SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de La Paz

INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
MAESTRÍA EN SISTEMAS COMPUTACIONALES

IMPLEMENTACIÓN DE UN CLUSTER DE GPGPU PARA ALGORITMOS DE OPTIMIZACIÓN

T E S I S

QUE PARA OBTENER EL GRADO DE
MAESTRO EN SISTEMAS COMPUTACIONALES

PRESENTA:
JESÚS IVÁN MERCADO BAREÑO

DIRECTOR DE TESIS:
MSC. ILIANA CASTRO LIERA

LA PAZ, BAJA CALIFORNIA SUR, MÉXICO, AGOSTO 2015.

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

TECNOLÓGICO NACIONAL DE MÉXICO
Instituto Tecnológico de La Paz

"2015, Año del Generalísimo José María Morelos y Pavón"

La Paz, B.C.S., **07/Agosto/2015**

DEPI/421/2015

Asunto: Autorización de impresión.

C.JESÚS IVÁN MERCADO BAREÑO,
ESTUDIANTE DE LA MAESTRÍA EN
SISTEMAS COMPUTACIONALES,
P R E S E N T E.

Con base en el dictamen de aprobación emitido por el Comité Tutorial de la Tesis denominada: **"IMPLEMENTACIÓN DE UN CLUSTER DE GPGPU PARA ALGORITMOS DE OPTIMIZACIÓN"** mediante la opción de tesis (Proyectos de Investigación), entregado por usted para su análisis, le informamos que se **AUTORIZA** la impresión.

A T E N T A M E N T E
"Ciencia es Verdad, Técnica es Libertad"

M.A.T.I. LUÍS ARMANDO CÁRDENAS FLORIDO,
JEFE DE LA DIV. DE EST. DE POSGRADO E INV.

ESTADOS UNIDOS MEXICANOS
INSTITUTO TECNOLÓGICO DE LA PAZ
DIVISIÓN DE ESTUDIOS DE POSGRADO
E INVESTIGACIÓN

C.c.p. Coordinación de la Maestría.
C.c.p. Depto. de Servicios Escolares.

LACF/fkso*

SEP
SECRETARÍA DE EDUCACIÓN PÚBLICA

Tecnológico Nacional de México
Instituto Tecnológico de La Paz

DICTAMEN DEL COMITÉ TUTORIAL

SUBDIRECCIÓN ACADÉMICA
DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN.

La Paz, B.C.S., **07/AGOSTO/2015**

C. M.A.T.I. LUÍS ARMANDO CÁRDENAS FLORIDO,
JEFE DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN,
P R E S E N T E.

Por medio del presente, enviamos a usted dictamen del Comité Tutorial de tesis para la obtención del grado de Maestro, con los siguientes datos generales:

No. de Control M12310024	Nombre JESÚS IVÁN MERCADO BAREÑO
Maestría en:	SISTEMAS COMPUTACIONALES
Título de la tesis: IMPLEMENTACIÓN DE UN CLUSTER DE GPGPU PARA ALGORITMOS DE OPTIMIZACIÓN	
DICTAMEN: Se autoriza el trabajo de investigación, en virtud de que realizó las correcciones correspondientes conforme a las observaciones planteadas por este Comité Tutorial.	

A t e n t a m e n t e.

El Comité Tutorial

MATI. LUIS ARMANDO CÁRDENAS FLORIDO

MC. JESÚS ANTONIO CASTRO

MSC. ILIANA CASTRO LIERA

c.c.p. Coordinador de la Maestría.
c.c.p. Departamento de Servicios Escolares.
c.c.p. Estudiante.

DEDICATORIA

A mi madre, por su amor infinito, por inculcarme valores, por ser el pilar y artífice de la persona que hoy soy.

AGRADECIMIENTOS

A mi Madre por su apoyo incondicional.
A mis primos, tíos y amigos Karina, Ramón (JR), Alejandra, Rubén, tía Mary y tío Rubén por alentarme cada día a culminar mi objetivo.

A mi Maestra Iliana Castro Liera quien me brindó su apoyo incondicional, me compartió su conocimiento y experiencia. Soporte fundamental para mi proyecto. Por ser mi tutora, maestra y socióloga, por alentarme cada día a culminar mi objetivo, maestra mi agradecimiento sincero.

A los maestros Marco Antonio Castro Liera y Jesús Castro por apoyarme y compartir sus conocimientos para soportar mi proyecto.

A la planta docente de la Maestría en Sistemas Computacionales, por nutrirme de herramientas para el desarrollo de este proyecto.

RESUMEN

En el presente trabajo se describe la configuración y puesta en marcha de un cluster de GPGPU, la implementación de aplicaciones paralelas conjugando dos arquitecturas de paralelización (MPI) de memoria distribuida y (CUDA) de memoria compartida sobre algoritmos de optimización basados en poblaciones: (PSO) optimización por enjambre de partículas y (AG) algoritmos genéticos. Se analizó la eficiencia y eficacia de los resultados entre los dos algoritmos de optimización. Se comparó el desempeño contra otra solución propuesta en 2011, teniendo como resultado una marcada mejora tanto en tiempo de ejecución como en calidad de la solución.

ABSTRAC

This work describes the configuration and commissioning of a GPGPU cluster, deploying parallel applications combining two architectures of parallelization (MPI) distributed memory and (CUDA) shared memory for optimization algorithms based on populations: (PSO) particle swarm optimization and (GA) genetic algorithms. The efficiency and effectiveness of the results between the two optimization algorithms are analyzed. The performance was compared against another solution proposed in 2011, resulting in a marked improvement in both runtime and quality of the solution.

CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPITULO 1.

Introducción.

1.1 INTRODUCCIÓN.

Con el paso del tiempo la sociedad ha tenido que aprender a resolver problemas que se le presentan en el acontecer de la vida cotidiana, con la finalidad de progresar y asegurar la existencia de la misma.

La capacidad que las personas hemos desarrollado para la resolución de los problemas se ha venido dando empíricamente con la observación y experimentación de soluciones.

En la actualidad muchos problemas cotidianos son resueltos aún de esa manera por la naturaleza de los mismos y por que la complejidad de éstos se lo permite.

Sin embargo, existen infinidad de problemas que no pueden ser resueltos de manera sencilla y se ha tenido que hacer uso de herramientas que ayuden a encontrar rápidamente la solución a dichos problemas.

Los equipos de cómputo son herramientas que se crearon a partir de la necesidad de la resolución de problemas de manera más rápida. El tipo de problemas que se abordaron con esta herramienta son básicamente los cálculos aritméticos. Hoy día las computadoras son referente y esenciales en muchos rubros de la vida cotidiana.

Las computadoras han ido evolucionando y esto conlleva a que han incrementado sus prestaciones como: procesadores con mayor capacidad de cálculo, más rápidos, equipos con más de un procesador, mayor cantidad de memoria, han reducido tamaño y costos. Este aumento en sus prestaciones hace posible que ahora las computadoras personales también puedan ser herramientas para tareas de cómputo científico.

Existen equipos llamados “supercomputadoras” los cuales son creados específicamente

con el propósito de realizar *Cómputo de Alto Rendimiento.* Estos equipos son grandes y muy costosos solamente accesibles por muy pocas instituciones privilegiadas en su presupuesto. El rendimiento de estos equipos se mide mediante la cantidad de cálculos por segundo que pueden realizar (flop/s), siendo hasta Noviembre del 2013 el equipo *Tianhe-2* de la Universidad Nacional de Tecnología de Defensa en China, la supercomputadora más poderosa con un rendimiento de 33.86 petaflop/s (cuatrillones 10^{24} de cálculos por segundo). [1]

Desde las últimas dos décadas las "supercomputadoras" se basan en computadoras paralelas; esto es, equipos que cuentan con mas de un procesador que trabajan en conjunto y cooperativamente en la resolución de un problema.

En la actualidad la mayoría de los equipos de cómputo cuentan con mas de un procesador o núcleos de ejecución. Incluso hasta los dispositivos móviles son ya multinúcleo, por lo que la programación paralela debería ser un estándar en la jerga de los programadores al optimizar los recursos de los equipos.

El cómputo científico es una de las ramas de la computación que más se beneficia del cómputo de alto rendimiento, al mismo tiempo que alienta a seguir desarrollando nuevas y mejores estrategias de paralelización.

El cómputo de alto rendimiento, además de implementarse en supercomputadoras, también tiene como alternativa las arquitecturas paralelas de memoria distribuida consistentes de clusters de computadoras, que son una alternativa de menor costo y, por lo tanto, mas accesibles.

Un cluster es un conjunto de computadoras interconectadas a través de una red, que trabajan de forma cooperativa mediante protocolos de paso de mensajes para emular una computadora paralela. [2]

Dentro de las arquitecturas paralelas de memoria compartida de bajo costo actualmente se encuentran las GPGPU, que son las Unidades de Procesamiento Gráfico de Propósito General. Estos dispositivos gráficos tienen una alta capacidad de cómputo dado que inicialmente fueron diseñadas para ejecutar videojuegos que demandaban muchos recursos, de tal modo que los fabricantes les añadieron un gran número de Unidades Aritmético Lógicas (ALUS) y también su propia memoria.

El número de Unidades de Procesamiento de la GPGPU superan con creces a los de un procesador multinúcleo, así pues, podemos ejecutar un número mayor de procesos simultáneos.

Los científicos e investigadores pronto dieron cuenta del poder de cómputo que estas tarjetas gráficas tenían y empezaron a usarlas para sus investigaciones. Sin embargo tenían la desventaja de que al ser diseñadas para videojuegos los resultados eran representados por un conjunto de pixeles.

NVIDIA es la empresa que atendió esta necesidad y empezó a fabricar tarjetas gráficas que además del rendimiento para videojuegos tuvieran la capacidad de procesamiento de propósito general, a la par que diseñó el lenguaje CUDA C para ese fin.

Nuestro interés en este proyecto es amalgamar ambas estrategias de paralelización y ofrecer una alternativa para problemas masivamente paralelos.

Por otro lado, la optimización es una las tareas que cotidianamente las personas realizamos de manera nata, pues siempre realizamos tareas tratando de minimizar el tiempo que empleamos, minimizar costos o bien maximizar beneficios o ganancias.

Muchos de estos problemas de optimización son resueltos empíricamente o bien con

herramientas que ya están diseñadas para ello.

Pero también existen una variedad de problemas que son lo bastante complejos en donde el número de variables que intervienen son demasiadas y no es posible resolverlos con herramientas tradicionales.

Ejemplo de este tipo de problemas son los llamados problemas de optimización continuos, en los cuales el espacio de búsqueda de solución puede ser muy grande. Para este tipo problemas existen los métodos heurísticos que buscan en conjunto definido de soluciones posibles mediante una función objetivo y operadores. Son los únicos que pueden emplearse para tratar de resolver los problemas para los que no existe un algoritmo eficiente para la solución del mismo o bien no se puede hacer una búsqueda exhaustiva del espacio completo, debido a sus dimensiones. Por tanto, no se garantiza la obtención de un óptimo global; pero se pueden obtener buenas aproximaciones que en muchos casos son satisfactorios para los investigadores. [3]

Dentro de este tipo de algoritmos heurísticos se encuentran los basados en poblaciones. En particular nos centraremos en los Algoritmos Genéticos y los Algoritmos de Optimización por Enjambre de Partículas, dado que han sido ampliamente estudiados. Además, existen en el Instituto Tecnológico de La Paz proyectos donde han sido objeto de estudio con diferentes estrategias, contra las cuales analizamos nuestra propuesta.

Los Algoritmos de Optimización por Enjambre de Partículas se basan en las bandadas de aves o cardúmenes de peces que siguen siempre al mejor individuo de su colonia. Una partícula sigue el mismo comportamiento y guarda una posición actual, una posición mejor histórica y tiene una función para determinar su velocidad. Con estos datos, la partícula se mueve dentro de su espacio de búsqueda definido.

Los Algoritmos Genéticos están basados en lo que conocemos sobre el proceso de la evolución natural de las especies y se basan en operadores como lo son la selección, el cruce y mutación para obtener "individuos" que se aproximen a una solución satisfactoria.

En este trabajo se conjugaron dos estrategias de paralelización, configurando una plataforma para cómputo de alto rendimiento a bajo costo sobre la cual se implementaron algoritmos de optimización para analizar su rendimiento frente a otras estrategias.

1.2 OBJETIVO.

Generar aplicaciones que implementen paralelización en un cluster de GPGPU, mediante la conjunción de las librerías de MPI y CUDA para algoritmos de optimización basados en poblaciones.

1.3 OBJETIVOS ESPECÍFICOS.

- Analizar el grado de paralelización que puede ser implementado en un cluster de GPGPU.
- Desarrollar las aplicaciones paralelas implementando CUDA y MPI, tanto en Optimización por Enjambre de Partículas como en Algoritmos Genéticos.
- Analizar la eficiencia y eficacia de la solución propuesta.
- Comparar el desempeño entre Optimización por Enjambre de Partículas y Algoritmos Genéticos en la solución propuesta.
- Comparar el desempeño de la solución propuesta contra las existentes.

- Contra un Cluster PVM.
- Contra una sola GPU.

1.4 HIPÓTESIS.

Se aumentará la eficacia de los resultados al poder procesar mayor número de individuos o partículas, según sea el caso, con igual o mayor eficiencia temporal que las soluciones ya existentes.

1.5 IMPORTANCIA DE LA INVESTIGACIÓN.

La investigación propuesta es importante porque abordamos la optimización de problemas generales mediante métodos heurísticos que ayudan a maximizar la eficiencia/eficacia de los resultados, minimizando el tiempo de la búsqueda de soluciones. Hasta ahora, el desarrollo de aplicaciones paralelas que hacen uso de los recursos multiproceso han tenido gran auge para resolver el problema de optimización, y mas recientemente con el desarrollo de aplicaciones paralelas en GPGPU, con resultados bastante satisfactorios. Si una sola GPGPU nos da resultados eficientes, comparados con otras arquitecturas de paralelización, imaginemos la calidad de los resultados y su eficiencia en una arquitectura de múltiples GPGPUS trabajando colaborativamente. Cabe señalar que la técnica propuesta puede ser adoptada para optimizar en distintas ramas de la investigación.

1.7 CONTRIBUCIÓN AL CONOCIMIENTO.

La investigación propuesta generará conocimiento en el ámbito de procesamiento paralelo, ya que conjugará dos herramientas para paralelización. Una de ellas es CUDA

(que se utiliza para paralelismo en las GPU) y la otra el manejo de paralemismo en un cluster a través de librerías de paso de mensajes, particularmente MPI. Estas herramientas se emplearán en algoritmos de optimización basados en poblaciones, específicamente algoritmos genéticos y optimización por enjambre de partículas, y se aplicarán por primera vez en el Instituto Tecnológico de La Paz para que sirvan como fuente de comparación contra soluciones de optimización existentes que se han estado desarrollado recientemente con otras metodologías.

CAPITULO 2.

Arquitecturas Paralelas de Bajo Costo.

2.1 MEMORIA COMPARTIDA

2.1.1 Cuda.

Las aplicaciones paralelas en un principio fueron desarrolladas y ejecutadas en equipos muy sofisticados y costosos a los que muy pocas instituciones tenían acceso. En la actualidad, los equipos personales cuentan con procesadores multinúcleo y algunos más también con tarjetas gráficas con capacidad de cómputo de propósito general.

Las GPGPU (General Purpose Computing on Graphics Processing Units) son actualmente una alternativa con gran auge en el ámbito de la Programación Paralela, dado su bajo costo y la gran capacidad de cómputo que tienen al incorporar muchas unidades Aritmético Lógicas (ALU) en su diseño.

En 2001 NVIDIA lanzó la gama de tarjetas GeForce 3 las cuales tenían, además de las funciones de procesamiento gráfico, la capacidad de ejecutar instrucciones de propósito general [4]. Los programadores "*engañaban*" a la GPU ejecutando instrucciones que daban como resultado pixeles y los interpretaban para problemas de propósito general.

Esto dio pie a que en 2006 NVIDIA lanzara la arquitectura CUDA (Compute Unified Device Architecture) con su gama de tarjetas GeForce 8800 GTX que tienen componentes específicamente diseñados para que la GPU soporte Cómputo de Propósito General.

En el mismo año se lanzó el compilador **CUDA C** para el que NVIDIA tomó el estándar de C y le agregó funciones específicas para programación paralela . Con ello pretendían captar el mayor número de desarrolladores posibles y ser una herramienta sencilla y poderosa en el cómputo paralelo [5].

