Hardware Gráfico Configurable. Una plataforma eficiente para la implementación de Algoritmos Genéticos.

Carmen Córdoba Juan Carlos Pedraz Christian Tenllado José Ignacio Hidalgo

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática
Universidad Complutense de Madrid
Calle Profesor García Santesmases s/n
28040 Madrid
tenllado@dacya.ucm.es , rlario@dacya.ucm.es , hidalgo@dacya.ucm.es

Resumen

Los algoritmos evolutivos (AEs) presentan propiedades que los hacen ser procesos eminentemente paralelos. Existen distintas implementaciones de Algoritmos Evolutivos Paralelos (AEPs) sobre distintas plataformas y la mayoría de ellas aprovechan bien los recursos disponibles. El hardware gráfico configurable permite la ejecución de procesos altamente paralelos de una forma eficiente. En este trabajo se presenta la implementación de un Algoritmo Genético Paralelo sencillo para demostrar que esta implementación es posible y aprovechar la capacidad de cómputo que durante la ejecución de programas de optimización suele desaprovecharse al no utilizarse este hardware.

1. Introducción

Los algoritmos genéticos (AG) [1] son mecanismos de búsqueda basados en las leyes de la selección natural y son muchas sus aplicaciones en ciencia e ingeniería. Para incrementar la eficiencia y la velocidad que se puede obtener con un AG secuencial, se han implementado distintas alternativas de algoritmos genéticos paralelos.

Muchas de las implementaciones de AG paralelos (AGP) han buscado hacer un uso óptimo de arquitecturas paralelas, por ejemplo Talbi et al presentaron un AGP para la resolución del problema de la partición de grafos. Para ello utilizaron una implementación sobre un Supernodo de Transputers [2]. El Supernodo consistía en una máquina altamente paralela basada en microprocesadores de 32-bit con una memoria on-chip y una unidad de Punto Flotante. La comunicación entre transputers se realizaba mediante cuatro conexiones serie asíncronas bidireccionales de punto a punto.

Hidalgo et al. utilizan un AGP implementado sobre una supercomputadora Cray T3E para resolver un problema de diseño de circuitos sobre FPGAs [3]. Además, Existen numerosas implementaciones de AGPs sobre clusters de computadoras con resultados excelentes.

Recientemente Herrera et al. han presentado una implementación orientada a entornos Grid. La mayoría de las dificultades de implementación surgen de las características propias del Grid como son la complejidad, heterogeneidad, dinamismo y alto porcentaje de fallos. Esto hace que no sea una implementación muy apropiada para problemas de optimización que requieran una solución rápida [4].

La mayoría de estas implementaciones son eficientes y aprovechan los recursos disponibles, pero sin embargo son recursos que durante ese proceso están solamente a disposición del usuario que ejecuta el AG. En el caso del Grid se utilizan recursos que de otra forma no estarían disponibles para el usuario, pero tienen una dependencia excesiva de que los recursos se estén utilizando por otros usuarios. La implementación que se presenta en este artículo aprovecha recursos que permanecen habitualmente inutilizados durante la ejecución de un programa, como es la tarjeta gráfica programable que solamente actúa de una forma eficiente cuando se utilizan aplicaciones gráficas, pero que sin embargo están disponibles y desaprovechadas el resto del tiempo. En este artículo se presenta una primera implementación de un algoritmo genético paralelo sobre una tarjeta gráfica programable Nvidia E-Force FX5950. EL objetivo es demostrar que los AG son programas

Actas del IV Congreso Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB2005, pp.829-836
ISBN: Vol I:84-9732-440-4 Vol II: 84-9732-467-6

de optimización que se pueden implementar eficientemente sobre este tipo de arquitecturas.

El resto de trabajo está organizado como sigue. En la sección 2 se realiza un breve repaso de los algoritmos genéticos paralelos. La sección 3 describe la adaptación e implementación del AG sobre la tarjeta gráfica. Finalmente se presentan unos resultados preeliminares así como las conclusiones y el trabajo en curso.

2. Algoritmos Genéticos Paralelos

Un proceso es una copia de un programa o de una parte de él. Un programa es paralelo si en cualquier momento de su ejecución puede ejecutar más de un proceso. Para crear programas paralelos eficientes hay que poder crear, destruir y especificar procesos así como la iteración entre ellos. Básicamente existen tres formas de paralelizar un programa [5]:

- Paralelización de grano fino: la paralelización del programa se realiza a nivel de instrucción.
- Paralelización de grano medio: los programas se paralelizan a nivel de bucle. Esta paralelización se realiza habitualmente de una forma automática en los compiladores.
- Paralelización de grano grueso: se basan en la descomposición del dominio de datos entre los procesadores, siendo cada uno de ellos el responsable de realizar los cálculos sobre sus datos locales.

