Alba y Tomassini (2002) proponen una visión moderna (para su época) de las técnicas utilizadas para la paralización de algoritmos evolutivos (del inglés, evolutionary algorithms,

abreviado EAs), cuyo trabajo está fundamentado en dos aspectos básicos: el primero se refiere a las diferentes familias de algoritmos evolutivos que han surgido en la última década mientras que los algoritmos evolutivos paralelos carecieron de estudios importantes, el segundo se refiere al gran número de mejoras en los algoritmos y en su paralelización, lo cual plantea la necesidad de analizar esta característica en profundidad. Los autores remarcan las diferencias entre el modelo de algoritmo evolutivo ejecutado de forma paralela a lo largo de su investigación, donde mencionan las ventajas e inconvenientes de los AEs; así como sus aplicaciones exitosas, y de igual manera plantean propuestas de posibles soluciones a los problemas identificados en los EAs. Finalmente proporcionan los fundamentos para describir los AEs paralelos con el objetivo de mostrar a los investigadores los beneficios que se pueden obtener al ejecutar de forma paralela y descentralizada este tipo de algoritmos.

Cristea y Godza (2000) describen cómo el campo de agentes inteligentes y la fuerte relación con el área de computación evolutiva son excelentes zonas de cultivo en las ciencias de la computación e ingeniería. En particular los algoritmos genéticos (del inglés, Genetic Algorithms, abreviado GAs) proporcionan una mejor solución a muchos problemas complejos de ingeniería para los cuales los métodos de optimización clásica no se pueden utilizar. Los autores mencionan la capacidad de los GAs para aprender de sí mismos y llegar a ser expertos en diferentes áreas, así como la característica que proveen los GAs de ser intrínsecamente paralelizables de manera natural y en consecuencia reducir el tiempo para obtener resultados. Además la paralelización produce soluciones de mejor calidad como el enfoque utilizado por los algoritmos genéticos paralelos, el cual se aproxima mejor al modelo de evolución natural, que el enfoque secuencial. La investigación describe el uso de paralelismo en el desarrollo de algoritmos genéticos y discuten dos aspectos de su diseño: la eficiencia y calidad de los resultados. Presentan varios enfoques comparativamente analizados, donde los principales criterios de comparación que consideran son: rendimiento, flexibilidad, facilidad de diseño y programación. El análisis realizado destaca las ventajas de los algoritmos genéticos diseñados con modelo de grano fino, los cuales son mejor adaptados a las arquitecturas actuales de computo de alto rendimiento.

Munawar y Munetomo (2008) realizan un estudio acerca del impacto de los paradigmas de cómputo paralelo/distribuido en los algoritmos genéticos ayudando a la comunidad científica interesada en estos algoritmos a reconfortarse con estos paradigmas de evolución paralela así como remarcando algunas áreas de investigación. Esto resulta inspirador para la comunidad científica relacionada con cómputo de alto rendimiento (del inglés High-Performance Computing, abreviado HPC) puesto que los paradigmas mencionados se encuentran inmersos en dos áreas principales que han evolucionado rápidamente en los últimos años: sistema multinúcleo (multi-core computing, GPUs) y sistemas en red (Grid computing, clusters). Finalmente se hace mención de los retos que implica el uso de los paradigmas modernos de cómputo paralelo; así como una estructura jerárquica de algoritmos genéticos paralelos conveniente para sistemas de cómputo paralelo en red y sistemas multinúcleo.

Loung et al. (2010) proponen un modelo de isla para los algoritmos evolutivos permitiendo retrasar la convergencia global del proceso de evolución y fomentar la diversidad. Sin embargo resolver problemas de optimización combinatoria de gran escala demandantes en tiempo con el modelo de isla requiere grandes cantidades de recursos computacionales, lo cual se puede resolver utilizando una GPU revelada recientemente como una poderosa herramienta de computo, la cual proporciona recursos de alto rendimiento, mediante una arquitectura multinúcleo capaz de realizar tareas de manera simultánea, permitiendo implementar algoritmos de forma paralela como en este caso el modelo de isla. Los autores realizan un re-diseño en el modelo de isla paralela sobre la GPU. Finalmente los resultados preliminares demuestran la eficacia del modelo propuesto y sus capacidades para aprovechar plenamente la arquitectura de la GPU.

1.7 Descripción del documento

El documento de tesis se encuentra organizado de la siguiente manera:

En el segundo capítulo se presentan las principales tecnologías y lenguajes para implementar códigos de forma paralela, de los cuales se eligió el lenguaje Cilk, el cual se describe a detalle.

En el tercer capítulo de esta tesis se muestran las características de las principales arquitecturas, así como también el conjunto de métricas para medir el desempeño de los programas ejecutados en paralelo.

En el cuarto capítulo se describen los aspectos fundamentales de los algoritmos genéticos.

En el quinto capítulo describen los detalles de la implementación del algoritmo genético paralelo.

En el sexto capítulo se revisaran los resultados obtenidos.

Por ultimo en el séptimo capítulo se enuncian las conclusiones principales de esta tesis y los posibles trabajos futuros, para continuar con el trabajo aquí presentado.

Capítulo 2: Tecnologías de Paralelización

Existen múltiples tecnologías que facilitan la escritura y el desarrollo de programas capaces de realizar ejecución de procesos paralelos. A continuación se presentan una serie modelos y lenguajes de programación más extendidos y usados para realizar implementaciones de algoritmos en forma paralela (Mattson et al., 2004).

2.1 Hilos de C++ 11

El estándar C++ 11 introduce por primera vez soporte nativo de programación multi-hilo. Con esta novedad es posible crear programas de C++ multi-hilo sin necesidad de recurrir a extensiones externas dependientes de la plataforma. Esta funcionalidad se ofrece en la nueva biblioteca estándar de hilos de C++.

Hasta la fecha, la programación multi-hilo de C++ se basaba en el uso de bibliotecas creadas por terceros que agrupan la funcionalidad necesaria y proporcionan una buena abstracción de alto nivel, que permite a los programadores la creación de código paralelizado. Uno de los problemas fundamentales es que el estándar previo de C++ no reconoce la existencia de hilos de ejecución, por lo que el modelo de memoria no estaba formalmente definido para este tipo de aplicaciones. En la mayoría de los casos las bibliotecas como Boos y ACE han servido como base sólida para la programación paralela en C++, pero al no poseer un modelo de memoria que tenga en cuenta la existencia de hilos surgen problemas, especialmente en aquellas aplicaciones que tratan de lograr alto rendimiento haciendo uso de conocimientos específicos del hardware, o para los desarrolladores creando programas multi-plataforma, donde el comportamiento del compilador varía según la plataforma.

Esta situación ha motivado la creación, por parte del comité de estandarización de C++, de una biblioteca estándar para hilos en este lenguaje, donde se define un modelo de memoria adecuado para la práctica de esta programación y se resuelven otros problemas subyacentes. Esta nueva biblioteca ofrece cuatro funcionalidades básicas: clases para el manejo de hilos, mecanismos de protección para datos compartidos, operaciones de sincronización entre hilos, y operaciones atómicas de bajo nivel.

2.2 OpenMP

OpenMP (Open Multi-Processing) es un API ideado para la programación de tareas multiproceso con memoria compartida, es decir, una encapsulación de hilos (Chapman et al., 2007). OpenMP se encuentra disponible para los lenguajes C, C++ y Fortran en la mayoría de las arquitecturas y sistemas operativos. EL API consiste en una agrupación de directivas para el compilador, rutinas de biblioteca y variables de entorno que permiten modificar el comportamiento de la ejecución de un programa. En el caso de sistemas grandes, como clusters, OpenMP puede ejecutarse simultáneamente con MPI (Message Passing Interface), donde esté adopta un modelo consistente en un clúster de computadoras donde los nodos no comparten memoria, y en el cual tanto el intercambio, como la interacción de los datos entre los nodos deben realizarse explícitamente a través del uso de paso de mensajes, de forma que el código pueda paralelizarse de manera sencilla en cada máquina del clúster con MPI, y dentro de cada procesador con las directivas de OpenMP, donde el funcionamiento básico de esté (Barney 1998) se basa en la ejecución de un hilo maestro, el cual se divide en diversos hilos ver figura 2.1. Esta división debe efectuarse en las secciones de código que lo soporten, ya que de otra forma en lugar de mejorar el rendimiento, este empeoraría. Una vez realizada la ejecución en paralelo, los hilos se juntan de nuevo para continuar con la ejecución del hilo principal.

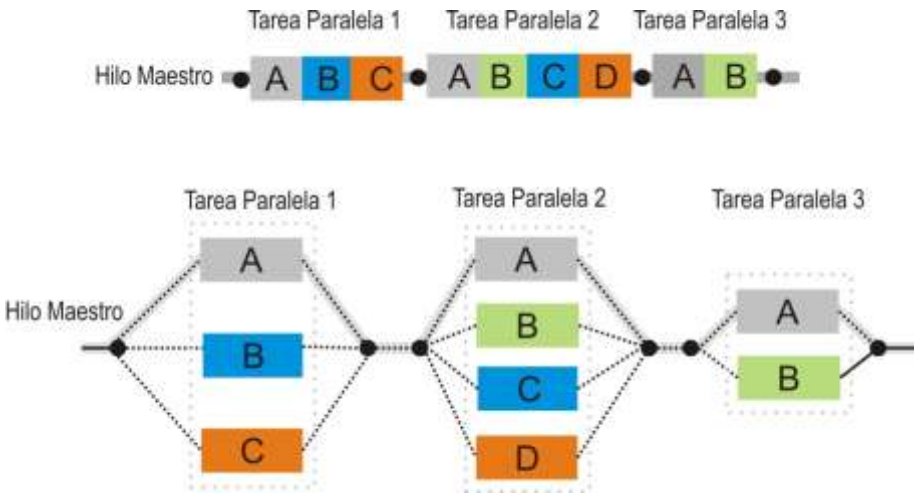


Figura 2.1 Paralelización en OpenMP

Cada hilo tiene su propio identificador, que puede usarse para mejorar la paralelización o comunicarse con un determinado hilo. Este modelo se basa en la memoria compartida, donde todos los procesadores o hilos tienen acceso a la misma sección de memoria. Esto hace que OpenMP proporcione cláusulas y directivas que permiten aislar ciertas variables o secciones de código dentro de una paralelización para evitar problemas de concurrencia o que puedan tener lugar condiciones de carrera, esto es, que el resultado no dependa del orden en que se ejecuten las tareas. Existen cláusulas de control sobre los datos y accesos a memoria, cláusulas de sincronización y cláusulas de planificación. Además, existen cláusulas extras que añaden funcionalidad a OpenMP, como cláusulas de inicialización de variables, cláusulas para la copia de variables, entre otras.

OpenMP ofrece una interfaz sencilla e intuitiva que permite una rápida paralelización del código, y potencia para desarrollar implementaciones más complejas que aprovechen mejor las ventajas de la ejecución simultánea. Por otro lado al ser un API externo, puede generar problemas de sincronización difíciles de depurar e imposibilitar el manejo manual de la distribución de hilos en los recursos disponibles.

2.3 TBB

TBB es el acrónimo de Threading Building Blocks, es una tecnología de la compañía Intel (Intel 2012), que se presenta en forma de biblioteca de C++ para el soporte y desarrollo de programación paralela. Diseñada para promocionar la programación escalable de datos paralelos, tiene además soporte para paralelismo anidado, lo que permite construir bloques paralelos tomando como base otros bloques paralelizados ver figura 2.2.

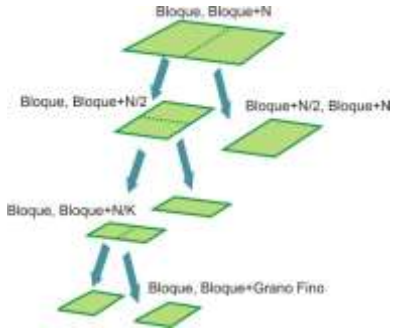


Figura 2.2 Paralelismo anidado de bloques en TBB

En TBB se especifican tareas, no hilos, y se deja que la biblioteca mapee dichas tareas en hilos de manera eficiente. La biblioteca ofrece una funcionalidad similar a la de OpenMP, aunque de diferente ejecución. TBB contiene algoritmos básicos y avanzados de paralelización, contenedores de datos concurrentes, distribución de memoria escalable, controles de concurrencia, operaciones atómicas, herramientas de medición de tiempos y un planificador de tareas.

La aproximación de paralelización de TBB consiste en dividir las tareas de forma equitativa entre los recursos disponibles. Si alguno de los recursos termina sus tareas mientras otros aún poseen una carga significativa de trabajo, TBB asignará parte de este trabajo, si es posible, al recurso que ha quedado libre. Este hecho permite, a una misma aplicación, escalar de forma diferente dependiendo de la máquina en que se ejecute sin necesidad de cambiar el código. TBB se encuentra disponible tanto de forma comercial como en forma open source.

2.4 OpenCL

Éste lenguaje tiene los mismos objetivos que CUDA: permitir derivar cálculos matemáticos a tarjetas gráficas, pero con la diferencia de que OpenCL trata las plataformas de forma homogénea. Esta característica, que permite hacer uso de distintos fabricantes de tarjetas y CPUs en la misma máquina, se deriva de su concepción como estándar gratuito y abierto. OpenCL ha sido desarrollado por el grupo Khronos, que ha contado con la colaboración de Intel, Apple, ATI y NVIDIA. El enfoque de OpenCL, pese a no estar orientado a ninguna arquitectura concreta, permite un control a más bajo nivel, con lo que se puede lograr un mayor rendimiento a costa de una mayor dificultad en su programación. Además, el código de OpenCL es totalmente portable, de forma que un mismo programa compilado puede ejecutarse sin problemas sobre máquinas con múltiples CPUs, máquinas con cluster de tarjetas gráficas, etc. de forma que opera como si fuese una única plataforma. La arquitectura lógica de OpenCL se basa en los dos siguientes elementos:

- Servidor: contiene dispositivos compatibles con OpenCL, una aplicación de OpenCL se ejecuta en el servidor, que es el encargado de realizar el reparto de tareas en los dispositivos disponibles.

- Unidades de procesamiento: cada dispositivo que forma parte del servidor puede tener a su vez una subdivisión en componentes capaces de realizar cálculos, como puede ser los núcleos de un procesador o las unidades de cálculo al igual que en una GPU éstos son los elementos de procesamiento que se encargan de realizar todas las operaciones, procesando cada una de ellas como un único flujo de instrucciones, ver figura 2.3.

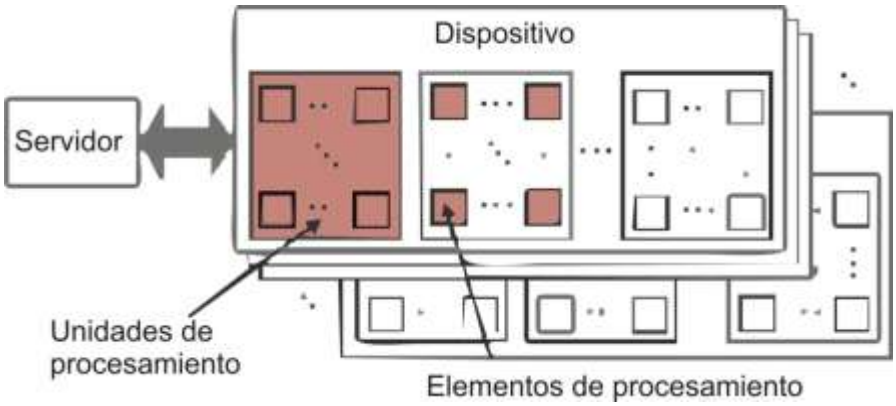


Figura 2.3 Organización de OpenCL

La ejecución de las aplicaciones de OpenCL se realiza en dos procesos. Primeramente el código anfitrión, que se ejecuta en el servidor y que es el encargado de gestionar los otros procesos, al igual que en CUDA se llaman núcleos (kernels). Estos núcleos son procesados por los dispositivos. La organización interna de los núcleos es similar a la explicada posteriormente en CUDA: cada núcleo se divide en un espacio de índices donde cada índice corresponde con un grupo de trabajo (work-group).

A continuación cada grupo de trabajo se separa en elementos de trabajo (work-items), éstos elementos de trabajo son equivalentes a los hilos (threads) de CUDA, en el sentido que cada uno ejecuta el mismo código pero con datos deferentes (la filosofía base de SIMD). Los elementos de trabajo tienen sus propios índices locales dentro de su grupo de trabajo y sus índices globales. Todos los elementos de un grupo de trabajo se ejecutan de manera concurrente en un elemento de procesamiento.

Los work-groups se organizan en grids que serán otro espacio de índices, como se muestra en la figura 2.4

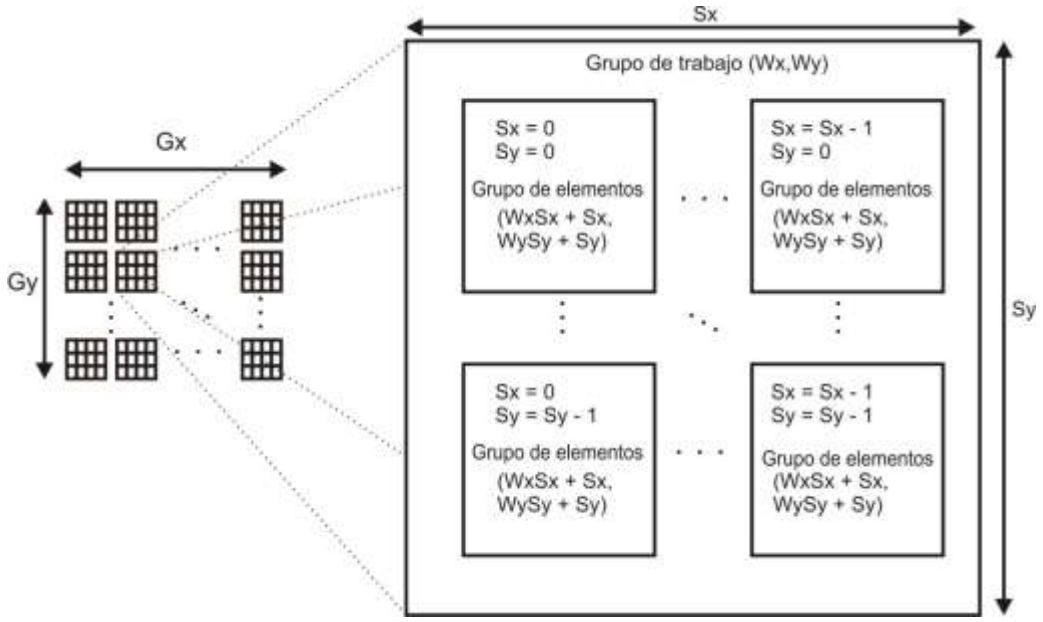


Figura 2.4 Estructura de los elementos y grupos de trabajo en un núcleo de OpenCL

Para permitir el procesamiento y que el servidor pueda repartir los nucleos, OpenCL hace uso de la siguiente estructura de memoria (AMD 2011):

- Memoria Global: Memoria principal a la que todos los elementos de trabajo y el servidor tienen acceso pleno. Esta memoria podrá hacer uso de cachés asociadas si el dispositivo dispone de ellas.
- Memoria Constante: Almacena los elementos constantes durante la ejecución de un núcleo. Solo el host tiene permisos de escritura en esta memoria.
- Memoria Local: Cada grupo de trabajo dispone de su propia memoria específica, a la que tienen acceso total y único. Suele usarse para guardar las variables locales del grupo de trabajo, pero también puede emplearse para mapear secciones de la memoria global.
- Memoria Privada: Memoria exclusiva de un elemento de trabajo. Las variables almacenadas en este espacio no podrán ser accedidas por ningún otro elemento.

Gracias a su modelo homogéneo, OpenCL ofrece tres posibilidades a la hora de realizar la programación concurrente: paralelización de datos, paralelización de tareas y una combinación de ambos. La paralelización de los datos es equivalente al modelo usado en CUDA, donde el programador puede gestionar personalmente la agrupación de elementos de trabajo o indicar el total de éstos y permitir que OpenCL haga la división de forma autónoma. La segunda opción es similar a la programación clásica de hilos (threads) donde se separan las tareas que son independientes y se ejecutan de forma paralela. La última opción consiste en dividir las tareas y gestionar manualmente los elementos de trabajo de cada una de ellas. Al ser un lenguaje que pretende ofrecer una interfaz homogénea para realizar cómputos en sistemas heterogéneos, hay que tener muy presente las capacidades del sistema en cuestión, para poder obtener la máxima aceleración, aunque de esta forma siempre se ha de sacrificar la portabilidad del programa. OpenCL además tiene la misma desventaja que CUDA: la conexión entre los diversos dispositivos depende de la velocidad de los buses, y por lo tanto puede perderse una gran cantidad de tiempo en transferencias de datos si éstas no son las mínimas posibles.

2.5 CUDA

El desarrollo de aplicaciones en paralelo ya no está restringido sólo a aplicaciones que se realizan en grandes y costosos equipos de cómputo. En los años recientes prácticamente todos los modelos de computadoras personales cuentan con al menos un procesador con dos núcleos y en algunos casos, poseen tarjetas gráficas con poder computacional suficiente para llevar a cabo operaciones de propósito general. Desde que, en el 2001, la compañía NVIDIA liberó la serie de tarjetas gráficas GeForce, que tuvieron la capacidad de ejecutar tanto instrucciones de propósito general como para el manejo de gráficos en ellas, la programación en paralelo ha avanzado rápidamente. Se ha evolucionado desde tener que "engañar" al GPU utilizando para otros propósitos sus instrucciones para el manejo de píxeles, hasta contar con las interfaces de programación específicas que existen actualmente, por ejemplo CUDA.

La tecnología CUDA, por sus siglas en inglés (Compute Unified Device Architecture) surgió en 2007, cuando NVIDIA lanzó sus tarjetas GeForce 8800 GTX las que por primera vez incluyeron elementos dirigidos específicamente a posibilitar la solución de problemas de propósito general. Poco después, NVIDIA lanzó el compilador CUDA C, el primer lenguaje

de programación para desarrollar aplicaciones de propósito general sobre un GPU; con la finalidad de captar la mayor cantidad de programadores posibles que adoptasen esta arquitectura, CUDA C es un lenguaje muy similar a C, al cual se le agregaron un conjunto de instrucciones que harían posible la programación en paralelo en un sólo equipo de cómputo (Sanders y Kandrot).

Este permite a los desarrolladores de software utilizar la capacidad multiprocesamiento de las tarjetas gráficas para realizar tareas de propósito general. Esto se conoce como GPGPU (General Purpose Computing on Graphics Processing Units) y la idea consiste en utilizar la alta concurrencia de las tarjetas de video para aplicaciones de cómputo general como pueden ser simulaciones de tráfico (Karimi et al., 2010), simulaciones físicas, algoritmos de búsqueda, cálculos con vectores o matrices de grandes tamaños, comparación de secuencias, procesamiento de imágenes médicas y en el presente trabajo para incrementar el desempeño de algoritmos genéticos aplicados a la solución de problemas de optimización con maximización.

Este modelo es relativamente nuevo y ha ganado muchos adeptos en los últimos años. Se basa en el hecho de que los desarrolladores usan las GPUs como procesadores de propósito general, entendiendo "propósito general" como aplicaciones intensivas en datos, usadas con fines científico-técnicos. Los programadores usan los sombreadores de píxel (del inglés pixel shader, abreviado PS) de las GPUs como si se tratasen de unidades de punto flotante; este PS sirve para manipular un píxel, o lo que es lo mismo, aplicar un efecto sobre la imagen (realismo, bump mapping, sombras, explosiones y efectos).

Si bien la idea de utilizar procesadores gráficos para tareas generales ya existía antes de la aparición de CUDA, este entorno presenta ciertas ventajas frente a los enfoques antiguos como lo son los lenguajes como CG y APIs como OpenGL.

Entre las ventajas encontramos principalmente la capacidad de acceder a la memoria compartida de las tarjetas gráficas, la cual alcanza velocidades mucho mayores que la memoria del dispositivo y más importante, la posibilidad de trabajar en contextos no gráficos. Sin embargo la más notoria desventaja de CUDA es que sólo se puede utilizar en GPUs de

NVIDIA, por lo que la portabilidad de las aplicaciones desarrollas para esta arquitectura está claramente limitada a las tarjetas gráficas de esta empresa.

Por otra parte el alto poder de cómputo y la flexibilidad introducida en las recientes GPUs están permitiendo solucionar problemas, hasta hace poco desafiantes para PCs, usando hardware de bajo costo. Esto ha generado cambios importantes en todo el contexto ya que ha puesto al alcance hardware paralelo de muy alto rendimiento, que puede incluso ser utilizado en resolución de problemas no necesariamente del dominio del cómputo gráfico (Fung et al., 2005), como en el caso de la visión por computadora.

Recientemente las capacidades de las tarjetas de video, han dejado de crecer linealmente para crecer exponencialmente, de manera específica las tarjetas a las cuales tiene acceso el consumidor común. Este tipo de tarjetas están avanzando mucho más rápido que la ley de More la cual estipula que el poder de procesamiento del hardware se duplica cada 1.5 años. En los últimos cuatro años ha habido un avance tan drástico que es difícil mantenerse al día en cuanto al desarrollo tecnológico de las GPU y el poder de procesamiento de estas tarjetas. Las, cuales se ha incrementado más de 10 veces en este periodo (Akenine-Möller, 2002). Las tarjetas de video que ahora están apareciendo en las computadoras de escritorio convencionales son casi tan poderosas como las que hay en las estaciones de trabajo un año atrás en laboratorios científicos de todo el mundo. Se han realizado comparaciones hechas entre los CPUs de las computadoras y los GPUs de las tarjetas de video actuales ver figura 2.5.

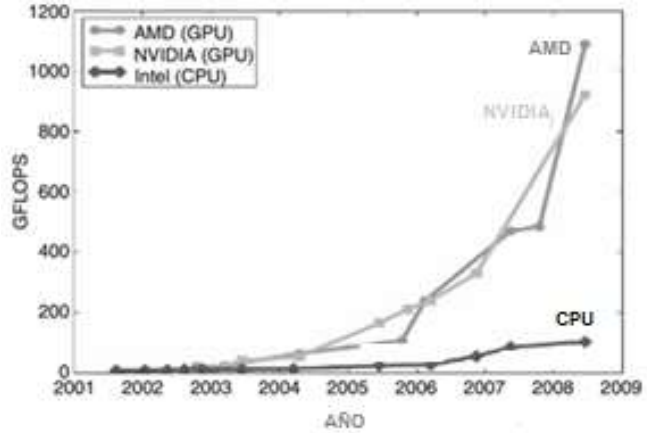


Figura 2.5 Diferencia de desempeño entre GPUs y CPUs

En cuanto al número de instrucciones de punto flotante que pueden realizar en un segundo (FLOP/s). En ellas se pueden apreciar que los CPUs se han incrementado en promedio 2.0 veces su poder de procesamiento, mientras que los GPUs lo han incrementado en promedio 3.7 veces cada 18 meses (Wloka y Huddy, 2003).