2.1.2 Arquitectura CUDA

La arquitectura CUDA y su lenguaje CUDA C hacen posible que el desarrollador pueda implementar programas que comuniquen al CPU con la GPU y aprovechar sus múltiples núcleos para ejecutar simultáneamente varias tareas. [4]

El rendimiento de la GPU se puede comparar contra el rendimiento de un CPU en cuanto a la cantidad de operaciones de punto flotante por segundo (FLOP/s) y por la cantidad memoria que pueden leer. [6]

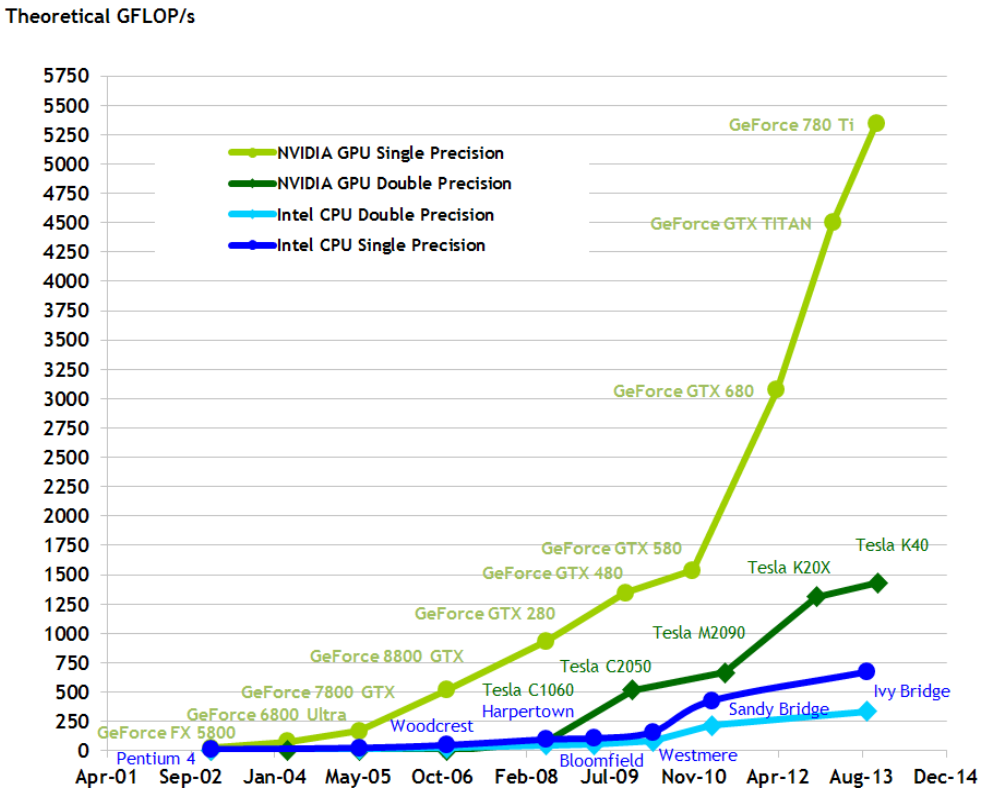

Figura 2.1 Operaciones de Punto Flotante por Segundo

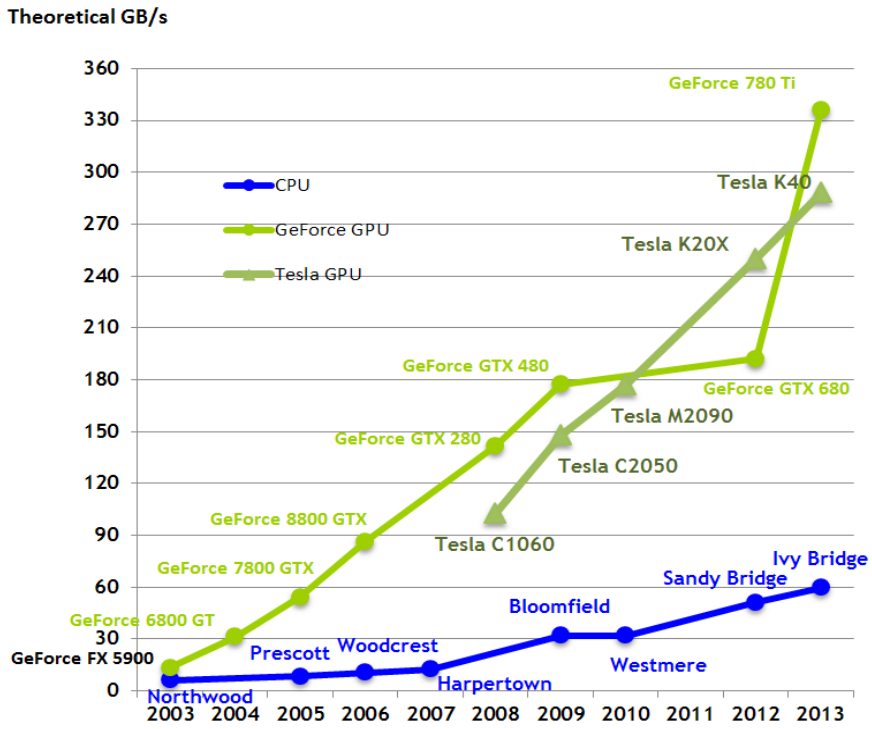

Figura 2.2 Cantidad de Memoria Leída.

En las figuras 2.1 y 2.2 se muestra que el desempeño de una GPU versus de CPU es considerablemente mayor y esto se debe a que las GPU están diseñadas para cómputo intensivo y altamente paralelo pues cuentan con mayor cantidad de unidades aritmético-lógicas (ALU´s) dedicadas a procesar información en lugar de almacenar datos en caché y a controlar el flujo de datos como se observa en la figura 2.3 [6].

La latencia por accesos a memoria disminuye y, puesto que cada unidad de procesamiento cuenta con sus propias localidades de memoria, no es necesario contar con un control de flujo sofisticado.

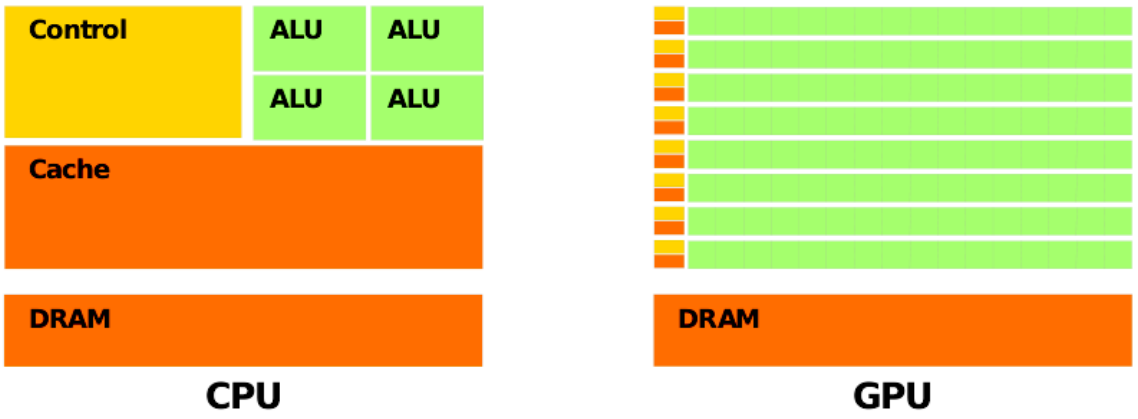

Figura 2.3 Comparativa entre CPU y GPU

La arquitectura CUDA cuenta con los siguientes elementos:

- **Threads, hilos:** Donde se ejecutan las funciones. Los hilos pueden procesarse simultáneamente y cuentan con memoria local para almacenar datos.
- **Blocks, bloques:** Un conjunto de hilos que se ejecutan en un multiprocesador. Cada bloque cuenta con una zona de memoria, denominada compartida, accesible para todos sus hilos.
- **Grid, malla:** Es un conjunto de bloques, cada malla se ejecuta sobre un GPU distinto y cuenta con memoria accesible para todos los bloques que la componen.
- **Host:** El CPU.
- **Device:** El GPU.

Los elementos de la arquitectura CUDA se muestran en la figura 2.4

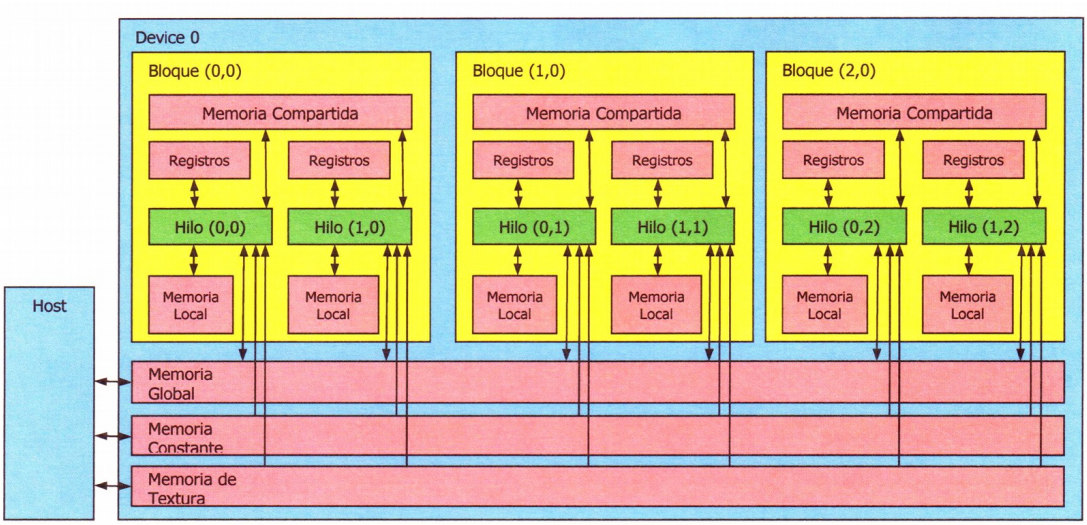

Figura 2.4 Arquitectura CUDA

Para resolver una tarea, el programador deberá analizar y determinar la manera de subdividir el problema en sub-problemas que puedan agruparse en hilos y bloques y lanzarlos en paralelo, esto es, determinar el grado de paralelización que puede ser de **grano fino** orientado a datos o bien de **grano grueso** orientado a tareas o hilos.

De acuerdo a la **Taxonomía de Flynn** [7-9], la arquitectura CUDA se clasifica como **SIMD** (Single Instruction Multiple Data).

Otra característica de la arquitectura CUDA es la **escalabilidad automática,** que consiste en que una aplicación paralela puede ser ejecutada en distintos modelos de GPU, lanzando un número de bloques simultáneos para ejecutarse en los multiprocesadores disponibles. Por ende, la aplicación paralela se ejecutará en menor tiempo en un GPU con mayor número de multiprocesadores que en uno con menor cantidad. Como lo podemos observar en la figura 2.5

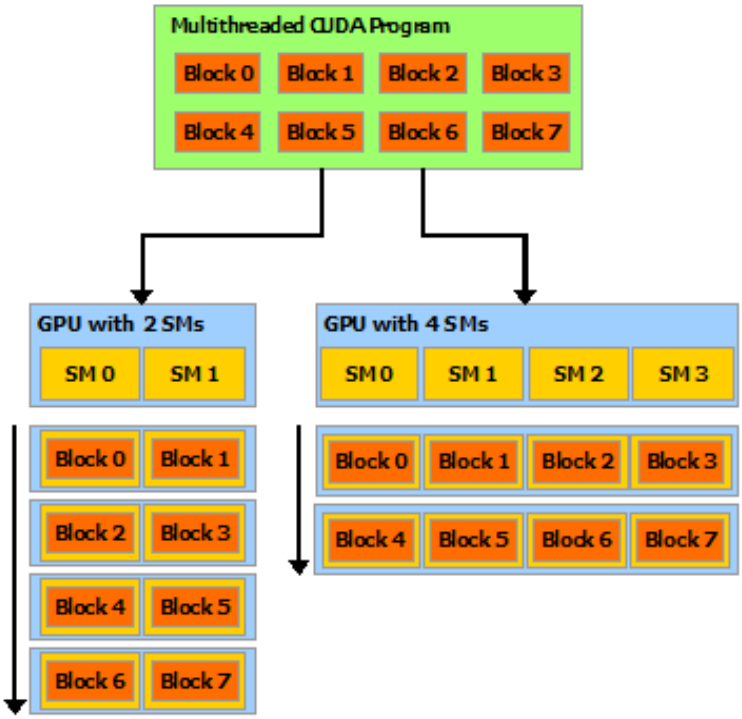

Figura 2.5 Escalabilidad Automática.

2.1.3 Instalación de CUDA en Fedora Linux.

La versión de CUDA que elegimos para este trabajo es el toolkit CUDA 5.5 debido a que es una versión reciente y tiene mejoras respecto a sus versiones previas.

Un requisito previo a la instalación es que el equipo debe tener instalada una tarjeta de video NVIDIA con soporte para CUDA. Los modelos habilitados para CUDA se pueden consultar en la liga: https://developer.nvidia.com/cuda-gpus.

La instalación sobre Linux Fedora se detalla en los siguientes pasos:

1. Tener instalado un compilador C/C++, en nuestro caso es GCC 4.7.
2. Descargar desde http://www.geforce.com/drivers/results/77525 el driver correspondiente al modelo de tarjeta de video, para Linux de 64 bits. En este trabajo para la tarjeta GeForce GTX 670
3. Descargar el CUDA toolkit 5.5 específicamente para Fedora Linux, desde la liga https://developer.nvidia.com/cuda-toolkit-55-archive.
4. Desde la terminal, como superusuario, se debe detener el servidor X para desactivar el entorno gráfico. Se invoca el nivel de corrida 3:

```
# init 3
```

5. Se debe desactivar el driver Nouveau para la tarjeta de video que, por default, instala el sistema operativo:

   a) Editar el archivo de configuración (si no existe, crearlo):

   ```
   # nano /etc/modprobe.d/blacklist.conf
   ```

   b) Añadir las directivas de configuración:

   ```
   blacklist nouveau

   options nouveau modeset=0
   ```

   c) Evitar que se cargue desde el arranque del sistema mediante los comandos:

   ```
   # cp /boot/initramfs-$(uname -r).img /boot/initramfs-$(uname
   -r).img.bak


   # dracut -f /boot/initramfs-$(uname -r).img $(uname -r)
   ```

6. Asignar permisos de ejecución al archivo de instalación del driver de la tarjeta de video e instalarlo:

```
# chmod 777 NVIDIA-Linux-x86_64-331.49.run

# ./NVIDIA-Linux-x86_64-331.49.run
```

7. Asignar permisos de ejecución al archivo de instalación de CUDA toolkit 5.5 e instalarlo. Elija "No instalar driver de desarrollo":

```
# chmod 777 cuda_5.5.22_linux_64.run

# ./cuda_5.5.22_linux_64.run
```

8. Agregar las rutas de binarios y librerías de Cuda 5.5. a las variables entorno:

```
/usr/local/cuda-5.5/bin     --> agregarla a la variable PATH

/usr/local/cuda-5.5/lib64 --> agregarla a la variable
LD_LIBRARY_PATH
```

9. Desde terminal y como superusuario instalar las siguientes librerías:

```
# yum -y install freeglut*

# ln -s /usr/lib/libglut.so.3 /usr/lib/libglut.so
```

10. Verificar la instalación compilando y ejecutando un ejemplo:

```
# /root/NVIDIA_CUDA-5.5_Samples/make

# cd /root/NVIDIA_CUDA-5.5_Samples/1_Utilities/DeviceQuery

#./DeviceQuery
```

2.1.4 Funciones y Gestión de Memoria en CUDA.

Las funciones que pueden ser ejecutadas en la GPU pueden ser invocadas desde el

Host y desde la GPU y se debe definir su cualificador:

- __device__ : Cualificador para las funciones que serán invocadas desde la GPU. El tipo de datos de retorno puede ser distinto de **void.** La invocación, parámetros y cuerpo de la función sigue las normas de C estándar.

- __global__ : Cualificador para las funciones que serán invocadas desde el Host. A estas funciones también se le conocen como ***kernels.*** El tipo de dato de retorno debe ser **void.** La invocación para los kernels de ser de la forma:

```
nombre_kernel<<<BLK,THR,MEMS,S>>> (parámetros);
```

donde:

  - **BLK:** Es el número de bloques que serán lanzados.

  - **THR:** El número de hilos lanzados por cada bloque.

  - **MEMS:** Parámetro opcional para especificar la cantidad de memoria compartida adicional a la asignada por defecto.

  - **S:** Parámetro opcional para especificar el tipo de flujo.

La gestión de la memoria en CUDA es un aspecto importante que debemos abordar, pues existen distintos tipos de memoria con un comportamiento diferente, las cuales son accedidas por hilos y bloques específicos.

A continuación se listan los diferentes tipos de memoria, así como los cualificadores de variables que residen en ellas:

- **Memoria Global:** Puede ser accedida por todos los hilos de todos los bloques del GPU. Para las variables que residan en dicha memoria se debe usar el

cualificador __device__. El tiempo de vida es el tiempo de la aplicación.

- **Memoria Compartida:** Pueden ser accedidas por todos los hilos de un mismo bloque. Para las variables que residen aquí se utiliza el cualificador __shared__. El tiempo de vida es el mismo tiempo que el del bloque. Este cualificador se puede utilizar junto con __device__. Las variables que son definidas con este cualificador no pueden inicializarse en su declaración. Si son utilizadas variables con este cualificador es necesario sincronizar los hilos después de cada operación de escritura en ellas mediante la instrucción __syncthreads().

- **Memoria Constante:** Son accesibles para todos los hilos de todos los bloques de la GPU. El cualificador para las variables que residen aquí es __constant__. Opcionalmente se puede usar junto con el cualificador __device__. La definición de estas variables se hace en el device y la inicilización solo puede hacerse desde el host.

- **Memoria Local:** Es accesible solo por el hilo donde esta declarada, cada hilo lanzado en la GPU cuenta con su memoria local. Al no ser especificado ningún cualificador las variables residen en memoria local.

La velocidad de acceso a la memoria en el GPU depende de su tipo y jerarquía como se observa en la figura 2.6:

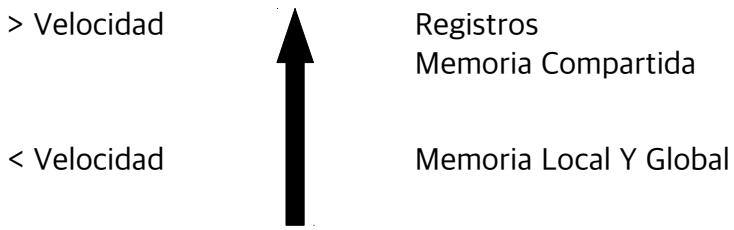

Figura 2.6 Jerarquización de memoria y velocidad de acceso.

2.1.5 Compilación y ejecución de programas en CUDA.

Para obtener un archivo ejecutable en CUDA se utiliza el compilador **nvcc** de NVIDIA, que tiene el propósito de invocar al compilador de C/C++ instalado en nuestro equipo Linux (gcc en nuestro caso), pero hace transparente al desarrollador el ligado de librerías y rutas específicas de CUDA.

La estructura básica de compilación en la línea de comandos es la siguiente:

```
#nvcc <fuente> -o <ejecutable>
```

donde :

<fuente>: Es el archivo fuente con extensión .cu que contiene el código CUDA que vamos a compilar.

<ejecutable>: Archivo de salida ejecutable que obtendremos de la compilación.

-o: Es la bandera que indica que el archivo es de salida.

Ejemplo:

```
# nvcc holamundo.cu -o holamundo
```

esto nos genera el archivo ejecutable **holamundo.** Para ejecutarlo usamos:

```
# ./holamundo
```

2.2 MEMORIA DISTRIBUIDA.

El cómputo paralelo consiste en la ejecución de múltiples instrucciones de un programa simultáneamente en varias unidades de procesamiento [7]

Una alternativa para implementar la programación paralela son los sistemas de Memoria Distribuida, que consisten en un conjunto de nodos interconectados en red que colaborativamente intercambian y procesan información. Los nodos son unidades independientes con procesador y memoria local. Solo los procesadores locales tienen acceso directamente a la memoria local en cada nodo. Por tanto, si un nodo requiere acceder a la memoria de otro nodo, se hace con el modelo de comunicación de Paso de Mensajes. Dentro del conjunto de nodos a uno de ellos se le asigna la tarea de iniciar, distribuir tareas y recopilar resultados de ejecución de la aplicación paralela, este nodo se le conoce como *maestro* [7], [8].