Este último tipo de paralelización se puede a su vez realizar siguiendo tres estilos distintos de programación: paralelismo en datos, programación por paso de mensajes y programación por paso de datos.

- Paralelismo en datos: El compilador se encarga de la distribución de los datos guiado por un conjunto de directivas que introduce el programador. Estas directivas hacen que cuando se compila el programa las funciones se distribuyan entre los procesadores disponibles. Como principal ventaja presenta su facilidad de programación. Sin embargo suelen tener una eficiencia inferior a la que se consigue con el paso de mensajes.
- Programación por paso de mensajes: El método más utilizado para programar sistemas de memoria distribuida es el paso de mensajes o alguna variante del mismo. La forma más básica consiste en que los procesos coordinan sus actividades mediante el envió y la recepción de mensajes. Las principales ventajas que presentan son la flexibilidad, la eficiencia, la portabilidad y la controlabilidad del programa. Por contra el tiempo de desarrollo puede ser más elevado que para un paralelismo en datos. Las librerías más utilizadas son por este orden la estándar MPI (Message Passing Interface) [6] y PVM (Parallel Virtual Machine) [7].
- Programación por paso de datos: A diferencia del modelo de paso de mensajes, la transferencia de datos entre los procesadores se realiza con primitivas unilaterales tipo put-get, lo que evita la necesidad de sincronización

Los principales métodos de paralelización de AGs consisten en la división de la población en varias sub-poblaciones (demes). Por ello, el tamaño y distribución de la población entre los distintos procesadores sería uno de los factores fundamentales a la ahora de paralelizar un algoritmo evolutivo. Al compartir la carga de trabajo entre N procesadores se puede esperar que el sistema trabaje N veces mas rápido que con un solo procesador, lo que permite tratar problemas mas grandes y complicados. Las cosas no son tan sencillas, ya que existen varios factores de sobrecarga que hacen disminuir el rendimiento previsible [8].

La paralelización global es la forma más sencilla de implementar un algoritmo genético paralelo. Como ya se ha dicho, consiste o en paralelizar la evaluación de los individuos o en realizar una ejecución simultánea de distintos AGs secuenciales. Sin embargo es muy útil, ya que permite obtener mejoras en el rendimiento con respecto al AG secuencial muy fácilmente sin cambiar la estructura principal de éste. En la mayoría de las aplicaciones de los AGs la parte que consume un mayor tiempo de cálculo es la evaluación de la función de coste. En estos casos se puede ahorrar mucho tiempo de cálculo simplemente encargando la evaluación de una parte de la población a distintos procesadores y de una forma simultánea.

Además de conseguir tiempos de ejecución menores, al paralelizar un AG estamos modificando el comportamiento algorítmico, y esto hace que podamos obtener otras soluciones y experimentar con las distintas posibilidades de

implementación y los distintos factores que influyen en ella [9][10][11][12]. Estas poblaciones van evolucionando por separado para detenerse en un momento determinado e intercambiar los mejores individuos entre ellas. Técnicamente hay 3 características importantes que influyen en la eficiencia de un algoritmo genético paralelo:

- La topología que define la comunicación entre sub-poblaciones.
- La proporción de intercambio: número de individuos a intercambiar
- Los intervalos de migración: periodicidad con que se intercambian los individuos.

Por supuesto al igual que en un AG secuencial en uno paralelo hay que tener en cuenta los operadores de selección cruce y mutación que se utilizan. La selección se encarga de escoger los individuos que participarán en la formación de la nueva población.

La selección utilizada es muy importante para la correcta convergencia del algoritmo. La presión de selección debe ser tal que consiga un equilibrio entre exploración y explotación. Si la presión de selección es muy alta se corre el peligro que individuos de la población inicial con fitness superior a la media, que representan óptimos locales pero no globales, se reproduzcan en exceso provocando una perdida de diversidad y una convergencia prematura. En este caso se prima la explotación frente a la exploración. El caso contrario, una presión de selección baja puede provocar una búsqueda aleatoria o en el mejor de los casos una enorme ralentización del algoritmo. Probablemente, lo óptimo fuera un operador de selección que fuera evolucionando con el algoritmo de manera que en las primeras fases se primara la exploración y cuya presión fuera creciendo hasta que se alcanzar un punto en el que se primara la explotación.