2.5.1 Ejecución de un kernel

La parte que ejecuta un programa en CUDA sobre la GPU se denomina kernel e inicia realizando una copia de los datos de memoria principal a la memoria del dispositivo, para posteriormente ceder la ejecución de la CPU a los stream multi-processor (SM) de la tarjeta, los SM son los procesadores físicos con los que cuenta una GPU también llamados núcleos de procesamiento. Posteriormente, el kernel se ejecuta sobre una configuración malla (grid), esta configuración se describe en el siguiente apartado ver figura 2.7, que gracias a las características multi-hilo y memoria compartida logra acelerar la ejecución. Por último, la GPU copia desde su memoria local a la principal los resultados obtenidos en el proceso y devuelve la ejecución a la CPU, dando así por finalizada la utilización de la tarjeta.

Se muestra en la figura 2.6 que la declaración de un kernel es prácticamente la misma que la de una función en C con la diferencia que se utiliza la declaración __global__ para especificar que se trata de una porción de código que habrá de ejecutarse en el GPU. Por otra parte, además de pasar parámetros al kernel como si de una función se tratara, se especifican los parámetros de ejecución del mismo de la siguiente forma: <<<Bloques, Hilos>>>

```
// Declaración de un kernel para multiplicar los
// elementos de dos vectores

__global__ void kernel0(
     float *a,
     float *b,
     float *c,
     int N)
{
     int idx = threadIdx.x;
     c[idx] = a[idx]*b[idx];
}

// Llamada el Kernel

int main(){
     kernel0<<<Blocks,Threads,SMem>>>(
               vector_a,
               vector_b,
               vector_c,
               N);
 }
```

Figura 2.6 Declaración y llamada de un kernel CUDA para multiplicar dos arreglos punto a punto.

2.5.2 Hilos, bloques y mallas

En la arquitectura de CUDA, se visualizan los siguientes elementos:

- Hilos (Threads): donde se ejecutan las funciones. Los hilos pueden procesarse simultáneamente y cuentan con memoria local para almacenar sus datos.
- Bloques (Blocks): un conjunto de hilos que se ejecutan en un multiprocesador. Cada bloque cuenta con una zona de memoria, denominada memoria compartida (Shared Memory), accesible para todos sus hilos.
- Malla (Grid): un conjunto de bloques. Cada malla se ejecuta sobre un GPU distinto y cuenta con memoria accesible (Global Memory) para todos los bloques que la componen.
- Host (CPU): funciones que se ejecutan sobre la CPU.
- Device (GPU): funciones que se ejecutan sobre la GPU.
- Núcleo (Kernel): función llamada desde el Host que se ejecuta en el Device.

Donde la unidad básica de operación es el hilo (Thread). Los hilos están organizados en grupos de (Blocks), y estos organizados en grupos de bloques (Grids) y estos últimos solo pueden ejecutarse dentro de un kernel. Para describir un flujo de trabajo de los elementos antes mencionados de acuerdo a una jerarquía de procesamiento de datos en CUDA un kernel se ejecuta mediante un arreglo de hilos en el que cada uno tiene un identificador (ID) que se usa para direccionar la memoria y tomar las decisiones de control.

2.5.3 Ejecución de un programa en CUDA

Con la consideración anterior se puede comprender mejor la ejecución de un programa en la arquitectura CUDA. Como se observa en la figura 2.7, la aplicación se ejecuta como un código secuencial en el CPU (Host) hasta que se llega a la llamada al Kernel 0, momento en el que se pasan los parámetros al kernel y este comienza a ejecutarse en el GPU (Device).Mientras sucede esto, el CPU está habilitado para continuar ejecutando las siguientes instrucciones del programa.

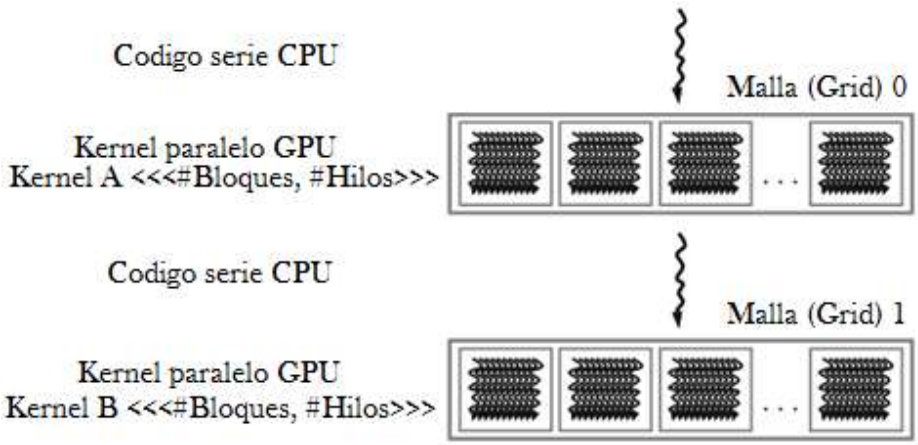


Figura 2.7 Flujo de la ejecución de un programa en la arquitectura CUDA.

La última generación de procesadores gráficos lanzada al mercado por NVIDIA cuenta con miles de procesadores, en contraste con los 512 de la generación previa. Las principales mejoras en la arquitectura de los procesadores de NVIDIA capaces de ejecutar código CUDA son: la disminución del tamaño de los transistores y el consumo energético. La estructura de memoria de las GPUs define diferentes localidades y tipos de memoria que pueden ser utilizadas mientras procesan se ejecuta un kernel en CUDA como sigue:

- Memoria Global (Global Memory): Memoria cuya tarea es comunicar la CPU con la tarjeta gráfica. Como esta memoria no dispone de caché propio es necesario aprovechar al máximo el ancho de banda del BUS para hacer el mínimo de transferencias posibles entre la GPU y la CPU.
- Memoria Constante (Constant Memory): Otra memoria que se puede usar para la comunicación entre la CPU y la GPU. Esta memoria sí tiene caché, pero al ser de sólo lectura (por parte de la GPU) su uso debe ser cuidadoso, ya que una vez que la CPU escribe un dato en la memoria, permanece constante para la tarjeta.
- Memoria Textura (Texture Memory): Como su nombre indica, esta memoria está destinada al tratamiento de gráficos, y ha sido ideada para tener mejor ancho de banda a costa de ofrecer una menor latencia.

- Memoria Local (Local Memory): Las memorias anteriores están compartidas por todos los GPCs (Graphics Computing Clusters). Sin embargo esta memoria es exclusiva para cada thread y se utiliza para sus variables locales.
- Memoria Compartida (Shared Memory): Memoria compartida por todos los hilos de un bloque. Es la más rápida de la tarjeta y si se programa adecuadamente puede igualar la eficiencia del uso de registros en una CPU.

Todas las descripciones anteriores son a nivel lógico, ya que a nivel físico las GPU siguen estando diseñadas para el tratamiento de gráficos y generación de la salida de vídeo. En la figura 2.8 se muestra un ejemplo del diseño físico de la arquitectura llamada Kepler, puede verse que la estructura en memoria descrita anteriormente encajará de forma aproximada con la arquitectura física.

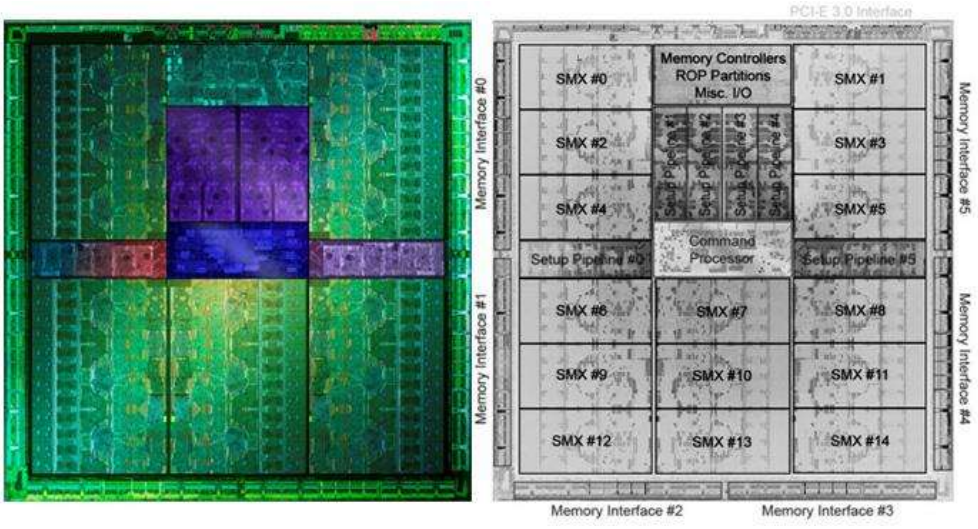


Figura 2.8 Arquitectura del Procesador Kepler en tarjetas gráficas NVIDIA

Además de tener en cuenta la estructura de memoria que plantea CUDA, el programador deberá familiarizarse también con la configuración de los threads. Previamente se ha explicado que cada bloque es una agrupación de threads, pero falta un componente intermedio: el WARP. Un WARP es la unidad mínima de threads que se agrupan en un bloque. Ésta agrupación será de 32 threads, tamaño que no puede ser cambiado por el usuario. Esto es importante porque un thread es la unidad mínima de ejecución, pero un warp es la unidad

mínima que puede manejar el programador, lo que conduce a que el uso de barreras de sincronización no se hace a nivel de thread, sino a nivel de warp. Además, esta agrupación obliga a que todos los threads de un warp deben realizar la misma tarea. Otra limitación importante es que el tamaño máximo de un bloque es de 1024 hilos, y que por cada dimensión tienen un máximo de: 1024 en X, 1024 en Y y 64 en Z; esto quiere decir que en cada bloque se pueden tener configuraciones del tipo 128 * 4 *2, 4 * 64 * 4, etc. El tamaño máximo de un grid es de 65536 por cada dimensión, dando un máximo de 64 bloques de 1024 threads. Éstos datos son ciertos para la versión 4.1 de CUDA [Nvidia 2012b], pero seguramente crecerán a futuro.

2.5.4 Concepto de granularidad

La granularidad se refiere al coeficiente entre el tiempo para una operación de comunicación básica entre una operación de cómputo (Culler et al., 1999). Esto es, el tamaño relativo de las unidades de cómputo que son ejecutadas al mismo tiempo. La granularidad se mide de acuerdo al grosor o fineza con que se dividen las tareas en un sistema de cómputo paralelo. Principalmente se clasifican en dos tipos de granularidad:

- Grano fino

Se trabaja a nivel de funciones, ciclos o instrucciones, en los que se requiere un alto grado de interacción y comunicación entre las unidades de procesamiento. Por lo general, consisten de una gran cantidad de procesadores simples y relativamente lentos.

- Grano grueso

En el caso de grano grueso una aplicación o grupo de procesos o aplicaciones, se dividen en procesos o tareas prácticamente independientes que se ejecutan al mismo tiempo y la comunicación entre las unidades de procesamiento es menor. Por lo general, consisten de pocos procesadores, aunque con mayor capacidad de cómputo que los de grano fino.

Los GPUs son unidades de procesamiento de grano fino, ya que, se conforman de una gran cantidad de unidades de procesamiento, aunque, con relativamente poco poder de cómputo

comparado con los procesadores del CPU y además, los GPUs, requieren un alto grado de coordinación en el cómputo paralelo.

2.5.5 GPU vs CPU

Una GPU es un procesador que inicialmente estaba destinado exclusivamente para desplegar gráficos en una computadora. Sin embargo, actualmente es posible desarrollar cómputo de propósito general (GPGPU) usando GPUs, los cuales se consideran una arquitectura SIMD en la clasificación de Flynn.

Se puede observar en la figura 2.9, la evolución en los últimos años de los CPUs frente a los GPUs fabricados por las empresas pioneras en el desarrollo de procesadores multinúcleo

Figura 2.9 Desempeño de doble precisión en GFLOPS

Con las últimas actualizaciones de las tarjetas gráficas el valor máximo se aproxima a los 700 GFlops en cambio en los CPUs no ha aumentado más allá de los 200 GFlops. Esto se debe a las diferencias estructurales entre el CPU y el GPU, un aspecto importante es la velocidad del bus (Actualmente PCI 3.0) que comunica la GPU con la CPU. La capacidad del bus de datos para trasferir información ya sea entre la CPU y la memoria residente en la placa base o entre la CPU y la GPU ha sido clave. Por desgracia el ancho de banda de los buses de datos nunca han estado al mismo nivel de las capacidades de computación entre CPUs y tarjetas gráficas Esta situación ha sido un constante cuello de botella en el mundo de los gráficos y por ello la conexión de la GPU se fue especializando al colocarse más cercana a la memoria y a la CPU,

unido a una organización más eficiente de la memoria en capacidad de cálculo, ver figura 2.10.

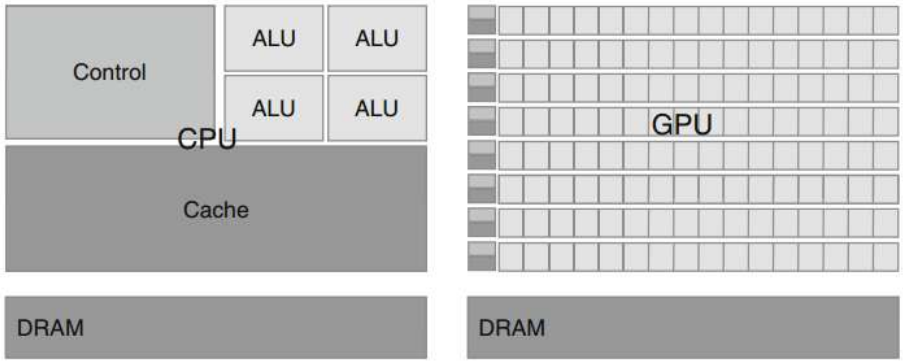


Figura 2.10 Diferencias fundamentales en diseño de CPU y GPU. Tomado de Programming Massively Parallel Processors a Hands-on Approach, David B. Kirk and Wen-mei W. Hwu

La razón de estas diferencias entre la capacidad de la CPU y la GPU, es que la GPU está diseñada para realizar computación-intensiva, altamente paralela y por ello está dotada de mayor cantidad de transistores que se dedican al procesamiento de datos en las unidades aritmético-lógicas (ALU) en lugar de almacenar datos en cache o controlar el flujo de información. Como se muestra en la figura 2.10.

Esto quiere decir que las GPU están especialmente diseñadas para llevar a cabo gran cantidad de operaciones en paralelo. Puesto que cada elemento de procesamiento posee sus propias localidades de memoria, no se requiere habilitar un control de flujo sofisticado. Además, la latencia por accesos a memoria disminuye considerablemente.

Los sistemas informáticos están pasando de realizar el "procesamiento central" en la CPU a realizar "co-procesamiento" repartido entre la CPU y la GPU. Para posibilitar este nuevo paradigma computacional, NVIDIA ha inventado la arquitectura de cálculo paralelo CUDA, que ahora se incluye en las tarjetas gráficas GeForce, Quadro y Tesla, lo cual representa una base instalada considerable para los desarrolladores de aplicaciones. En el mercado de consumo, prácticamente todas las aplicaciones de vídeo se han acelerado, o pronto se acelerarán, a través de CUDA, como demuestran diferentes productos de Elemental Technologies, MotionDSP y LoiLo, Inc. CUDA ha sido recibida con entusiasmo por la

comunidad científica. Por ejemplo, se está utilizando para acelerar AMBER, un simulador de dinámica molecular empleado por más de 60.000 investigadores del ámbito académico y farmacéutico de todo el mundo para acelerar el descubrimiento de nuevos medicamentos. En el mercado financiero, Numerix y CompatibL introdujeron soporte de CUDA para una nueva aplicación de cálculo de riesgo de contraparte y, como resultado, se ha multiplicado por 18 la velocidad de la aplicación. Cerca de 400 instituciones financieras utilizan Numerix en la actualidad. Un buen indicador de la excelente acogida de CUDA es la rápida adopción de la GPU Tesla para aplicaciones de GPU Computing. En la actualidad existen más de 700 clústers de GPUs instalados en compañías de todo el mundo, lo que incluye empresas como Schlumberger y Chevron en el sector energético o BNP Paribas en el sector bancario.

Por otra parte, la reciente llegada de los últimos sistemas operativos de Microsoft y Apple (Windows 8 y Snow Leopard) está convirtiendo el GPU Computing en una tecnología de uso masivo. En estos nuevos sistemas, la GPU no actúa únicamente como procesador gráfico, sino como procesador paralelo de propósito general accesible para cualquier aplicación.

El principal inconveniente del uso de tarjetas gráficas a la hora de hacer computación de alto rendimiento, radica en las transferencias de datos, ya que el bus entre los puertos PCIe donde se conectan las tarjetas y la memoria principal es mucho más lento que la conexión entre la CPU y la memoria. El resultado es que una operación pequeña tal vez se resuelva de forma más rápida en una GPU, pero debido al tiempo que se pierde en las transferencias de entrada y salida de datos es más eficiente hacer la misma operación en la CPU. Sin embargo para operaciones que pueden llevar un largo tiempo, resulta mejor el uso de tarjetas gráficas.

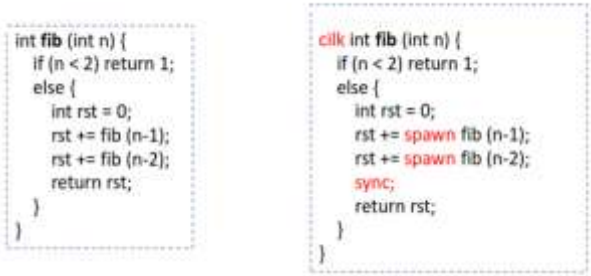
2.6 Cilk

Cilk es un lenguaje de programación multi-hilo basado en ANSI C añadiendo palabras claves específicas para Cilk (constructores paralelos). Diseñado para programación paralela de propósito general, especialmente efectivo para explotar paralelismo dinámico y altamente asíncrono. Desarrollado en el laboratorio de inteligencia artificial y ciencias computacionales del Instituto Tecnológico de Massachusetts (MIT) por el Prof. Charles E. Leiserson, la primera versión de cilk fue liberada en 1994 y actualmente se encuentra en la versión Cilk-5.4.6

desarrollado principalmente en plataformas Unix/Linux. Posteriormente Intel adquirió el proyecto creando Cilk++ que soporta C/C++ y desde entonces es el encargado del mantenimiento y mejora del lenguaje. La versión actual que proporciona Intel mantiene la filosofía básica de Cilk, pero simplifica el lenguaje y añade extensiones para matrices, así como compatibilidad con C++ y una variedad de compiladores (Intel 2009). El objetivo de Intel es convertir Cilk en un estándar de la industria.

El funcionamiento de Cilk se basa en el hecho de que un programador debe desarrollar un algoritmo preparado para ejecutarse en paralelo, identificando los elementos que pueden ejecutarse como tal de forma segura, siempre y cuando sea posible (programa adaptado a las necesidades de Cilk), sin tener que preocuparse por la distribución del procesamiento paralelo, puesto que al ejecutar un programa en Cilk, básicamente se asegura que se asignen las cargas de trabajo, comunicación y sincronización entre los procesadores de forma eficiente, puesto que el planificador de Cilk (algoritmo de planificación work-stealing) decide en tiempo de ejecución cómo dividir el trabajo entre los procesadores y obtener el máximo rendimiento de ese programa. Haciendo uso de arquitecturas paralelas de tipo múltiples instrucciones y múltiples datos (MIMD); así como arquitecturas de memoria compartida distribuida multi-hilo y soporte para tareas paralelas asíncronas.

Prácticamente Cilk se puede apreciar como una extensión del lenguaje C/C++, puesto que solo se utilizan seis palabras reservadas: cilk, spawn, sync, inlet, abort, synched, las cuales solo se beben agregar en la ubicación correcta dentro del programa para implementar códigos de forma paralela como se puede observar en el ejemplo de la figura 2.11 del cálculo de la secuencia Fibonacci.



```
int fib (int n) {
  if (n < 2) return 1;
  else {
    int rst = 0;
    rst += fib (n-1);
    rst += fib (n-2);
    return rst;
  }
}
```

```
cilk int fib (int n) {
  if (n < 2) return 1;
  else {
    int rst = 0;
    rst += spawn fib (n-1);
    rst += spawn fib (n-2);
    sync;
    return rst;
  }
}
```

Figura 2.11 Programa Fibonacci en versión serie y versión cilk. Tomado de "Introduction to Cilk Programming, Feng Liu, 2001"

2.6.1 Constructores paralelos

El lenguaje de programación Cilk utiliza palabras reservadas para definir los constructores que definen y controlan los procesos que se ejecutan en paralelo, a continuación se describe cada constructor:

- **Constructor cilk:** la función principal "main" solo necesita iniciar con la palabra clave cilk para ser capaz de generar un procedimiento paralelo.
- **Constructor spawn:** informa al planificador que se crea un nuevo hilo para realizar una tarea (código del programa). El planificador decide si el hilo se lanzará en paralelo.
- **Constructor sync:** a manera de control no se puede pasar de este punto, porque se debe esperar que todos los hilos (creados por spawn) hayan terminado y devuelto los resultados de la ejecución al padre, es decir, los resultados queden almacenados en variables que sólo conoce el padre, únicamente para aquellos de padres actuales, no actúa de forma global.
- **Constructor inlet:** Es una función interna, que se opera de forma atómica.
- **Constructor abort:** Usado dentro de un inlet, e indica al planificador que cualquier procedimiento generado fuera del padre puede ser abortado de forma segura.

2.6.2 Gráfico acíclico dirigido

La ejecución de un programa Cilk se puede apreciar de forma gráfica en lo que se denomina grafo acíclico dirigido o DAG (del inglés Directed Acyclic Graph) el cual ilustra como los procedimientos principales (padres) y subprocesos (hijos) creados por el constructor spawn, se ejecutan en un programa Cilk donde se divide una sola tarea en procedimientos independientes que se pueden ejecutar en paralelo. Cilk es un lenguaje de programación multi-hilo, donde un hilo se refiere a una cierta cantidad de código serie C, el cual es equivalente a una secuencia de instrucciones sin palabras reservadas de Cilk equivalente a un programa C común, que se puede ejecutar correctamente en un solo procesador. Cilk trabaja en procesadores con memoria compartida, es decir múltiples hilos en un programa Cilk tienen acceso a este tipo de memoria; en otras palabras un programa Cilk se puede ejecutar en un solo procesador o 'N' procesadores, denominados "workers, responsables de un conjunto de

hilos” determinados por el algoritmo de planificación work-stealing. En el DAG cada círculo representa un hilo, los hilos dentro de un recuadro representan un procedimiento y la secuencia de ejecución en cada procedimiento es de izquierda a derecha horizontalmente como lo denotan las flechas. Una flecha hacia abajo indica la creación de un procedimiento hijo y el hilo que lo ejecuta a partir del procedimiento padre al comienzo de esa flecha. Una flecha hacia arriba indica una dependencia de datos, donde un procedimiento hijo ha terminado su ejecución y enviará su valor de retorno a su padre. Se puede ahora analizar el procedimiento que Cilk realiza para el cálculo de la secuencia Fibonacci mediante el DAG correspondiente Figura 2.12.

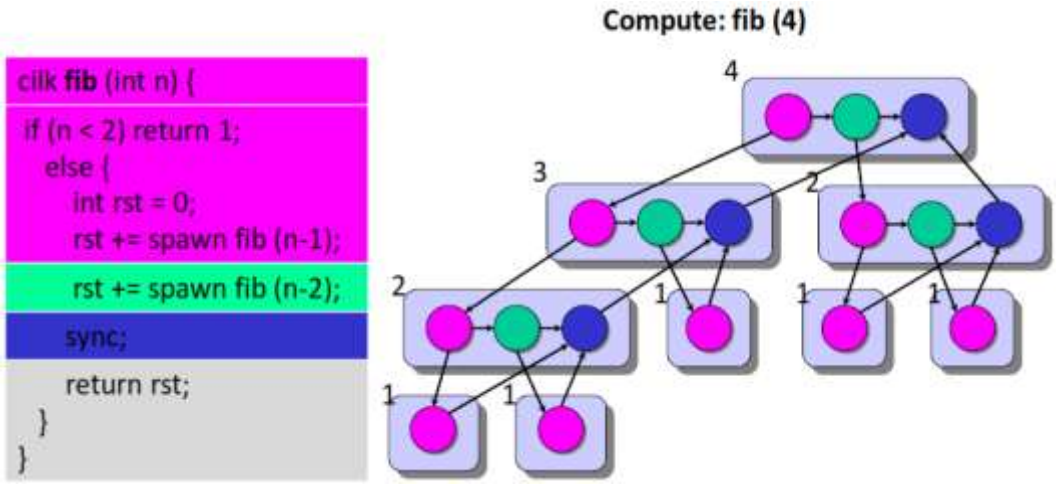


Figura 2.12 Gráfico acíclico dirigido. Tomado de http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Capítulo 3: Arquitecturas Paralelas

3.1 Clasificación de arquitecturas paralelas

Existen diferentes clasificaciones de arquitecturas paralelas. Una de las más conocidas es la clasificación propuesta por Flynn (Flynn 1972), quien clasifica las computadoras paralelas en cuatro tipos diferentes basándose en el número de instrucciones que se ejecutan y en el número de datos sobre los que se aplican estas instrucciones:

- SISD (Single Instruction, Single Data)

En este caso hablamos de una computadora secuencial en la que no hay paralelismo en las instrucciones, ni en el flujo de los datos. Un ejemplo son las computadoras tradicionales con un solo procesador.

- MISD (Multiple Instruction, Single Data)

Esta arquitectura es poco común. En ella, múltiples instrucciones se aplican a un solo flujo de datos. Esta arquitectura se usa en situaciones de paralelismo redundante en donde se requieren sistemas tolerantes a fallos.