A esta alternativa de programación paralela, se le denomina cluster de computadoras y es muy utilizado en las instituciones educativas debido a su bajo costo.

Sin embargo, los sistemas de memoria distribuida, pese a múltiples beneficios, también presentan limitantes que debemos que tener en cuenta como la ***latencia en el medio de comunicación,*** que puede generar un cuello de botella respecto del ancho de banda de la memoria y procesadores locales, retrasando el tiempo total de ejecución de la aplicación paralela.

Por lo general asumimos que, a mayor cantidad de nodos en el sistema distribuido, mejor será el desempeño de nuestra aplicación paralela. Pero no siempre es así, pues al tener mayor cantidad de nodos es mayor la comunicación entre ellos y, por consiguiente, pueden producirse retrasos temporales.

Para medir el desempeño tenemos dos factores:

- **El Factor de Aceleración:**

  Que se obtiene dividiendo el tiempo de ejecución logrado con el mejor

algoritmo secuencial en un sistema de un solo procesador, entre el tiempo de ejecución de un sistema de varios procesadores [7], [9]

Este factor se obtiene mediante (2.1)

$$S(p)=\frac{t_s}{t_p} \tag{2.1}$$

donde:

p Es el número de procesadores en el sistema.

t_s Es el tiempo de ejecución del mejor algoritmo secuencial con un solo procesador.

t_p Es el tiempo de ejecución en el sistema con varios procesadores.

Lo ideal sería tener un factor de aceleración lineal, sin embargo, hay factores que influyen en el desempeño como son la latencia en el medio, procesadores desocupados, partes del algoritmo que no se pueden paralelizar.

- **El Factor de Eficiencia.**

En términos computacionales la Eficiencia es realizar una tarea con un buen resultado y empleando la menor cantidad de recursos. En nuestro caso particular, el tiempo de procesamiento.

En sistemas que emplean paralelización la Eficiencia se denota por (2.2):

$$E=\frac{S(p)}{p} \tag{2.2}$$

Donde:

p Es el número de procesadores del sistema.

$S(p)$ Es el factor de aceleración.

El resultado hay que multiplicarse por 100 para obtener una escala en porcentaje. La Eficiencia nos indica el porcentaje efectivo de la ocupación de los procesadores en tiempo de procesamiento y, por ende, el desempeño de nuestro sistema.

Existen varias alternativas para la implementación del modelo de comunicación de Paso de Mensajes como son: **PVM (Parallel Virtual Machine)** y **MPI (Message Passing Interface) .**

En el presente trabajo se decidió emplear el estándar MPI debido a que:

- Es mas reciente que PVM.
- Se puede generar código más compacto que con PVM.
- En versiones recientes de CUDA, existen librerías para trabajo conjunto con MPI.

2.2.1 Estándar MPI.

MPI es una estandarización de las librerías de paso de mensajes que fue creada por el *MPI-FORUM,* un grupo formado por investigadores de universidades, laboratorios y empresas involucrados en la computación de altas prestaciones. Se abocaron a estandarizar las principales funciones de las diferentes librerías que existían para el paso de mensajes y lograron presentar un primer estándar en Noviembre de 1993.

MPI, como estándar, es portable y originalmente tenía librerías para ser implementaciones en lenguaje C, C++ y Fortran. Actualmente también hay librerías

para Java, Phyton, Perl entre otros.

Existen varias implementaciones del estándar, por ejemplo, empresas como Intel, IBM, Cray Research tienen sus propias versiones del estándar; también existen implementaciones de software libre como son : MPICH y OpenMPI.

En el presente trabajo se optó por la implementación OpenMPI debido a que en sus versiones recientes cuenta con librerías para poder compilarse con librerías CUDA.

2.2.2 Configuración del Cluster.

El cluster está conformado por 6 equipos ubicados en el Laboratorio de Programación Paralela, con las siguientes características:

- Procesador Intel Core I5 @ 3.3 Ghz
- 4 GB Memoria RAM
- Disco Duro de 465 Gb.
- Tarjeta de Red Ethernet 1 Gb/s.
- GPU Nvidia GTX 670.
    - 2 GB Ram.
    - 7 Multicores.
    - 1344 Cuda Cores.

La conectividad consta de 1 Switch Gigabit Ethernet y cableado categoría 6.

En lo que respecta al S**oftware:**

Decidimos utilizar el sistema operativo **Fedora Linux 17 de 64 bits** debido a que en el Instituto Tecnológico ya se había utilizado esta plataforma en trabajos previos con buenos resultados.[10]

Otra razón es que los cluster "*Beowulf*", basados en software libre, están teniendo mayor auge y confiabilidad en proyectos de investigación con equipos de bajo costo. [11]

Debido a que el cluster puede ser afectado por factores como la temperatura de los equipos, la ejecución de plataformas gráficas y los mensajes que llegan a través de la red, fue necesario hacer los siguiente ajustes:

- En cada equipo se configuró una tarjeta de red virtual con dirección IP de rango distinto al segmento de la institución, con el fin de aislar el cluster de la red institucional y evitar recibir mensajes broadcast que afectan negativamente el desempeño del cluster. En particular se utilizaron direcciones virtuales del rango 10.0.0.x

- Es necesario seleccionar un equipo maestro en el que se iniciará la ejecución en los equipos esclavos y donde se recibirán y almacenarán los resultados de cada uno de ellos.

- En cada equipo debemos configurar el archivo **/etc/hosts** con los nombres y direcciones IP correspondientes a cada equipo, como se indica a continuación:

```
#  cat /etc/hosts
10.0.0.1  clcuda01
10.0.0.2  clcuda02
10.0.0.3  clcuda03
```

```
10.0.0.4  clcuda04

10.0.0.5  clcuda05

10.0.0.6  clcuda06
```

- Se debe crear un archivo de texto que contenga el nombre de los equipos que van a ejecutar el trabajo de cluster, al que se hará referencia cuando se ejecute la aplicación (como se detalla en la sección 2.2.5). Por ejemplo, si queremos que los 6 equipos ejecuten una prueba, deberemos tener un archivo de texto con el siguiente contenido:

```
# cat equipos

clcuda01

clcuda02

clcuda03

clcuda04

clcuda05

clcuda06
```

- Es necesario instalar las librerías para envío de mensajes en cada equipo del cluster. En nuestro caso utilizamos el estándar MPI (Message Passing Interface) en particular la distribución **OpenMPI 1.7.2,** pues ésta versión y las posteriores tienen la característica de poder compilar librerías para trabajo con CUDA. En la sección 2.2.3 se muestran con detalle los pasos para instalar OpenMPI 1.7.2.

- Para ejecutar la aplicación en el cluster es necesario que el equipo maestro tenga acceso al resto de los equipos a través del protocolo SSH sin tener que escribir la contraseña por cada equipo. Sin embargo, debemos hacer notar

enfáticamente que: para trabajar en la versión OpenMPI 1.7.2 y posteriores (al igual que el equipo maestro) **es necesario que todos los equipos en el cluster tengan acceso a todos los demás mediante SSH obviando la contraseña.** Cuando se ejecuta una aplicación paralela en mas de 3 equipos, se delegan funciones en otros equipos para que también manden ejecutar la aplicación hacia otros. Se debe generar la llave pública de cada equipo y se va agregando al archivo de llaves autorizadas de tal modo, que al final, el archivo de llaves autorizadas contenga la llave pública de cada equipo. Esta situación no viene explícita en la documentación oficial de OpeMPI [12] y fue precisada por la experimentación en el desarrollo de este proyecto.

2.2.3 Instalación de OpenMPI en Linux.

Una vez realizados los pasos descritos en la sección 2.2.2 detallamos la instalación de **OpenMPI 1.7.2.**

1. Tener instalado un compilador C/C++. En nuestro caso es GCC 4.7
2. Descargar los archivos de instalación correspondientes a la versión 1.7.2 de OpenMPI desde el siguiente enlace:

   http://www.open-mpi.org/software/ompi/v1.7/.
3. Descomprimir el archivo openmpi-1.7.2.tar.gz en una carpeta, de tal modo que los archivos queden en la carpeta openmpi-1.7.2.
4. Ubicar los archivos **cuda.h** y **libcuda.so,** correspondientes a librerías de CUDA. En nuestro caso están ubicados en :

   ```
   /usr/local/cuda-5.5/       --> para cuda.h
   ```

```
/usr/lib64/              --> para libcuda.so
```

5. Debemos estar ubicados en de la carpeta donde se descomprimieron los archivos de OpenMPI:

```
#  cd /openmpi-1.7.2
```

6. Configurar los parámetros para OpenMPI antes de compilar y enlazar las librerías de CUDA, para prever el posible uso de una librería conjunta *cuda-aware-MPI,* mediante la siguiente instrucción:

```
./configure –prefix=/usr/local/openmpi –with-cuda=/usr/local/cuda-5.5 –with-cuda-libdir=/usr/lib64
```

7. Una vez configurados los parámetros necesarios, debemos compilar e instalar los archivos binarios mediante la instrucción:

```
./make all install
```

8. Debemos agregar las rutas de binarios y librerías de openmpi a las variables de entorno:

```
/usr/local/openmpi/bin  --> agregarla a la variable PATH

/usr/local/openmpi/lib  --> agregarla a la variable LD_LIBRARY_PATH
```

2.2.4 Funciones de OpenMPI.

MPI, como estándar, tiene variedad de funciones para trabajo con procesamiento paralelo. En lo que se refiere al trabajo con el lenguaje C, las funciones están definidas en el archivo de cabecera **mpi.h,** por tanto siempre es necesaria su inclusión en los programas.

Las funciones esenciales para un programa con MPI son :

`int MPI_Init(init *argc, char ***argv)` para inicializar la aplicación paralela.

`int MPI_Finalize(void)` para finalizar la aplicación paralela.

`int MPI_Comm_size(MPI_Comm comm, int *size)` para determinar el número total de procesos lanzados.

Donde:

- **comm** es el comunicador seleccionado. Por lo regular se usa un comunicador universal.
- **size** es valor del total de procesos lanzados.

`Int MPI_Comm_rank (MPI_Comm comm, int *rank)` para determinar el identificador de cada proceso lanzado.

Donde:

- ***rank*** es el valor del identificador del proceso.

El tipo de dato de retorno de las funciones es **int** y el valor esperado es **MPI_SUCCESS**, que indica que la función fue realizada satisfactoriamente.

Existen varias funciones para la comunicación entre procesos, las hay para comunicación colectiva y comunicación entre pares de procesos (uno a uno). La comunicación puede ser con bloqueo o no. Así que tenemos distintas posibilidades de envío y recepción de mensajes.

Las funciones de comunicación mas sencillas entre procesos son:

`MPI_Send(void* buf, int count, MPI_Datatype datatype,int`

```
dest, int tag, MPI_Comm comm)
```

Se utiliza para el envío de mensajes. Con bloqueo.

```
MPI_Recv(void* buf, int count, MPI_Datatype datatype, int
source, int tag, MPI_Comm comm, MPI_Status *status);
```

Para la recepción de mensajes con bloqueo.

Los parámetros:

- ***buf*** es el apuntador de inicio al buffer de envío o bien el apuntador hacia al buffer donde se almacenará el mensaje recibido.
- ***count*** es el número de elementos a enviar y recibir.
- ***datatype*** es el tipo de dato definido que se envía y recibe.
- ***dest*** en la función de envío es el identificador del proceso al que vamos a enviar el mensaje.
- ***source*** en la función de recepción es el identificador del proceso del que esperamos recibir el mensaje.
- ***tag*** se puede definir una etiqueta para diferenciar distintos mensajes o bien, si esto no es primordial, se puede definir con *MPI_ANY_TAG* en la función de recepción.
- ***status*** en la función de recepción almacena información acerca del estado en que se recibió un mensaje, como son la etiqueta, el origen y el tamaño.

Los tipos de datos definidos en el estándar MPI se muestran (2.1):

Tipos MPI	Equivalente en C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Sin equivalente

Tabla 2.1 Tipos de datos definidos en MPI

2.2.5 Compilación y Ejecución de Programas con OpenMPI

Para obtener un archivo ejecutable MPI en necesario utilizar la orden **mpicc** para fuentes C ó **mpicxx** para fuentes C++. Es un *guión* que tiene como función ejecutar el compilador de C/C++ de nuestro sistema, pero que facilita la ubicación de los archivos de cabecera y librerías necesarios en la creación del archivo objeto [13].

La estructura básica en es la siguiente:

```
#  mpicc –o <ejecutable> <fuente>
```

Donde:

<ejecutable> es el archivo de salida ejecutable

<fuente> es el archivo del programa con extensión .c

-o: Es la bandera con que indica el archivo de salida.

Ejemplo:

```
#  mpicc –o mpiholamundo mpiholamundo.c
```

Para la ejecución la aplicación MPI se utiliza el comando **mpirun** y la sintaxis básica es la siguiente:

```
#  mpirun  -np <número de procesos> ./<ejecutable>
```

donde:

<número de procesos> indica cuantos procesos ejecutará la aplicación paralela.

<ejecutable> Es el archivo ejecutable.

Existen distintas opciones para ejecutar una aplicación paralela. Por ejemplo:

```
#  mpirun –np 4 ./mpiholamundo
```

Ejecuta el programa **mpiholamundo** en 4 procesos en un solo equipo.

```
#  mpirun  ./mpiholamundo
```

Ejecuta el programa en un solo proceso en un solo equipo.

```
#  mpirun  -np 6 –hostfile equipos ./mpiholamundo
```

Ejecuta el programa **mpiholamundo** en 6 procesos, pero se distribuyen entre los equipos que están listados en el archivo **equipos.**

2.2.6 Compilación y Ejecución de Programas con MPI-CUDA.

En la sección 2.1.5 y 2.2.5 se revisó como obtener archivos ejecutables en CUDA y MPI independientemente. Para el proyecto, es necesario que enlacemos cada archivo en un archivo de ejecución principal que se ejecutará en el cluster.

Se compila el archivo "maestro" para obtener el archivo objeto:

```
mpicxx -o maestroag.o -c maestroag.cpp
```

Se compila el archivo "cuda" para obtener el archivo objeto:

```
nvcc  -I /usr/local/openmpi/include -o agcuda.o -c agcuda.cu
```

Finalmente debemos enlazar ambos archivos en uno principal:

```
mpicxx   -o  mainag  maestroag.o  agcuda.o   -L"/usr/local/cuda-5.5"/lib64
-lcudart
```

El archivo resultante se ejecutará en el cluster conforme se revisó en la sección 2.2.5.

CAPITULO 3.

Algoritmos de Optimización.

3.1 OPTIMIZACIÓN

La optimización es una tarea que la sociedad realiza día con día y muchas veces de manera nata siempre estamos buscando la manera de maximizar las utilidades invirtiendo la menor cantidad de recursos posibles o bien minimizar el tiempo en que realizamos alguna tarea para obtener beneficios. La optimización busca valores para parámetros bajo ciertas restricciones para satisfacer alguna condicionante.

El término ***optimizar*** se usa indistintamente de los términos ***maximizar y minimizar;*** maximizar una función *f* es equivalente a minimizar la función *–f.*

De acuerdo a la manera en la que son modelados, los problemas de optimización pueden clasificarse como: [4], [14]

- Lineales: Modelo lineal. Para resolverlos, se utiliza la programación lineal.
- *No lineales: Son generalmente bastante difíciles de modelar y resolver. Requieren el uso de técnicas de programación paralela.*

Existen varios tipos de algoritmos de optimización de propósito general, entre otros[3]:

- Búsqueda aleatoria
- Escaladores de colinas
  - Clásico
  - Estocástico con Mejor Acceso
  - Estocástico con Primer Acceso
- Métodos derivados de un solo punto

- Escalador de colinas con reiniciación
- Recocido ó enfriamiento simulado
- Búsqueda tabú

- Métodos con poblaciones de puntos.
  - Basados en mutaciones
  - Basados en cruzamiento: Algoritmos Genéticos.
  - Algoritmos de estimación de distribuciones.
  - Enjambre de Partículas

Como vemos, la optimización es una área muy amplia e interdisciplinaria la cual se resume a buscar mejoras, solución y/o aproximación a resolución de problemas.

Para este trabajo elegimos los métodos basados en poblaciones en particular Enjambres de Partículas y Algoritmos Genéticos dado que queremos analizar y comparar nuestra propuesta contra soluciones anteriores que utilizan métodos basados en poblaciones.

3.2 OPTIMIZACIÓN POR ENJAMBRE DE PARTÍCULAS

3.2.1 Optimización por Enjambre de Partículas.

La Optimización por Enjambre de Partículas (PSO, Particle Swarm Optimization), es un método metaheurístico proporcional basado en poblaciones que fue propuesto por primera vez por Kenedy y Eberhart [15], [16].

El método PSO está inspirado en el comportamiento social del vuelo de las bandadas de aves y el movimiento de los cardúmenes de peces. Consiste en mantener una población de partículas, cada una de las cuales es una solución potencial y está descrita por los siguientes vectores:

$\vec{x}_i$ Posición actual.

$\vec{v}_i$ Velocidad actual.

$\vec{y}_i$ Mejor posición histórica.

La mejor posición histórica de la partícula, cuando se trata de minimizar, es el valor mas bajo que ha tenido la posición de la partícula de todos aquellos que ha tomado.

El movimiento de las partículas durante la ejecución del algoritmo se da dentro de un espacio de búsqueda acotado y este movimiento está determinado por su mejor posición histórica y por la mejor posición de un conjunto de partículas, como se muestra en la figura 3.1.

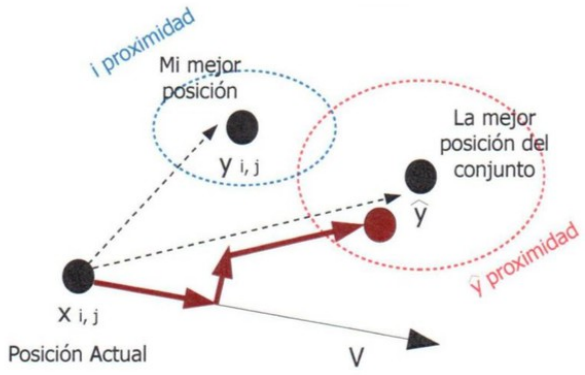

Figura 3.1 Representación Gráfica del movimiento de una partícula.