2.1. Selección por torneo (Tournament selection)

Muchos métodos de selección necesitan dos fases; en la primera se calcula el fitness medio de la población y en la segunda se computa el valor esperado de cada individuo. La selección por torneo es una técnica similar a las basadas en ranking en lo que a medida de presión se refiere, pero es eficiente computacionalmente hablando y se puede paralelizar más cómodamente. Funciona de la siguiente forma:

- Se escogen dos individuos aleatoriamente de la población,
- se genera un numero aleatorio r comprendido entre 0 y 1.
- Si r < k, donde k es un parámetro, se selecciona el mejor de los dos individuos en caso contrario se selecciona el peor.

Los dos se devuelven a la población inicial para que puedan ser seleccionados de nuevo.

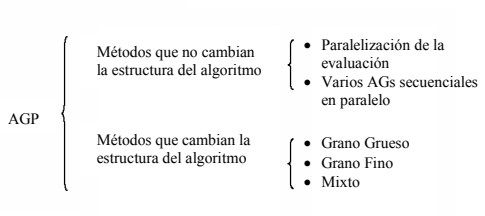

Figura 1: Clasificación de los Algoritmos genéticos paralelos

La Figura 1 resume la clasificación más aceptada de los algoritmos genéticos paralelos. La propuesta que se hace en este trabajo pretende demostrar la viabilidad de implementación de algoritmos genéticos paralelos utilizando hardware gráfico configurable. Para ello implementamos como primera implementación un algoritmo genético secuencial en la tarjeta y comparamos resultados con la CPU. Los resultados demostrarán que es una buena opción utilizar este tipo de plataformas para ejecutar AGPs.

3. Algoritmos Genéticos sobre una Tarjeta Gráfica Configurable

En esta sección presentamos todas las transformaciones del algoritmo genético necesarias para su implementación sobre una tarjeta gráfica programable del estilo de la GPU. En la subsección 3.1 la plataforma se describe objetivo como una arquitectura de procesamiento de flujos, en la Subsección 3.2 damos los detalles de implementación y la descomposición en *kernels* del algoritmo genético.

3.1. GPU: un procesador de flujos.

Las tarjetas gráficas modernas utilizan una arquitectura de procesamiento de flujos segmentada para realizar una parte significativa de los cómputos en el proceso de renderizado. La Figura 2 ilustra este flujo. Un programa envía un conjunto de polígonos a la tarjeta gráfica. Cada vértice en un polígono es procesado de manera independiente atendiendo al *view point*. Una vez transformados, los vértices se ensamblan en sus polígonos constituyentes que son rasterizados convirtiéndolos en un conjunto de fragmentos. Tras algunas operaciones de *raster* los fragmentos finalmente seleccionados son enviados al *frame buffer* constituyendo los *píxels* que forman la imagen visualizada.

Dos de las etapas en el hardware gráfico moderno son programables, el procesador de vértices y el procesador de fragmentos (vertex y fragment engines/shaders). El primero se utiliza para realizar transformaciones sobre los atributos de cada vértice (normal, posición, color, textura, ...). El segundo en cambio se utiliza para realizar transformaciones sobre los distintos fragmentos que constituyen un polígono. Ambos procesadores son extremadamente paralelos, procesando varios elementos en paralelo y haciendo uso intensivo de unidades SIMD.

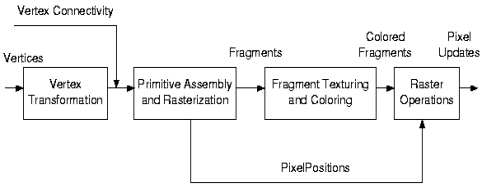

Figura 2: Flujo lógico del proceso de renderizado.

Este hardware programable de propósito específico puede ser utilizado en cómputos de propósito general aprovechando así sus cualidades paralelas. El principal obstáculo es que el algoritmo debe ser expresado como un programa de procesado de flujos (*streaming application*) adecuado para una de las dos unidades programables de la tarjeta.

Un algoritmo genético trabaja básicamente sobre un conjunto de individuos que pueden ser modelados como vectores . Los individuos forman entonces una población que puede ser modelada como un array 2D, en el que cada fila representa un individuo. El procesador de fragmentos está diseñado para trabajar sobre estructuras 2D con lo que parece la unidad más adecuada sobre la que mapear nuestro algoritmo genético.