- SIMD (Single Instruction, Multiple Data)

La arquitectura SIMD es cuando una misma instrucción es ejecutada sobre un conjunto de datos distintos. En esta arquitectura se cuenta con varios procesadores capaces de ejecutar la misma instrucción simultáneamente, sobre un segmento de datos que previamente fue asignado a cada procesador. Esta arquitectura es muy utilizada en las aplicaciones científicas debido a que en ellas suele requerirse un gran número de operaciones sobre matrices y vectores. Ejemplos de este tipo de arquitectura son las unidades de procesamiento gráfico (GPUs) o las máquinas vectoriales.

- MIMD (Multiple Instruction, Multiple Data)

En las computadoras MIMD se realiza la ejecución simultánea de un grupo de instrucciones sobre distintos flujos de datos. Los sistemas distribuidos se ubican dentro de este tipo de

arquitectura. A su vez, este tipo de arquitectura puede subdividirse en MIMD de memoria compartida, en el caso de que todos los procesadores puedan acceder a todas las regiones de la memoria disponible o MIMD de memoria distribuida en el caso de que cada procesador posea su propia región de memoria y no pueda acceder directamente a los datos contenidos en la memoria de otro procesador.

Estas cuatro arquitecturas se muestran gráficamente en la figura 3.1 en donde cada "PU" (processing unit) es una unidad de procesamiento.

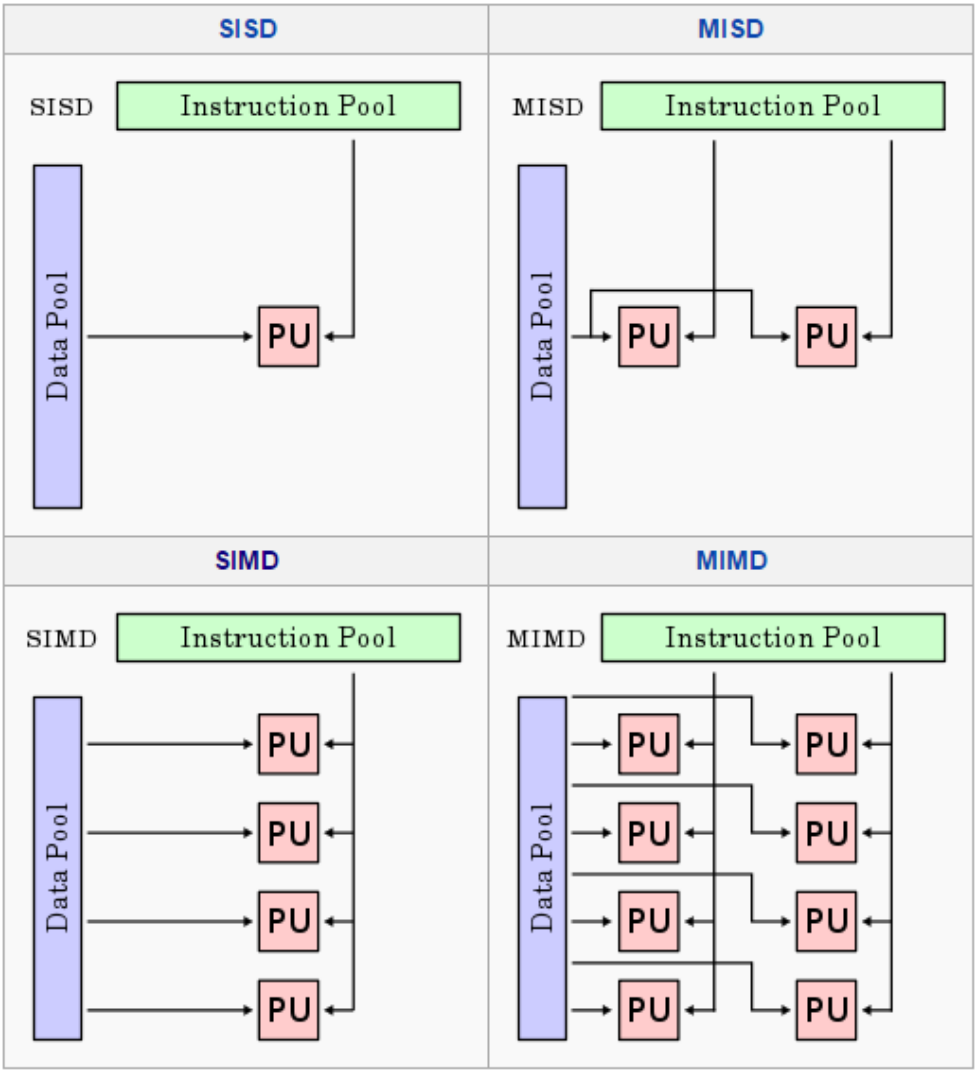


Figura 3.1 Diagrama de comparación en las clasificaciones de arquitecturas paralelas

3.2 Medidas de desempeño

Existen varios parámetros para evaluar el desempeño de un programa paralelo como se definen en (Lee 1980). A continuación se muestra la definición de dichos parámetros.

Eficiencia del sistema. Sea $O(n)$ el número total de operaciones elementales realizadas por un sistema con n elementos de proceso, y $T(n)$ el tiempo de ejecución en pasos unitarios de tiempo. En general, $T(n) < O(n)$ si los n procesadores realizan más de una operación por unidad de tiempo, donde $n \geq 2$. Supongamos que $T(1) = O(1)$ en un sistema mono-procesador. El factor de mejora del rendimiento (speed-up) se define como:

$$S(n) = T(1)/T(n) \tag{3.1}$$

La eficiencia del sistema para un sistema con n procesadores se define como .

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{3.2}$$

La eficiencia es una comparación del grado de speed-up conseguido frente al valor máximo. Dado que $1 \leq S(n) \leq n$, tenemos $1/n \leq E(n) \leq 1$.

La eficiencia más baja ($E(n) \rightarrow 0$) corresponde al caso en que todo el programa se ejecuta en un único procesador de forma serie. La eficiencia máxima *(E(n) = 1*) se obtiene cuando todos los procesadores están siendo completamente utilizados durante todo el periodo de ejecución.

Escalabilidad. Un sistema se dice que es escalable para un determinado rango de procesadores $[1..n]$, si la eficiencia $E(n)$ del sistema se mantiene constante y cercana a la unidad en todo ese rango. Normalmente todos los sistemas tienen un determinado número de procesadores a partir del cual la eficiencia empieza a disminuir de forma más o menos brusca. Un sistema es más escalable que otro si este número de procesadores, a partir del cual la eficiencia disminuye, es menor que el otro.

No hay que confundir escalabilidad con ampliabilidad. Un sistema es ampliable si físicamente se le pueden poner más módulos (más memorias, procesadores, tarjetas de entrada/salida, etc.). Que un sistema sea ampliable no significa que sea escalable, es decir, que un sistema sea

capaz de ampliarse con muchos procesadores no significa que el rendimiento vaya a aumentar de forma proporcional, por lo que la eficiencia no tiene por qué mantenerse constante y por tanto el sistema podría no ser escalable.

Redundancia y utilización. La redundancia en un cálculo paralelo se define como la relación entre $O(n)$ y $O(1)$:

$$R(n) = O(n)/O(1) \tag{3.3}$$

Esta proporción indica la relación entre el paralelismo software y hardware. Obviamente $1 \leq R(n) \leq n$. La utilización del sistema en un cálculo paralelo se define como:

$$U(n) = R(n)E(n) = \frac{O(n)}{nT(n)} \tag{3.4}$$

La utilización del sistema indica el porcentaje de recursos (procesadores, memoria, recursos, etc.) que se utilizan durante la ejecución de un programa paralelo. Es interesante observar la siguiente relación: $1/n \leq E(n) \leq U(n) \leq 1\, y\, 1 \leq R(n) \leq 1/E(n) \leq n$.

Calidad del paralelismo. La calidad de un cálculo paralelo es directamente proporcional al speed-up y la eficiencia, inversamente proporcional a la redundancia, por lo cual, tenemos:

$$Q(n) = \frac{S(n)E(n)}{R(n)} = \frac{T^3\,(1)}{nT^2\,(n)O(n)} \tag{3.5}$$

Dado que *E(n)* es siempre una fracción y *R(n)* es un número entre 1 y n, la calidad *Q(n)* está siempre limitada por el speed-up *S(n)*.

Para complementar las medidas métricas antes decritas, usamos el speed-up *S(n)* para indicar el grado de ganancia de velocidad de una computación paralela. La eficiencia E(n) mide la porción útil del trabajo total realizado por n procesadores.

La redundancia *R(n)* mide el grado del incremento de la carga. La utilización *U(n)* indica el grado de utilización de recursos durante un cálculo paralelo. Finalmente, la calidad *Q(n)* combina el efecto del speed-up, eficiencia y redundancia en una única expresión para indicar el mérito relativo de un cálculo paralelo sobre un sistema.

Grado de paralelismo. Es el número de procesos paralelos en los que se puede dividir un programa en un instante dado. La ejecución de un programa en un ordenador paralelo puede utilizar un número diferente de procesadores en diferentes periodos de tiempo. Para cada periodo de tiempo, el número de procesadores que se puede llegar a usar para ejecutar el programa se define como el grado de paralelismo (GDP). . En la figura 3.2 se muestra un ejemplo de perfil del paralelismo en función del tiempo.

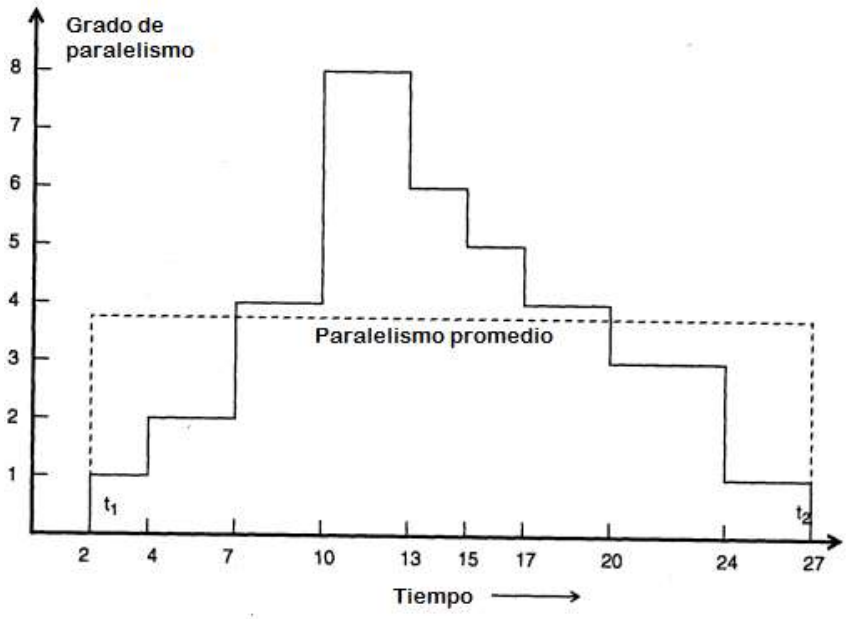


Figura 3.2 Perfil del paralelismo de un algoritmo del tipo divide y vencerás.

3.3 Ley de Amdahl

La ley de Amdahl es un modelo matemático que describe la relación entre la aceleración esperada de la implementación paralela de un algoritmo y la implementación serial del mismo algoritmo (Amdahl 1967).

Técnicamente la ley de Amdahl trata sobre la aceleración S que se puede alcanzar a partir de las modificaciones (mejoras) de una porción P de un cálculo.

$$\frac{1}{(1-P)+\frac{P}{S}} \tag{3.6}$$

Por ejemplo, si se realiza una mejora del 30% en el tiempo de ejecución del cálculo, entonces la porción modificada (mejorada) será 0.3 Y si la porción modificada se ejecuta el doble de rápido, entonces la aceleración será igual a 2. Esta fórmula se deriva de lo siguiente: Asumiendo que el tiempo que toma el cálculo original es 1 (para una unidad de tiempo cualquiera). El tiempo que toma el nuevo cálculo es igual al tiempo que toma la ejecución de la porción no modificada

$$(1 - P) \tag{3.7}$$

Más el tiempo que toma la ejecución de la porción modificada. El tiempo que toma la ejecución de la porción modificada es igual al tiempo que le tomaba originalmente dividida por la aceleración.

$$\frac{P}{S} \tag{3.8}$$

Si se considera el número de procesadores *(N)* en la relación tendremos:

$$S = \frac{1}{(1-P)+\frac{P}{N}} \tag{3.9}$$

Si consideramos el rendimiento que se obtiene cuando se agregan más procesadores a una máquina. La ley de Amdahl puede fusionarse con la ley de disminución del rendimiento.

La aceleración de un programa paralelo está limitada por la porción serial del mismo, es decir Si el 95% de un programa es paralelizable la máxima aceleración obtenida es de 20x como se muestra en la figura 3.3.

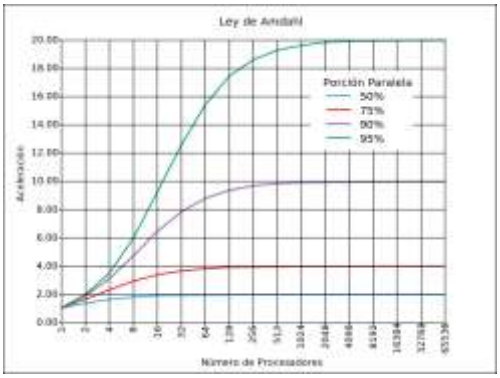


Figura 3.3 Incremento de velocidad de un programa utilizando múltiples procesadores.

Capítulo 4: Algoritmos genéticos

4.1 Algoritmos genéticos

Los Algoritmos Genéticos (AGs) son una técnica de búsqueda usada en análisis numérico para encontrar soluciones más precisas a problemas de optimización. Fueron propuestos inicialmente por Holland (1975), y están basados en analogías con los procesos de evolución biológica (evolución simulada).

Los AGs se implementan mediante una población de representaciones abstractas (llamadas cromosomas) de soluciones candidatas (llamadas individuos o hipótesis), representadas como cadenas de bits que resuelven un problema de optimización, el cual consiste en buscar e identificar dentro del conjunto de soluciones candidatas, la mejor, siendo aquella que optimiza una medida numérica predefinida para el problema, llamada aptitud (fitness function) esta función define el criterio para evaluar las hipótesis potenciales y seleccionarlas probabilísticamente, para su inclusión en la siguiente generación de la población. En un AG, la probabilidad de que una hipótesis sea seleccionada está dada por la tasa (%) de su aptitud con respecto de los demás miembros de la población actual como se observa en la expresión 4.1, este método se conoce como selección proporcional a la aptitud, o selección de ruleta.

$$Pr = \frac{Aptitud\ (h_i)}{\sum_{j=1}^{p} Aptitud\ (h_i)} \tag{4.1}$$

Donde h_i es la hipótesis o solución y P es el número de hipótesis en la población.

El AG inicia operando iterativamente, actualizando un conjunto de hipótesis llamado población, en cada iteración, todos los miembros de la población son evaluados de acuerdo a una función de aptitud, entonces una nueva población es generada, seleccionando probabilísticamente los individuos de mayor aptitud en la población actual, donde algunos de estos individuos pasan intactos a la siguiente generación y otros son utilizados como base para crear nuevos individuos (descendientes o hijos), aplicando operaciones genéticas como cruza y mutación.

Aunque los detalles de implementación varían entre diferentes algoritmos genéticos, todos comparten en general la siguiente estructura:

Función AG(aptitud, umbral, p, r, m)

Aptitud: función de aptitud;

Umbral: determina criterio de paro(iteraciones, aptitud >= umbral);

p: número de hipótesis en la población;

r: porcentaje de población a reemplazar mediante operador de cruza;

m: tasa(%) de mutación;

- Inicia población: P ← genera p hipótesis aleatoriamente;
- Evaluación: para cada h en P, calcula Aptitud(h);
- Mientras [maxAptitud(h)] < umbral, hacer:

Crea nueva generación P_s:

1.- Selección: selecciona probabilísticamente (pierde el mejor individuo) (1 - r)p miembros de P para agregarlos a P_s (Probabilidad **Pr(h_i)**) [Elitismo selección determinista: pasan los mejores]

2.- Cruza: selecciona probabilísticamente $\frac{r \cdot p}{2}$ pares de hipótesis de P de acuerdo a **Pr(h_i)**, para cada par $\{h_1, h_2\}$, produce dos descendientes (hijos), por medio de la aplicación del operador de cruza. Agrega todos los descendientes a P_s.

3.- Mutación: elige m porcentaje (bajo) de los miembros de P_s con probabilidad uniforme. Para cada miembro invierte un bit seleccionado aleatoriamente en esta representación.

4.- Actualiza: P ← P_s;

5.- Evaluación: para cada h en P, calcula Aptitud(h);

- Regresa la hipótesis de P que tenga la Aptitud más alta.

Fin Función;

Se observa que cada iteración de este algoritmo produce una nueva generación de hipótesis, con base en la población actual de hipótesis, un cierto número de hipótesis en la población es seleccionado probabilísticamente para su inclusión en la siguiente generación. La probabilidad de una hipótesis $\{hi \in pob\}$ de ser seleccionada está dada por la expresión (4.1). Por lo tanto, la probabilidad de que una hipótesis sea seleccionada es proporcional a su propio índice

de aptitud, e inversamente proporcional a la aptitud de las hipótesis concurrentes en la misma población. Una vez que estos miembros de la población son seleccionados, se aplica el operador de cruza para crear nuevos individuos también llamados hipótesis o soluciones de la población.

Por lo general en los AGs las hipótesis son casi siempre representadas como cadenas de bits, cuya interpretación depende de la aplicación del algoritmo, de forma que las hipótesis o soluciones puedan ser fácilmente manipuladas por los operadores de cruza y mutación, dichas interpretaciones incluyen pares atributo-valor, expresiones simbólicas, e incluso programas de cómputo. La búsqueda de la hipótesis apropiada comienza con una población inicial, o colección, de hipótesis posibles. En la cual los miembros de ésta, dan lugar a la población de la siguiente generación (descendencia) por medio de operaciones como cruza y mutación que siguen el modelo del proceso en la evolución biológica.

4.2 Operadores genéticos

La generación de las nuevas poblaciones en un algoritmo genético está determinada por un conjunto de operadores que combinan y mutan a los miembros seleccionados de la población actual, el operador de cruza, produce dos nuevos descendientes, a partir de dos cadenas de bits, copiando bits seleccionados de cada padre, en los hijos, el bit en la posición *i* de cada hijo, es copiado del bit en la posición *i* de uno de los padres; la decisión de que padre contribuye con el bit *i* está determinada por una cadena adicional llamada máscara de cruza, ver figura 4.1

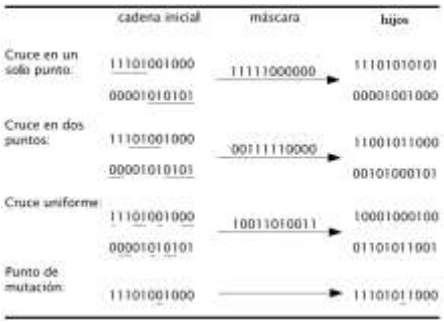


Figura 4.1 Operadores genéticos comunes, la máscara determina la contribución de cada padre. Algunos algoritmos genéticos emplean operadores especializados para una representación de hipótesis en particular

4.3 Algoritmos genéticos en paralelo

Los AGs pueden implementarse en paralelo de manera natural, diversos enfoques de implementación en paralelo han sido estudiados.

Los enfoques llamados de grano grueso (coarse grain) suelen dividir la población entre grupos distintos de individuos llamados demes. Cada deme es asignado a un nodo de computación y un algoritmo genético estándar se aplica en cada nodo.

La paralelización de grano fino, generalmente asigna un procesador a cada miembro de la población. Las combinaciones se dan sólo entre vecinos y diferentes tipos de vecindad son propuestas.

Las regiones que tienen un mejor desempeño que sus vecinas, pero que, en el espacio global no son las óptimas, se conocen como regiones sub-óptimas. Los algoritmos genéticos que se ejecutan en una sola computadora pueden quedar atrapados en regiones sub-óptimas del espacio de búsqueda, convergiendo de manera prematura.

Los algoritmos genéticos en paralelo, al buscar simultáneamente en distintas regiones del espacio de búsqueda, tienen menor probabilidad de quedar atrapados en una región sub-óptima y converger de manera prematura.

Por otro lado, al procesarse de manera simultánea, el cálculo de la aptitud y la aplicación de los operadores genéticos en varios individuos de distintas poblaciones, se disminuye considerablemente el tiempo de ejecución del algoritmo.

Una de las ventajas de implementar el AG en paralelo es que reducen en gran medida el fenómeno de *crowding* que es común en versiones no paralelas de los algoritmos genéticos, en dicho fenómeno, un individuo que es más apto que otros miembros de la población, comienza a reproducirse rápidamente de tal forma que copias de este individuo, o bien de individuos muy parecidos, ocupan un porcentaje importante de la población. El efecto negativo de este fenómeno es que reduce la diversidad de la población, haciendo la búsqueda más lenta.

4.4 Taxonomía de los algoritmos genéticos en paralelo.

La clasificación de los algoritmos genéticos de acuerdo a su distribución en paralelo se compone de ocho categorías (Nowostaki y Poli 1999), considerando los siguientes rasgos:

- La forma de evaluar la aptitud y aplicar la mutación.
- Si se implementa en una sola población o varias sub-poblaciones.
- En el caso de que se implemente en varias sub-poblaciones, la forma en la que se intercambian individuos entre ellas.
- Si se hace selección global o localmente.

Las categorías propuestas son:

1. Maestro-Esclavo.

- Una sola población.
- Generalmente se paraleliza el cálculo de aptitud y la mutación, a cargo de varios procesos esclavos.
- La selección y el cruce se llevan a cabo de manera global, a cargo de un proceso maestro, el cual puede ser:
    - Síncrono: el maestro espera a que todos los esclavos terminen de aplicar sus operadores, antes de iniciar la siguiente generación.
    - Asíncrono: el maestro sólo espera a una fracción de los esclavos para iniciar la siguiente generación.

2. Sub-poblaciones estáticas con migración.

- También conocido como: modelo de islas, algoritmo genético distribuido o algoritmo genético de grano grueso.
- Conjunto estático de sub-poblaciones independientes.
- Requiere implementar un operador genético adicional: migración, que consiste en intercambiar individuos entre sub-poblaciones cada cierto número de generaciones. La migración permite compartir material genético entre las sub-poblaciones y para

implementarla requiere que se incluya en el algoritmo un conjunto de parámetros adicionales.

- Los modelos de migración pueden ser:
    - Anillo: antes de iniciar el algoritmo se determina la población destino de cada elemento a migrar, esto se muestra en la Figura 4.2

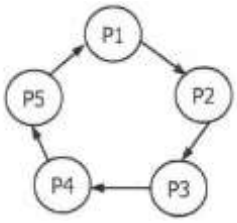


Figura 4.2 Migración en anillo de diferentes poblaciones del algoritmo

    - Aleatoria: Se elige al azar la población destino, en el momento de ejecutar la migración, ver Figura 4.3

Figura 4.3 Migración aleatoria de diferentes poblaciones del algoritmo

3. Sub-poblaciones estáticas superpuestas.

- A diferencia de la categoría anterior, no existe un operador de migración.
- El intercambio de información genética se hace mediante individuos que pertenecen a más de una sub-población.

4. Algoritmos genéticos masivamente paralelos.

- También conocidos como algoritmos genéticos de grano fino.
- Básicamente funcionan como la categoría anterior, solamente disminuye el número de individuos en la sub-población.

- Se implementan en computadoras masivamente paralelas.

5. Sub-Poblaciones dinámicas.

- Una sola población.
- Se divide en sub-poblaciones en tiempo de ejecución.

6. Algoritmos genéticos de estado estable.

- Se distinguen de los algoritmos maestro-esclavo con evaluación asíncrona de aptitud por la forma en la que aplican la selección.
- El algoritmo no espera a que una fracción de la población haya concluido su evaluación, sino que continúa con la población existente.

7. Algoritmos genéticos desordenados.

- Constan de tres fases:
    - Inicialización: se crea la población inicial, usando algún método de enumeración parcial.
    - Primaria: se reduce la población mediante alguna forma de selección y se evalúan sus individuos.
    - Yuxtaposición: se unen las soluciones parciales resultantes de las fases anteriores.

8.- Métodos híbridos.

- Combinan características de los métodos anteriores.

Cada una de las categorías anteriores puede presentar mejores rendimientos, dependiendo del tipo de problema a tratar y de la arquitectura de la máquina paralela en la que se implementará la solución, en la figura 4.4 se muestra el diagrama de flujo de un algoritmo genético en paralelo.

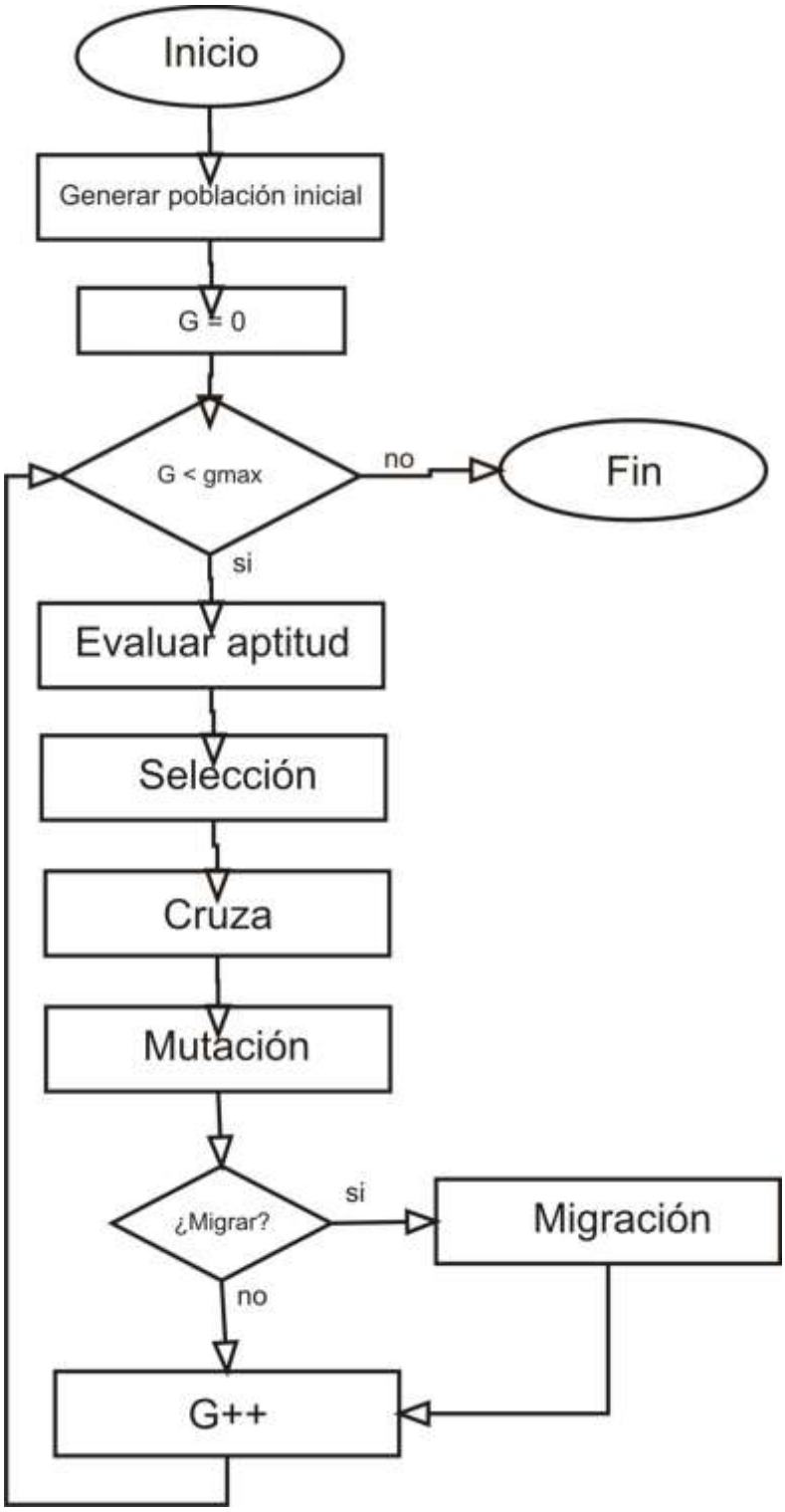


Figura 4.4 Diagrama de flujo de un algoritmo genético en paralelo con operador de migración.