El algoritmo utiliza dos parámetros aleatorios independientes $r_1 \sim U(0,1)$ y $r_2 \sim U(0,1)$, donde U es una distribución uniforme. Estos valores son escalados por las constantes C_1 y C_2, para las que se sugiere un valor en el intervalo de (0,2] [14], [17]

A estas constantes se les conoce como coeficientes de aceleración y representan el grado en que variará la posición de la partícula en cada iteración. La velocidad se actualiza por separado para cada dimensión $j \in 1..n$, de tal manera que $v_{i,j}$ denota $j_{ésima}$ dimensión del vector de velocidad asociado con la $i_{ésima}$ partícula. En (3.1) se muestra la ecuación para actualizar la velocidad:

$$v_{i,j}(t+1) = v_{i,j}(t) + c_1 r_{1,j}(t)[y_{i,j}(t) - x_{i,j}(t)] + c_2 r_{2,j}(t)[\hat{y}_j(t) - x_{i,j}(t)] \quad (3.1)$$

De acuerdo a (3.1) se observa que c_2 regula cuanto avanza la partícula hacia la mejor partícula del enjambre y c_1 regula el avance hacia la mejor posición histórica de la partícula.

El valor de $v_{i,j}$ se limita al rango $[-v_{max}, v_{max}]$ para reducir la posibilidad de que la partícula abandone el espacio de búsqueda. Si el espacio de búsqueda se define en el rango $[-x_{max}, x_{max}]$, entonces el valor de v_{max} se establece de tal manera que $v_{max} = k \times x_{max}$, donde $0.1 \leqslant k \leqslant 1.0$ [18]

La posición de cada partícula se actualiza utilizando su vector de velocidad, como se muestra en (3.2):

$$x_i(t+1) = x_i(t) + v_i(t+1) \quad (3.2)$$

El algoritmo consta de los siguientes pasos:

1. Inicializar la posición de cada partícula p_i, estableciendo cada una de sus coordenadas x_j de manera aleatoria y uniformemente distribuidas en el intervalo $[-x_{max}, x_{max}]$, donde $i \in 1..s$, s es el tamaño del enjambre, $j \in 1..n$ y n representa la cantidad de dimensiones del problema a resolver.

2. Inicializar el vector de velocidad v_j para cada partícula p_i asignando valores aleatorios uniformemente distribuidos en el intervalo $[-v_{max}, v_{max}]$, para las $i \in 1..s$, partículas y las $j \in 1..n$ dimensiones.

3. Establecer las mejores posiciones históricas $y_i = x_i \forall i \in 1..s$.

4. Repetir para todas las partículas p_i en el enjambre s:

   a. Calcular su velocidad y nueva posición, según (3.1) y (3.2).

   b. Actualizar su mejor posición histórica.

   c. Actualizar la mejor posición del enjambre.

   d. Cada cierta cantidad de iteraciones, envía la información de la mejor partícula del enjambre hacia un enjambre vecino.

5. Hasta que se cumpla la condición de parada.

Generalmente, la condición de parada se refiere a una cantidad establecida de iteraciones, a las que denominaremos generaciones. Figura 3.2.

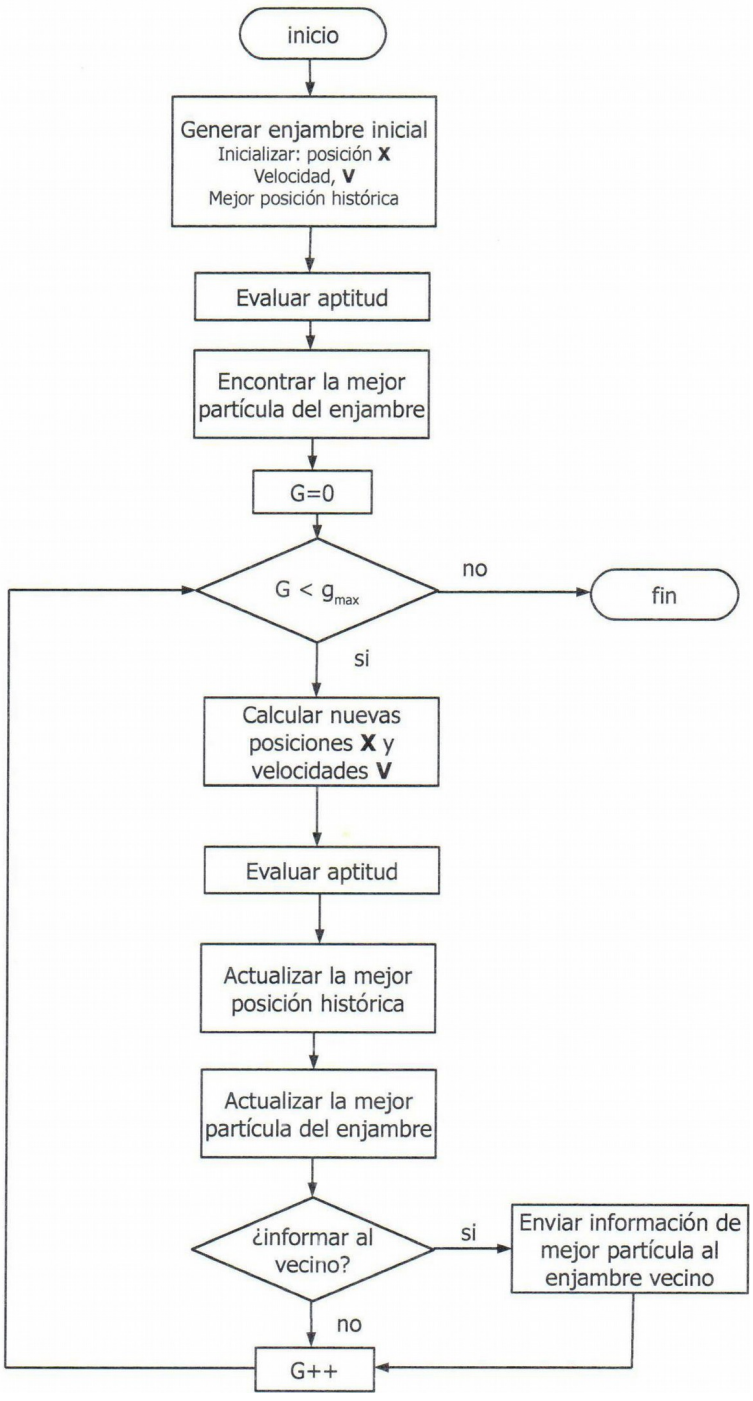

Figura 3.2 Algoritmo de PSO.

Para incrementar la tasa de convergencia de PSO se agregó el parámetro de inercia w a la fórmula para obtener la velocidad, como se muestra en (3.3):

$$v_{i,i}(t+1)=wv_{i,i}(t)+c_1r_{1,i}(t)[y_{i,i}(t)-x_{i,i}(t)]+c_2r_{2,i}(t)[\hat{y}_i(t)-x_{i,i}(t)] \quad (3.3)$$

Se han hecho muchas investigaciones para determinar el valor de los parámetros w, c_1 y c_2 que aseguren la convergencia del algoritmo; para w, Shi y Eberhart [19] sugieren un valor $w\in[0.8,1.2]$ y conjuntamente con Clerc [20], [21], sugieren $c_1,c_2=1.4692$.

3.2.2 PSO Paralelo.

Dependiendo del tamaño de B y de la forma en que se intercambia información entre los conjuntos, la paralelización del algoritmo es [22], [23]

- **De grano grueso**, también conocido como el modelo de islas en el cual se procesan varios enjambres con intercambio ocasional de partículas entre ellos.
- **De grano fino**, que consiste en un solo enjambre organizado en una malla ortogonal donde los vecinos intercambian información.

Si se utiliza paralelización de grano grueso, la actualización de los parámetros de velocidad y posición de la partícula i en un enjambre k se hace hasta que se actualizaron los de la partícula $i-1$. La paralelización consiste en que se hace esa actualización simultáneamente en todos los enjambres que se hayan definido.

En la implementación de grano fino, se actualizan simultáneamente los parámetros de velocidad y posición de todas las partículas del enjambre.

No obstante que la paralelización del algoritmo disminuye notablemente el tiempo de ejecución del algoritmo, a mayor grado de paralelización son necesarias mayores cantidades de iteraciones para que el algoritmo converja [17].

3.2.3 Implementación de PSO.

Para la implementación del algoritmo de PSO se utilizó el lenguaje C++ con el compilador GCC 4.7, las librerías del Toolkit de CUDA 5.5 y las librerías de OpenMPI 1.7.2 para el paso de mensajes entre los nodos del cluster, sobre el sistema operativo Linux Fedora 17 de 64 bits.

El cluster está conformado por 6 equipos intel Core i5 y sus características fueron descritas en la sección 2.2.2. Cada equipo cuenta con una tarjeta gráfica NVIDIA GeForce GTX 670 con 2 GB de memoria global, 7 multiprocesadores, 1344 cuda cores.

Para la paralelización del problema se utilizó la siguiente estrategia:

- Se definió un Enjambre global, el cual está distribuido en los 6 nodos del cluster.
- En cada nodo se definieron 21 Subenjambres de 256 partículas, a cada subenjambre le corresponde ser procesado por un bloque. Esto se ilustra mejor en la figura 3.3
- Hay un hilo de ejecución por cada partícula de cada bloque que actualiza sus parámetros, como se ilustra en la figura 3.3
- Cada nodo tiene 5376 partículas y, por consiguiente, nuestro enjambre global tiene 32256 partículas que son procesadas paralelamente.

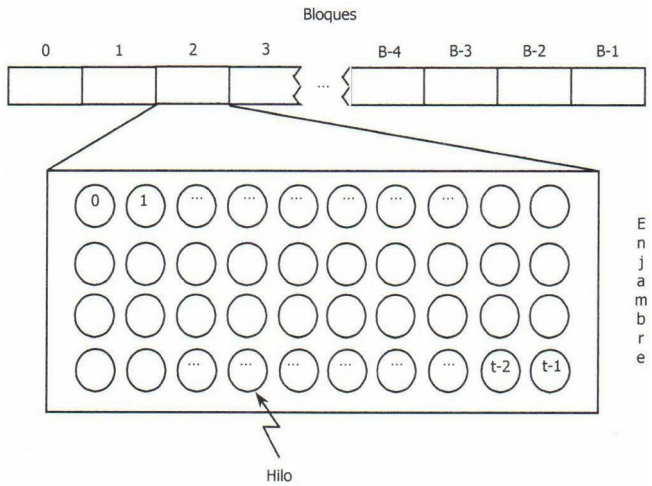

Figura 3.3 Un enjambre de *t* partículas se procesa en un bloque. Cada partícula se procesa por un hilo.

- Se definió un periodo migratorio entre bloques de cada 2% del total de las generaciones y un periodo migratorio entre nodos de cada 4% del total de las generaciones. Esto es que, por cada dos veces que se realiza migración entre bloques, ocurre una migración entre nodos.

- La migración entre bloques se realiza de manera circular, recibiendo la información de la mejor partícula del bloque siguiente, como se ilustra en la Figura 3.4

- La migración entre nodos también es de forma circular. Se recibe la información de la mejor partícula del nodo siguiente. Para migrar entre nodos se copia la información de la partícula desde el device hacia host, una vez en el host se pasa por mensaje al nodo siguiente y se hace la operación inversa se copia desde el host hacia el device para colocarla en el lugar que le corresponde y seguir con las iteraciones del algoritmo.

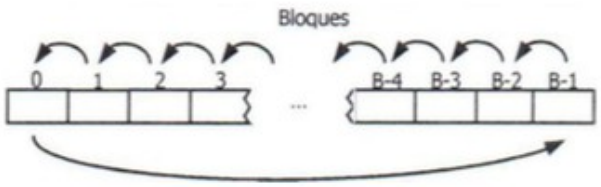

Figura 3.4 Migración entre bloques.

- Se definió un nodo maestro que tiene la función de mandar la instrucción a los esclavos para que inicien la ejecución del algoritmo con sus partículas (incluso el mismo) y al termino recibe las mejores de cada esclavo y determina la mejor del enjambre global.

3.5 ALGORITMOS GENÉTICOS.

3.5.1 Algoritmos Genéticos.

Los algoritmos genéticos (AG) son algoritmos de búsqueda heurística que fueron propuestos originalmente por Holland [24] y están inspirados en la evolución natural de las especies, en las teorías de Darwin y Mendel. El libro de Holland "Adaptaciones en Sistemas Naturales y Artificiales" [25] se presentó por primera vez el Algoritmo Genético en 1975. A partir de ese momento los algoritmos genéticos son estudiados mas ampliamente para aplicarlos a la búsqueda de soluciones a diversos problemas.

El concepto central de los algoritmos genéticos es el de **población**, que se define como un conjunto de individuos C_i donde $i=1,..,\mu$. Donde μ es el tamaño de la población.

Existen diversas formas de representar a los individuos en problemas de optimización y, por lo general, depende de la naturaleza del problema en donde se están empleando.

En la versión original de Holland conocida como “Algoritmo Genético Simple”, los individuos se representan como cadenas binarias, conocidas como cromosomas, en las que cada bit, al que se denomina 'gen', representa una respuesta a una proposición lógica sobre una de las características de de la solución candidato al problema.

En la versión con representación real del AG, cada individuo es un vector de números reales que representan un punto en el espacio de búsqueda R^n.

$$C=[x_1, x_2, \ldots, x_{n-1}, x_n] \quad \forall j \mid j \in [1, n],\ x_j \in R \tag{3.4}$$

Los algoritmos genéticos inician con una población P_g generada de forma aleatoria. Posteriormente, de manera iterativa, se calcula la aptitud de sus individuos y se aplican los operadores genéticos para determinar a los individuos que formarán la siguiente población P_{g+1}, donde g es el número de generaciones o iteraciones que se han ejecutado. Generalmente, el criterio de parada es el arribo a una generación g_{max}, fijada antes de iniciar el algoritmo.

La aptitud de los individuos se evalúa por medio de la función de aptitud $f(C)$ y permite medir el grado en que el individuo cumple con el objetivo del problema de optimización. Antes de iniciar el algoritmo, el diseñador establece valores mínimo f_{min} y máximo f_{max} para $f(C)$, que marcarán los límites de rechazo para soluciones candidato con desempeño muy deficiente.

3.5.2 Operadores Genéticos.

Los operadores genéticos que son aplicados a la población son:

- **Selección:** La selección es la operación que nos sirve para escoger a los

individuos que formarán parte de la siguiente generación. Existen varios tipos de selección entre los cuales podemos mencionar: selección por torneo, selección por ruleta, selección directa. Dependerá de cada tipo de selección que los individuos con mejor aptitud aparezcan más de una vez en la siguiente generación y, por el contrario, los menos aptos puedan no aparecer.

- **Selección Proporcional (Ruleta):** Requiere que la función de aptitud no genere valores negativos. Este método toma la aptitud relativa p_i como la probabilidad del individuo de ser seleccionado. Para determinar cuáles individuos se seleccionarán para formar parte de la siguiente generación, puede utilizarse una variable aleatoria uniforme en el intervalo (0,1) y asignar porciones de este rango a cada individuo proporcionales a su aptitud relativa. Se generan μ valores aleatorios correspondientes a la misma cantidad de individuos en la siguiente población.[4]

- **Selección por torneo:** Consiste en insertar en la nueva población al individuo con mayor adaptación de una muestra aleatoria de z individuos de la población actual. A z se le conoce como tamaño del torneo. Este método tiene la ventaja de que no requiere valores positivos para la función de aptitud. El proceso de torneo se repite μ veces para generar una nueva población de igual tamaño que la población actual.

  La medida en que los individuos con mayor aptitud tienen una mayor probabilidad de formar parte de la siguiente población, se conoce como presión de selección. A mayor tamaño de torneo, mayor presión de selección.

Una mayor presión de selección tiende a hacer que el algoritmo converja mas rápidamente, pero se incrementa la probabilidad de que converja hacia un óptimo local. Este fenómeno se conoce como "convergencia prematura". [4].

- **Selección Directa:** Consiste en ordenar primero la población y dependiendo del porcentaje de selección determinado antes del inicio del algoritmo, serán los n primeros individuos de nuestra población P los que pasen a la siguiente generación. Con este método se garantiza que solo los más aptos estén en la generación siguiente, aunque esto conlleva un costo en el tiempo de ejecución ya que en cada generación debemos ordenar nuestra población. [26].

- **Cruzamiento:** Para generar nuevo material genético se eligen al azar dos individuos (padres) y se combinan sus cromosomas, dando lugar así a la formación de nuevos individuos. Cuando se realiza cruce en poblaciones con representación discreta. La Fig. 3.5 nos ilustra esto de mejor manera.

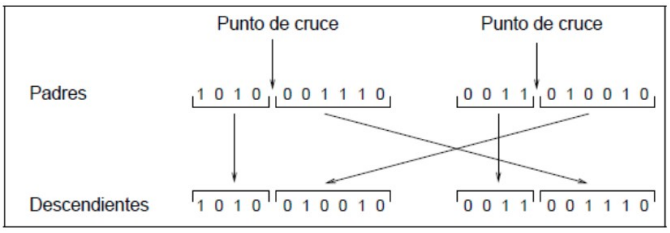

Figura 3.5: Operador de cruce representación discreta

Pero cuando la representación es real, se tiene un escalar λ el cual tiene valores del intervalo de (0,1) y con esto nos permite generar vectores con valores proporcionales de los padres :

$$C_1^{'} = \lambda\, C_1 + (1-\lambda) C_2 \tag{3.5}$$

$$C_2^{'} = (1-\lambda) C_1 + \lambda\, C_2 \tag{3.6}$$

Se define la probabilidad de cruce p_c como el cociente entre la cantidad de individuos generados por medio de este operador y el tamaño de la población μ [4].

- **Mutación:** Este operador es otra manera de generar cambios genéticos en los individuos pero primero debemos saber si un individuo debe o no ser mutado. Esto lo hacemos generando aleatoriamente un valor entre (0 y 1) y lo comparamos con el valor que le asignamos inicialmente a la Pm (probabilidad de mutación), de tal forma que si el valor generado es menor o igual a la probabilidad de mutación este individuo es alterado para generar uno nuevo, y si es mayor que la probabilidad de mutación no sufre alteraciones.

  Para aquellos individuos que les corresponda aplicarles este operador, hay que decidir de manera aleatoria cuál de sus genes se mutará.

  En la representación real, el nuevo gen $\tilde{x}_m$ puede ser generado aleatoriamente de manera uniforme en el rango (x_{min}, x_{max}) prefijado por el diseñador.

  Opcionalmente, puede generarse el gen a mutar $\tilde{x}_m$ de manera no uniforme, aprovechando el hecho de que las soluciones candidato tienen mas probabilidad de encontrarse en la vecindad del óptimo conforme se incrementa el número de generación. En este caso se calcula $\tilde{x}_m$, haciendo que la distancia entre el

individuo original y el mutado sea en promedio menor conforme aumenta el número de generación, como se muestra en (3.6):

$$\tilde{x_m} = \begin{cases} x_m + \Delta(g, x_{max} - x_m), & si \quad h = 0 \\ x_m + \Delta(g, x_m - x_{min}), & si \quad h = 1 \end{cases} \tag{3.6}$$

donde h es un bit aleatorio (0 ó 1) que determina si el gen mutado x_m será incrementado o decrementado. El incremento Δ se calcula como:

$$\Delta(g, y) = y \cdot r \cdot \left(1 - \frac{g}{g_{max}}\right)^b \tag{3.7}$$

Donde r es un número real en el intervalo [0,1] y b es un parámetro que determina cuánto disminuye el incremento Δ conforme aumentan las generaciones.

Los algoritmos genéticos incorporan la mutación como una forma de incluir soluciones en regiones distintas del espacio de búsqueda; pero en general, la probabilidad de mutación p_m de que un individuo mute suele ser baja. [4]

3.5.3 Algoritmos Genéticos en Paralelo.

Se conoce como *regiones sub-óptimas* a aquellas regiones que tienen un mejor desempeño que sus vecinas pero que en el espacio global no son las óptimas.

Cuando los algoritmos genéticos se ejecutan en una sola computadora es probable que caigan en regiones sub-óptimas y por ende que converjan prematuramente.

En cambio cuando los algoritmos genéticos se ejecutan paralelamente existe menor probabilidad de que caigan en estas regiones sub-óptimas y que converjan

prematuramente. Otra beneficio del procesamiento en paralelo de los algoritmos genéticos es que el tiempo de ejecución se reduce notablemente al procesarse simultáneamente el cálculo de la aptitud y la aplicación de operadores en distintas poblaciones [4].

Nowostawski y Poli [27] propusieron una clasificación de algoritmos paralelos considerando los siguientes aspectos :

- 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:

- Maestro - Esclavo.
  - Síncrono.
  - Asíncrono.
- Sub-poblaciones estáticas con migración.
- Sub-poblaciones estáticas sobrepuestas.
- Algoritmos genéticos masivamente paralelos.
- Sub-poblaciones dinámicas.
- Algoritmos genéticos de estado estable.

- Algoritmos genéticos desordenados.
- Métodos híbridos.

Cada categoría puede presentar mejores rendimientos, dependiendo del problema a tratar y de la arquitectura de la máquina paralela donde se implemente la solución. En nuestro caso, el algoritmo genético es masivamente paralelo pues la estrategia de paralización en cluster aunado a las características de las GPU nos permitieron lanzar gran cantidad de hilos simultáneamente que representan a su vez a los individuos de la población global. En la Figura 3.6 se ilustra el algoritmo genético en general.

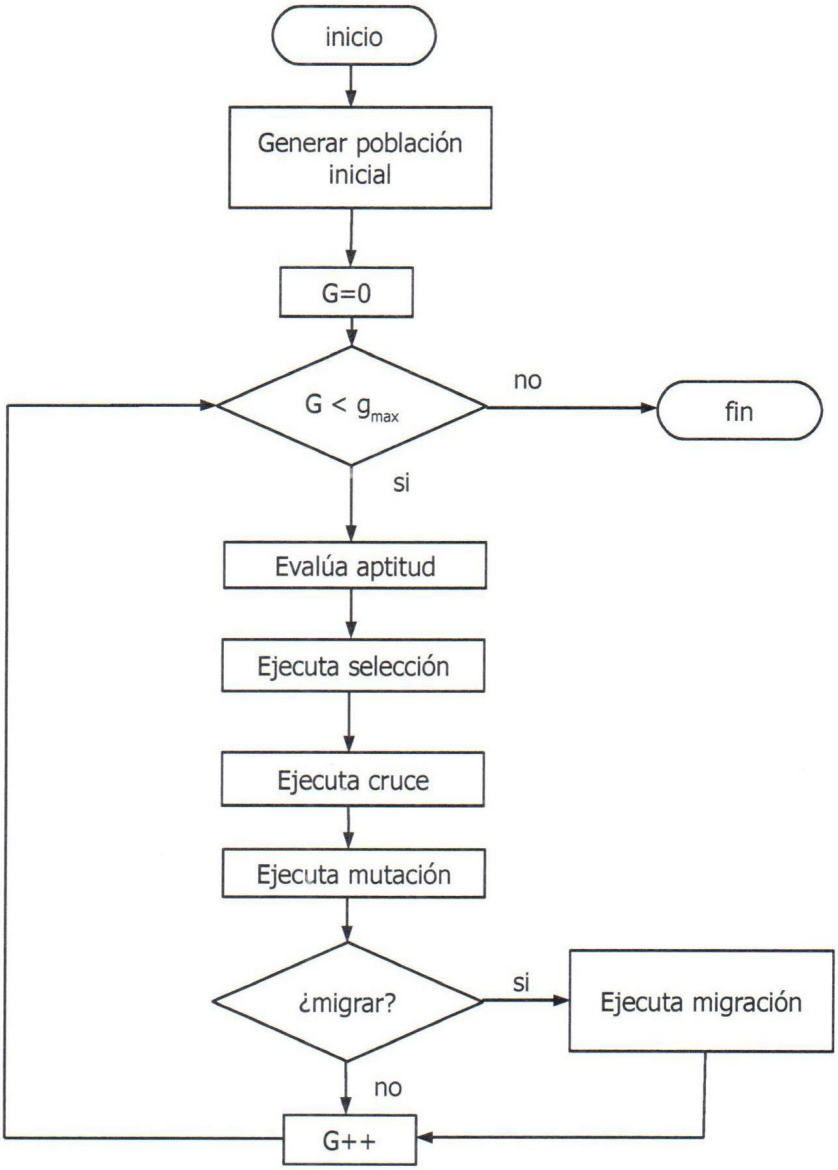

Figura 3.6: Algoritmo Genético

3.5.4 Implementación de Algoritmos Genéticos.

Para la implementación del Algoritmo Genético se utilizó el lenguaje C++ con el compilador GCC 4.7, las librerías del Toolkit de CUDA 5.0 y las librerías de OpenMPI 1.7.2 para el paso de mensajes entre los nodos del cluster, sobre el sistema operativo Linux Fedora 17 de 64 bits.

El cluster está conformado por 6 equipos intel Core i5 y sus características fueron descritas en la sección 2.2.2. Cada equipo cuenta con una tarjeta gráfica NVIDIA GeForce GTX 670 con 2 GB de memoria global, 7 multiprocesadores y 1344 cuda cores.

Para la paralelización del problema se utilizó la siguiente estrategia:

- Se definió una Población global el cual está distribuido en los 6 nodos del cluster.
- En cada nodo se definieron 14 Subpoblaciones de 64 individuos, cada subpoblación le corresponde ser procesado por un bloque. Esto se ilustra mejor en la Figura 3.7
- Hay un hilo de ejecución por cada individuo de cada bloque que actualiza sus parámetros. Como se ilustra en Figura 3.7
- Cada nodo tiene entonces 896 individuos y por consiguiente nuestra Población global tiene 5376 individuos que son procesados paralelamente.

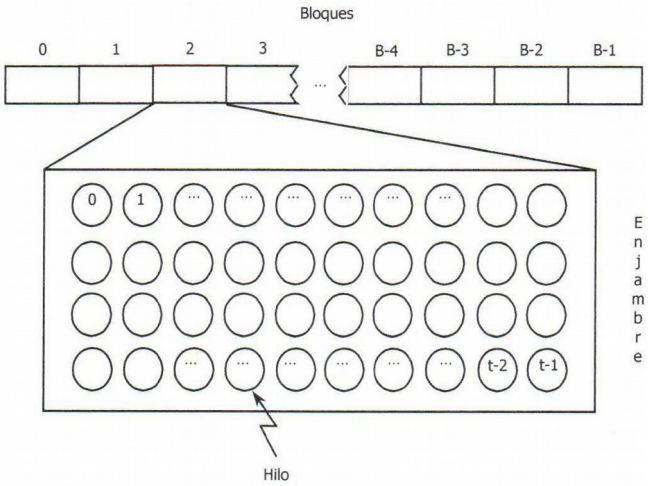

Figura 3.7 Una población es procesada en un bloque, cada hilo corresponde a un individuo.

- Se definió un periodo migratorio entre bloques de cada 10% del total de las generaciones y un periodo migratorio entre nodos de cada 25% del total de las generaciones y una tasa de migración de aproximadamente 3%. Esto es que la migración entre poblaciones tuvo una frecuencia de 10 veces y la migración entre nodos tuvo una frecuencia de 4 ocasiones del total de las generaciones

- La migración entre bloques se realiza de manera circular. Primero se ordena la población para recibir la información de los MR mejores individuos del bloque siguiente en la posición de los MR peores del bloque actual, como se ilustra en las Figuras 3.8 y 3.9

- La migración entre nodos también es de forma circular, se recibe la información del mejor individuo del nodo siguiente. Para esto es necesario copiar desde el device hacia host el individuo que se va a migrar. Una vez en el host, se envía por mensaje hacia el nodo siguiente y realiza la operación inversa. Cuando se

recibe el individuo se copia desde el host hacia el device y se coloca en el lugar que le corresponde para seguir iterando.

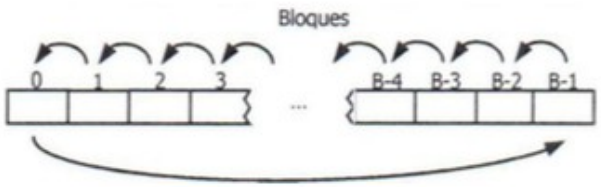

Figura 3.8 Migración entre bloques de los MR mejores del bloque siguiente.

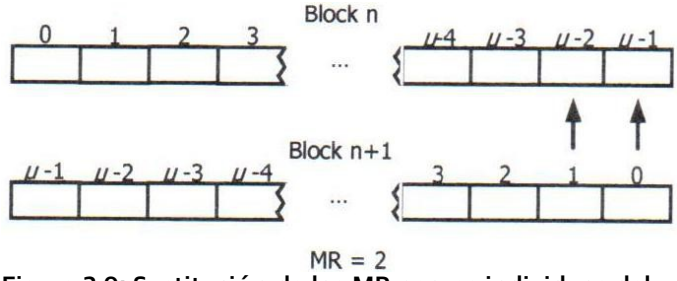

Figura 3.9: Sustitución de los MR peores individuos del bloque actual con los MR mejores del bloque siguiente.

- Se definió un nodo maestro quien tiene la función de mandar la instrucción a los esclavos para que inicien la ejecución del algoritmo con sus individuos (incluso el mismo). Al termino recibe los mejores de cada esclavo y determina el mejor de la población global.

CAPITULO 4.

Resultados y Conclusiones.

4.1 FUNCIONES DE PRUEBA.

Las funciones de prueba que se utilizaron en las implementaciones de PSO y AG son las propuestas por Suganthan et al [28] para la "Competencia de Computación Evolutiva 2005" de las cuales ya se han obtenido resultados con la implementación en un sistema con 1 GPU [4].

De la batería de 5 funciones unimodales y 7 funciones básicas propuestas, se seleccionaron las siguientes:

- **Rastrigin**:

$$f(x)=\sum_{i=1}^{D}\left(x_i^{\,2}-10\cos(2\Pi\, x_i)+10\right) \qquad D:\ dimensiones$$

Propiedades:

- Multimodal, Desplazable, Separable, Escalable.
- Gran cantidad de óptimos locales.
- $x\in[-5,5]^D$

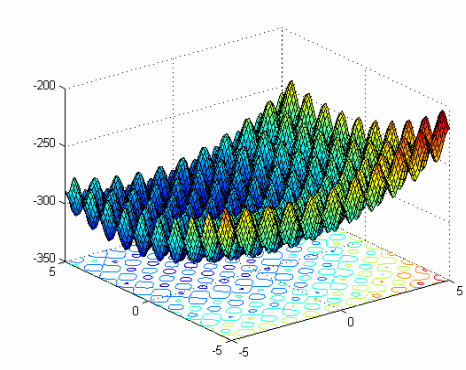

Figura 4.1 Función de Rastrigín.

- **Ackley:**

$$f(x) = -20\ exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x_i^{\,2}}\right) - exp\left(\frac{1}{D}\sum_{i=1}^{D} cos(2\,\Pi\ x_i)\right) + 20 + e$$

$$D : dimensiones$$

Propiedades:

- Multimodal, Rotable, Desplazable, No-separable, Escalable.
- Óptimo global en la frontera.
- Si se inicializa la población cerca de la frontera, el problema se resuelve fácilmente.
- $x \in [-32,32]^D$

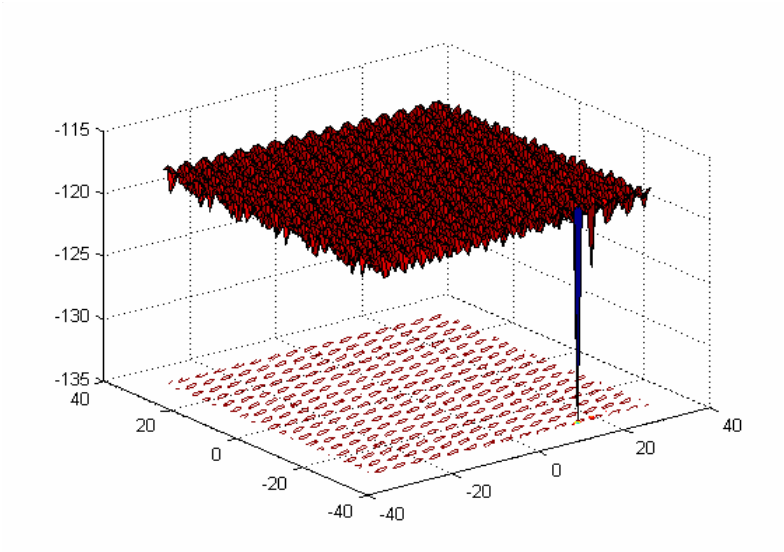

Figura. 4.2 Función de Ackley

4.2 OPTIMIZACIÓN POR ENJAMBRE DE PARTÍCULAS EN CLUSTER.

En la optimización de las funciones propuestas para Optimización por Enjambre de Partículas se utilizaron los siguientes parámetros :

Nodos	6
Enjambres por Nodo	21
Partículas por Enjambre	256
Periodo Migratorio e/ nodos	4% *
Periodo Migratorio e/ Enjambres	2% *
Dimensiones	30

Tabla 4.1 Parámetros de las funciones (PSO).

***** Expresa cada que porcentaje del total de generaciones se llevará acabo la migración.

Para 1500 muestras se obtuvieron los resultados siguientes:

Generaciones	Emax	Aciertos	Tiempo(s)
45000	1.0 e^{-6}	1499*	3.44

Tabla 4.2 Resultados para función Rastrigin para PSO en Cluster

***** Una muestra presentó un error de 2.5 e^{-6} .

Generaciones	Emax	Aciertos	Tiempo(s)
3600	1.0 e^{-6}	1500	0.33

Tabla 4.3 Resultados para función Ackley PSO en Cluster

La ***Aceleración*** y ***Eficiencia*** en el cluster evaluando la función **Ackley** se comportó de la siguiente manera:

Nodos	Tiempo (s)	Aceleración	Eficiencia
1	1.71	1.00	100.%
2	0.87	1.95	97.61%
3	0.61	2.80	93.37%
4	0.47	3.60	89.99%
6	0.33	5.14	85.65%

Tabla 4.4 Aceleración y Eficiencia Cluster (PSO)

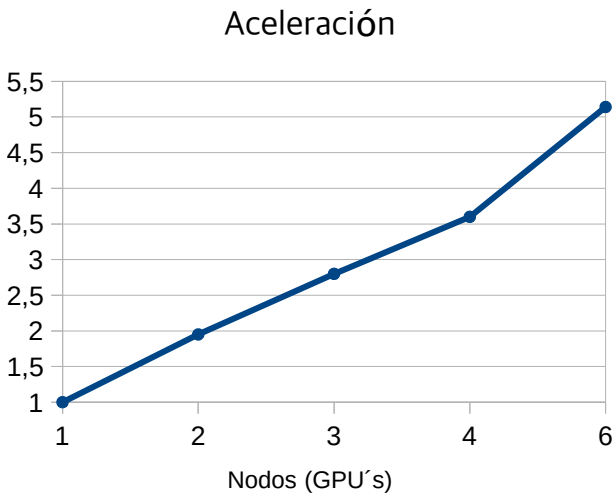

Figura. 4.3 Gráfica de Aceleración en Cluster (PSO)

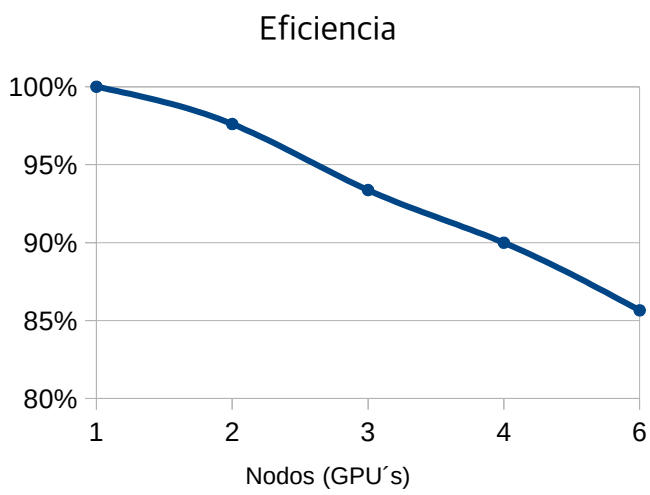

Figura. 4.4 Gráfica de Eficiencia en Cluster (PSO)

4.3 ALGORITMOS GENÉTICOS EN CLUSTER.

En la optimización de las funciones propuestas para Algoritmos Genéticos se utilizaron los siguientes parámetros:

Nodos	6
Poblaciones por Nodo	14
Individuos por Población	64
Periodo Migratorio e/ Nodos	25% *
Periodo Migratorio e/ Poblaciones	10% *
Probabilidad de Cruce	0.7
Probabilidad de Mutación	0.16
Tamaño de Torneo	2
Dimensiones	30

Tabla 4.5 Parámetros de las funciones AG.