Capítulo 5: Implementación

En este capítulo se especifican las herramientas y tecnologías seleccionadas, así como una descripción del proceso en el desarrollo del algoritmo.

5.1 Lenguaje de programación y tecnología de paralelización

De acuerdo al estudio realizado en el segundo capítulo, se ha decidido utilizar C como lenguaje base para la programación del algoritmo debido a los siguientes puntos:

- Es un lenguaje abierto que permite la implementación de programas siguiendo prácticamente cualquier paradigma de programación, de forma que el programador no está restringido a diseñar la solución de una forma concreta por limitaciones en la herramienta.
- C/C++ es un lenguaje compilado, no interpretado, por lo que el rendimiento es mayor y ofrece un amplio marco de opciones de optimización mediante instrucciones al compilador, sin necesidad de cambiar e l código del programa.
- Es portable, en el sentido de que se puede compilar para funcionar en la plataforma deseada.
- Todas las herramientas de paralelización están disponibles para este lenguaje, de forma que pueden realizarse diferentes optimizaciones y comparaciones de rendimiento sin necesidad de tener que reescribir el programa.

Como herramienta de paralelización se ha decidido usar Cilk porque ofrece una interfaz sencilla que permite realizar la paralelización introduciendo pocos cambios sobre la implementación secuencial. Esta herramienta en su versión original se encuentra disponible de forma libre para el lenguaje C estándar. Se utilizó el sistema operativo Linux Ubuntu debido a que Cilk está diseñado originalmente para sistemas Unix.

Las arquitecturas de hardware utilizadas son: equipo portatil DELL PRECISION M4500 equipada con un procesador multinúcleo Intel(R) Core(TM) i7 CPU Q840 @ 1.87 GHz, 1867Mhz, 4 procesadores principales, 8 procesadores lógicos; así como en un clúster de

computadoras de 16 nodos con procesadores Intel Xeon de doble núcleo por lo cual su arquitectura proporciona 32 procesadores principales.

5.2 Diseño

El algoritmo genético está diseñado para resolver problemas de optimización con maximización de 'n' variables donde se debe encontrar el valor de $'x'$ que hace que la función $'f(x)'$ alcance su valor máximo pero restringiendo a la variable $'x'$ a tomar valores dentro de los límites permitidos. Tomando en cuenta que la tarea de optimizar implica determinar los valores para una serie de parámetros de tal manera que, bajo ciertas restricciones, se satisfaga alguna condición (el más grande, el más pequeño, el menos caro, etc.). La optimización es muy importante en muchos campos y especialidades e implica tareas tanto de minimización como de maximización.

Una aplicación muy popular de los algoritmos genéticos es la optimización numérica. La optimización, en general, puede ser definida como el proceso mediante el cual se tiene como objetivo el encontrar el valor máximo o mínimo de una función en todo el espacio de búsqueda. La optimización es una rama de las matemáticas que tiene muchas aplicaciones en la vida real, sobre todo en diseño e ingeniería. De manera general, un problema de optimización consta de los siguientes componentes:

- Variables de decisión: Son los valores que pueden tener las variables para resolver el problema.
- Función objetivo: Es la expresión matemática que sirve para evaluar las variables de decisión con el fin de poder determinar si una solución es mejor que otra.
- Restricciones: Son las ecuaciones de igualdad o desigualdad que limitan los valores que las variables del problema pueden tomar.

De acuerdo a los componentes anteriores un problema de optimización se enuncia de la siguiente manera: Encontrar los valores de las n variables $[x_1, x_2, \dots x_n]$, denotadas por el vector $\vec{x}$, que optimicen (o sea, minimicen o maximicen) la función objetivo $f(\vec{x})$. Debido a que minimizar $f(\vec{x})$ es equivalente a maximizar $-f(\vec{x})$, el problema de optimización puede ser escrito siempre como uno de minimización:

Encontrar $\vec{x} = [x_1, x_2, \ldots x_n]^T$ que minimice $f(\vec{x})$

Donde $\vec{x}$ es un vector n-dimensional llamado vector de diseño, $f(\vec{x})$ es la función objetivo.

5.3 Algoritmo genético

El algoritmo está basado en módulos siendo cada uno de éstos una función que realiza una operación, esto se realizó con la intención de que finalmente el AG realice las operaciones dentro de una función principal haciendo llamadas a cada uno de los módulos donde los principales operadores genéticos son: selección proporcional con modelo elitista, cruza de un punto, mutación uniforme y la representación de los individuos en la población es mediante cadenas de dígitos binarios (genotipo). A continuación se describen y ejemplifican los cálculos realizados por los módulos principales que forman el AG:

Función de inicio: el AG inicia con el archivo de entrada "entradag.txt" en el cual el número de líneas corresponde al número de variables. Cada línea proporciona el límite superior y el límite inferior de cada variable, posteriormente se generan valores aleatoriamente entre estos límites y se inicializa (a cero) todos los valores de aptitud para cada individuo; creando la población inicial se forma una matriz, en donde cada fila representa un cromosoma equivalente a un individuo el cual está compuesto por bits generados aleatoriamente, ver figura 5.1; cabe señalar que el tamaño de estos individuos depende del dominio de las variables de la función a optimizar, p. ej. la que se muestra en la expresión 5.1.

$$f(x_1, x_2) = 21.5 + x_1 * \sin(4\pi x_1) + x_2 * \sin(20\pi x_2) \qquad (5.1)$$

La función de la expresión (5.1) depende de la variable x_1 la cual está definida en el intervalo: $-3 \leq x_1 \leq 12.1$ y de la variable x_2 que está definida en el intervalo: $4.1 \leq x_2 \leq 5.8$ suponiendo que la precisión requerida es de cuatro posiciones decimales (miles) para cada variable se deduce que se necesitan 18 y 15 bits respectivamente; por lo tanto el tamaño total del cromosoma (individuo o solución) es 18+15 =33 bits, una vez definida la codificación de las variables se continua con la creación de un determinado número de individuos (tamaño de la población), como se muestra en la figura 5.1

Figura 5.1 Población inicial de 20 individuos, cada individuo es representado por una cadena binaria, los primeros 18 bits codifican el valor de la variable x_1 y los 15 bits restantes (del bit 19 al 33) codifican el valor de la variable x_2 como se muestra en el cromosoma C1.

Función Objetivo: también definida como función de aptitud (fitness), el AG únicamente maximiza, pero la minimización puede realizarse fácilmente utilizando el inverso aditivo de la función definida por el usuario, la característica principal de esta función es que tiene ser capaz de evaluar que tan buena o mala es una solución o individuo de la población, es decir se determina la aptitud de cada individuo (cromosoma); retomando la función de la expresión (5.1) se muestra a continuación la evaluación correspondiente a cada individuo de la población

generada anteriormente ver figura 5.2 donde se observa que el cromosoma C15 es el mejor o más apto y el cromosoma C2 es el peor o menos apto.

f(x1, x2)

eval(C1) = f(6.084492, 5.652242) = 26.019600
eval(C2) = f(10.348434, 4.380264) = 7.580015
eval(C3) = f(-2.516603, 4.3903381) = 19.526329
eval(C4) = f(5.278638, 5.593460) = 17.406725
eval(C5) = f(-1.255173, 4.734458) = 25.341160
eval(C6) = f(-1.811725, 4.391937) = 18.100417
eval(C7) = f(-0.991471, 5.680258) = 16.020812
eval(C8) = f(4.910618, 4.703018) = 17.959701
eval(C9) = f(0.795406, 5.381472) = 16.127799
eval(C10) =f(-2.554851, 4.793707) = 21.278435
eval(C11) = f(3.130078, 4.996097) = 23.410669
eval(C12) = f(9.356179, 4.239457) = 15.011619
eval(C13) = f(11.134646, 5.378671) = 27.316702
eval(C14) = f(1.335944, 5.151378) = 19.876294
eval(C15) = f(11.089025, 5.054515) = 30.060205
eval(C16) = f(9.211598, 4.993762) = 23.867227
eval(C17) = f(3.367514, 4.571343) = 13.696165
eval(C18) = f(3.843020, 5.158226) = 15.414128
eval(C19) = f(-1.756635, 5.395584) = 20.095903
eval(C20) = f(7.935998, 4.757338) = 13.666916

Figura 5.2 Evaluación de la población mediante la función objetivo, la cual determina la aptitud de cada cromosoma.

Función de selección: realiza una selección proporcional estándar para problemas de maximización que incorpora modelo de elite el cual asegura que el mejor individuo sobreviva; el AG debe otorgar la misma probabilidad a cada individuo de la población de ser seleccionado de acuerdo a su valor de aptitud con respecto a la de los demás miembros, para lo cual calcula la aptitud total de la población de acuerdo a la expresión 5.2.

$$A = \sum_{j=1}^{p} eval(C_i) \tag{5.2}$$

Entonces la probabilidad de selección para cada cromosoma se calcula con $P_i = eval(C_i)/A$ a continuación se muestra la probabilidad de selección de los cromosomas generados en la población inicial, ver la figura 5.3.

P1 = eval(C1)/A = 0.067099
P2 = eval(C2)/A = 0.019547
P3 = eval(C3)/A = .0.050355
P4 = eval(C4)/A = 0.044889
P5 = eval(C5)/A = 0.065350
P6 = eval(C6)/A = 0.046677
P7 = eval(C7)/A = 0.041315
P8 = eval(C8)/A = 0.046315
P9 = eval(C9)/A = 0.041590
P10=eval(C10)/A=0.054873
P11=eval(C11)/A=0.060372
P12=eval(C12)/A=0.038712
P13=eval(C13)/A=0.070444
P14=eval(C14)/A=0.051527
P15=eval(C15)/A=0.077519
P16=eval(C16)/A=0.061549
P17=eval(C17)/A=0.035320
P18=eval(C18)/A=0.039750
P19=eval(C19)/A=0.051823
P20=eval(C20)/A=0.035244

Figura 5.3 Probabilidad de selección P_i para cada cromosoma C_i

Posteriormente el algoritmo acumula esta probabilidad, similar a como lo realizó con el valor de aptitud, ver figura 5.4

q1 = p1 = 0.067099
q2 = q1+p2 = 0.086647
q3 = q2+p3 = 0.137001
q4 = q3+p4 = 0.181890
q5 = q4+p5 = 0.247240
q6 = q5+p6 = 0.293917
q7 = q6+p7 = 0.355232
q8 = q7+p8 = 0.381546
q9 = q8+p9 = 0.423137
q10=q9+p10=0.478009
q11=q10+p11=0.538381
q12=q11+p12=0.577093
q13=q12+p13=0.647537
q14=q13+p14=0.698794
q15=q14+p15=0.776314
q16=q15+p16=0.837863
q17=q16+p17=0.875182
q18=q17+p18=0.912932
q19=q18+p19=0.964756
q20=q19+p20=1.000000

Figura 5.4 Probabilidad acumula q_i para cada cromosoma C_i

Ahora genera una secuencia de números aleatorios r dentro del intervalo [0…1] ver figura 5.5.

r1 = 0.513870 r2 = 0.175741 r3 = 0.308652 r4 = 0.947628
r5 = 0.947628 r6 = 0.171736 r7 = 0.702231 r8 = 0.226431
r9 = 0.494773 r10 = 0.424720 r11 = 0.703899 r12 = 0.389647
r13 = 0.277226 r14 = 0.368071 r15 = 0.983437 r16 = 0.005398
r17 = 0.765682 r18 = 0.646473 r19 = 0.767139 r20 = 0.780237

Figura 5.5 Números generados aleatoriamente r_i

Con cada número generado selecciona un cromosoma para formar una población nueva de la siguiente manera, se compara el número r_i con la probabilidad acumulada q_i de cada cromosoma C_i, p. ej. El primer número r_1 es mayor que q_{10} y menor que q_{11} por lo tanto el

cromosoma C_{11} es seleccionado para incluirlo en la nueva población; el segundo número r_2 es mayor que q_3 y menor que q_4 por lo tanto el cromosoma C_4 es seleccionado para incluirlo en la nueva población, prosiguiendo de esta manera hasta que finalmente la población nueva este completa con los cromosomas seleccionados como se muestra en la figura 5.6.

C'1=01100111111011010110000110111100 (C11)
C'2=10001100010110100111100000111001 0 (C4)
C'3=00100010000011010111101101111101 1 (C7)
C'4=01100111111011010110000110111100 0 (C11)
C'5=00010101001111111111000011000110 0 (C19)
C'6=10001100010110100111100000111001 0 (C4)
C'7=11101111101000100011000000100011 0 (C13)
C'8=00011101100101001101011111100010 1 (C5)
C'9=01100111111011010110000110111100 0 (C11)
C'10=00001000001100100000101011101110 1 (C3)
C'11=11101111101000100011000000100011 0 (C13)
C'12=01000000010110001011000000011111 00 (C9)
C'13=00010100001001010100101011111101 1 (C6)
C'14=10000110000111010001011010110011 1 (C8)
C'15=10111001011001111001100010111111 0 (C20)
C'16=10011010000000111111101001101111 1 (C1)
C'17=00000111100011000001101000011101 1 (C10)
C'18=11101111101000100011000000100011 0 (C13)
C'19=11101110110111000010001111101111 0 (C15)
C'20=11001111000001111110000110100101 1 (C16)

Figura 5.6 Población nueva cuyos cromosomas están denotados ahora por C'_i

Función de cruza: realiza la cruza de dos cromosomas seleccionados en la nueva población de acuerdo a la probabilidad de cruza Pc la cual se especifica en los parámetros de inicio del algoritmo genético, suponiendo que $Pc = 0.25$ se tiene una expectativa de que él 25% de los cromosomas sean sometidos a la operación de cruza, es decir 5 de 20 cromosomas para el caso de nuestra población, procediendo de la siguiente manera: por cada cromosoma en la población nuevamente se genera un número aleatorio r'dentro del intervalo $[0\ldots1]$; sí $r' < Pc$, selecciona el cromosoma C'_i correspondiente a su r' que cumple con la condición anterior, asumiendo que los cromosomas seleccionados para realizar la cruza son: C'_2, C'_{11}, C'_{13} y C'_{18} se continua acoplándolos en parejas las culés en términos de evolución se definen como padres, la primera pareja será formada por C'_2 y C'_{11} la segunda por C'_{13} y C'_{18}. Para cada

pareja se genera un número aleatorio pos dentro del rango [1..33] siendo pos el que indica la posición del punto de cruza; la primera pareja se muestra a continuación:

si $pos = 9$ esta pareja de cromosomas será dividida en 2 partes la primera del bit 1 al 9 y la segunda del bit 10 al 33 para posteriormente combinar las partes del padre 1 con las partes del padre 2 lo que da como resultado un par nuevo de cromosomas C''_i, llamados hijos como se muestra a continuación:

En la segunda pareja de cromosomas:

el proceso es el mismo, solo cambia el número aleatorio, si $pos = 20$ ahora el punto de cruza será en el bit 20 lo que ocasiona que la parte que proporciona cada padre para la cruza sea diferente y en consecuencia su descendencia tenga características diferentes, a continuación se muestran los hijos generados por la segunda pareja de padres:

Una vez aplicado el operador de cruza, los cromosomas generados C''_i se reemplazan por sus predecesores C'_i actualizando la población nueva como se muestra en la figura 5.7

Figura 5.7 Actualización de la población nueva después de aplicar el operador de cruza.

Función de mutación: realiza mutación aleatoria bit a bit en base a la probabilidad de mutación Pm especificada al inicio del algoritmo, el tamaño de la población es de 20 cromosomas de 33 bits cada uno, por lo tanto en total son $20x30 = 660$ bits en toda la población, si $Pm = 0.01$ se tiene una expectativa de que el 1% de bits sean sometidos a la operación de mutación, es decir 6.6 bits en cada población, procediendo de la siguiente manera: cada bit tiene la misma oportunidad de ser mutado, por lo que, para cada bit en la población se genera un número aleatorio r dentro del intervalo $[0 \ldots 1]$; si $r < 0.01$ muta el bit, esto quiere decir que se tienen que generar 660 números aleatorios, se muestran a continuación un ejemplo con 5 de estos números que son más pequeños que 0.01 con el número de bit correspondiente.

Número de bit	Número aleatorio
112	0.000213
349	0.009945
418	0.008809
429	0.005425
602	0.002856

Posteriormente se traslada cada número de bit al número de cromosoma correspondiente, así como la posición que ocupa dentro del cromosoma, como se muestra en la tabla 5.1

Tabla 5.1 Bits seleccionados para mutación.

Número de bit	**Número de cromosoma**	**Posición del bit en el cromosoma**
112	4	13
349	11	19
418	13	22
429	13	33
602	19	8

Los datos en la tabla 5.1 indican que 4 cromosomas serán afectados por el operador de mutación, se observa que el cromosoma 13 tendrá 2 bits modificados; una vez concluida la mutación se vuelve a actualizar la población (los bits mutados aparen en amarillo) como se muestra en la figura 5.8

Figura 5.8 Actualización de la población después de aplicar operador de mutación.

Se ha completado la primera iteración (generación) del ciclo evolutivo del algoritmo genético, resulta interesante examinar los resultados hasta este punto, la aptitud total de la población actual (447.049668) es más alta que la aptitud total de la población inicial (387.776822) con respecto al mejor cromosoma actual (C_{11}) tiene una mejor evaluación (33.776822) que el mejor cromosoma (C_{15}) de la población inicial (30.060205), esto se debe a que el cromosoma (C_{11}) fue uno de los que resulto modificado por los operadores de cruza y mutación, ver figura 5.8, lo que ocasionó una mejora considerable en su evaluación, de esta forma se comprueba que el proceso evolutivo del algoritmo genético funciona correctamente y esto solo durante una sola generación, por consiguiente se deben obtener mejores resultados al incrementar el número de generaciones y/o tamaño de la población, así como especificar los parámetros de inicio, esto de acuerdo a los requerimientos del problema, puesto que el hecho de incrementar por incrementar no garantiza mejor desempeño y puede incluso resultar

contraproducente ya que los algoritmos genéticos también presentan algunos inconvenientes como la perdida de diversidad en la población (óptimos locales).

Función de reporte: el AG finaliza enviando los resultados a un archivo de salida "salidag.txt" separado por comas el cual muestra la mejor aptitud y el valor promedio de cada generación, así como la mejor solución para cada variable.

5.4 Análisis

El análisis del AG se realizó de acuerdo a su estructura y flujo de la función principal, ver figura 5.9 para identificar los bloques de código que demandan mayores recursos de cómputo; es decir, donde se procese gran cantidad de datos o realice un número importante de operaciones, siendo dichos bloques los candidatos a ser procesados de forma paralela.

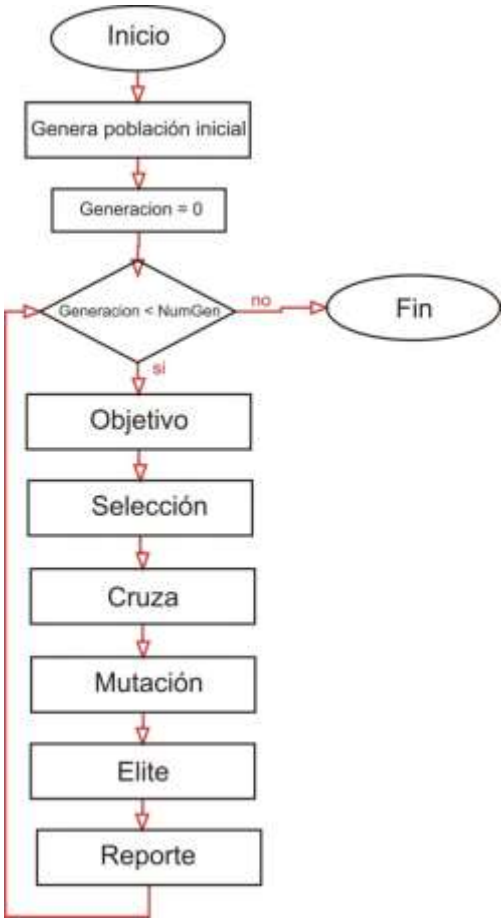


Figura 5.9 Diagrama de flujo de la función principal del algoritmo genético.

Aunque cabe mencionar que en algunos casos resulta contraproducente paralelizar bloques de código puesto que puede tomar más tiempo mover los datos al hardware multinúcleo (en el caso del cluster) para que sean procesados de forma paralela, que el proceso en sí mismo; por estas razones se analizó a detalle la estructura del AG tomando en cuenta dos aspectos importantes:

- Diseño del algoritmo: se analizó el grado de paralelismo en el código fuente del AG puesto que la programación se realizó de forma tradicional; es decir secuencialmente en una sola función, por lo cual se modificó la estructura del código fuente para que las cálculos realizados por el AG se ejecutaran por medio de funciones separadas como se mostró en el apartado anterior, lo cual permitió especificar de mejor manera los bloques de código a paralelizar con Cilk.
- Cálculos realizados: los cálculos se refieren a los operadores genéticos que se utilizan en el AG para emular el proceso de evolución natural (Población, Selección, Cruza, Mutación, etc.), se analizaron los bloques de código correspondientes a cada operador genético del algoritmo para determinar cuál de ellos procesa mayor cantidad de datos y en consecuencia demanda mayores recursos de cómputo. De acuerdo a la literatura especializada existen dos caminos para implementar un algoritmo genético de forma paralela: paralelizar todos los operadores genéticos o solamente la función objetivo (evaluación de los individuos de la población), en este caso se realizaron dos pruebas, en la primera se paralelizaron todos los operadores y en la segunda solamente tres: objetivo, selección y elite.

5.5 Clúster

En el transcurso de la implementación del algoritmo se tuvo la oportunidad de acceder a un clúster de computadoras de 16 nodos, durante la estancia de investigación realizada en la Facultad de Ciencias de la Computación (FCC) de la Benemérita Universidad Autónoma de Puebla (BUAP), en el cual se realizaron conjuntos de pruebas con las versiones paralelas del algoritmo genético, a continuación se describe las principales características del clúster:

- Dirección: cluster.cs.buap.mx
- SO: Linux
- Nodos: 16
- Procesador: Xenon Dual 64 Bits
- Servicios: http, https, ftp, ssh, mysql

De acuerdo a las características anteriores, la que fue de mayor utilidad para realizar las pruebas con el AG es el número de procesadores; puesto que se tienen 16 nodos con un procesador de doble núcleo cada uno, entonces se obtiene un total de 32 procesadores físicos, esto fue un complemento perfecto con la implementación paralela del algoritmo genético, puesto que Cilk permite especificar el número de procesadores con los que se ejecuta un programa en Cilk.

Cabe mencionar que el clúster brinda diferentes servicios y está disponible las 24 horas para los usuarios que se les permite el acceso, por lo que el potencial del mismo se reduce de acuerdo a la carga de trabajo que tenga en el momento de acceder para realizar las pruebas correspondientes.

El acceso al clúster es controlado por la FCC de la BUAP la cual se encarga de administrar el clúster y los servicios que este brinda, los cuales están dirigidos principalmente a docentes y estudiantes de maestría que realizan investigación en alguna de las áreas que el clúster permite hacer experimentación, como lo es en este caso el computo paralelo. Para acceder al clúster se utilizó un cliente para la transferencia de archivos (File Transfering Protocol, FTP) con el cual

se pueden subir y gestionar los archivos y directorios necesarios, esto de acuerdo a los privilegios que tenga cada usuario; finalmente se utilizó un intérprete de órdenes seguras (Secure Shell, SSH) ver figura 5.10 para entrar de forma remota a la línea de comandos del cluster y poder ejecutar el algoritmo genético paralelizado en Cilk.

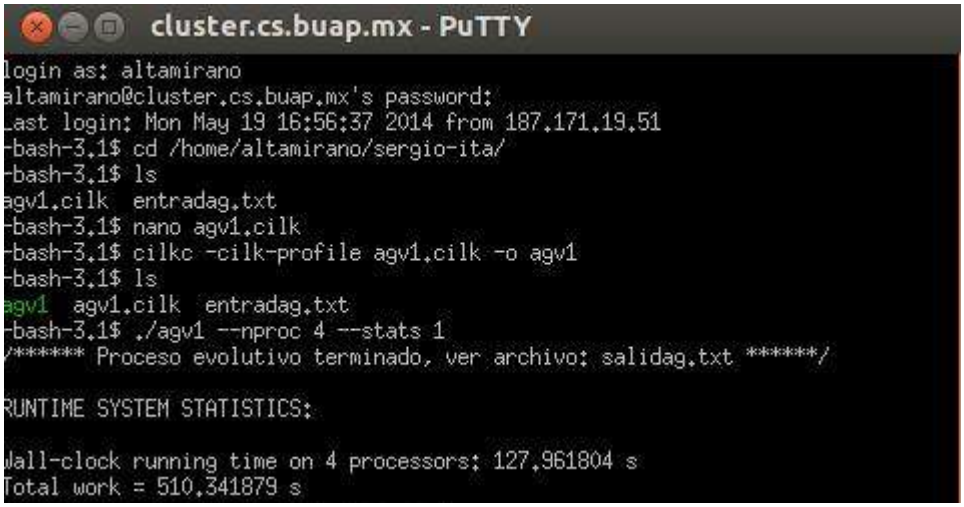


Figura 5.10 Acceso al cluster mediante cliente SSH

5.6 Paralelización

El método de paralelización utilizado fue básico, en términos del algoritmo genético existirá una sola población global que mejora durante el proceso de evolución. Este mecanismo no es el mejor desde el punto de vista del paralelismo, pero la intención es realizar la implementación utilizando el lenguaje Cilk en conjunto con una arquitectura multinúcleo, en este caso el cluster de computadoras descrito anteriormente para demostrar que es viable la combinación del algoritmo genético y el computo paralelo en la solución de problemas de optimización.

Ahora se describe la forma en que se realizó el proceso de paralelización del algoritmo genético, cabe mencionar que se realizaron 2 implementaciones paralelas con respecto a la versión secuencial, en la primera se paralelizaron todas las operaciones a manera de experimentación, posteriormente en la segunda solo se paralelizaron 3 operaciones, siendo esta última la que presento mejor desempeño por lo cual se está tomando como la

implementación principal a la que se hace referencia durante el proceso descrito a continuación.

Con la finalidad de mejorar tanto el desempeño como la calidad de las soluciones encontradas en un espacio de búsqueda grande y de acuerdo al análisis realizado previamente, se optó por paralelizar las operaciones objetivo (evaluación aptitud), selección y elite tal como se muestra en el diagrama de la figura 5.11

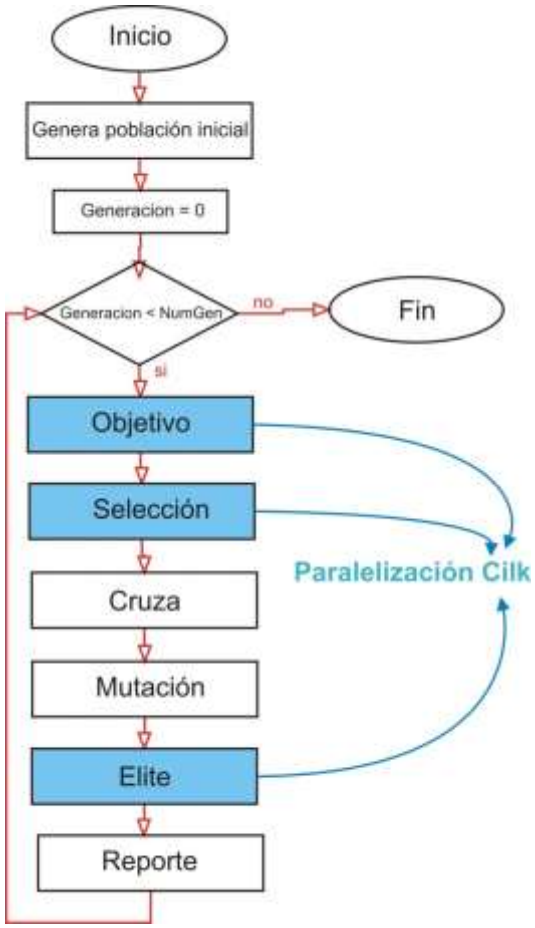


Figura 5.11 Diagrama de flujo de la función principal del algoritmo genético, los bloques en color azul están implementados en Cilk para ejecutarse de forma paralela.

Como puede observarse en la figura 5.11 el flujo de la función principal del algoritmo genético es en forma secuencial y dentro de este flujo se están ejecutando 3 funciones de forma paralela en Cilk, la primera de estas es la función objetivo encargada de evaluar la aptitud de cada individuo de la población, ver listado 5.1.

```
cilk void objetivo(void)
{
int i, j;
double x[NUMVAR+1];

cilk_for (i = 0; i < TAMPOB; i++)
      cilk_for_set = 32
      {
      for (j = 0; j < NUMVAR; j++)
            x[i+1] = pob[i].gen[i];

      pob[i].aptitud = spawn 21.5 + (x[i]*(sin((4*PI)*(x[i]))) +
(x[i]*(sin((20*PI)*(x[i]));
      sync;
      }
}
```

Listado 5.1 Paralelización del bloque de código de la función objetivo en Cilk

Para realizar la paralelización de un bloque de código en específico, se debe agregar la directiva Cilk al inicio de la función y posteriormente colocar el constructor spawn en las líneas que se van a ejecutar de forma paralela o el constructor correspondiente al ciclo for de Cilk , ver figura 5.12, la recomendación para utilizar este constructor es que se coloque donde se realice un número considerable de operaciones, puesto que el trabajo se distribuye en la cantidad de hilos que el planificador de Cilk determina adecuado para realizar el proceso con el mejor desempeño posible, finalmente se utiliza el constructor sync esperar que todos los hilos (creados por spawn) hayan terminado y devuelto los resultados, ver listado 5.1.

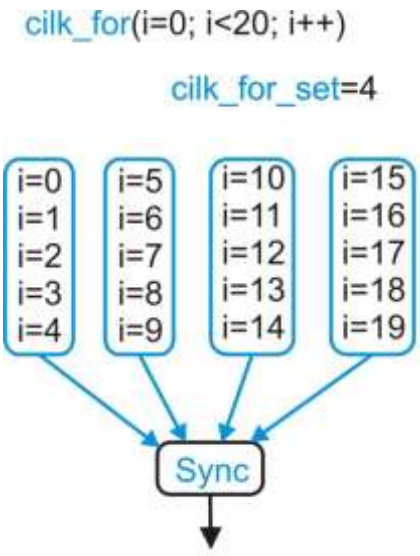


Figura 5.12 Ciclo for en Cilk

Para las funciones de selección y elite el proceso es similar al descrito anteriormente, una vez terminada la implementación en Cilk de cada una de estas funciones se tiene completo el primer nivel de la paralelización puesto que la función principal del algoritmo genético también se implementa de forma paralela ver listado 5.2, la cual completa el segundo nivel de paralelización.

```
cilk int main(void)
{
int i;

if ((salidag = fopen("salidag.txt","w"))==NULL)
      {
      exit(1);
      }
generacion = 0;

fprintf(salidag, "\n Numero      Mejor    Valor \n");
fprintf(salidag, " generacion  aptitud  promedio \n");

while(generacion<NUMGENS)
      {
      generacion++;
    spawn objetivo();
    spawn seleccion();
        /*spawn*/ cruza();
        /*spawn*/ mutacion();
    spawn elite();
        reporte();
      }
fprintf(salidag,"\n\n /***** Proceso evolutivo terminado ******/\n");
fprintf(salidag,"\n Mejor individuo(gen): \n");

for (i = 0; i < NUMVAR; i++)
   {
   fprintf (salidag,"\n var(%d) = %3.3f",i,pob[TAMPOB].gen[i]);
   }
fprintf(salidag,"\n\n Mejor aptitud = %3.3f",pob[TAMPOB].aptitud);
fclose(salidag);
printf("/****** Proceso evolutivo terminado, ver archivo: salidag.txt ******/\n");
sync;
return 0;
}
```

Listado 5.2 Paralelización de la función principal del algoritmo genético en Cilk

Con lo que se obtiene una paralelización anidada en la que la primera parte correspondiente a las operaciones dentro de alguna fase como por ejemplo la evaluación de los individuos en la que se despliega paralelización de grano fino a nivel de instrucciones y la segunda parte correspondiente al proceso evolutivo del algoritmo se despliega paralelización de grano grueso a nivel de funciones como se muestra en la figura 5.13.

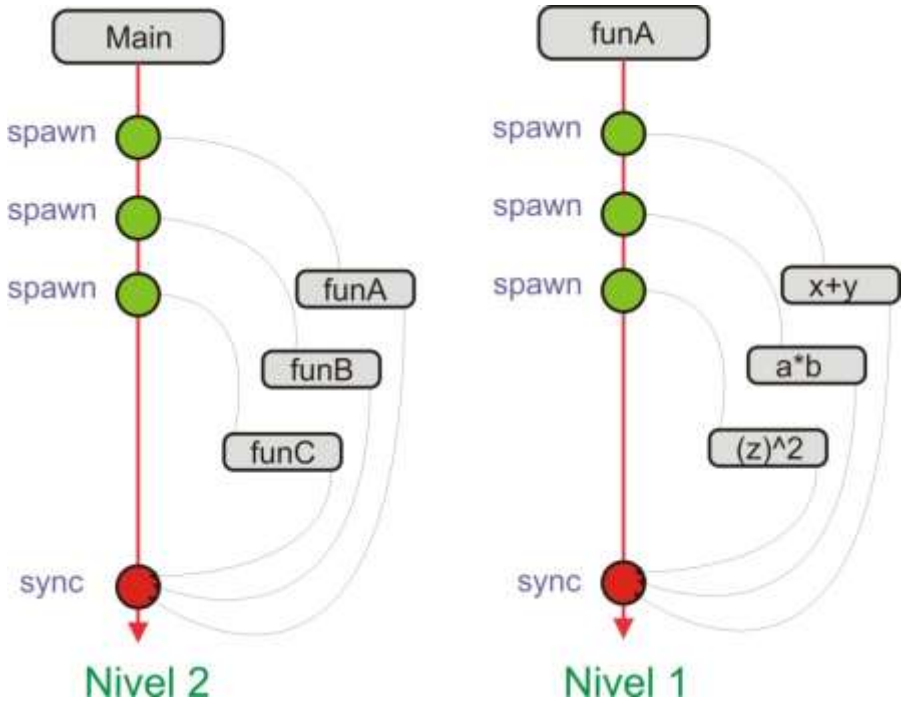


Figura 5.13 Niveles de paralelización del algoritmo genético en Cilk.

De acuerdo al funcionamiento de Cilk el constructor paralelo spawn es el encargado de crear un determinado número de hilos los cuales el algoritmo de planificación de Cilk distribuye en los procesadores que se especifiquen en el momento de ejecutar un programa en Cilk (Blumofe et al., 1995), para complementar esta tarea se considera un conjunto de procesadores capaces de cooperar simultáneamente en la solución de un problema, el número de procesadores está limitado de acuerdo a la arquitectura paralela que en nuestro caso es un cluster con $P = 32$ procesadores que están conectados mediante un costo de comunicación mínimo. Finalmente se realizaron conjuntos de pruebas ejecutando el AG paralelizado en Cilk utilizando los procesadores disponibles en el cluster, en el primer conjunto se ejecutó el AG en cada uno de los procesadores, es decir la primera ejecución especificando un solo procesador la segunda especificando 2 procesadores y así sucesivamente hasta 32 procesadores, obteniendo tiempos muy cercanos entre sí; por lo cual en el segundo conjunto de pruebas se utilizaron 4, 8, 16 y 32 procesadores los cuales se identificaron como los tiempos más significativos durante las ejecuciones del AG, tal como se muestra en los resultados obtenidos.

Capítulo 6: Resultados

6.1 Primera implementación paralela

La primera implementación del algoritmo genético consistió en paralelizar todos los módulos en Cilk, posteriormente se realizaron ejecuciones del algoritmo utilizando un tamaño de población y número de generaciones fijos especificando desde 1 hasta 32 procesadores, los resultados se muestran en la tabla 6.1.

Tabla 6.1: Ejecución de la primera versión paralela del algoritmo genético en la arquitectura 1

	Parámetros		**tiempo (s)**
Tamaño Pob	Generaciones	# Procesadores	agv1.cilk
4096	800	1	60.3
4096	800	2	38.8
4096	800	3	33
4096	800	4	25.4
4096	800	5	25.2
4096	800	6	27.3
4096	800	7	24.8
4096	800	8	25.2
4096	800	9	27
4096	800	10	25.9
4096	800	11	26.8
4096	800	12	24
4096	800	13	23.2
4096	800	14	23.4
4096	800	15	25
4096	800	16	23.1
4096	800	17	25
4096	800	18	23.9
4096	800	19	22.4
4096	800	20	23.5
4096	800	21	22.5
4096	800	22	23.4
4096	800	23	21.4
4096	800	24	23.5
4096	800	25	22
4096	800	26	23.5
4096	800	27	22.8
4096	800	28	22.4
4096	800	29	22.7
4096	800	30	22.5
4096	800	31	22.8
4096	800	32	29.8

Los tiempos obtenidos en la tabla 6.1 se obtuvieron al ejecutar el algoritmo en la arquitectura correspondiente al equipo portátil, el cual cuenta con 4 procesadores físicos, por lo que se demuestra que el compilador de Cilk permite especificar un número mayor de procesadores de los que realmente se encuentran físicamente en el equipo de cómputo que se esté utilizando, en consecuencia la paralelización a partir de 5 procesadores solo es virtual y no la esperada realmente. Luego si se observa el tiempo de ejecución utilizando un solo procesador es muy similar al que se obtiene al ejecutar el algoritmo en forma secuencial con los mismos parámetros, es decir únicamente en lenguaje C, el cual registró un tiempo de 65.7 segundos, con esto se comprueba que los tiempos obtenidos son congruentes: el tiempo secuencial y el tiempo paralelizado utilizando un solo procesador, como se muestra en la figura 6.1. También, se puede observar una reducción de tiempo en el segundo y tercer procesador, a partir del cuarto procesador los tiempos se encuentran dentro del rango de 20 a 30 segundos lo que demuestra que Cilk mantiene un nivel constante a partir del último procesador físico que encuentra.

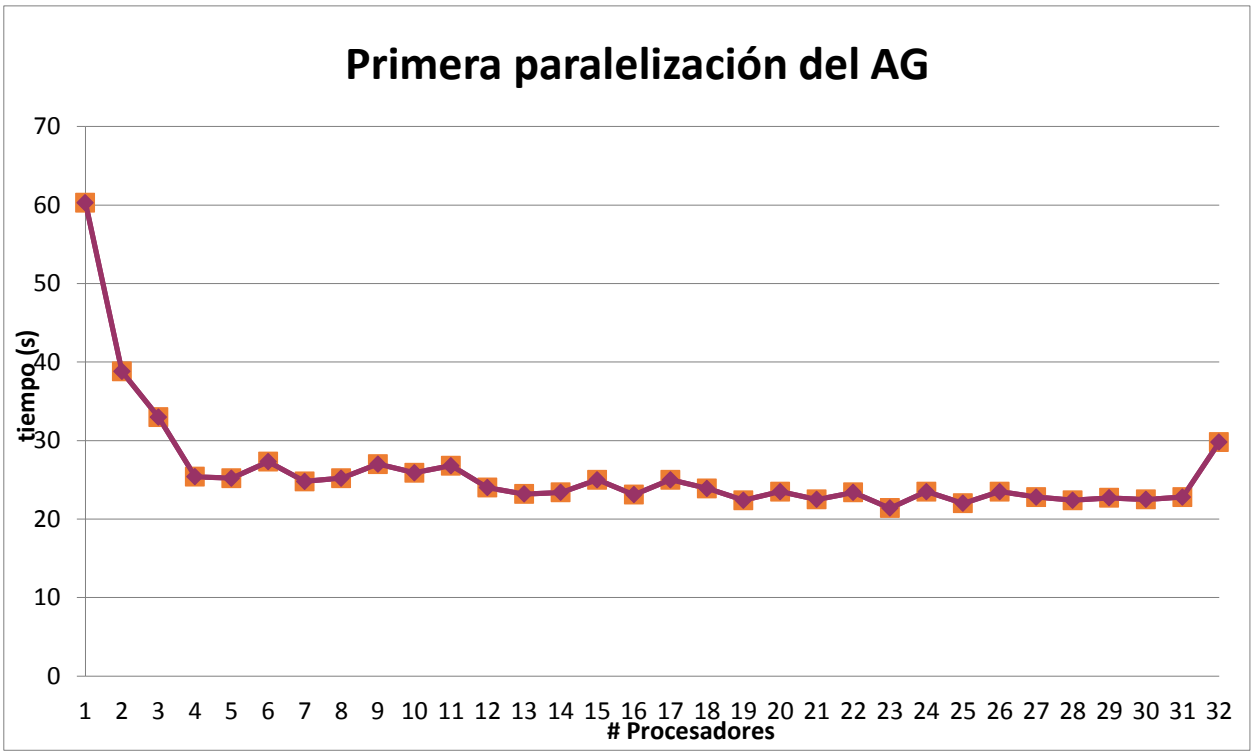


Figura 6.1: Gráfica de los tiempos obtenidos en la primera versión paralela del algoritmo genético en la arquitectura 1

6.2 Conjunto de pruebas

Continuando con la versión del algoritmo genético donde se paralelizaron todos los módulos en Cilk, se procedió a ejecutarlo en las dos arquitecturas de hardware multinúcleo, especificando tamaños de población y número de generaciones diferentes, así como un determinado número de procesadores tal como se muestra en las tablas 6.1, 6.2, 6.3, 6.4 y 6.5

Tabla 6.2: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 64 individuos y 200 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
64	200	4	344.91 ms	3.02	26.09 ms	57.74
64	200	8	436.15 ms	5.67	38.92 ms	74.13
64	200	16	1.009 s	4.15	32.71 ms	90.86
64	200	17	746.39 ms	2.31	29,28 ms	38.17
64	200	18	943.41 ms	7.84	30.06 ms	27.05
64	200	19	251.05 ms	1.84	33.02 ms	33.27
64	200	20	499 ms	4.86	35.19 ms	26.51
64	200	32	270.77 ms	23.27	46.95 ms	37.58

Tabla 6.3: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 128 individuos y 300 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
128	300	4	279.59 ms	9.23	57.60 ms	68.45
128	300	8	320.01 ms	8.33	90.37 ms	42.03
128	300	16	1.75 s	13.89	74.61 ms	80.1
128	300	17	831.33 ms	6.97	78.44 ms	107.5
128	300	18	1.12 s	2.41	80.51 ms	36.57
128	300	19	286.98 ms	9.34	83.65 ms	92.87
128	300	20	1.53 s	6.35	77.31 ms	100.13
128	300	32	2.15 s	2.22	77.84 ms	90.4

Tabla 6.4: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 256 individuos y 400 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
256	400	4	969.42 ms	5.47	119.24 ms	108.9
256	400	8	1.16 s	2.13	210.46 ms	110.82
256	400	16	1.58 s	4.89	204.49 ms	170.69
256	400	17	765.96 ms	7.05	200 99 ms	161.78
256	400	18	1.51 s	4.64	205.72 ms	190.99
256	400	19	1.51 s	4.7	202.88 ms	177.75
256	400	20	1.00 s	4.9	202.85 ms	168.21
256	400	32	6.24 s	1.78	209.97 ms	220.61

Tabla 6.5: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 512 individuos y 500 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
512	500	4	3.86 s	3.99	324.46 ms	149.71
512	500	8	2.36 s	9.38	404.89 ms	151.16
512	500	16	2.34 s	33.09	494.46 ms	345.33
512	500	17	2.21 s	32.11	494,84 ms	359.84
512	500	18	3.06 s	9.71	497.01 ms	306.92
512	500	19	2.75 s	24.59	493.54 ms	333.56
512	500	20	2.83 s	12.35	488.64 ms	324.37
512	500	32	2.63 s	23.86	487.09 ms	280.93

Tabla 6.6: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 1024 individuos y 600 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
1024	600	4	7.65 s	32.18	1.15 s	204.14
1024	600	8	6.86 s	8.62	1.04 s	207.97
1024	600	16	7.07 s	13.22	1.10 s	247.12
1024	600	17	7.02 s	13.95	1.09 s	338.67
1024	600	18	9.72 s	11.34	1.01 s	259.93
1024	600	19	8.78 s	8.47	1.10 s	372.67
1024	600	20	12.65 s	15.32	1.13 s	235.02
1024	600	32	7.40 s	22.71	1.08 s	406.06

Tabla 6.7: Comparación de los tiempos de la primera versión paralela del algoritmo entre ambas arquitecturas para 2048 individuos y700 generaciones.

			Arquitectura 1		**Arquitectura 2**	
# Indv (Pob)	# Gen	# Procesadores	Tiempo	Paralelismo	Tiempo	Paralelismo
2048	700	4	29.77 s	90.23	5.40 s	303.95
2048	700	8	30.11 s	29.54	4.55 s	361.53
2048	700	16	28.59 s	19.22	4.52 s	442.94
2048	700	17	29.90 s	22.31	4.48 s	431.72
2048	700	18	29.13 s	42.98	4.51 s	423.68
2048	700	19	29.40 s	34.68	4.47 s	387.25
2048	700	20	29.83 s	30.04	4.51 s	408.69
2048	700	32	30.22s	45.92	4.40 s	532.94

Como se puede observar en las tablas anteriores los tiempos de ejecución en la arquitectura 2 correspondiente al cluster (32 procesadores) son menores, también se puede observar que se incluye el paralelismo obtenido, esta es una métrica que proporciona el compilador de Cilk. Durante este conjunto de pruebas el comportamiento del tiempo en el cluster es muy similar en cada uno de los procesadores por lo que se decide modificar en primer lugar la cantidad de módulos paralelizados y en segundo lugar los tamaños de población y número de generaciones, esto nos lleva a una segunda versión del algoritmo cuyos resultados se presentan en la siguiente sección.

5.3 Implementación principal

Finalmente se realizaron pruebas con diferentes combinaciones de número máximo de generaciones y tamaños de población utilizando únicamente la arquitectura del cluster y la segunda versión del algoritmo genético en la que solo se paralelizan los módulos: objetivo, selección y elite; los resultados se muestran en las tablas 6.1 y 6.2

		tiempos (seg)				
		AG secuencial	AG Paralelo			
			# Procesadores			
Tamaño Población	Generación		4	8	16	32
32	100	0.01	0.01	0.01	0.08	0.03
64	200	0.04	0.04	0.05	0.18	0.07
128	300	0.19	0.27	0.18	0.12	0.21
256	400	0.21	1	0.63	0.75	0.64
512	500	0.64	1	0.6	1.3	1
1024	600	2.72	3.3	2.2	2.5	2.6
2048	700	12.72	9.3	7.4	7	7.6
4096	800	58.35	30.3	26.1	25.2	27.3
8192	900	259.6	110.8	92.3	89.3	95.1
16384	1000	1129	472.6	398.6	381.8	459.1

Tabla 6.8: Comparación de los tiempos computacionales entre la versión secuencial y la versión paralela (solo con los operadores: objetivo selección y elite) implementada en Cilk del algoritmo genético, se puede observar que la reducción significativa de tiempo es a partir de una población de 2048 individuos.

			tiempo (s)			
T. Población	Generaciones	# Procesadores	agv1.cilk	Aceleración	agv2.cilk	Aceleración
4096	800	4	42.8	3.1	28.4	5.34
4096	800	8	36	4.21	22.7	6.6
4096	800	16	33.1	4.58	29	5.2
4096	800	32	35	4.33	37.5	4

Tabla 6.9: Comparación de los tiempos computacionales del algoritmo genético en las versiones paralelas, la versión uno (agv1) presenta paralelización en todos los módulos y la versión dos (agv2) únicamente se paralelizan los módulos: objetivo, selección y elite

De acuerdo a los resultados de las tablas 6.1 y 6.2 la segunda versión del algoritmo genético paralelizado en Cilk (agv2.cilk) muestra mejor rendimiento, esto debido a que solo se están ejecutando de forma paralela los bloques de código que realizan las operaciones con mayor demanda de cómputo, con lo cual se comprueban dos cosas: la primera es que el hecho de paralelizar todos los bloques de código no garantiza mejor rendimiento y la segunda es que usar mayor número de procesadores puede resultar contraproducente puesto que la comunicación y flujo de datos entre los procesadores genera sobrecarga y por consiguiente el tiempo de proceso tiende a aumentar como se puede observar en las gráficas de las figuras: 6.2 y 6.3 al utilizar 32 procesadores el tiempo de procesamiento aumenta nuevamente.

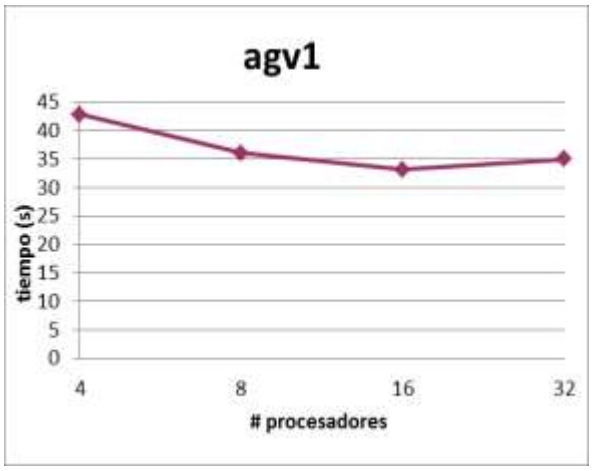


Figura 6.2:Desempeño del algoritmo genético con paralelización de todos los módulos

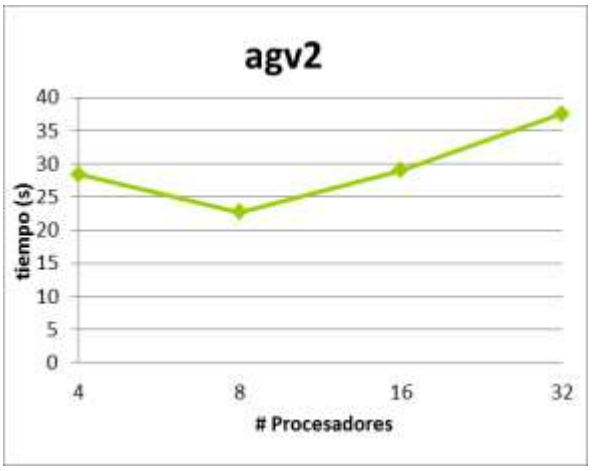


Figura 6.3:Desempeño del algoritmo genético con paralelización únicamente en los módulos: objetivo selección y elite

Los resultados indican que la implementación en Cilk propuesta en esta tesis produce reducciones significativas en el tiempo necesario para realizar la búsqueda y optimización durante el proceso evolutivo del algoritmo genético, con respecto a la ejecución secuencial. Para la configuración, en la que se cuenta con 4096 individuos y 800 generaciones, se obtuvo una aceleración de más de 4 veces con respecto a la versión secuencial, ver tabla 6.2 de tal forma que en la versión implementada en forma secuencial el tiempo necesario para finalizar la ejecución del algoritmo es de 151.7 segundos, mientras que en la versión paralela fue de 22.7 segundos utilizando 8 procesadores y paralelizando solo tres módulos en Cilk: objetivo selección y elite.