***** Expresa cada que porcentaje del total de generaciones se llevará acabo la migración.

Para 1500 muestras se obtuvieron los resultados siguientes:

Generaciones	Emax	Aciertos	Tiempo (s)
400	1.0 e^{-6}	1500	0.78

Tabla 4.6 Resultados para función Rastrigin AG

Generaciones	Emax	Aciertos	Tiempo (s)
10000	1.0 e^{-6}	1293*	12.80

Tabla 4.7 Resultados para función Ackley AG

Los 207 resultados tienen un error por debajo de 2.5 e^{-6}

La ***Aceleración*** y ***Eficiencia*** en el cluster evaluando la función **Rastrigin** se comportó de la siguiente manera:

Nodos	Tiempo (s)	Aceleración	Eficiencia
1	3.70	1.00	100.%
2	2.25	1.64	82.27%
3	1.52	2.42	80.89%
4	1.16	3.17	79.43%
5	0.95	3.88	77.78%
6	0.78	4.69	78.31%

Tabla 4.8 Aceleración y Eficiencia Cluster (AG)

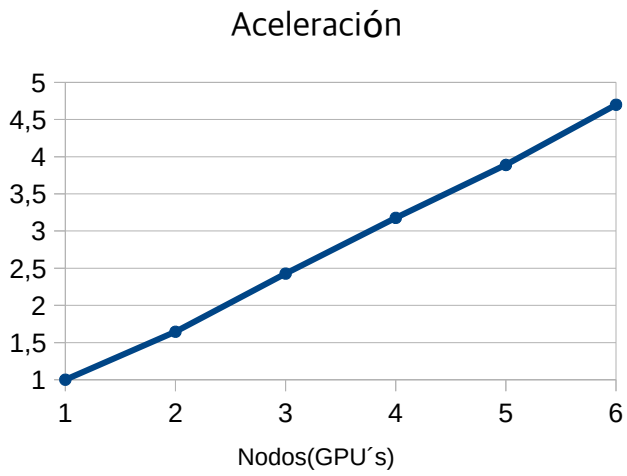

Figura 4.5 Gráfica de Aceleración en Cluster (AG)

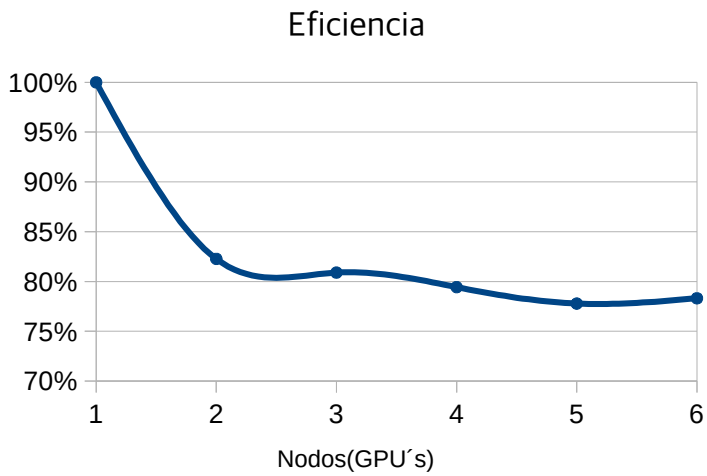

Figura 4.6 Gráfica de Eficiencia en Cluster (AG)

4.4 CONCLUSIONES.

En el presente trabajo se obtuvieron como producto las librerías que se implementaron en MPI y CUDA C, para Optimización por Enjambre de Partículas y Algoritmos Genéticos. Las librerías pueden ser modificadas de forma sencilla para servir como base

a trabajos futuros.

La hipótesis planteada en este trabajo se cumplió dado que con la conjugación de arquitecturas de paralelización propuesta, se obtuvieron resultados satisfactorios al mejorar la eficiencia (tiempo de ejecución) y la eficacia (calidad de resultados) respecto de trabajos previos [4], como se observa en las tablas 4.9 y 4.10

	Función			
	Ackley		Rastrigin	
	Emax	Tiempo (s)	Emax	Tiempo (s)
1 GPU/CUDA 2011	2.5 e $^{-6}$	1.02	1.0 e $^{-6}$	5.89
Cluster CUDA	1.0 e $^{-6}$	0.33	1.0 e $^{-6}$	3.44
Mejora		**67.64%**		**41.60%**

Tabla 4.9 Comparación de resultados de cluster de GPGPU versus 1 GPU(2011) PSO

	Función			
	Ackley		Rastrigin	
	Emax	Tiempo (s)	Emax	Tiempo (s)
1 GPU/CUDA 2011	2.5 e $^{-6}$	22.99	1.0 e $^{-6}$	4.33
Cluster CUDA	1.0 e $^{-6}$	12.80	1.0 e $^{-6}$	0.78
Mejora		**44.32%**		**81.98%**

Tabla 4.10 Comparación de resultados de cluster de GPGPU versus 1 GPU(2011) AG

Del análisis de resultados de las funciones de prueba se concluye que PSO tiene un mejor desempeño en cuanto a tiempo de ejecución en la función de ***Ackley.*** En la función ***Rastrigin*** el Algoritmo Genético tuvo mejor desempeño y debido a que Rastrigin tiene una gran cantidad de óptimos locales.

El ancho de banda del medio de transmisión es mucho menor al ancho de banda de la memoria de la GPU y el CPU. Por consiguiente, al diseñar la estrategia de paralelización se determinó una tasa de migración entre nodos pequeña para ocupar lo menos posible el medio de transmisión y no generar retardos en tiempo de ejecución, pero lo suficiente para que las poblaciones o enjambres coadyuven entre sí a encontrar soluciones en el rango establecido.

El desempeño del cluster fue satisfactorio dado que se tuvo una aceleración casi uniforme al ir aumentando el número de nodos y por ende un menor tiempo de ejecución. Sin embargo, también se debe revisar la eficiencia del sistema la cual los resultados nos indican que, al aumentar la cantidad de nodos, la eficiencia fue en decremento (aspecto comprensible dado que a mayor número de nodos habrá una mayor comunicación entre los mismos, teniendo un pequeño costo en tiempo).

La conjunción de las dos arquitecturas paralelas: de memoria distribuida y compartida (MPI y CUDA) es una alternativa rentable para el cómputo de alto rendimiento (HPC) de bajo costo, pues no es tan costoso como las supercomputadoras y los clusters de cientos de equipos diseñados ex profeso para tal propósito.

4.5 TRABAJO FUTURO.

Existen diversas alternativas para desarrollar como trabajo futuro del presente trabajo; una de ellas es el realizar una aplicación híbrida entre Optimización de Enjambre de Partículas y Algoritmos Genéticos (adecuada al cluster de GPGPU) y comparar sus resultados.

Otra alternativa sería diseñar aplicaciones para un cluster heterogéneo, donde puedan participar en el procesamiento tanto GPU´s como CPU´s en conjunto. E incluso

distintas arquitecturas de GPU.

Sería interesante implementar distintos tipos de modelo altamente paralelos en el cluster de GPGPU, como pueden ser el procesamiento de imágenes y bioinformática para su tratamiento y comparativa de resultados.

APÉNDICE 1.

Código De Funciones De Aptitud.

ARCHIVO "ACKLEY.H"

```
//Función 8, Pag 11 CEC05



#define PII 3.14159265358979323846264338327950288
#define E  2.71828182845904523536028747135266625
#define XMAX 32.0

//funcion de aptitud
__device__ float aptitud(float *x)
  {
    int i;
    float r;
    float sum1, sum2,res,D;
    sum1 = 0.0;
    sum2 = 0.0;
    D=(float)(DIMENSIONES*1.0);
    for (i=0; i<DIMENSIONES; i++)
    {
        sum1 += (x[i]+BIAS)*(x[i]+BIAS);
        sum2 += __cosf(2.0*PII*(x[i]+BIAS));
    }
    sum1 = -0.2*sqrtf(sum1/D);
    sum2 /=D;
    res = 20.0 + E - 20.0*expf(sum1) - expf(sum2);
    r =-res;
    return (r);
 }
```

ARCHIVO “RASTRIGIN.H”

```
//Función 9, Pag 12 CEC05


#define PI 3.14159265358979323846264338327950288
#define E 2.71828182845904523536028747135266625
#define XMAX 5.0

__device__ float aptitud(float x[DIMENSIONES])
{
    int i;
    float res;
    res = 0.0;
    for (i=0; i<DIMENSIONES; i++)
    {
        res += ((x[i]+BIAS)*(x[i]+BIAS) - 10.0 *
        __cosf(2.0*PI*(x[i]+BIAS)) + 10.0);
    }
    return (-res);
}
```

APÉNDICE 2.

Código De Optimización Por Enjambre De Partículas.

ARCHIVO “ARCHIVOS.H”

```
#include<mpi.h>

#define DIMENSIONES    30      //variables a encontrar
#define GMAX          3600         //generaciones
#define PMB             72     // periodo para comunicarse con el siguiente blk
#define PMN            144     //perio migratorio entre nodos
#define POBS            21     // cantidad de enjambres                    21
#define TENJ           256     // tamaño del enjambre (cantidad de
partículas)256
#define C1            1.62// constante de aceleración cognitiva
#define C2            1.62// constante de aceleración social
#define W              0.8     // peso inercial
#define EMAX          0.01     // error máximo permitido
#define BLK           POBS// en un bloque se aloja un enjambre
#define THR           TENJ// cada hilo procesa una partícula
#define TMUESTRA       1500         // cantidad de veces que se ejecuta PSO
#define TOTALH 6
#define FUNCION "ackley"
//#define FUNCION "rastrigin"

typedef struct{
     float x[DIMENSIONES];
     float v[DIMENSIONES];
     float ap;
     float mx[DIMENSIONES];
     float map;
}particula;

typedef struct{
     float x[DIMENSIONES];
     float ap;
}mejores;


extern "C" {


     void generarn(mejores *mbloque, float *rr, int neighbor, int nblo, int
nhil);
     void detmejbloque(mejores *mbloque, float *rr);
     void envianodo(int neighbor, float *particle);
     void recibenodo(float *particleOr);
}
```

ARCHIVO “MAESTRO.CPP”

```
#include "archivos.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>


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

     int Size, Rank;
     int muestra,prematuros,ceros;
     int i,j,h,destinoE,destinoR;
     float error,eprom;
     FILE *informe;
     FILE *tiempos;
     char salida[256];

     double tini,tfin;
     float ttotal,tpromedio;

//------------------------------Incializo los archivos antes que el MPI



     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &Size);
     MPI_Comm_rank(MPI_COMM_WORLD, &Rank);

     int N = THR*BLK;



     mejores *mbl = NULL;
     mbl= new mejores[BLK];

     float *resultado=NULL;
     resultado = new float[DIMENSIONES+1];

     float *rema=NULL;
     rema = new float[DIMENSIONES+1];

     float *dimej=NULL;
```

```
      dimej = new float[DIMENSIONES+1];


      int idmejor;



                              //CICLO Principal
      if (Rank == 0)   // es el maestro
      {
         prematuros=0;
         ceros=0;
         error=0.0;
         eprom=0.0;
         tpromedio=0.0;
         //Crear archivos
         printf("\n \n Empezando prueba en cluster...  %d Muestras
\n\n",TMUESTRA);
         printf("\n ---Cluster trabajando...  NO APAGAR EQUIPOS ---\n\n");
         sprintf(salida,"v2PSOgpu%d-%s-%02d-%03d-%03d-%06d-%06d.txt",Size-
1,FUNCION,BLK,THR,DIMENSIONES,GMAX,TMUESTRA);
         informe = fopen(salida,"w");
         sprintf(salida,"v2t-PSOgpu%d-%s-%02d-%03d-%03d-%06d-%06d.txt",Size-
1,FUNCION,BLK,THR,DIMENSIONES,GMAX,TMUESTRA);
         tiempos = fopen(salida,"w");
         for (muestra=0;muestra<TMUESTRA;muestra++)
         {
           tini=tfin=0;
           tini = MPI_Wtime();

           for (i=1;i<Size;i++)
           {
                destinoE= (i+1) % Size;
                if (destinoE==0)
                    destinoE=1;
                MPI_Send(&destinoE,1,MPI_INT,i,0,MPI_COMM_WORLD);
           }   // fin for  prueba envio de vecino

           for (i=1;i<Size;i++)
           {



      MPI_Recv(rema,DIMENSIONES+1,MPI_FLOAT,i,4,MPI_COMM_WORLD,&rec_stat);
                if (i==1||rema[DIMENSIONES]>dimej[DIMENSIONES])
                {
                     dimej[DIMENSIONES]=rema[DIMENSIONES];
                     for(j=0;j<DIMENSIONES;j++)
```

```
                    {
                         dimej[j]=rema[j];
                    }
               idmejor=i;
               }

          }// FIN CILO i

               if(dimej[DIMENSIONES]>EMAX)
               {
                    prematuros++;
               }
               else
               {
                    error += dimej[DIMENSIONES];
               }
               tfin= MPI_Wtime();
               ttotal=tfin-tini;
               tpromedio+=ttotal;
               if(dimej[DIMENSIONES]<0.000001) ceros++;   //incrementamos
cuantos ceros encontramos


               fprintf(tiempos,"%.06f\t\n",ttotal);
               fprintf(informe,"%.07f\n",dimej[DIMENSIONES]);
               eprom=error/(TMUESTRA-prematuros);

       }  // fin ciclo muestra

       tpromedio/=TMUESTRA;
       fprintf(tiempos,"Tpromedio: %.06f\t\n",tpromedio);
       fprintf(informe,"CEROS...: %d \n", ceros);
       fprintf(informe,"PREMATUROS ...: %d \n", prematuros);
       fprintf(informe,"ERROR PROMEDIO ...: %f \n", eprom);
       fclose(informe);
       fclose(tiempos);
       printf("\n\n---PRUEBA FINALIZADA...\n");
       printf("\n\n CEROS: %d\t TIEMPO PROMEDIO: %.06f\t\n",ceros,tpromedio);

    } //fin if rank = 0
    else  //son los esclavos
    {
        for (muestra=0;muestra<TMUESTRA;muestra++)
        {
         MPI_Recv(&destinoR,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
         generarn(mbl, resultado,destinoR, BLK, THR);
         MPI_Send(resultado,DIMENSIONES+1,MPI_FLOAT,0,4,MPI_COMM_WORLD);
           }
```

```
        }


        delete [] mbl;
        delete [] resultado;
        delete [] rema;
        delete [] dimej;

        MPI_Finalize();


        return 0;
    }
```

ARCHIVO “CUDA.CU”

```
#include "archivos.h"

#include "aptitud.h"
#include <curand_kernel.h>
#include <sys/time.h>
#include <stdlib.h>

__device__ long semillasd[POBS*TENJ];
__device__ curandState estadosd[POBS*TENJ];
__device__ particula pd[POBS*TENJ];
__device__ mejores mejorbd[POBS];
__device__ mejores ptempD;



__global__ void inicializa_random()
{
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    curand_init(semillasd[tid],tid,0,&estadosd[tid]);
}


__global__ void genera()
{
    int d, it, ib, tid,mitad;
    float UNI, AUX;

    it = threadIdx.x;
    ib = blockIdx.x;

    tid = threadIdx.x + blockIdx.x*blockDim.x;
    curandStateXORWOW_t miestado=estadosd[tid];

    for(d=0;d<DIMENSIONES;d++)
    {
        UNI = curand_uniform(&miestado);//}while(UNI==1.0);
        #ifdef ASIMETRICO //Si el espacio de busqueda es asimetrico
            pd[tid].mx[d] =p[tid].x[d] =UNI*XMAX; //UNI distribucion uniforme
(0,1)
        #else
            /*AUX=curand_uniform(&miestado);
            if(AUX>0.5)
               UNI = -UNI; */
            UNI=1.0-2.0*curand_uniform(&miestado);
            pd[tid].mx[d]=pd[tid].x[d]=UNI*XMAX; //UNI distribucion uniforme (-
```

```
1,1)
        #endif

        UNI=1.0-2.0*curand_uniform(&miestado);
        pd[tid].v[d] = (UNI*XMAX) /10; //Velocidad inicial
    }// fin ciclo d

    pd[tid].map = pd[tid].ap = aptitud(pd[tid].x);      // Se calcula la
aptitud inicial
    estadosd[tid] = miestado; //actualiza estado del generador de aleatorios
    __syncthreads();

    //Encuentra el mejor del enjambre usando reducción binaria
    __shared__ int idmejor[TENJ];
    idmejor[it] = it;
    //__syncthreads();
    mitad=TENJ/2;
    do{
        if(it<mitad)
        {
            if(pd[ib*TENJ+idmejor[it]].ap < pd[ib*TENJ+idmejor[it+mitad]].ap)
                idmejor[it] = idmejor[it+mitad];
            __syncthreads();
        }
        mitad /= 2;
    }while(mitad>0); // el it de la mejor ap quedó en idmejor[0]    //corregi
reduccion binaria   tenia 1
    //asigna el mejor resultado del bloque
    if (it ==0)
    {
        mejorbd[ib].ap = pd[ib*TENJ+idmejor[0]].ap;
        //y sus dimensiones
        for(d=0;d<DIMENSIONES;d++)
            mejorbd[ib].x[d] = pd[ib*TENJ+idmejor[0]].x[d];
    }
}


__global__ void pso()
{
    __shared__ int idm[TENJ]; //para encontrar il it de la mejor del enjambre
    __shared__ float ap[TENJ];     //aptitudes de las partículas del enjambre
    __shared__ mejores mpb;        //Mejor partícula del bloque

    particula pl; // partícula local al hilo

    //Declaracion de variables
    int it, ib, tid, mitad,d,k;
```

```
    it=threadIdx.x;

    ib=blockIdx.x;

    tid=threadIdx.x + blockIdx.x*blockDim.x;    //secuencia para el generador de
números aleatorios
    curandState miestado=estadosd[tid];

    float UNI,AUX;

    // COPIA EN VARIABLES LOCALES AL BLOQUE los datos de las partículas del
enjambre
    // INICIALIZA EL MEJOR DEL VECINDARIO CON LA PROPIA PARTÍCULA
    //la aptitud
    pl.ap = pd[tid].ap; //aptitud de c/partícula en el enjambre
    pl.map = pd[tid].map; // mejor aptitud histórica de c/partícula en el
enjambre
    if (it == 0)
    {
        mpb.ap = mejorbd[ib].ap; // mejor aptitud del enjambre
    }

    //sus dimensiones y velocidad
    for(d=0; d<DIMENSIONES;d++)
    {
        pl.x[d] = pd[tid].x[d]; //x partícula
        pl.v[d] = pd[tid].x[d]; // velocidad
        pl.mx[d] = pd[tid].mx[d]; // mejor x de la partícula
        if (it == 0)
        {
            mpb.x[d] = mejorbd[ib].x[d]; // x del mejor del enjambre
        }
    }
    __syncthreads();

    ////////////////////////////////////////////////
    ////////////////////////////////////////////////
    //////   EMPIEZA OPTIMIZACIÓN    ////////
    ////////////////////////////////////////////////
    ////////////////////////////////////////////////
    for(k=0;k<PMB;k++)    //Hacer 'PMB' veces
    {
        //Calcular Velocidad y Posicion nuevas de la particula
        for(d=0;d<DIMENSIONES;d++)
        {
            ////////////////////////////////////////////////////////////////////
///////////////
            ////////    v e l o c i d a d
```

```
            ////////////////////////////////////////////////////////////////
//////////////
            UNI = curand_uniform(&miestado);
            AUX = curand_uniform(&miestado);
            pl.v[d] = W*pl.v[d] + C1*UNI*(pl.mx[d] - pl.x[d]) +
                                  C2*AUX*(mpb.x[d] - pl.x[d]);


            if(pl.v[d] > XMAX)
                pl.v[d] = XMAX;
            else
                if(pl.v[d]< -XMAX)
                    pl.v[d] = -XMAX;
            ////////////////////////////////////////////////////////////////
///////////////
            ////////    p o s i c i ó n
            ////////////////////////////////////////////////////////////////
//////////////
            pl.x[d] += pl.v[d];
            if(pl.x[d] > XMAX)
                 pl.x[d] =XMAX;
            else
                if(pl.x[d] < -XMAX)
                   pl.x[d] = -XMAX;
        } // for dimensiones
        ////////////////////////////////////////////////
        //calcula aptidud de la partícula
        ////////////////////////////////////////////////
        pl.ap = aptitud(pl.x);

        ////////////////////////////////////////////////
        //Verifica si la actual es mejor que la histórica
        ////////////////////////////////////////////////
        if(pl.ap > pl.map)
        {
            pl.map = pl.ap;
            for(d=0;d<DIMENSIONES;d++)
            {
                pl.mx[d]=pl.x[d];
            }
        }
        ////////////////////////////////////////////////
        //encuentra la mejor del enjambre por reducción binaria
        ////////////////////////////////////////////////
        ap[it] = pl.ap; // copia las aptitudes a la memoria compartida
        idm[it] = it;//índices de las aptitutdes
        __syncthreads();
        mitad = THR/2;
```

```
            do{
                if(it<mitad)
                {
                    if(ap[idm[it+mitad]]>ap[idm[it]])
                        idm[it] = idm[it+mitad];
                }
                __syncthreads();
                mitad/=2;
            }while(mitad>0);
            // el hilo con mejor aptitud actualiza la mejor ap del enjambre
            if(it==idm[0])
            {
                if(pl.ap> mpb.ap)
                {
                    mpb.ap =pl.ap;
                    mejorbd[ib].ap = mpb.ap;
                    for(d=0;d<DIMENSIONES;d++)
                    {
                        mpb.x[d]=pl.x[d];
                        mejorbd[ib].x[d] = mpb.x[d];
                    }
                }
            }
            __syncthreads();
        }//for PMB
        ////COPIAR RESULTADOS A MEMORIA GLOBAL
        //aptitudes
        pd[tid].ap = pl.ap;
        pd[tid].map = pl.map;

        //dimensiones
        for(d=0;d<DIMENSIONES;d++)
        {
            pd[tid].x[d] = pl.x[d];
            pd[tid].mx[d] = pl.mx[d];

        }

    }


    __global__ void comunicaVecino()
    {
        //Declaracion de variables
        int ib, ibnext, d;
        ib=blockIdx.x;
    //  ibnext =(ib+1)%BLK;
        ibnext = (ib+1)&(BLK-1);
```

```
    //si la del bloque siguiente es mejor que la del actual, la sustituye
    if(mejorbd[ibnext].ap>mejorbd[ib].ap)
    {
        //aptitud
        mejorbd[ib].ap = mejorbd[ibnext].ap;
        //dimensiones
        for(d=0; d<DIMENSIONES;d++)
            mejorbd[ib].x[d] = mejorbd[ibnext].x[d];
    }
}

__global__ void dispersaparticle()
{
    int l;
    int ib;
    ib = blockIdx.x;
    mejorbd[ib].ap=ptempD.ap;

    for (l=0; l<DIMENSIONES;l++)
        mejorbd[ib].x[l]=ptempD.x[l];
}

void envianodo(int neighbor, float *particle)
{

    MPI_Send(particle,DIMENSIONES+1,MPI_FLOAT,neighbor,1,MPI_COMM_WORLD);
}

void recibenodo(float *particleOr)
{
    int l;
    mejores ptemp;
    float *particleRec=NULL;
    particleRec=new float[DIMENSIONES+1];

    MPI_Recv(particleRec, DIMENSIONES+1, MPI_FLOAT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    if (particleRec[DIMENSIONES] > particleOr[DIMENSIONES])    //la particula
que estoy recibiendo tiene mejor aptitud
    {
        ptemp.ap = particleRec[DIMENSIONES];
        for (l=0;l<DIMENSIONES;l++)
            ptemp.x[l]=particleRec[l];      //copio las dimensiones a la
particula;

        cudaMemcpyToSymbol(ptempD,&ptemp,sizeof(mejores));
        dispersaparticle<<<1,1>>>();
```

```
    }

}

void detmejbloque(mejores *mbloque, float *rr)
{
    int r,imr,d;
    mejores res;
    for(r=0;r<BLK;r++)
        {
            mbloque[r].ap = -mbloque[r].ap;
            if(r==0 || mbloque[r].ap<res.ap)
            {
                res.ap=mbloque[r].ap;  //actualiza mejor resutlado global
                imr=r;   //renglón del mejor resultado global
            }
        }

    //copia las dimensiones del mejor resultado global

        for(d=0;d<DIMENSIONES;d++)
            rr[d]=res.x[d]=mbloque[imr].x[d];

        rr[DIMENSIONES]=res.ap;

}


void generarn(mejores *mbloque, float *rr, int neighbor, int nblo, int nhil)
{
    struct timeval tv;
    struct timezone tz;
    int total,r;
    long si; //Segundos iniciales
    //long sf; //Segundos finales
    long ui;  // microsegundos inicales y finales
    total = nblo * nhil;
    long semillas[total]; // semillas para generador de aleatorios



    gettimeofday(&tv,&tz);  //obtiene la hora, fecha y zona horaria
    si=tv.tv_sec;     // segundos tiempo inicial
    ui=tv.tv_usec;  //milisegundos tiempo inicial
```

```
    srand(ui);
    for(r=0;r<total;r++)
        {
            semillas[r]=rand();

        }

    ////// copia semillas al device
    cudaMemcpyToSymbol(semillasd,semillas,sizeof(semillas));
    cudaDeviceSetLimit(cudaLimitStackSize,60000);
    cudaThreadSynchronize();
    inicializa_random<<<nblo,nhil>>>();
    cudaThreadSynchronize();
    genera<<<nblo,nhil>>>();
    int j,k;
    int vecesExt=GMAX/PMN;
    int vecesInt=PMN/PMB;
    for(j=0;j<vecesExt;j++)      //ciclo Ext  migra entre nodos
    {

        for(k=0;k<vecesInt;k++)      //ciclo interno migra enrtre bloques
        {
            cudaThreadSynchronize();
            pso<<<nblo,nhil>>>();
            cudaThreadSynchronize();
            comunicaVecino<<<nblo,1>>>();
        }  //fin ciclo int

        cudaMemcpyFromSymbol(mbloque,mejorbd,sizeof(mejorbd));
        detmejbloque(mbloque, rr);
        envianodo(neighbor,rr);
        recibenodo(rr);

    }  //fin del ciclo ext
    cudaMemcpyFromSymbol(mbloque,mejorbd,sizeof(mejorbd));
    detmejbloque(mbloque, rr);
    cudaDeviceReset();
}
```

APÉNDICE 3.

Código De Algoritmo Genético.

ARCHIVO “ARCHIVOS.H”

```
#include<mpi.h>
#define DIMENSIONES   30
#define TMUESTRA  1500          //Veces que se ejecuta el algoritmo
#define MU        64            //tamaño de la población
#define POBS      14            //número de poblaciones
#define GMAX      400       //generaciones
#define PC            0.7           //probabilidad de cruce
#define PM            0.16      //probabilidad de mutación
#define MPB           40        //periodo migratorio
#define MPN           100
#define MR            2             //tasa de migración
#define Z             2             //tamaño de torneo
#define EMAX      0.01      //error máximo permitido
#define BLK           POBS
#define THR           MU
#define TOTAL     POBS*MU

//#define FUNCION "ackley"
#define FUNCION "rastrigin"


typedef struct
{
   float x[DIMENSIONES];  //Cromosoma.
   float ap;    //Aptitud.
} individuo;


extern "C" {


    void h_ag(int neighbor, float *rr);
    void detmejorgpu(individuo *h_bloques, float *rr);
    void envianodo(int neighbor, float *indi);
    void recibenodo(float *indiOr);
}
```

ARCHIVO “MAESTROAG.CPP”

```
#include "archivos.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>


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

    int Size, Rank;
    int muestra,prematuros,ceros;
    int i,j,h,destinoE,destinoR;
    float error,eprom;
    FILE *informe;
    FILE *tiempos;
    char salida[256];

    double tini,tfin;
    float ttotal,tpromedio;

//------------------------------Incializo los archivos antes que el MPI



    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &Size);
    MPI_Comm_rank(MPI_COMM_WORLD, &Rank);

    int N = THR*BLK;



    float *resultado=NULL;
    resultado = new float[DIMENSIONES+1];

    float *rema=NULL;
    rema = new float[DIMENSIONES+1];

    float *dimej=NULL;
    dimej = new float[DIMENSIONES+1];
```

```
    int idmejor;



                                    //CICLO Principal
    if (Rank == 0)   // es el maestro
    {
       prematuros=0;
       ceros=0;
       error=0.0;
       eprom=0.0;
       tpromedio=0.0;
       //Crear archivos
       printf("\n \n Empezando prueba en cluster...  %d Muestras
\n\n",TMUESTRA);
       printf("\n ---Cluster trabajando...  NO APAGAR EQUIPOS ---\n\n");
       sprintf(salida,"AGgpu%d-%s-%02d-%03d-%03d-%06d-%06d.txt",Size-
1,FUNCION,BLK,THR,DIMENSIONES,GMAX,TMUESTRA);
       informe = fopen(salida,"w");
       sprintf(salida,"t-AGgpu%d-%s-%02d-%03d-%03d-%06d-%06d.txt",Size-
1,FUNCION,BLK,THR,DIMENSIONES,GMAX,TMUESTRA);
       tiempos = fopen(salida,"w");
       for (muestra=0;muestra<TMUESTRA;muestra++)
       {
        tini=tfin=0;
        tini = MPI_Wtime();

        for (i=1;i<Size;i++)
        {
            destinoE= (i+1) % Size;
            if (destinoE==0)
                destinoE=1;
            MPI_Send(&destinoE,1,MPI_INT,i,0,MPI_COMM_WORLD);
        }   // fin for  prueba envio de vecino

        for (i=1;i<Size;i++)
        {



    MPI_Recv(rema,DIMENSIONES+1,MPI_FLOAT,i,4,MPI_COMM_WORLD,&rec_stat);
            if (i==1||rema[DIMENSIONES] > dimej[DIMENSIONES])
            {
                dimej[DIMENSIONES]=rema[DIMENSIONES];
                for(j=0;j<DIMENSIONES;j++)
                {
                    dimej[j]=rema[j];
```

```
                }
            idmejor=i;
            }

        }// FIN CILO i
            dimej[DIMENSIONES]= -dimej[DIMENSIONES];   //Lo hago positivo para
comparar
            if(dimej[DIMENSIONES]>EMAX)
            {
                prematuros++;
            }
            else
            {
                error += dimej[DIMENSIONES];
            }
            tfin= MPI_Wtime();
            ttotal=tfin-tini;
            tpromedio+=ttotal;
            if(dimej[DIMENSIONES]<0.000001) ceros++;   //incrementamos cuantos
ceros encontramos


            fprintf(tiempos,"%.06f\t\n",ttotal);
            fprintf(informe,"%.07f\n",dimej[DIMENSIONES]);
            eprom=error/(TMUESTRA-prematuros);

      }  // fin ciclo muestra

      tpromedio/=TMUESTRA;
      fprintf(tiempos,"Tpromedio: %.06f\t\n",tpromedio);
      fprintf(informe,"CEROS...: %d \n", ceros);
      fprintf(informe,"PREMATUROS ...: %d \n", prematuros);
      fprintf(informe,"ERROR PROMEDIO ...: %f \n", eprom);
      fclose(informe);
      fclose(tiempos);
      printf("\n\n---PRUEBA FINALIZADA...\n");
      printf("\n\n CEROS: %d\t TIEMPO PROMEDIO: %.06f\t\n",ceros,tpromedio);

    } //fin if rank = 0
    else  //son los esclavos
    {
        for (muestra=0;muestra<TMUESTRA;muestra++)
        {
        MPI_Recv(&destinoR,1,MPI_INT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        h_ag(destinoR, resultado);
        MPI_Send(resultado,DIMENSIONES+1,MPI_FLOAT,0,4,MPI_COMM_WORLD);
            }
    }
```

```
        delete [] resultado;
        delete [] rema;
        delete [] dimej;

        MPI_Finalize();


        return 0;
}
```