Capítulo 7: Conclusiones y Trabajos Futuros

7.1 Conclusiones

Los algoritmos evolutivos son técnicas metaheurísticas de búsqueda y optimización que se utilizan cuando otras alternativas no son viables. Éste tipo de técnicas resultan particularmente útiles en problemas de optimización donde los algoritmos de programación matemática presentan diversas limitantes. Por ejemplo, cuando las funciones objetivo no son diferenciables. Dentro de los algoritmos evolutivos, el algoritmo genético ha mostrado ser una técnica eficaz en espacios de búsqueda continuos. Por esta razón se eligió este algoritmo evolutivo como motor de búsqueda para el trabajo aquí reportado, debido a la simplicidad en su implementación ya que ha sido utilizado en trabajos anteriores relacionados con la implementación de esta tesis, en la cual se abordó la implementación de un algoritmo genético sobre arquitecturas de hardware multinúcleo, para resolver problemas de optimización con maximización de 1 a N variables. La complejidad de este tipo de problemas es directamente proporcional al número de individuos dentro de la población, número de dimensiones y tamaño del espacio de búsqueda que el mismo problema a optimizar requiera. Sin embargo, el impacto que puede tener el aplicar una técnica de cómputo evolutivo a este tipo de problemas es muy importante para diversas áreas como el diseño (Hongying y Jingsong 2010), finanzas (Brabazon et al., 2008), robótica y control (Meng y Song 2007), bioinformática (Foget et al., 2007), procesamiento de imágenes (Zhang et al., 2010), reconocimiento de patrones (Espejo et al., 2010), sistemas de potencia (Rudolf y Bayrleithner 1999) y control (Abido y Abdel-Magid 2000), entre otras. Debido a que el tiempo computacional requerido para obtener un modelo aceptable es muy alto y crece conforme se incrementa la dimensión del problema, en esta tesis se propuso el uso de cómputo evolutivo basado en arquitecturas de hardware multinúcleo para resolver esta desventaja.

Los resultados presentados en esta tesis indican que es posible obtener reducciones significativas en el tiempo requerido para resolver este problema, se lograron aceleraciones de hasta 6 veces el tiempo de cómputo usando el lenguaje Cilk y hardware multinúcleo ver tabla 6.2, con respecto a la versión secuencial.

7.2 Contribuciones

Las principales contribuciones de esta tesis son:

- Se mostró que el uso de hardware multinúcleo es una forma eficiente de realizar cómputo paralelo.
- Se demuestra que los algoritmos genéticos pueden implementarse eficientemente en arquitecturas multinúcleo.
- Este parece ser el primer algoritmo genético implementado en el lenguaje de programación paralela Cilk para resolver problemas de optimización.
- Se obtuvo una importante reducción en el tiempo de cómputo necesario para la solución de problemas de optimización.

7.3 Trabajos futuros

El trabajo se puede continuar migrando la implementación al lenguaje CUDA y hacer uso de tarjetas gráficas NVIDIA.

Desarrollar una interfaz gráfica para introducir parámetros y mostrar resultados, así como estadísticas del algoritmo genético.

En cuanto a la estructura del algoritmo se pueden realizar las siguientes mejoras:

- Determinar las variaciones en los resultados, modificando los parámetros del algoritmo.
- Modificar el procedimiento para poder resolver problemas de minimización.
- Paralelizar otros operadores genéticos, como cruza multipunto y cruza uniforme, o cruza multi-padres.
- Paralelizar otros métodos de selección métodos de selección (rango o torneo)
- Paralelizar otras funciones objetivo funciones objetivo. (funciones De Jong)

Referencias bibliográficas

1. Rosete, A. (2000). Una solución flexible y eficiente para el trazado de grafos basada en el escalador de colinas estocástico, CEIS, La Habana.
2. Vose, M. (1999). The Simple Genetic Algorithm: Foundations and Theory. MIT Press. 251 Knowledge-Based Intelligent Information Engineering Systems.
3. Cordon, O. y Herrera F. (1999). Solving Electrical Distribution Problem Using Hybrid Evolutionary Data Analysis.
4. Chow, K. y Rad A. (2002). On-line fuzzy identification using genetic algorithms. Fuzzy Sets and Systems, Elseriver(132): 147-171.
5. Carmona, J. y Castro, Liera. (2009). Adaptative modeling of the near-shore behavior on the Baja California Sur region." Memorias del CIPITECH 09 ISBN 978-1-4276-4108-3.
6. Herrera, F. y Martinez, J. (2003). Aplicación de las Técnicas de la Inteligencia Artificial en un Proceso Biotecnológico de Reproducción Celular (Embriogénesis Somática).
7. Martínez, S. (2010) Fingerprint verification with non-linear composite correlation filters. LNCS: Advances in Pattern Recognition, Vol. 6256, ISBN 978-3-642-15991-6.
8. Mussi, L., Daolio, F., y Cagnoni, S. (2009). GPU-based Road Sign Detection Using Particle Swarm Optimization. 9th International Conference on Intelligent Systems Design and Applications.
9. Hernández, M. y Herrera F. (2005). Use of an ANFIS Network for Relative Humidity Behaviour Modelling on the South Region of Jalisco, México. Journal of Research in Computing Science, IPN 17.
10. Karimi, K., Dickson, N. y Hamze, F. (2010). A Performance Comparison of CUDA and OpenCL. CoRR, Vol. abs/1005.2581.
11. Cg Toolkit, <http://developer.nvidia.com/cg-toolkit>
12. OpenGL: Open Graphics Library <http://www.opengl.org>
13. Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. IEEE Transactions Computer.
14. Mattson, T., Sanders, B. y Massingill, B. (2004). Patterns for Parallel Programming. Addison-Wesley Professional, 1 edition.
15. Chapman, B., Jost, G. y Van der Pas R. (2007). Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press.
16. Culler, D., Singh, J. y Gupta, A. (1999). Parallel computer architecture: a hardware/software approach. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers.
17. http://www.nvidia.es/object/cuda-parallel-computing-es.html

18. Nowostaki, M. y Poli R. (1999). Parallel Genetic Algorithm Taxonomy. Third International Conference of Knowledge-Based Intelligent Information Engineering Systems.
19. Guoyan, Y., Zhen, H., Chaoan, L. y Bing, L. (2006). The Application of Interactive Evolutionary Algorithm in Product Design. The Sixth World Congress on Intelligent Control and Automation (WCICA), Vol. 2, pp 6758–6762.
20. Hongying, Y. y Jingsong, H. (2010). Evolutionary design of operational amplifier using variable-length diferential evolution algorithm. International Conference on Computer Application and System Modeling (ICCASM), Vol. 4, pp 610–614.
21. Lee, V., Ting, Y. y Kwok, T. (2007). Evolutionary Finance Simulation of Long Term Equity Portfolio Management System. International Conference on Service Systems and Service Management, pp 1–6.
22. Brabazon, A., Neill, M. y Dempsey, I. (2008). An Introduction to Evolutionary Computation in Finance. Computational Intelligence Magazine IEEE, Vol. 3, pp 42 – 55.
23. McGregor, D., Odetayo, M., y Dasgupta, D. (1992). Adaptive control of a dynamic system using genetic-based methods. Proceedings of the 1992 IEEE International Symposium on Intelligent Control, pp 521–525.
24. Meng, X. y Song, B. (2007). Fast genetic algorithms used for pid parameter optimization. IEEE International Conference on Automation and Logistics, pp 2144–2148.
25. Pal, S., Bandyopadhyay, S. y Ray. S. (2006). Evolutionary computation in bioinformatics: a review. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews Vol 36, pp 601–615.
26. Fogel, G., Corne, D. y Pan, Y. (2007). Evolutionary Feature Selection for Bioinformatics. Computational Intelligence in Bioinformatics, pp 117–139.
27. Aravind, I., Chandra, C., Guruprasad, M., y Dev, P. (2002). Implementation of image segmentation and reconstruction using genetic algorithms. IEEE International Conference on Industrial Technology, IEEE ICIT '02, Vol. 2, pp 970–975.
28. Zhang, N., Shan Chen, Y., y Li Wang, J., (2010). Image parallel processing based on gpu. 2nd international Conference on Advanced Computer Control, ICACC, Vol. 3, pp 367 – 370.
29. Pei-Fang, G., Bhattacharya, P. y Kharma, N. (2009). An efficient image pattern recognition system using an evolutionary search strategy. IEEE International Conference on Systems, Man and Cybernetics (SMC), pp 599–604.
30. Espejo, P., Ventura, S. y Herrera, F. (2010). A Survey on the Application of Genetic Programming to Classification. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, Vol.4, pp 121–144.
31. Wong, K. y Wong, Y. (1995). Thermal generator scheduling using hybrid genetic/simulated-annealing approach. IEEE Proceedings-Generation, Transmission and Distribution, Vol. 142, pp 372–380.

32. Rudolf A. y Bayrleithner R. (1999). A genetic algorithm for solving the unit commitment problem of a hydro-thermal power system. IEEE Transactions on Power Systems, Vol. 14, pp 1460–1468.
33. Abido, M. y Abdel-Magid, Y. (2000). Robust design of multimachine power system stabilisers using tabu search algorithm. IEEE Proceedings Generation Transmission Distribution, Vol. 147, pp 387–394.
34. Amdahl, G. (1967). Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings, pp. 483-485.
35. Sanders, J. Y Kandrot, E. Cuda by Example. An Introduction to General-Purpose GPU Programming. 1 Ed. Ann Harbor, Michigan, Pearson, 290 páginas.
36. Barney, B. (1998). Tutorial: OpenMP. Lawrence Livermore National Laboratory.
37. Intel. (2012). Intel threading building blocks for open source documentation.
38. AMD. (2011). Opencl and the amd sdk
39. Lee, R. (1980). Empirical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations. Proceedings of the 1980 International Conference on Parallel Processing, IEEE Computer Society.
40. Fung, E., Wong, W.W., Suen, J.K., Bulter, T., Lee, S.G., and Liao, J.C. (2005) A synthetic gene-metabolic oscillator. Nature, 435, pp. 118-122.
41. Akenine-Möller, T. y Assarsson, U. (2002). Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. 13th Eurographics Workshop on Rendering, pp. 309-318.
42. Wloka, M. y Huddy, R. (2003). Optimization for DirectX 9 Graphics
43. Blumofe, D. R., Joerg, C., Kuszmaul, B., Leiserson, C., Randal, H. y Zhou Y. (1995). Cilk: An Efficient Multithreaded Runtime System, MIT Laboratory for Computer Science, Cambridge.

Producción académica

1 Artículos publicados

Principios de programación paralela sobre arquitecturas GPUs con tecnología CUDA en aplicaciones de propósito general GPGPU

S. Palafox-Ugarte, J. F. Ramírez-Cruz, E. Carreón-Esteban y B.E. Pedroza-Méndez

***Resumen*-- CUDA, es una tecnología de programación paralela de aparición relativamente reciente (2006), permite escribir códigos paralelos de propósito general (GPGPU) y ejecutarlos en unidades de procesamiento gráfico (GPUs) conocidas como tarjetas gráficas, relativamente económicas que prácticamente están presentes, frecuentemente en computadoras personales (PCs) convencionales. Así, es posible aprovechar el gran poder computacional de las tarjetas gráficas, mediante la paralelización de algoritmos secuenciales, de una manera simple, sobre todo para los programadores familiarizados con el lenguaje C. En este artículo se exponen los principios básicos para utilizar la tecnología CUDA en la programación paralela de dispositivos GPUs de NVIDIA, y se dan recomendaciones prácticas, tanto para garantizar el máximo aprovechamiento del GPU como para justificar el empleo del mismo, así como la configuración de un sistema de cómputo de alto rendimiento al combinar CPU (Host) + GPU (Device) para la ejecución de códigos paralelos en aplicaciones de propósito general.**

***Índice de Términos*— Arquitecturas GPUs, CUDA, Programación paralela.**

I. Introducción

Recientemente las capacidades de las Unidades de Procesamiento Gráfico (del inglés Graphics Processing Units, abreviado GPUs), comúnmente llamados chips o tarjetas de gráficos han dejado de crecer linealmente para crecer exponencialmente, de manera específica las tarjetas gráficas a las cuales tiene acceso el consumidor común. Este tipo de tarjetas están avanzando mucho más rápido que la ley de Moore, la cual estipula que aproximadamente cada dos años se duplica el número de transistores en un circuito integrado y en consecuencia el poder de procesamiento del hardware, característica que se ha incrementado más de 10 veces de este periodo en las GPUs.

En los últimos años las unidades de procesamiento gráfico (GPU), que eran originalmente un dispositivo para aplicaciones gráficas, han atraído mucho interés en el campo de la investigación desde el punto de vista de computación de alto rendimiento de bajo costo. Las GPUs se han convertido en dispositivos altamente paralelos con una gran cantidad de núcleos de procesamiento y gran poder de cómputo. En comparación con los sistemas convencionales multiprocesador (workstations, clusters, grids y supercomputadoras) En la actualidad las GPUs son un medio alternativo mucho más accesible para la obtención de un gran poder computacional, brindando el mismo poder de cómputo que los sistemas multiprocesador más recientes, a una décima parte del precio y consumiendo una vigésima parte de la energía. Sin embargo, el rendimiento de las GPU rápidamente ha ido mejorando en los últimos años de tal manera que su máximo rendimiento es mucho mayor que la de las CPUs. El concepto de usar la GPU no sólo para aplicaciones gráficas, sino también para aplicaciones de propósito general se llama computación de propósito general en GPU (GPGPU) [1].

La idea consiste en utilizar la alta concurrencia (ejecutar varias tareas de forma simultanea) de las tarjetas gráficas para aplicaciones de cómputo general como pueden ser simulaciones de tráfico, simulaciones físicas, algoritmos de búsqueda, cálculos con vectores o matrices de grandes tamaños, comparación de secuencias, procesamiento de imágenes médicas y exploración del subsuelo por mencionar algunos propósitos; desde este punto de vista, tanto los fabricantes como los desarrolladores, han considerado esta nueva aplicación de la computación paralela como una prometedora área de investigación, sobre todo, por la amplia variedad de posibles aplicaciones que se pueden aprovechar en el paralelismo disponible en los actuales dispositivos GPUs.

II. Arquitecturas

A. GPU NVIDIA

La tendencia actual, observada tanto en el desarrollo como en la aplicación de los dispositivos GPU ofrecidos por el fabricante NVIDIA, permite acompañar y contribuir a la consolidación del nuevo modelo de programación paralela soportado por la herramienta de programación CUDA (del inglés Compute Unified Device Architecture), donde la GPU, además de ofrecer una mayor capacidad de cálculo paralelo, tiene un papel importante como administrador de múltiples hilos. Esta GPU proporciona una arquitectura unificada, tanto para gráficos como para cálculos, que es fundamentalmente un arreglo escalable de multiprocesadores (MP) multi-hilo, donde cada MP consiste en 'n' unidades de procesamiento gráfico a los cuales se conoce como núcleos (CUDA Cores) cuya

actividad primordial es la manipulación de pixeles, porque las GPUs normalmente procesan un conjunto complejo de polígonos, renderizado de escenas, donde aplica texturas a los polígonos y luego se realiza el sombreado, así como los cálculos de iluminación; debido a esta función es común referirse a los núcleos de un MP como sombreador de pixel (del inglés pixel shader, abreviado PS). Uno de los pasos importantes fue el desarrollo de PS programables, estos fueron efectivamente pequeños programas que la GPU procesó para calcular diferentes efectos. Básicamente el modelo de hardware en las arquitecturas de las GPUs (o tarjetas de video) contienen 'N' MP, y cada uno de estos 'M' procesadores (núcleos) junto con memoria compartida (muy rápida y pequeña), caches de constantes y de texturas (sólo lectura) y finalmente una memoria global, todo dentro del circuito integrado como se muestra en la Fig. 1. La cantidad de MP es directamente proporcional a la capacidad de cómputo del GPU. Cada MP se encarga de la creación, manejo y ejecución de los hilos que están físicamente activos en el dispositivo GPU, teóricamente hasta 512 con el esquema denominado por NVIDIA como "una instrucción y múltiples hilos" (del inglés Single Instruction, Multiple Thread, abreviado SIMT). Debido a que un multiprocesador asigna cada uno de sus hilos a un núcleo y que cada hilo es ejecutado independientemente de los demás, con su propia dirección de instrucción y sus propios registros de estado, las herramientas de programación de NVIDIA ofrecen algunas funciones que son enfocadas precisamente al manejo y optimización de múltiples hilos, como la posibilidad de sincronizar el proceso que realizan todos los hilos en un bloque de código específico.

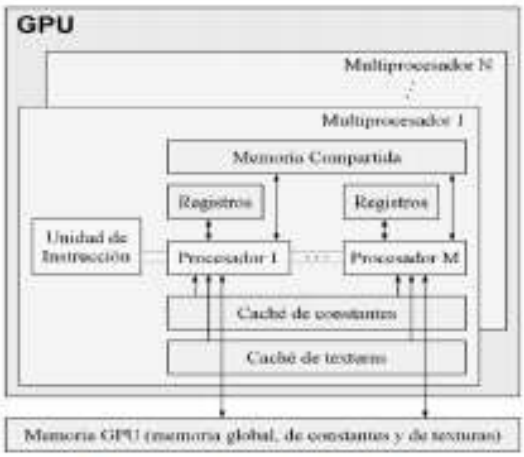


Fig. 1. Arquitectura GPU NVIDIA.

B. CUDA

La arquitectura CUDA, surgió en 2006, cuando NVIDIA lanzó sus tarjetas GeForce 8800 GTX [2], las que por primera vez incluyeron elementos dirigidos específicamente a posibilitar la solución de problemas de propósito general. Poco después, NVIDIA lanzó el compilador CUDA C, el primer lenguaje de programación para desarrollar aplicaciones de propósito general sobre una GPU; con la finalidad de captar la mayor cantidad de programadores que pudieran adoptar esta arquitectura. Esta herramienta promete a la amplia comunidad de desarrolladores lograr aceleraciones importantes durante procesamiento de datos mediante el uso de la tecnología CUDA, la cual se puede ejecutar en tarjetas gráficas NVIDIA para los principales sistemas operativos Linux y Windows. El paralelismo natural de la computación en la GPU se puede aprovechar en técnicas de optimización como lo son los algoritmos genéticos [3]. CUDA C es un lenguaje muy similar a C, al cual se le agregaron un conjunto de instrucciones que harían posible la programación en paralelo en un sólo equipo de cómputo. Se ha evolucionado al grado de tener que "engañar" al GPU utilizando para otros propósitos sus instrucciones para el manejo de pixeles, hasta contar con las interfaces de programación específicas que existen actualmente, por ejemplo CUDA, la cual está diseñada para soportar el esquema SIMT, de tal manera que múltiples hilos pueden ser ejecutados sobre muchos datos.

La herramienta CUDA permite que los programadores escriban el código paralelo, usando lenguaje C estándar más algunas extensiones de NVIDIA, permitiendo organizar el paralelismo en un sistema jerárquico de tres niveles: malla, bloque, e hilo. El proceso comienza cuando el procesador anfitrión (CPU Host) invoca una función para el dispositivo (GPU Device), llamada kernel, enseguida se crea una malla (o arreglo) con bloques de múltiples hilos, para distribuirla en algún MP disponible. Con CUDA, dentro del programa se arranca la ejecución de los kernels paralelos mediante la siguiente sintaxis extendida de llamada a función:

kernel <<<dimGrid, dimBlock>>> (...parameters...);

Donde dimGrid y dimBlock son parámetros especializados que especifican, respectivamente, la dimensión (en bloques) de la malla de procesamiento paralelo y la dimensión (en hilos) de cada uno de los bloques. Prácticamente un Kernel en CUDA es una función en C que se ejecutará 'N' veces en paralelo por 'n' hilos.

El modelo de programación CUDA permite a los programadores lanzar bloques de software para ejecutarse en paralelo especificando el número de hilos por bloque que deben ser generados por los multiprocesadores de la GPU [4].

III. Modelos de Memoria

Durante la ejecución del kernel, los hilos tienen acceso a seis tipos de memoria dentro del dispositivo GPU, de acuerdo a los siguientes niveles de acceso predefinidos por NVIDIA:

- Memoria global. Es una memoria de lectura/escritura y se localiza en la tarjeta del GPU.
- Memoria para constantes. Es una memoria rápida (cache) de lectura y se localiza en la tarjeta del GPU.
- Memoria para texturas. Es una memoria rápida (caché) de lectura y se localiza en la tarjeta del GPU.

- Memoria local. Es una memoria de lectura/escritura para los hilos y se localiza en la tarjeta del GPU.
- Memoria compartida. Es una memoria de lectura/escritura para los bloques y se localiza dentro del circuito integrado del GPU.
- Memoria de registros. Es la memoria más rápida, de lectura/escritura para los hilos y se localiza dentro del circuito integrado del GPU.

Es importante especificar la ubicación de los diferentes tipos de memoria en la arquitectura del GPU para comprender el flujo y acceso de datos entre (CPU Host) y (GPU Device). La memoria compartida es la más rápida, pero su tamaño está limitado porque está ubicada dentro del circuito integrado [5], la memoria global es la encargada de recibir y retornar los datos a procesar en la GPU. Los threads son capaces de acceder a los espacios de memoria durante su periodo de vida. En la Fig. 2 se describe el acceso a la memoria local por parte de cada thread.

Los bloques de threads tienen un espacio de memoria compartida a la cual pueden acceder todos los threads dentro del bloque. Así mismo, todos los threads tienen acceso a la memoria global de la GPU durante la ejecución de un kernel. Estos espacios de memoria tienen un periodo de vida igual al tiempo de ejecución del kernel que ha sido invocado por los threads.

Tanto la memoria constante como la memoria compartida son recursos escasos, 16 KB y 64 KB, respectivamente, por lo que su uso debe ser limitado de acuerdo a las necesidades particulares de cada aplicación.

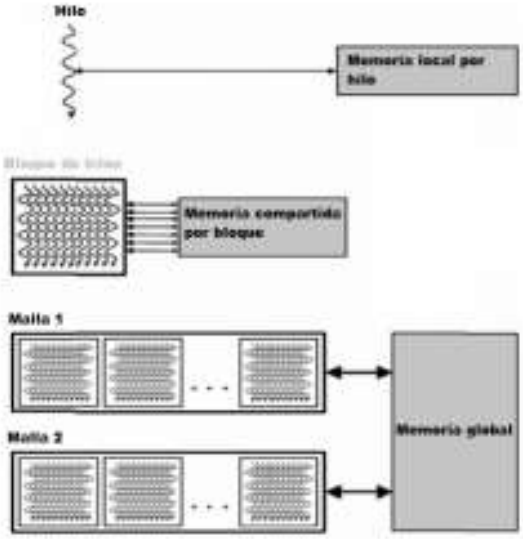


Fig. 2. Niveles de memoria en GPUs NVIDIA.

En la práctica es muy importante hacer énfasis en que, debido a los tiempos relativamente grandes, de retardo (latencia) y al bajo ancho de banda en las trasferencias de memoria, entre la computadora anfitrión [6] (CPU Host) y el dispositivo (GPU Device), es altamente recomendable dividir a la aplicación, de tal manera que cada parte del sistema (hardware) haga únicamente el trabajo que mejor realiza. El uso de la GPU se recomienda si:

- La complejidad de las operaciones justifica el costo de mover datos, hacia el dispositivo GPU. El ejemplo ideal es aquel en el que muchos hilos ejecutan una cantidad considerable de trabajo, ya que las transferencias deben ser minimizadas, los datos deberían mantenerse en le GPU tanto como sea posible.
- La aplicación tiene numerosos datos que pueden ser calculados simultáneamente en paralelo. Esto frecuentemente involucra operaciones aritméticas sobre un gran conjunto de datos, donde la misma operación puede ser realizada sobre miles de elementos al mismo tiempo.
- La aplicación puede ser dividida en operaciones simples, que pueden ser asignadas a gran número de hilos ejecutándose en paralelo.

Con estas consideraciones en mente, lo primero que se tiene que hacer es determinar cuál es la parte del código secuencial que se puede paralelizar mejor usando el GPU. Generalmente se elige como posibles candidatos a todos los segmentos de código que son especialmente demandantes de recursos computacionales (tiempo de procesamiento y memoria). Finalmente, se otorga al GPU aquellos segmentos de código que cumplen con las recomendaciones prácticas antes mencionadas Fig. 3.

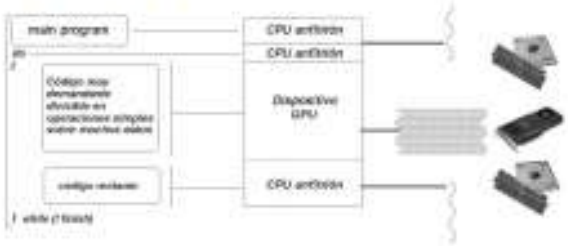


Fig. 3. Estrategia de paralelización.

IV. Codificación

Generalmente, una aplicación codificada en CUDA debe incluir los siguientes pasos:

1. El CPU Host llama al cuerpo principal del programa (main()).
2. Se reserva memoria dentro del dispositivo GPU.
3. Se copian los datos del CPU al dispositivo GPU.
4. El CPU llama a la función kernel.
5. El dispositivo GPU ejecuta el código paralelamente.
6. Se copian los resultados nuevamente a la memoria

del CPU principal.

7. Se libera la memoria reservada dentro del dispositivo GPU.

V. Metodología

A. Selección de Características del GPU Device

Las principales características a considerar para seleccionar la GPU son: en primer lugar el uso que se le dará a la tarjeta gráfica; en este caso será para realizar programación paralela de propósito general, concepto utilizado frecuentemente hoy en día en las comunidades científicas por estudiantes e investigadores, pues han encontrado una excelente opción para ejecutar aplicaciones fuera de los gráficos por computadora, desarrollando técnicas para la implementación de simulaciones, bases de datos y algoritmos que requieren computación de alto rendimiento.

En segundo lugar existe una de las características que ha logrado aumentar exponencialmente el uso de GPUs como recurso de cómputo de alto rendimiento y es el bajo precio en relación a su potencia de cálculo, gran paralelismo, optimización para cálculos en coma flotante y sumando el hecho de trabajar en conjunto con el CPU de una computadora de escritorio común, se puede obtener un sistema de cómputo de alto rendimiento, discreto y eficiente sin la necesidad de recurrir al uso de ambientes monumentales como un clúster de computadoras. Con millones de GPU NVIDIA vendidas hasta la fecha, los desarrolladores de software, científicos e investigadores encuentran usos más amplios para la computación de la GPU con CUDA.

En base a las características antes descritas se ha seleccionado la tarjeta GeForce GT 640 que se muestra en la Fig. 4, tomando en cuenta la capacidad de cómputo y por supuesto bajo precio, así como el fabricante (ASUS). Esta tarjeta de video proporciona características acordes para incursionar en el campo de programación paralela con CUDA.

Fig. 4. Tarjeta gráfica seleccionada en relación desempeño-precio.

B. Instalación y Configuración GPU Device

Las consideraciones previas para la instalación son principalmente el tipo de puerto, buses de memoria y consumo de potencia que necesita la GPU para realizar interfaz con la CPU, especificaciones que debe satisfacer la tarjeta madre. En el caso de una computadora de escritorio para conectar una tarjeta de video generalmente se utilizan dos puertos: AGP y PCI, siendo mayormente utilizado este último, pues proporciona prestaciones superiores en cuanto a transferencia de datos. El PCI se encuentra actualmente en la versión 3.0, los buses de memoria actuales son DDR3 y DDR5 y la fuente de energía que suministra la potencia en PC´s de escritorio oscila entre los 500w y 650w. Continuando con las consideraciones previas, dos en especial se deben cuidar a detalle la potencia y la temperatura de la tarjeta gráfica, cuyas especificaciones en este aspecto se suman al consumo de todos los dispositivos que integran el CPU. La GPU tiene una relación potencia-watts mayor que la CPU, para determinar correctamente la potencia total que necesitan funcionando en conjunto CPU + GPU se encuentra disponible un calculador de potencia en la página web de ©ASUSTeK Computer Inc. [7], el cual permite especificar los componentes instalados y como resultado muestra el suministro total de energía en watts. Para finalizar la configuración se instalan los controladores y software proporcionados por el fabricante de la tarjeta gráfica en este caso ASUS, el cual provee la utilidad GPU Tweak, usada para monitorear en tiempo real el estado de la GPU y optimizar su configuración para conseguir un rendimiento óptimo de la tarjeta gráfica.

C. Software CUDA

Para usar la arquitectura de programación paralela CUDA en un sistema operativo Windows, es necesario:

- Tarjeta gráfica compatible con CUDA, en este caso la tarjeta seleccionada es la GeForce GT 640 habilitada con 384 núcleos CUDA.
- Controlador de la tarjeta gráfica, en el caso de GPUs NVIDIA el controlador es general.
- Microsoft Windows XP, Vista, 7 y 8 en arquitecturas de 32 o 64 bits.
- NVIDIA CUDA Toolkit, la versión a instalar de este software es CUDA 5.

Microsoft Visual Studio 2008 o 2010, donde se utilizará como entorno de desarrollo el editor para C++ [8].

VI. Implementación

Para mostrar las ventajas del desempeño de un algoritmo paralelizado en la tecnología CUDA, en comparación con la programación secuencial usada tradicionalmente, se realiza una operación donde se toman dos vectores (v1 y v2) de N elementos de tipo entero. Posteriormente se inicializa v1 con una secuencia ascendente, desde 0 hasta N-1, y v2 con la suma de los valores de v1 en las posiciones i e i+1 elevadas al cubo. Es importante mencionar que esta operación se ha ejecutado tanto de forma paralela (GPU), como de forma secuencial (CPU) para hacer notar el rendimiento en cada uno de los dispositivos.

Se realizaron 10 ejecuciones del programa con diferente número de elementos para los vectores de prueba. Como puede notarse, la aceleración es directamente proporcional al número de elementos a procesar; es decir, mientras mayor sea la cantidad de datos a procesar menor será el tiempo de ejecución.

Cabe mencionar que cuando el número de elementos es menor que 50 millones, el tiempo de proceso en ambos dispositivos es similar, y por ende la aceleración no es

significativa. Esto se debe a que en la transferencia de datos desde la GPU a la CPU se consume mucho tiempo, y esta operación puede resultar innecesaria si la cantidad de datos a procesar es relativamente pequeña.

TABLA I
COMPARACIÓN DE LOS TIEMPOS REGISTRADOS EN AMBOS DISPOSITIVOS

Número de elementos (Millones)	Tiempo (s) CPU	Tiempo (s) GPU	Aceleración
50	0.59	0.418	x1.411
75	0.858	0.562	x1.526
100	1.734	1.093	x1.586
125	2.028	1.248	x1.625
150	1.482	0.889	x1.667
175	1.186	0.702	x1.689
192	2.246	1.311	x1.713

VII. CONCLUSIONES

En este artículo se han expuesto los conceptos básicos, el potencial y las restricciones de los dispositivos de procesamiento de gráficos (GPUs), al emplearse como procesadores paralelos de propósito general. En particular, se ha dado énfasis a las recomendaciones prácticas que procuran el mejor aprovechamiento de esta poderosa herramienta de cómputo.

Finalmente es importante aclarar que debido a la creciente demanda de los gráficos 3D de alta definición en aplicaciones interactivas, las GPUs han evolucionado en arquitecturas de procesamiento masivamente paralelas. La arquitectura de las GPUs modernas es muy eficiente, con la capacidad para realizar cómputo de propósito general ejecutando ciertos algoritmos de forma más rápida y eficiente que una CPU; siendo necesario considerar que no todos los problemas complejos pueden ser resueltos eficientemente en una GPU, la cual es adecuada para resolver problemas complejos que puedan ser expresados principalmente como el cálculo de datos en paralelo.

VIII. REFERENCES

[1] Oiso, Masashi and Matsumura, Yoshiyuki and Yasuda, Toshiyuki and Ohkura, Kazuhiro, "Implementing genetic algorithms to CUDA environment using data parallelization", Tehni{\v{c}}ki vjesnik, vol.18, pp. 511-517,2011.

[2] Technical Brief NVIDIA GeForce 8800 GPU Architecture Overview November 2006 TB-02787-001_v01

[3] P. Pospichal, J. Jaros and Josef Schwarz, "Parallel Genetic Algorithm on the CUDA Architecture", Springer, pp. 442 - 451, 2005.

[4] Sarnath Kannan and Raghavendra Ganji, Porting Autodock to CUDA, 2010.

[5] NVIDIA CUDA C Programming Guide, Versión 4.2, 2012

[6] CUDA C Best Practices Guide, Version 5.0, 2012

[7] http://support.asus.com/PowerSupply.aspx

[8] Installation and Verification on Windows DU-05349-001_v5.0 | October 2012

IX. BIOGRAFÍAS

Sergio Palafox Ugarte
Graduado de la Licenciatura en Informática en 2006 en el Instituto Tecnológico de Apizaco.
Actualmente estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco.
Áreas de interés: Cómputo Paralelo y Estrategias de Optimización.

José Federico Ramírez Cruz
Graduado de la ingeniería Industrial en Electrónica en el Instituto Tecnológico de Puebla en 1993.
Graduado de la Maestría en Ciencias en la especialidad de Electrónica en el Instituto Nacional de Astrofísica y Óptica en 1994.
Graduado del Doctorado en Ciencias, con especialidad en el área de Ciencias Computacionales en el Instituto Nacional de Astrofísica y Óptica en 2003.
Realizó una estancia postdoctoral en la Universidad de Texas, en El Paso, en 2011.
Es docente de tiempo completo de la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Apizaco
Áreas de interés: Algoritmos Evolutivos, Procesamiento Paralelo y Aprendizaje Automático.

Eustolia Carreón Esteban. Graduada de la Ingeniería en Sistemas Computacionales en 2009 en el Instituto Tecnológico Superior de Libres. Actualmente, estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco.
Áreas de interés: Cómputo Paralelo y Algoritmos Evolutivos

M.C. Blanca Estela Pedroza Méndez. Estudió la licenciatura en Matemáticas Aplicadas en la Universidad Autónoma de Tlaxcala y se graduó en 1993. Posteriormente se graduó como Maestro en Ciencias Computacionales en la Benemérita Universidad Autónoma de Puebla en 1998. Es profesora de tiempo completo de la División de Estudios de Posgrado e Investigación del Instituto Tecnológico de Apizaco y Coordinadora de la Maestría en Sistemas Computacionales. Áreas de interés: Procesamiento de Lenguaje Natural, Procesos Estocásticos y Tutoriales Inteligentes.

ALGORITMO EVOLUTIVO PARALELO PARA TOMOGRAFÍA SÍSMICA

Eustolia Carreón Esteban

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

euce_79@hotmail.com

José Federico Ramírez Cruz

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

federico_ramirez@yahoo.com.mx

Sergio Palafox Ugarte

Instituto Tecnológico de Apizaco, Av. Instituto Tecnológico s/n, 01 (241) 417 2010 ext 138

serggio.23@gmail.com

Resumen

En este trabajo de investigación se realiza la paralelización de un algoritmo de tomografía sísmica, cuyo objetivo es trazar los rayos generados por 7 fuentes de energía sísmica artificiales (SP) hacia cientos de dispositivos receptores (geófonos) y obtener el tiempo mínimo residual, utilizando la arquitectura CUDA. El cálculo del tiempo residual es realizado en la GPU y se obtiene con la diferencia del tiempo observado (registrado por un geólogo), y el calculado previamente en el algoritmo. El proceso de cálculo del tiempo residual y el trazo de la trayectoria del rayo sísmico que viaja desde un SP hasta un geófono es asignado a una unidad mínima de ejecución en la GPU (hilo), que para la aplicación realizada en este trabajo fueron 4,439 ejecutándose en paralelo. Los datos usados en este trabajo se obtuvieron de un experimento realizado en el campo volcánico El Potrillo, ubicado en el sur de Nuevo México, a cargo del Departamento de Ciencias Computacionales y del Departamento de Ciencias Geológicas de la Universidad de Texas, en El Paso. Las pruebas fueron realizadas en una GPU NVIDIA GeForce GTX 480 con 448 núcleos CUDA y con una CPU Intel Core I7 de 3.6 GHz.

Palabra(s) Clave(s): cómputo paralelo, CUDA C, tomografía sísmica, algoritmos evolutivos.

1. Introducción

La tomografía sísmica es una técnica de imagen que asimila observaciones del movimiento de la tierra, recolectadas con sismómetros, para mejorar los modelos estructurales, y ha sido uno de los medios más efectivos para obtener imágenes del interior del planeta en las últimas décadas [1]. Los modelos de velocidades de ondas, propagadas a través de la corteza terrestre por fuentes de energía naturales, como terremotos; o artificiales, como explosiones, proporcionan información que puede ser utilizada para una amplia variedad de aplicaciones tales como investigaciones arqueológicas, control de calidad y evaluación de proyectos de ingeniería, además del descubrimiento de depósitos de agua, aceite o material de desecho [2], para ello se usan diversas técnicas de búsqueda, en este caso en particular se utilizan los Algoritmos Evolutivos (AEs), ya que tienen la capacidad de buscar en espacios grandes y no son dependientes de una solución inicial aproximada a la óptima. La Computación Evolutiva (CE) es reconocida como un método eficaz para resolver problemas de optimización difíciles; sin embargo, conlleva grandes costos computacionales, ya que generalmente se evalúa a todos los candidatos de soluciones en una población para todas las generaciones [3] y por lo tanto ralentiza el proceso por horas o días, dependiendo de la cantidad de los datos que se usen. Debido a que muchas aplicaciones actualmente requieren mayor poder de cómputo del que una computadora secuencial es capaz de ofrecer, el cómputo paralelo ofrece la distribución del trabajo entre diferentes unidades de procesamiento, resultando mayor poder de cómputo y rendimiento del que se puede obtener mediante un sistema tradicional de un procesador [4]. Con el desarrollo de herramientas de programación de Unidades de Procesamiento Gráfico (GPU's, por sus siglas en inglés), varios algoritmos han sido adaptados a este hardware satisfactoriamente y la plataforma de computación híbrida GPU - CPU, comparada con las implementaciones en CPU's ha alcanzado aceleraciones importantes [5]

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

En los últimos años se han implementado los AEs en GPU's y se ha visto que son capaces de mejorar decenas de veces el desempeño mediante el uso de este hardware, sobre todo cuando se utilizan tamaños grandes de población [6].

En este trabajo se hace uso de un AE y de un Algoritmo de Tomografía Sísmica, que tienen como objetivo generar un modelo de velocidades de acuerdo a los tiempos de llegada de las ondas, desde una fuente de energía (SP por sus siglas en inglés Shot Point) hacia cada uno de los receptores (geófonos). Los datos utilizados fueron producto de un estudio en la región del campo volcánico El Potrillo, ubicado en el sureste de Nuevo México, cuyo volumen a evaluar fue de 414 414 m^3. Se realiza además la paralelización en una GPU y bajo la arquitectura de CUDA, en el lenguaje C, con la finalidad de reducir el tiempo de ejecución.

En la sección 2 se describirán los algoritmos ED, de tomografía sísmica y el modelo de paralelización sobre la GPU. En la sección 3 se presentan resultados experimentales obtenidos y se hace una comparación entre el desempeño de las versiones secuencial y paralela. En la sección 4 se mencionan las conclusiones y los trabajos futuros

2. Desarrollo

2.1 Algoritmo de Evolución Diferencial (ED)

El algoritmo de Evolución Diferencial es uno de los más potentes algoritmos de optimización actualmente usados [7, 8]. Para la generación del modelo de velocidades se implementa una ED, el cual mediante los operadores de selección, cruza, mutación y evaluación mejora con en el transcurso de decenas de generaciones.

El pseudocódigo del algoritmo evolutivo empleado en este trabajo se muestra en la Fig. 1.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

Fig. 1. Pseudocódigo de una Evolución Diferencial

2.1.2 Representación de los individuos

En este algoritmo cada individuo de la población representa un conjunto de profundidades y velocidades en 8 diferentes capas de la tierra.

En la Fig. 2 se representa un individuo, el cual al ser visto como un vector, está dividido en dos partes. En la primera, los valores 1 y 69 ocupan las posiciones 0 y 7, respectivamente, que son el mínimo y el máximo de la profundidad en kms, y las posiciones intermedias son ocupadas por valores de tipo entero que no pueden repetirse y que deben ser ordenados de menor a mayor. En la segunda parte del individuo se asignan las velocidades en un rango de 3 a 8 km/s; que cumplen con las mismas restricciones de la primera pero son valores reales.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

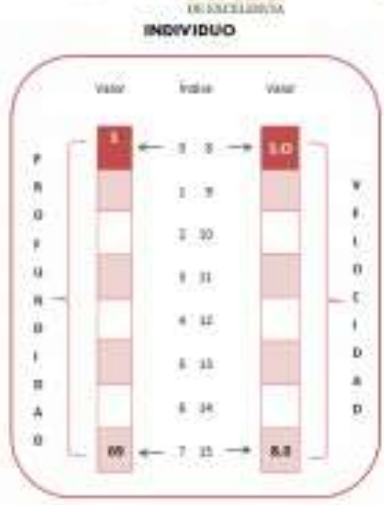


Fig. 2. Representación de un individuo

2.1.3 Recombinación o cruza

La recombinación o cruza es una de las operaciones evolutivas implicadas en la generación de nuevos individuos. Esta operación es llevada a cabo con la participación de dos o más padres, quienes heredan rasgos o características a sus descendientes mediante una mezcla de información que se da de manera aleatoria. También existe la posibilidad de que los padres sobrevivan para formar parte de la siguiente generación, esto se da en caso de que la aptitud sea mejor que la de los hijos.

Aunque la cruza no es el operador principal en una ED, a diferencia de un Algoritmo Genético, éste se implementa debido a que es parte de los principios originales de los AEs, y depende de un parámetro muy importante: la tasa de recombinación o cruza, que de acuerdo a la literatura consultada está en un rango de [0 a 1]. Para este caso se estableció en 0.8, y el tipo de cruza es discreta binomial como se representa en la Fig. 3.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

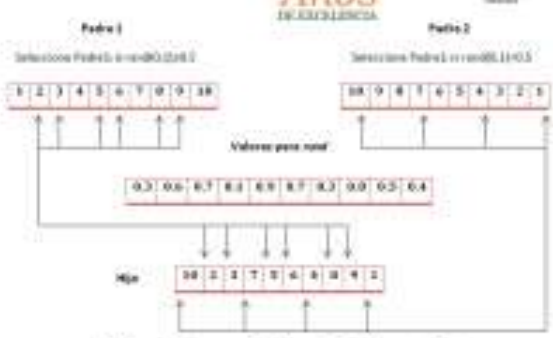


Fig. 3. Cruza Discreta Binomial

2.1.4 Mutación

El operador de mutación proporciona diversidad a la población a través de la introducción de nuevas soluciones, lo cual evita la convergencia prematura en el algoritmo; es decir, la posibilidad de una rápida obtención no necesariamente del óptimo global. Para superar este problema es necesario conservar la diversidad en las generaciones, lo cual permite la exploración de otras áreas en el espacio de búsqueda.

Para llevar a cabo la operación de mutación, por cada individuo de la población x_i, se eligen aleatoriamente tres individuos de población, $x_{r1}, x_{r2}, y\, x_{r3}$, con la condición de que sean diferentes entre sí y diferentes de x_i, el operador de mutación se calcula como la diferencia de dos de ellos multiplicada por un factor $F \in [0..1]$, y el resultado es sumado al tercer individuo, esto es, $x_i' = x_{r3} + F(x_{r2} - x_{r1})$, como se muestra en la Fig. 4. El valor de F puede ser establecido por el diseñador o puede ser un número aleatorio.

2.1.5 La Función de evaluación

Cada vez que un individuo x_i' es generado, debe ser evaluado para determinar si forma parte de la siguiente generación, en dado caso que el valor de la evaluación $f(x_i')$ sea mayor que la de su padre $f(x_i)$, este individuo pasa a la siguiente generación, de lo contrario se queda el padre x_i.

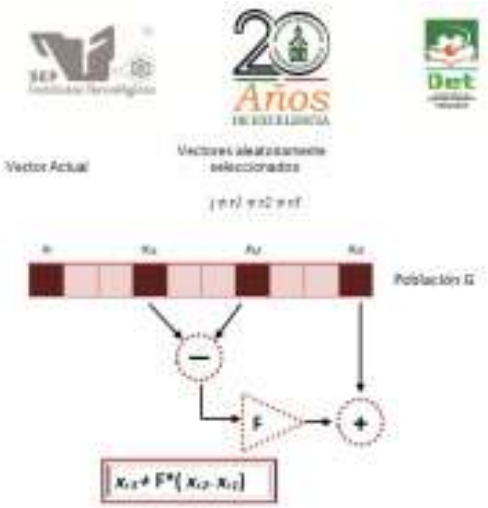


Fig. 4. Operación de Mutación en la ED

La función de evaluación de este algoritmo consiste en medir los tiempos residuales (diferencia entre el tiempo de llegada del rayo sísmico a cada geófono y el tiempo observado). Para calcular el tiempo de llegada del rayo sísmico se realiza una simulación de la propagación del frente de onda sísmica dentro del volumen a evaluar utilizando el modelo de Vidale [9]. Para calcular la posición del rayo sísmico dentro de la celda donde se encuentra el geófono se utiliza una interpolación tri-lineal utilizando los tiempos de llegada del frente de onda a los ocho vértices de la celda. Estos dos procesos son parte de un Algoritmo de Tomografía Sísmica (ATS) propuesto por Vidale and Holes [9, 10]. En este trabajo, el proceso del cálculo de del tiempo residual es paralelizado sobre una GPU.

El ATS inicia leyendo los datos de las ubicaciones tanto de las 7 fuentes de energía, de los cientos de receptores distribuidos, y de los tiempos de llegada en base al análisis de estudios sísmicos anteriores realizados por un geólogo. Se crea una red uniforme en las tres dimensiones del volumen de estudio, como se muestra en la Fig. 5. En este caso el tamaño de cada una de las celdas es de 1 km^3. De acuerdo a la velocidad y a la profundidad (representadas en cada uno de los individuos), se genera mediante el método de interpolación un modelo inicial de tiempos de llegada de los frentes de onda para rastrear las trayectorias de los rayos a lo largo de la

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

pendiente más pronunciada (la línea perpendicular a cada frente de onda) de un receptor a la fuente.

Después de que se han generado los tiempos de viaje se calcula el tiempo residual mediante la diferencia del observado y el calculado. Una vez que se han realizado estas operaciones para cada rayo que atraviesa el modelo desde el geófono hasta la fuente, se repite el proceso para el resto de las fuentes y de los receptores y se obtiene un promedio.

El modelo de velocidades tiene que ir mejorando haciendo cada vez mínimo el tiempo residual a través de las generaciones.

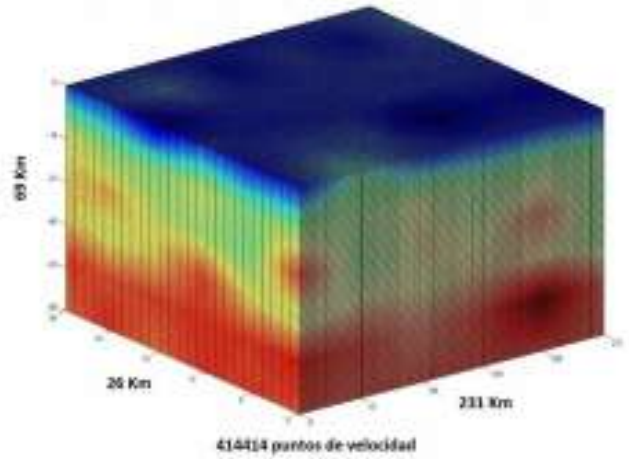


Fig. 5. Modelo de velocidades de la corteza terrestre

2.2 Modelo de paralelización

El modelo paralelo que se propone en este trabajo consiste en hacer el trazo de cada uno de los rayos de todas las fuentes, en forma independiente, debido a que no necesitan información de cálculos previos al que se está evaluando.

La información correspondiente a las ubicaciones de las fuentes de energía y a los receptores, así como la de los tiempos de llegada es transferida desde la CPU a la

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

GPU, con la finalidad de tener acceso de manera más rápida, ya que el pase de datos de un dispositivo a otro es el inconveniente que tiene el uso de este hardware.

El algoritmo de evolución diferencial implementado en la GPU adopta el esquema maestro-esclavo de una sola población, en donde el nodo maestro se encuentra representado por la CPU mientras que los esclavos serán cada uno de los procesadores dentro de la GPU.

En la arquitectura CUDA los hilos se encuentran dentro de bloques. El número de hilos por bloque (NH) puede ser desde 16 hasta 512 o 1024 dependiendo del modelo. En este caso se hace la prueba con NH= 64, para un mejor rendimiento. El número de bloques es calculado mediante la siguiente ecuación: $n = Ngeofonos/NH$.

En la Fig. 6 se muestra de manera general el modelo paralelo implementado.

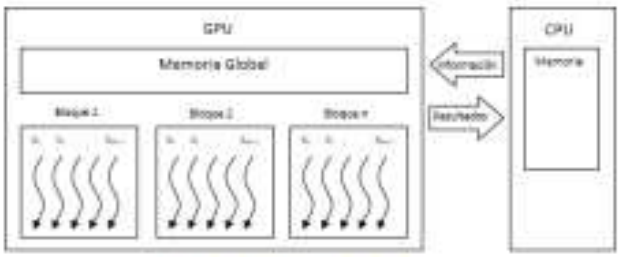


Fig. 6. Modelo paralelo

2.1.2 Reservación de memoria y transferencia de datos

Para la transferencia de datos desde la CPU a la GPU es necesario reservar un espacio en la memoria global mediante la instrucción cudaMalloc, que incluye como parámetros el número de valores a recibir multiplicado por el tipo de dato y un puntero para ir almacenándolos. Posteriormente se realiza la copia de la información con la función cudaMemcpy la cual es caracterizada por un parámetro que indica la fuente y el destino para realizar la transferencia, y que pueden ser:

cudaMemcpyHostToDevice, para copiar desde la CPU a la GPU; o cudaMemcpyDeviceToHost, para copiar desde la GPU a la CPU.

3. Resultados

En la versión secuencial se leen datos para hacer el trazo de rayos de un sólo un geófono. Se sigue la trayectoria celda por celda, desde el receptor hasta la fuente de energía, pero no se puede continuar con el siguiente sino hasta que termina el actual; en contraste con el modelo paralelo propuesto, donde se hace el cálculo simultáneamente para todos los geófonos hacia todas las fuentes de energía, ya que en cada uno de ellos se usan valores diferentes por lo que cada hilo toma los valores necesarios.

La versión secuencial realiza el cálculo de los tiempos residuales en un tiempo aproximado de 100 ms; sin embargo, la versión paralela registra un tiempo menor a 1 ms sin considerar la trasmisión de datos de la memoria del CPU al GPU.

5. Conclusiones

El concepto básico en un esquema de procesamiento paralelo es la división de un proceso en varios sub-procesos, los cuales son resueltos simultáneamente empleando múltiples procesadores.

En este trabajo se realizó la paralelización de la función objetivo de un algoritmo de evolutivo (ED), que consiste en el cálculo del promedio cuadrático de los tiempos residuales de un algoritmo de tomografía sísmica.

Se calculó el tiempo residual de 4440 los geófonos de forma paralela utilizando una GPU, cada hilo de la GPU calcula el tiempo y la trayectoria del rayo sísmico que va desde 7 fuentes de energía (SP) hasta cada uno de los geófonos.

En este trabajo se utilizó la memoria global del dispositivo para pasar la información de los cuadrados de los tiempos residuales de la GPU al CPU, en un trabajo futuro se utilizará la memoria compartida del dispositivo para realizar en la tarjeta GPU el promedio cuadrático de los tiempos residuales, se asume que este proceso acelerará el cálculo.