ARCHIVO “AGCUDA.CU”

```
#include <curand_kernel.h>

#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include "archivos.h"
#include "aptitud.h"
#include "cuda_runtime.h"
#include "device_launch_parameters.h"


//*********VARIABLES GLOBALES


__device__ long d_semillas[POBS*MU];            //arreglo para recibir las
semillas.
__device__ curandState d_estados[POBS*MU];      //estados
__device__ individuo d_pob[POBS*MU];
__device__ individuo d_mejores[BLK];   //arreglo para guardar los mejores de
cada bloque
__device__ individuo itempD;


//////// KERNEL inicializa aleatorios
//////////////////////
__global__ void inicializa_random()
{
    int tid = threadIdx.x + blockIdx.x*blockDim.x;
    curand_init(d_semillas[tid],tid,0,&d_estados[tid]);
}   //FIN INICIALIZA



//GENERAR LOS INDIVIDUOS, DIMENSIONES Y APTITUD
////////////////////////////////////////////
__global__ void genera()
{
    int tid,d;
    float UNI;


    tid = threadIdx.x + blockIdx.x *blockDim.x;

    curandStateXORWOW_t miestado=d_estados[tid];
```

```
    for(d=0;d<DIMENSIONES;d++)
        {

            #ifdef ASIMETRICO //Si el espacio de busqueda es asimetrico
                UNI = curand_uniform(&miestado);//}while(UNI==1.0);
                d_pob[tid].x[d] =UNI*XMAX; //UNI distribucion uniforme (0,1)
            #else

                UNI=1.0-2.0*curand_uniform(&miestado);
                d_pob[tid].x[d]=UNI*XMAX; //UNI distribucion uniforme (-1,1)
            #endif


        }// fin ciclo d

        d_pob[tid].ap =  aptitud(d_pob[tid].x);     // Se calcula la aptitud
inicial

        d_estados[tid] = miestado; //actualiza estado del generador de
aleatorios

        __syncthreads();

}  //////////FIN GENERA




//////////////////////////////ORDENA LOS INDIVIDUOS DE CADA BLOQUE (POBLACIÓN)
__global__ void ordena()
{

    __shared__ individuo pob[MU];

    int d,i, im,j;
    int it, ib,tid;
    it=threadIdx.x;
    ib=blockIdx.x;
    tid = blockIdx.x * blockDim.x  +threadIdx.x;
    // Variables  auxiliares
    individuo aux;

    ////////////////////////////////////////////////////////////////
    /////////  COPIA LA población actual de las BLK poblaciones recibidas
    ////////////////////////////////////////////////////////////////
    //la aptitud
```

```
pob[it].ap = d_pob[tid].ap;
//sus dimensiones
for(d=0; d<DIMENSIONES; d++)
    pob[it].x[d] = d_pob[tid].x[d];
__syncthreads();

//ordena la población
if(it==0)
{
    for(i=0; i<MU-1; i++)
    {
        im=i;
        for(j=i+1; j<MU; j++)
        {
            // busca desde i hasta MU al mejor
            if(pob[j].ap > pob[im].ap)
            {
                im =j;
            }
        }
        //pone en la posición i al mejor (para que ya no participe en la
        //búsqueda del mejor)
        aux.ap     = pob[im].ap;
        pob[im].ap = pob[i].ap;
        pob[i].ap  = aux.ap;

        // y sus dimensiones
        for(d=0; d<DIMENSIONES; d++)
        {
            aux.x[d] = pob[im].x[d];
            pob[im].x[d] = pob[i].x[d];
            pob[i].x[d] = aux.x[d];
        }

    }
    //actualiza el mejor de cada bloque
    d_mejores[ib].ap=pob[0].ap;

    //sus dimensiones
    for(d=0;d<DIMENSIONES;d++)
        d_mejores[ib].x[d] = pob[0].x[d];
}
__syncthreads();

//actualiza variable global que tiene la población actual
d_pob[tid].ap = pob[it].ap;
//y sus dimensiones
for(d=0; d<DIMENSIONES; d++)
```

```
    {
        d_pob[tid].x[d] = pob[it].x[d];
    }
    __syncthreads();
}  ////FIN ORDENA



__global__ void mejor()      //ENCUENTRA EL MEJOR DE CADA BLOQUE
{
    // Población ACTUAL   está en
    __shared__ individuo pob[MU];


    int d, i, im, it, ib,tid;

    it=threadIdx.x;
    ib=blockIdx.x;
    tid=blockIdx.x*blockDim.x + threadIdx.x;

    ////////////////////////////////////////////////////////////////
    /////////  COPIA LA población actual de las BLK poblaciones recibidas
    ////////////////////////////////////////////////////////////////
    //la aptitud
    pob[it].ap = d_pob[tid].ap;
    //sus dimensiones
    for(d=0; d<DIMENSIONES; d++)
        pob[it] = d_pob[tid];

    __syncthreads();

    if(it==0)
    {
        im=0;
        for(i=1;i<MU;i++)
        {
            if(pob[i].ap > pob[im].ap)
            {
                im=i;
            }
        }
        d_mejores[ib].ap = pob[im].ap; // copia en memoria global el mejor
indiviuo de
                                //la población (bloque) actual para comunicarlo
al HOST
        for(d=0;d<DIMENSIONES;d++) // y sus dimensiones
```

```
            d_mejores[ib].x[d] = pob[im].x[d];
        }
}   ///FIN DE KERNEL MEJOR.

__global__ void mejoredb()    //ENCUENTRA MEJOR DE CADA BLOQUE   REDUCCIÓN
BINARIA
{
    int mitad,tid,it,ib,d;
    __shared__ individuo pob[MU];
    __shared__ int idmejor[MU];

    ib= blockIdx.x;
    tid = blockIdx.x * blockDim.x + threadIdx.x;
    it = threadIdx.x;

    pob[it].ap = d_pob[tid].ap;
    for (d=0;d<DIMENSIONES;d++)
        pob[it].x[d]=d_pob[tid].x[d];

    idmejor[it] = it;
    __syncthreads();


    mitad=MU/2;
    do{
        if(it<mitad)
        {
            if(pob[idmejor[it]].ap < pob[idmejor[it+mitad]].ap)
                idmejor[it] = idmejor[it+mitad];
            __syncthreads();
        }
        mitad /= 2;
    }while(mitad>0); // el it de la mejor ap quedó en idmejor[0]    //corregi
reduccion binaria   tenia 1
    //asigna el mejor resultado del bloque
    if (it ==0)
    {
        d_mejores[ib].ap = pob[idmejor[0]].ap;
        //y sus dimensiones
        for(d=0;d<DIMENSIONES;d++)
            d_mejores[ib].x[d] = pob[idmejor[0]].x[d];
    }

}


////////////KERNEL PARA MIGRAR DATOS ENTRE POBLACIONES
__global__ void migra()
```

```
{
    //Declaracion de variables
    int d,u;
    int it;
    it=threadIdx.x;
    int ib, ibnextB;  // indices para migración vertical
    ib=blockIdx.x;
    ibnextB= (ib+1) & (BLK-1);     // (ib+1) % (BLK)
    u = MU-1;
    if(it<MR)
    {
        //copia en los peores del bloque bloque actual, los mejores del
        //bloque siguiente
        d_pob[ib*MU+(u-it)].ap = d_pob[ibnextB*MU+it].ap;
        // y sus dimensiones
        for(d=0; d<DIMENSIONES; d++)
        {
            d_pob[ib*MU+(u-it)].x[d] = d_pob[ibnextB*MU+it].x[d];
        }

    }
    __syncthreads();
}   //FIN MIGRA


__global__ void ag_esc()
{
    // Población ACTUAL   está en
    __shared__ individuo pob[MU];


    // Población SIGUIENTE
    __shared__ individuo psig[MU];



    // Variables  auxiliar para la MUTACIÓN, CRUCE y migración
    individuo aux;


    // mejor INDIVIDUO
    individuo mejor_ap;            //Mejor Aptitud


    //Declaracion de variables
    int it;
    it=threadIdx.x;
```

```
    int ib;  // indices para migración vertical
    ib=blockIdx.x;

    int  i,j, k, g, l,d,im,ip,gen,CZ;    //generación, periodo, cantidad
                                                    //de veces e índices para
los ciclos

    float r,xnew1, xnew2, lambda, xold1, xold2;

    float UNI;
    //secuencia para el generador de números aleatorios
    int tid=threadIdx.x + blockIdx.x*blockDim.x;

    curandStateXORWOW_t miestado=d_estados[tid];

    CZ = (MU*PC)/2;
    ////////////////////////////////////////////////////////////////
    ////////  COPIA LA población actual de las BLK poblaciones recibidas
    ////////////////////////////////////////////////////////////////

    //la aptitud
    pob[it].ap = d_pob[tid].ap;
    //sus dimensiones
    for(d=0; d<DIMENSIONES; d++)
        pob[it].x[d] = d_pob[tid].x[d];

    __syncthreads();

    ////////////////////////////////////////////////////////////////////
    /// copia el mejor /////
    ////////////////////////////////////////////////////////////////////
    if(it==(THR-1))
    {
        // copia en mejor_ap el mejor indiviuo de la población actual
        mejor_ap.ap=d_mejores[ib].ap;
        for(d=0;d<DIMENSIONES;d++) // y sus dimensiones
            mejor_ap.x[d]=d_mejores[ib].x[ib];
    }
    ////////////////////////////////////////////////////////////////
    ////////////// HACER MP veces
    ////////////////////////////////////////////////////////////////
    for(g=0;g<MPB;g++)
    {
        ////////////////////////////////////////////////////////////////
        ////////    s e l e c c i ó n
        ////////////////////////////////////////////////////////////////
        UNI = curand_uniform(&miestado);
        k=UNI*MU;
```

```
if (k==MU)
    k--;
for(j=1;j<Z;j++)
{
    UNI=curand_uniform(&miestado);
    l=UNI*MU;
            if (l==MU)
        l--;
    if(pob[k].ap<pob[l].ap)
        k=l;
}
//copia en poblaciòn siguiente
psig[it].ap = pob[k].ap;  //aptitud
for(d=0;d<DIMENSIONES;d++)  //dimensiones
    psig[it].x[d]=pob[k].x[d];

__syncthreads();

////////////////////////////////////////////////////////////////////
/////////    c r u c e
////////////////////////////////////////////////////////////////////
if(it==(THR-1))
{
    lambda =curand_uniform(&miestado);
    for(i=0; i<CZ; i++)   //CZ=(MU*PC)/2;
    {
        j=curand_uniform(&miestado)*MU;
        k=curand_uniform(&miestado)*MU;
        for(d=0; d<DIMENSIONES; d++)
        {
            xold1=psig[j].x[d];
            xold2=psig[k].x[d];

            xnew1 = (xold1 * lambda) + (xold2 * (1-lambda));
            xnew2 = (xold1 * (1-lambda)) + (xold2 * lambda);


            psig[j].x[d]= xnew1;
            psig[k].x[d] = xnew2;
        }
        psig[j].ap = aptitud(psig[j].x);
        psig[k].ap = aptitud(psig[k].x);
    }
}
__syncthreads();

////////////////////////////////////////////////////////////////////
/////////    m u t a c i ó n
```

```
        ////////////////////////////////////////////////////////////////
        // Para decidir si se aplica el operador en base a PM
        r=curand_uniform(&miestado);
        if(r<=PM)
        {
            //copia dimensiones de psig en el arreglo auxiliar para mutación
            for(d=0;d<DIMENSIONES;d++)
                aux.x[d] = psig[it].x[d];
            UNI=curand_uniform(&miestado);
            //Para decidir cual gen mutar x[0], x[1],...,x[gen]
            gen=UNI*DIMENSIONES;
            //h=curand_uniform(&miestado); //incremento o decremento
            UNI=1.0-2.0*curand_uniform(&miestado);
            aux.x[gen] = UNI*XMAX;


            aux.ap=aptitud(aux.x);

            psig[it].ap = aux.ap;
//          for(d=0; d<DIMENSIONES; d++)
                psig[it].x[gen] = aux.x[gen];
        }
        __syncthreads();

        ////////////////////////////////////////////////////////////////
        //////  copia en p (pob. actual) la población psig (pob. siguiente)
        ////////////////////////////////////////////////////////////////

        //// la aptitud
        pob[it].ap = psig[it].ap;
        /// sus dimensiones
        for(d=0;d<DIMENSIONES;d++)
            pob[it].x[d] = psig[it].x[d];

        __syncthreads();

        ////////////////////////////////////////////////////////////////
        /////////    encuentra el mejor y el peor
        ////////////////////////////////////////////////////////////////
        if(it==(THR-1))
        {
            im=ip=0;
            for(i=1;i<MU;i++)
            {
                if(pob[i].ap > pob[im].ap)
                {
                    im=i;
                }
```

```
                if(pob[i].ap < pob[ip].ap)
                {
                    ip=i;
                }
            }

            // si el mejor de la generación anterior, es mejor que el de la
            // generación actual, lo incluye en la POS DEL PEOR
            if(pob[im].ap < mejor_ap.ap)
            {
                pob[ip].ap = mejor_ap.ap;
                for(d=0 ; d<DIMENSIONES; d++)
                    pob[ip].x[d] = mejor_ap.x[d];
            }
            else
            {
                mejor_ap.ap =pob[im].ap;
                for(d=0 ; d<DIMENSIONES; d++)
                    mejor_ap.x[d]=pob[im].x[d];
            }
            //actualiza el mejor del bloque
            d_mejores[ib].ap= mejor_ap.ap;
            //sus dimensiones
            for(d=0; d<DIMENSIONES;d++)
                d_mejores[ib].x[d] =mejor_ap.x[d];
        }
        __syncthreads();
    } //fin de ciclo g<MP

    //actualiza variable global que tiene la población actual
    d_pob[ib*MU+it].ap = pob[it].ap;
    //y sus dimensiones
    for(d=0; d<DIMENSIONES; d++)
        d_pob[ib*MU + it].x[d] = pob[it].x[d];

    d_estados[tid] = miestado; //actualiza variable gener. de #s aleatorios
    __syncthreads();
}  // fin __global__ de ag_esc


__global__ void dispersaindi()     //////////COPIA EL INDIVIDUO RECIBIDO EN LA
ULTIMA POSICION DEL PRIMER BLOQUE
{
    int l;
    //int ib;
    //ib = blockIdx.x;
    d_pob[MU-1].ap = itempD.ap;
```

```
    for (l=0; l<DIMENSIONES;l++)
        d_pob[MU-1].x[l]=itempD.x[l];
}


void envianodo(int neighbor, float *indi)
{

    MPI_Send(indi,DIMENSIONES+1,MPI_FLOAT,neighbor,1,MPI_COMM_WORLD);
}

void recibenodo(float *indiOr, FILE *aptitud)
{

    int l;
    individuo itemp;
    float *indiRec=NULL;
    indiRec=new float[DIMENSIONES+1];

    MPI_Recv(indiRec, DIMENSIONES+1, MPI_FLOAT, MPI_ANY_SOURCE, 1,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    fprintf(aptitud,"AP: %.06f\tAPRec:
%.06f\n",indiOr[DIMENSIONES],indiRec[DIMENSIONES]);
    if (indiRec[DIMENSIONES] > indiOr[DIMENSIONES])    //el individuo que estoy
recibiendo tiene mejor aptitud
    {
        itemp.ap = indiRec[DIMENSIONES];
//      fprintf(aptitud,"AP: %.06f\t\n",itemp.ap);
        for (l=0;l<DIMENSIONES;l++)
            itemp.x[l]=indiRec[l];      //copio las dimensiones a el individuo;

        cudaMemcpyToSymbol(itempD,&itemp,sizeof(individuo));
        dispersaindi<<<1,1>>>();


    }

}



//////////////DETERMINA EL MEJOR INDIVIDUO DEL HOST
void detmejorgpu(individuo *h_bloques, float *rr)
{
  int r,imr,d;
  float resap=0.0;
    for (r=0;r<BLK;r++)
    {
```

```
//        h_bloques[r].ap = -h_bloques[r].ap;
          if(r==0 || h_bloques[r].ap > resap)
          {
               resap= h_bloques[r].ap;  //actualiza mejor resutlado global
               imr=r;   //renglón del mejor resultado global
           }
      }

      for(d=0;d<DIMENSIONES;d++)
          rr[d]=h_bloques[imr].x[d];

          rr[DIMENSIONES]=resap;



}



/////////////////FUNCIÓN PRINCIPAL//////////////
void h_ag(int neighbor, float *rr)    // Función que llama la ejecución del
Algoritmo AG
{
     FILE *aptitud;
     char texto[256];
     struct timeval tv;
     struct timezone tz;
     long si; //Segundos iniciales
     long ui;  // microsegundos inicales     solo para generar los aleatorios
     long h_semillas[TOTAL];  //semillas
     int r;
     int j,k;
     individuo h_bloques[BLK];    //guarda los mejores de cada bloque de cada gpu


     ////ABRE ARCHIVO PARA GUARDAR APTITUDES EN CADA MIGRACION
     sprintf(texto,"aptitud.txt");
     aptitud = fopen(texto,"w");

     gettimeofday(&tv,&tz);  //obtiene la hora, fecha y zona horaria
     si=tv.tv_sec;     // segundos tiempo inicial
     ui=tv.tv_usec;  //milisegundos tiempo inicial
     srand(ui);
     for(r=0;r<TOTAL;r++)
          {
               h_semillas[r]=rand();

          }
```

```
    cudaMemcpyToSymbol(d_semillas,h_semillas,sizeof(h_semillas));
    //INICIALIZA ALEATORIOS
    ////////////////////////
    cudaDeviceSetLimit(cudaLimitStackSize,30000);
    cudaThreadSynchronize();
    inicializa_random<<<BLK,THR>>>();
    //GENERA POBLACIONES
    ////////////////////////
    cudaThreadSynchronize();
    genera<<<BLK,THR>>>();
    //ORDENA LA POBLACION
    ////////////////////////
    cudaThreadSynchronize();
    ordena<<<BLK,THR>>>();

    int vecesExt=GMAX/MPN;   //para ciclo externo migración entre nodos
    int vecesInt=MPB/MPN;  // ciclo interno para migración entre bloques
    for(j=0;j<vecesExt;j++)      //ciclo Ext  migra entre nodos
    {

        for(k=0;k<vecesInt;k++)      //ciclo interno migra enrtre bloques
        {
            cudaThreadSynchronize();
            ag_esc<<<BLK,THR>>>();
            ordena<<<BLK,THR>>>();
            cudaThreadSynchronize();
            migra<<<BLK,THR>>>();
        }  //fin ciclo int
        mejoredb<<<BLK,THR>>>();   //encuentra el mejor del bloque
        cudaMemcpyFromSymbol(h_bloques,d_mejores,sizeof(d_mejores));
        detmejorgpu(h_bloques,rr);
        envianodo(neighbor,rr);
        recibenodo(rr,aptitud);
    }  //fin del ciclo ext
    mejoredb<<<BLK,THR>>>();   //encuentra el mejor del bloque

    cudaMemcpyFromSymbol(h_bloques,d_mejores,sizeof(d_mejores));
    detmejorgpu(h_bloques,rr);
    cudaDeviceReset();
    fclose(aptitud);

}  // Fin h_ag
```

REFERENCIAS

[1] Top500.org, "Top500 Supercomputers sites." [Online]. Available: www.top500.org.

[2] M. A. Castro Liera, "Un Algoritmo Genético Distribuido con Aplicación en la Identificación Difusa de un Proceso Fermentativo.," 2009.

[3] A. Rosete, "UNA SOLUCIÓN FLEXIBLE Y EFICIENTE PARA EL TRAZADO DE GRAFOS BASADA EN EL ESCALADOR DE COLINAS ESTOCÁSTICO.," Facultad de Ingeniería Industrial, CEIS., 2000.

[4] I. Castro Liera, "Paralelización de Algoritmos de Optimización Basados en Poblaciones Usando GPGPU," Instituto Tecnológico de La Paz, 2011.

[5] J. Sanders and E. Kandrot, *CUDA by Example*, vol. 54. 2010.

[6] Nvidia, "Cuda C Programming Guide," *Des. Guid.*, p. 228, 2014.

[7] J. A. Castro, M. A. Castro Liera, and I. Castro Liera, *Programación Paralela Aplicada en Optimización*, 1st ed. La Paz, B.C.S., México: Instituto Tecnológico de La Paz, 2012.

[8] T. Rauber and G. Rünger, *Parallel programming: For multicore and cluster systems*. 2010.

[9] B. Wilkinson and M. Allen, *Parallel Programming. Techniques and Applications using networked workstations and Parallel Computers*, Second. 2005.

[10] C. L. A. Castro, M.A. Morales, J.A. Castro, I. Castro, J.A., "Distributed Particle Swarm Optimization Using Clusters and GPGPU," in *I Congreso Internacional de Ingeniería Electrónica y Computación 2011*, 2011.

[11] T. Sterling, *Beowulf Cluster Computing with Linux*. 2001.

[12] O. M. Project, "Open MPI: Open Source High Performance Computing," 2014. [Online]. Available: http://www.open-mpi.org.

[13] P. Pacheco, *An Introduction to Parallel Programming*, vol. 79. 2011.

[14] F. Van Den Bergh, "An Analysis of Particle Swarm Optimizers," *PhD Thesis, Univ. Pretoria*, vol. 200, no. November, pp. 1–300, 2001.

[15] J. Kennedy and C. Eberhart, "Particle Swarm Optimization.," *Proc. IEEE Int. Conf. Neural Networks*, vol. IV, pp. 1942-1948, 1995.

[16] R. C. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," *Proc. Sixth Int. Symp. Micro Mach. Hum. Sci.*, pp. 39-43, 1995.

[17] M. Clerc, *Particle Swarm Optimization*, 1st ed. New Port Beach ISTE Ltd, USA., 2006.

[18] D. Corne, M. Dorigo, and F. Glover, *New Ideas in Optimization*. 1999.

[19] Y. Shi and R. C. Eberhart, "A Modified Particle Swarm Optimizer.," in *IEEE International Conference of Evolutionary Computation*, 1998.

[20] M. Clerc, "The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization.," in *Proceedings or the Congress on Evolutionary Computation*, 1999, pp. 1951-1957.

[21] R. C. Eberhart and Y. Shi, "Compuarting Inertia Weights and Constriction Factors in Particle Swarm Optimization.," in *Proceedings of the Congress on Evolutionary Computing*, 2000, pp. 84-89.

[22] M. Waintraub and R. Schirru, "Multiprocessor modeling of Particle Swarm Optimization applied to nuclear engineering problems," *Prog. Nucl. Eng. 51*, pp. 680-688, 2009.

[23] B. Li and K. Wada, "Parallelizing particle swarm optimization," in *IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, 2005, p. 288 —291.

[24] M. D. Vose, *The Simple Genetic Algorithm: Foundations and Theory*, vol. Complex ad. 1999.

[25] J. H. Holland, *Adaptation in Natural and Artificial Systems*, vol. Ann Arbor. 1975.

[26] D. Higuera Balderrama, "Optimización Híbrida del Modelo Difuso de un Proceso Fermentativo," Instituto Tecnológico de La Paz, 2014.

[27] M. Nowostawski and R. Poli, "Parallel genetic algorithm taxonomy," *1999 Third Int. Conf. Knowledge-Based Intell. Inf. Eng. Syst. Proc. (Cat. No.99TH8410)*, 1999.

[28] P. N. Suganthan, N. Hansen, J. J. Liang, K. Deb, Y.-P. Chen, A. Auger, and S. Tiwari, "Problem Definitions and Evaluation Criteria for the CEC 2005 Special Session on Real-Parameter Optimization," 2005.