Instituto Tecnológico Superior de Misantla
Km. 1.8, Carretera a Loma del Cojolite
Tel. 01 (235) 323 15 45, www.itsm.edu.mx
C.P 93821, Misantla, Veracruz, México

6. Referencias

1. En-Jui , L., He, H., John M., D., Po, C., & Liqiang, W. An optimized parallel LSQR algorithm for seismic tomography. *Computers & Geosciences*, 2013, pp. 184 - 197
2. J. F. Ramírez Cruz, O. Fuentes, R. Romero, & A. Velasco, A Hybrid Algorithm for Crustal Velocity Modeling. *In Advances in Computational Intelligence*. Springer Berlin Heidelberg, 2013, pp. 329-337
3. M. Oiso, Y. Matsumura, T. Yasuda and K. Ohkura. Implementing genetic algorithms to CUDA environment using data parallelization. *Technical Gazette, Hrcak Portal of scientific journals of Croatia*, 2011, Vol.18, No.4, 2011 pp. 511 – 517
4. Kirk, D. B., & Hwu, W.-m. W. Programming Massively Parallel Processors: A Hands-on Approach. San Francisco, CA, USA: Elsevier, 2010.
5. M, Dawei, P. Chen, & L. Wang, Accelerating the discontinuous Galerkin method for seismic wave propagation simulations using multiple GPUs with CUDA and MPI. Earthquake Science, Vol 26, No. 6, 2013, pp. 377-393
6. Chitty, Darren M. "A data parallel approach to genetic programming using programmable graphics hardware." Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO 07. ACM, 2007, pp. 1566-1573.
7. S. Das, and P. N. Suganthan, Differential Evolution: A Survey of the State-of-the-Art. IEEE Transactions on Evolutionary Computation, Vol. 15, No. 1, 2011, pp. 4-31
8. R. Storn and K. V. Price, "Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces," ICSI, USA, Tech. Rep. TR-95-012, 1995 [Online]. Available: http://icsi.berkeley.edu/~storn/litera.html
9. J E. Vidale, Finite-difference calculation of traveltimes in three dimensions, Geophysics, vol. 55, No. 5, 1990, pp 521-526.

10. J. A. Hole, Nonlinear high-resolution three-dimensional seismic travel time tomography: Journal of Geophysical Research, Vol. 97, No. 85, 1992, pp. 6553–6562.

7. Autores

Ing. Eustolia Carreón-Esteban. Graduada de la Ingeniería en Sistemas Computacionales en 2009 en el Instituto Tecnológico Superior de Libres. Actualmente, estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco. Áreas de interés: Cómputo Paralelo y Algoritmos Evolutivos.

Dr. José Federico Ramírez-Cruz, Graduado de la ingeniería Industrial en Electrónica en el Instituto Tecnológico de Puebla en 1993, Graduado de la Maestría en Ciencias en la especialidad de Electrónica en el Instituto Nacional de Astrofísica y Óptica en 1994, Graduado del Doctorado en Ciencias, con especialidad en el area de Ciencias Computacionales en el Instituto Nacional de Astrofísica y Óptica en 2003, Realizó una estancia postdoctoral en la Universidad de Texas, en El Paso, en 2011, Es docente de tiempo completo del Departamento de Sistemas y Computación del Instituto Tecnológico de Apizaco. Áreas de interés: Algoritmos Evolutivos, Procesamiento Paralelo y Aprendizaje Automático.

Lic. Sergio Palafox-Ugarte. Graduado de la Licenciatura en Informática en 2006 en el Instituto Tecnológico de Apizaco. Actualmente estudiante de la Maestría en Sistemas Computacionales en el Instituto Tecnológico de Apizaco. Áreas de interés: Cómputo Paralelo y Optimización Evolutiva.

2 Estancias de Investigación

BENEMERITA
UNIVERSIDAD
AUTÓNOMA DE PUEBLA

FACULTAD DE
CIENCIAS DE LA COMPUTACIÓN

Oficio SIEPFCC/05/2014.

ASUNTO: Carta de Aceptación.

MTRO. FELIPE PASCUAL ROSARIO AGUIRRE
DIRECTOR DEL INSTITUTO TECNOLÓGICO DE APIZACO, TLAX.
PRESENTE.

Dentro del marco de colaboración que nuestras Instituciones tienen bajo el convenio firmado desde el año 2012, aprovecho para enviarle un cordial saludo, y a la vez hacer de su conocimiento que nos es grato hacer constar que el:

Lic. Sergio Palafox Ugarte

quien es estudiante de la Maestría en Sistemas Computacionales del Instituto Tecnológico de Apizaco, Tlax., fue **ACEPTADO** en nuestro programa de Maestría en Ciencias de la Computación, para realizar la estancia técnica y de investigación descrita en el plan de trabajo que nos hizo llegar, con el objetivo de fomentar el intercambio académico, la movilidad y la vinculación entre nuestras Instituciones en general, y entre nuestros Programas de Posgrado de Excelencia registrados en el PNPC del CONACyT en particular, durante los meses de Enero a Abril del año actual. Cabe mencionar que yo personalmente fungiré como co-asesor del citado estudiante por parte de la BUAP.

Sin más por el momento, agradezco su atención para la presente y quedo de usted.

Recibí original

ATENTAMENTE
"PENSAR BIEN, PARA VIVIR MEJOR"
H. Puebla de Z., a 10 de Enero de 2014.

Benemérita Universidad Autónoma de Puebla
SECRETARÍA DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO
FACULTAD CS. DE LA COMP.

DR. LUIS CARLOS ALTAMIRANO ROBLES
Secretario de Inv. y Estudios de Posgrado

c.c.p.- Archivo.

Avenida San Claudio y 14 Sur
Edificio 104 A, Ciudad Universitaria
Puebla, Pue., CP. 72570

Teléfono: 01 (222) 229 55 00
Exts. 7206 y 5673
Fax: 01 (222) 229 56 72

BENEMÉRITA
UNIVERSIDAD
AUTÓNOMA DE PUEBLA

FACULTAD DE
CIENCIAS DE LA COMPUTACIÓN

Oficio SIEPFCC/62/14.

MTRO. FELIPE PASCUAL ROSARIO AGUIRRE
DIRECTOR DEL INSTITUTO TECNOLÓGICO DE APIZACO
PRESENTE

Por medio de la presente y a la par de enviarle un cordial saludo, aprovecho la oportunidad de hacer de su conocimiento que el C:

Lic. Sergio Palafox Ugarte

quien es estudiante de la Maestría en Sistemas Computacionales del Instituto Tecnológico de Apizaco, Tlax., y que fue aceptado para realizar una Estancia Técnica en nuestra Benemérita Institución para apoyar la realización de su trabajo de Tesis de Maestría, co-asesorado por un servidor, ha cumplido con el periodo establecido para la citada Estancia, a saber, **del 06 de Enero al 07 de Abril de 2014** y ha culminado exitosamente el plan de trabajo acordado con antelación y que forma parte de su trabajo de Tesis de Maestría.

A petición del interesado(a) y para los fines legales que estime pertinentes, se extiende la presente en la ciudad Heroica de Puebla de Zaragoza a los veintinueve días del mes de mayo del año dos mil catorce.

Recibí original

ATENTAMENTE
"PENSAR BIEN, PARA VIVIR MEJOR"
H. Puebla de Z., a 29 de Mayo de 2014.

Benemérita Universidad Autónoma de Puebla
SECRETARÍA DE INVESTIGACIÓN Y ESTUDIOS DE POSGRADO
FACULTAD CS. DE LA COMP.

DR. LUIS CARLOS ALTAMIRANO ROBLES
Secretario de Investigación y Estudios de Posgrado

c.c.p.- Archivo.

Avenida San Claudio y 14 Sur, Edificio 104 A, Ciudad Universitaria, Puebla, Pue., CP. 72570
Teléfono: 01 (222) 229 55 00 Exts. 7206 y 5673 Fax: 01 (222) 229 56 72

Anexos

1 Instalación de CUDA e interfaz con Visual Studio

Para usar la arquitectura de programación paralela CUDA en un sistema operativo Windows, es necesario:

- Tarjeta gráfica compatible con CUDA, en este caso la tarjeta seleccionada es la GeForce GT 640 habilitada con 384 núcleos CUDA.
- Controlador de la tarjeta gráfica, en el caso de GPUs NVIDIA el controlador es general.
- Microsoft Windows XP, Vista, 7 y 8 en arquitecturas de 32 y 64 bits, se usara Windows 7 en 64 bits.
- NVIDIA CUDA Toolkit, la versión a instalar de este software es CUDA 5.
- Microsoft Visual Studio 2008 o 2010, donde se utilizara como entorno de desarrollo el editor para C++.

La instalación del software CUDA en el sistema operativo debe realizarse en orden para que funcione correctamente junto con el entorno de desarrollo, a continuación se describen los pasos a seguir:

Verificar que el sistema tenga instalada una tarjeta gráfica compatible con CUDA, la cual se debe verificar por medio del modelo y fabricante para identificar que las especificaciones indiquen que la tecnología CUDA está habilitada, si la tarjeta gráfica es NVIDIA se puede consultar una lista con tarjetas gráficas compatibles con CUDA en http://www.nvidia.com/object/cuda_gpus.html

El software CUDA está disponible en la página de NVIDIA ttps://developer.nvidia.com/cuda-downloads disponible para diferentes sistemas operativos, en este caso se utiliza el adecuado para sistemas Windows.

El Toolkit de CUDA contiene el driver y herramientas necesarias para crear, compilar y ejecutar aplicaciones CUDA, así como librerías y encabezados códigos fuente de ejemplos

CUDA y otros recursos, cabe mencionar que en la versión 5 de CUDA integra el driver, toolkit, sdk e integración automática con Visual Studio 2008 o 2010.

El directorio de instalación automáticamente está en la ruta: C:\Program Files\NVIDIA GPU ComputingToolkit\CUDA\v5.0 este directorio contiene:

- Bin\ Compilador y librerías
- Include\ archivos de cabecera necesarios para compilar programas CUDA
- Lib\ archivos de librerías necesarios para ligar programas CUDA
- Doc\ guías de programación en CUDA C y documentación de librerías y toolkit de CUDA

Nota: a partir del CUDA toolkit 3.2 pueden ser instaladas múltiples versiones del toolkit.

Antes de continuar es importante verificar que los programas CUDA pueden encontrar y comunicarse correctamente con el hardware compatible con CUDA, para hacer esto se necesitan compilar y ejecutar algunos programas de ejemplo incluidos en el software CUDA que se encuentran tanto en forma de código fuente y compilada. Para verificar una correcta configuración entre el hardware (GPU) y software (CUDA) es altamente recomendable ejecutar los siguientes programas: deviceQuery y bandwidthTest.

2 Instalación y configuración de Cilk

El software Cilk se distribuye bajo Licencia Pública General (GNU General Public License), al día de hoy la versión oficial liberada más reciente es Cilk-5.4.6, la cual funciona en plataformas GNU/Linux principalmente, así como MacOS X y Microsoft Windows (necesita Cygwin). La instalación de Cilk se puede realizar en distribuciones Linux Ubuntu recientes con algunos inconvenientes que se pueden reparar sin mayor problema una vez que se conocen los paquetes del sistema necesarios, directorios de la instalación y el proceso de compilación de Cilk.

A continuación se describe el proceso para instalar Cilk 5.4.6 en la distribución Linux Ubuntu 12.04 LTS (Precise Pangolin), antes de iniciar es conveniente verificar los paquetes necesarios: GNU Make, GNU linker y gcc, los cuales vienen incluidos en Ubuntu de forma

automática, prosiguiendo el proceso de instalación:

Primero se debe obtener el software el cual está disponible en la página web del proyecto Cilk http://supertech.csail.mit.edu/cilk/cilk-5.4.6.tar.gz asumiendo que el directorio donde se almacena el archivo es *"~/Descargas* " y la ubicación de instalación debe ser un directorio oficial como "*/usr/local*" se realiza la extracción del archivo comprimido de la siguiente forma:

sergio@sergio-Precision-M4500:~$ ***cd /usr/local/***

sergio@sergio-Precision-M4500:/usr/local$ ***sudo tar xvf /home/sergio/Descargas/cilk-5.4.6.tar.gz***

Con lo anterior se extraen 1824 elementos en la carpeta de instalación ubicada en */usr/local/cilk-5.4.6*

A continuación se accede a esta ubicación para realizar la instalación de Cilk la cual se lleva a cabo con dos comandos: *./configure y make*

sergio@sergio-Precision-M4500:/usr/local/cilk-5.4.6$ ***./configure***

Al comando anterior se debe agregar los siguientes parámetros que no se especifican en la documentación oficial de Cilk lo que causa fallas en la instalación.

./configure CFLAGS="-D_XOPEN_SOURCE=600 -D_POSIX_C_SOURCE=200809L"

sergio@sergio-Precision-M4500:/usr/local/cilk-5.4.6$ ***make***

sergio@sergio-Precision-M4500:/usr/local/cilk-5.4.6$ ***make check***

sergio@sergio-Precision-M4500:/usr/local/cilk-5.4.6$ ***sudo make install***

Lo anterior instala en el sistema:

- Librerías en **/usr/local/lib**
- Archivos de cabeceras en **/usr/local/include**
- Compilador en **/usr/local/bin**

3 Compilación y ejecución de programas Cilk

Ahora que el software Cilk está instalado en el sistema se puede realizar la compilación de programas especificando el número de procesadores a usar para ser ejecutados de forma paralela.

Compilación básica desde la línea de comandos donde se especifica el programa a compilar:

cilkc nombre.cilk -o nombre-ejecutable

Ejecución básica desde la línea de comandos donde se especifica el número de procesadores a usar:

./nombre-ejecutable --nproc 4

Si se tienen errores durante la compilación como los siguientes:

/usr/include/bits/waitstatus.h:76: syntax error

/usr/include/bits/waitstatus.h:90: syntax error

Borrar ambas líneas en la cabecera waitstatus.h

Durante la ejecución de un programa el compilador de Cilk puede proporcionar información de acuerdo al desempeño del programa mostrando datos útiles como el tiempo total de la ejecución, para lo cual se debe compilar utilizando las siguientes banderas:

cilkc -cilk-profile -cilk-critical-path nombre.cilk -o nombre-ejecutable

La bandera ***-cilk-profile*** recolecta datos acerca de cuánto tiempo toma a cada procesador realizar la carga de trabajo asignada, cuantas migraciones de hilos ocurrieron, cantidad de memoria usada etc. La bandera ***-cilk-critical-path*** habilita la medición de la ruta crítica (tiempo de ejecución ideal sobre un número infinito de procesadores), aunque su uso hace que el rendimiento en la ejecución del programa disminuya considerablemente debido a la sobrecarga que se genera al hacer gran número de llamadas a rutinas del compilador de cilk. Por lo anterior se recomienda usar solamente la bandera ***-cilk-profile***, para obtener los datos en pantalla se debe agregar la bandera --***stats 1*** entonces la compilación quedaría de la siguiente forma:

cilkc -cilk-profile nombre.cilk -o nombre-ejecutable

./nombre-ejecutable --nproc 4 --stats 1

4 Código fuente

```
/*****************************************************************/
/* Algoritmo genetico, implementacion paralela en Cilk 5.4.6   */
/*****************************************************************/

#include <cilk-lib.cilkh>
#include <cilk.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

/* Parametros de inicio */

#define TAMPOB 16384            //TamaÃ±o de la poblacion (# individuos)
#define NUMGENS 1000               //Numero de generaciones
#define NUMVAR 3                //Numero de variables del problema
#define PROBCRUZA 0.8             //Probabilidad de cruza
#define PROBMUTA 0.15           //Probabilidad de mutacion
#define TRUE 1
#define FALSE 0

int generacion;                //No. generacion actual
int mej_ind;                    //Mejor individuo actual
FILE *salidag;                //Archivo de salida con resultados

/* genotipo de un individuo de la poblaciÃ³n */

struct genotipo
{
  double gen[NUMVAR];         //Cadena de variables
  double aptitud;              //Aptitud
  double superior[NUMVAR];  //limite superior/
  double inferior[NUMVAR];    //limite inferior
  double rAptitud;          //Aptitud relativa
  double aAptitud;            //Aptitud acumulada
};

struct genotipo pob[TAMPOB+1];     //Poblacion
struct genotipo npob[TAMPOB+1]; //Nueva poblacion

/* Funciones utilizads por el algoritmo genetico */

/*cilk*/ void iniciar(void);
double valeatorio(double, double);
cilk void objetivo(void);
/*cilk*/ void mejores(void);
cilk void elite(void);
cilk void seleccion(void);
/*cilk*/ void cruza(void); //
/*cilk*/ void X_cruza(int,int);
/*cilk*/ void interc(double *, double *); //
/*cilk*/ void mutacion(void);
void reporte(void);

/*********************************************************************/
/* Funcion de inicio: crea los valores de cada gen dentro de        */
/* los limites de las variables.                                     */
/*********************************************************************/

/*cilk*/ void iniciar(void)
{
FILE *infile;
int i, j;
double lim_inf, lim_sup;
```

```
if ((infile = fopen("entradag.txt","r"))==NULL)
      {
      fprintf(salidag,"\nNo se encuentra archivo de entrada!\n");
      exit(1);
      }

/* iniciar variables dentro de los limites */

for (i = 0; i < NUMVAR; i++)
      {
      fscanf(infile, "%lf",&lim_inf);
      fscanf(infile, "%lf",&lim_sup);

      for (j = 0; j < TAMPOB; j++)
           {
           pob[j].aptitud = 0;
           pob[j].rAptitud = 0;
           pob[j].aAptitud = 0;
           pob[j].inferior[i] = lim_inf;
           pob[j].superior[i]= lim_sup;
           pob[j].gen[i] = valeatorio(pob[j].inferior[i],
                                      pob[j].superior[i]);
           }
      }

fclose(infile);
}

/*************************************************************************/
/* Generador de numeros aleatorios: genera valores dentro de los limites */
/*************************************************************************/

double valeatorio(double inf, double sup )
{
double val;
srand(time(NULL));
val = ((double)(rand()%1000)/1000.0)*(sup  - inf) + inf;
return(val);
}

/*************************************************************************/
/* Funcion objetivo: funcion definida por el usuario                     */
/*************************************************************************/

cilk void objetivo(void)
{
int individuo, i;
double x[NUMVAR+1];

for (individuo = 0; individuo < TAMPOB; individuo++)
      {
      for (i = 0; i < NUMVAR; i++)
            x[i+1] = spawn pob[individuo].gen[i];

      pob[individuo].aptitud = spawn (x[1]*x[1]) - (x[1]*x[2]) + x[3];
      sync;
      }
}

/*************************************************************/
/* Funcion mejores: realiza seguimiento del mejor individuo  */
/* de la poblacion.                                          */
/*************************************************************/

/*cilk*/ void mejores()
{
int individuo;
int i;
```

```
mej_ind = 0; /* almacena el indice del mejor individuo actual */

for (individuo = 0; individuo < TAMPOB; individuo++)
      {
      if (pob[individuo].aptitud > pob[TAMPOB].aptitud)
            {
            mej_ind = individuo;
            pob[TAMPOB].aptitud = pob[individuo].aptitud;
            }
      }
/* una vez que el mejor individuo de la poblacion se encuentra, almacena los genes */
for (i = 0; i < NUMVAR; i++)
      pob[TAMPOB].gen[i] = pob[mej_ind].gen[i];
}

/*********************************************************************/
/* Funcion de elite: el mejor individuo de la generacion anterior es */
/* almacenado como el ultimo en el arreglo.                          */
/*********************************************************************/

cilk void elite()
{
int i;
double mejor, peor;             /* mejor-peor valores aptitud */
int mejor_individuo, peor_individuo; /* indices del mejor y peor individuos */

mejor = pob[0].aptitud;
peor = pob[0].aptitud;
for (i = 0; i < TAMPOB - 1; ++i)
      {
      if(pob[i].aptitud > pob[i+1].aptitud)
            {
            if (pob[i].aptitud >= mejor)
                  {
                  mejor = pob[i].aptitud;
                  mejor_individuo = i;
                  }
            if (pob[i+1].aptitud <= peor)
                  {
                  peor = pob[i+1].aptitud;
                  peor_individuo = i + 1;
                  }
            }
      else
            {
            if (pob[i].aptitud <= peor)
                  {
                  peor = pob[i].aptitud;
                  peor_individuo = i;
                  }
            if (pob[i+1].aptitud >= mejor)
                  {
                  mejor = pob[i+1].aptitud;
                  mejor_individuo = i + 1;
                  }
            }
      }
/*Si el mejor individuo de la nueva poblacion es mejor, que el mejor de la poblacion anterior
entonces copia el mejor de la nueva poblacion*/

if (mejor >= pob[TAMPOB].aptitud)
    {
    for (i = 0; i < NUMVAR; i++)
       pob[TAMPOB].gen[i] = pob[mejor_individuo].gen[i];
    pob[TAMPOB].aptitud = pob[mejor_individuo].aptitud;
    }
else
    {
```

```
    for (i = 0; i < NUMVAR; i++)
       pob[peor_individuo].gen[i] = pob[TAMPOB].gen[i];
    pob[peor_individuo].aptitud = pob[TAMPOB].aptitud;
    }
}
/*************************************************************/
/* funcion de seleccion: selecciÃ³n proporcional estÃ¡ndar     */
/*************************************************************/

cilk void seleccion(void)
{
int individuo, i, j;
double sum = 0;
double p;

/* busca la aptitud total de la poblacion */
for (individuo = 0; individuo < TAMPOB; individuo++)
      {
      sum += pob[individuo].aptitud;
      }

/* calcula aptitud relativa */
for (individuo = 0; individuo < TAMPOB; individuo++)
      {
      pob[individuo].rAptitud =  pob[individuo].aptitud/sum;
      }
pob[0].aAptitud = pob[0].rAptitud;

/* calcula aptitud acumulada */
for (individuo = 1; individuo < TAMPOB; individuo++)
      {
      pob[individuo].aAptitud =  pob[individuo-1].aAptitud +
                        pob[individuo].rAptitud;
      }

/* finalmente selecciona sobrevivientes utilizando la aptitud acumulada */

for (i = 0; i < TAMPOB; i++)
      {
      p = rand()%1000/1000.0;
      if (p < pob[0].aAptitud)
            npob[i] = pob[0];
      else
            {
            for (j = 0; j < TAMPOB;j++)
                  if (p >= pob[j].aAptitud &&
                              p<pob[j+1].aAptitud)
                        npob[i] = pob[j+1];
            }
      }
/* una vez que se crea una nueva poblaciÃ³n, reemplaza la anterior */

for (i = 0; i < TAMPOB; i++)
      pob[i] = npob[i];
}

/*************************************************************/
/* funcion de cruza: selecciona dos individuos (padres)      */
/*************************************************************/

/*cilk*/ void cruza(void) //
{
int  individuo, uno;
int padres  =  0; /* contador del numero de individuos seleccionados */
double x;

for (individuo = 0; individuo < TAMPOB; ++individuo)
      {
```

```
      x = rand()%1000/1000.0;
      if (x < PROBCRUZA)
            {
            ++padres;
            if (padres % 2 == 0)
                  /*spawn*/ X_cruza(uno, individuo); // indicado el uso de spawn por el
compilador de Cilk
            else
                  uno = individuo;
            }
      }
}
/*****************************************************************************/
/* funcion operacion(X) de cruza: realiza la cruza de los dos padres seleccionados*/
/*****************************************************************************/

/*cilk*/ void X_cruza(int p1, int p2)
{
int i, punto_X; /* punto de cruza */

/* selecciona punto de cruza */
if(NUMVAR > 1)
   {
   if(NUMVAR == 2)
         punto_X = 1;
   else
         punto_X = (rand() % (NUMVAR - 1)) + 1;

   for (i = 0; i < punto_X; i++)
        /*spawn*/ interc(&pob[p1].gen[i], &pob[p2].gen[i]); // indicado el uso de spawn por el
compilador de Cilk

   }
}

/*****************************************************************************/
/* Funcion de intercambio: Un procedimiento de ayuda, intercambiando 2 variables */
/*****************************************************************************/

/*cilk*/ void interc(double *x, double *y)
{
double temp;

temp = *x;
*x = *y;
*y = temp;

}

/*********************************************************************/
/* Funcion de mutacion
/*********************************************************************/

/*cilk*/ void mutacion(void)
{
int i, j;
double lim_i, lim_s;
double x;

for (i = 0; i < TAMPOB; i++)
      for (j = 0; j < NUMVAR; j++)
            {
            x = rand()%1000/1000.0;
            if (x < PROBMUTA)
                  {
                  /* encontrar los limites en la variable que sera mutada */
                  lim_i = pob[i].inferior[j];
                  lim_s = pob[i].superior[j];
```

```
                pob[i].gen[j] = valeatorio(lim_i, lim_s);
                }
            }
}

/*************************************************************************/
/* Funcion de reporte: la informacion con los resultados se almacenan */
/* dentro de un archivo de salida "salidag.txt" separado por comas    */
/*************************************************************************/

void reporte(void)
{
int i;
double mejor_val;          //mejor aptitud en la poblacion
double promedio;             //promedio de la aptitud en la poblacion
double sum_cuad;            //suma de cuadrados para calculo desviacion est.
double cuad_sum;          //cuadrado de sumas para calculo desviacion est.
double sum;                  // aptitud total en la poblacion

sum = 0.0;
sum_cuad = 0.0;

for (i = 0; i < TAMPOB; i++)
      {
      sum += pob[i].aptitud;
      sum_cuad += pob[i].aptitud * pob[i].aptitud;
      }

promedio = sum/(double)TAMPOB;
cuad_sum = promedio * promedio * TAMPOB;
mejor_val = pob[TAMPOB].aptitud;

fprintf(salidag, "\n%5d,      %6.3f, %6.3f \n\n", generacion, mejor_val, promedio);
}

/*************************************************************************************/
/* Funcion principal: cada generacion evoluciona seleccionando los mejores     */
/* individuos, realizando cruza y mutacion, evaluando cada poblacion nueva     */
/* hasta que la condicion de terminacion se cumpla                             */
/*************************************************************************************/

cilk int main(void)
{
int i;

if ((salidag = fopen("salidag.txt","w"))==NULL)
      {
      exit(1);
      }
generacion = 0;

fprintf(salidag, "\n Numero      Mejor    Valor \n");
fprintf(salidag, " generacion  aptitud  promedio \n");

/*spawn*/ iniciar();
spawn objetivo();
/*spawn*/ mejores();

while(generacion<NUMGENS)
      {
      generacion++;
    spawn objetivo();
    spawn seleccion();
        /*spawn*/ cruza();
        /*spawn*/ mutacion();
        spawn elite();
        reporte();
      }
```

```
fprintf(salidag,"\n\n /***** Proceso evolutivo terminado ******/\n");
fprintf(salidag,"\n Mejor individuo(gen): \n");

for (i = 0; i < NUMVAR; i++)
   {
   fprintf (salidag,"\n var(%d) = %3.3f",i,pob[TAMPOB].gen[i]);
   }
fprintf(salidag,"\n\n Mejor aptitud = %3.3f",pob[TAMPOB].aptitud);
fclose(salidag);
printf("/****** Proceso evolutivo terminado, ver archivo: salidag.txt ******/\n");
sync;
return 0;
}
/*****************************************************************************/
```