La Figura 3 es una representación lógica del procesador de fragmentos. El algoritmo debe ser representado como un conjunto de *kernels* que trabajan sobre algún flujo de datos de entrada produciendo uno o más flujos de datos de salida. En el contexto del procesador de fragmentos los *kernels* se denominan *fragment programs*. Los flujos de datos de entrada iniciales son enviados a la GPU como texturas donde son almacenados en *buffers* de textura. El flujo de salida es escrito por la GPU en el *frame buffer*. Con el fin de poder utilizar estos resultados como entrada para otro *fragment program* con la mínima sobrecarga posible, es conveniente componer todas las texturas de entrada en un sólo OpenGl *pbuffer* y utilizar el modo *render to texture*.

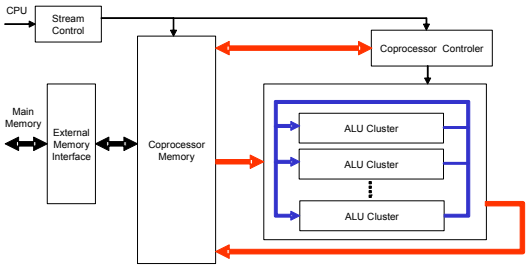

Figura 3: Vista lógica de un procesador de fragmentos moderno.

Inicialmente los kernels y las texturas de entrada son transferidas de la memoria principal del sistema a la memoria integrada en la tarjeta gráfica, que puede operar en paralelo con la CPU. En la implementación final, sólo existirá un reducido número de puntos de sincronización en los que la CPU dictaminará el orden de ejecución de los distintos *kernels*.

3.2. Modelado de un algoritmo genético en flujos

A la hora de transformar un algoritmo secuencial en un programa de procesado de flujos debemos tomar una serie de decisiones de diseño.

Consideramos dos problemas principales. Primero el algoritmo debe ser descompuesto en una serie de *kernels* sencillos que trabajan sobre flujos de datos homogéneos produciendo nuevos flujos de datos. Una vez realizada esta división, los datos deben ser reorganizados en los flujos necesarios para los distintos *kernels*. Estos flujos de datos se agrupan en una sola textura 2D.

El Algoritmo 1 representa el flujo lógico de kernels en los que hemos descompuesto el algoritmo genético. En este proceso estamos limitados por las características de la plataforma objetivo. Por ejemplo, no tenemos la posibilidad de generar números aleatorios en la tarjeta gráfica. En nuestro caso, solucionamos el problema introduciendo un *kernel* que obtiene nuevos números aleatorios a partir de un conjunto de ellos presentes en la textura. Estos números aleatorios iniciales son generados en la CPU y se escriben en la textura inicial. El flujo de *kernels* es controlado por la CPU, es decir, la GPU actúa como un coprocesador que inicia la ejecución de un determinado kernel sobre un determinado flujo de datos de entrada cuando se lo ordena la CPU.

```
Initialize_texture();
Initialize_population();

foreach generation
   foreach gene in an individual
        K1: Evaluate_gen_column();
   end foreach;
   K3: Select_individuals();
   K2: Refresh_random_stream();
   K4: Cross_Selected_Individuals();
   K2: Refresh_random_stream();
   foreach gene in an individual
        K5: Mutate_gen_column();
        K2: Refresh_random_stream();
   end foreach;
end foreach;
```

Algoritmo 1. Kernel Flow.

El proceso comienza por la transmisión de la textura inicial de la CPU a la GPU. Esta textura está lógicamente dividida en 5 regiones como se describe en la Figura 4. El *frame buffer* será mapeado sobre alguna región de esta textura gracias al uso del modo *render to texture*. Hasta seis regiones de la textura puede tomar cada *kernel* como flujos de datos de entrada.

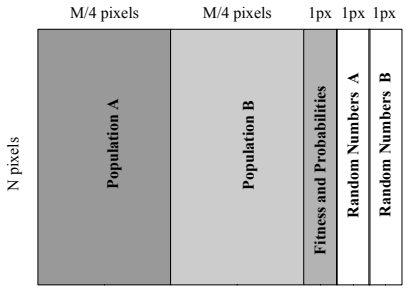

Figura 4: Vista lógica de la textura para un algoritmo genético con N individuos de M genes.

El *kernel* K2 se utiliza para regenerar el juego de números aleatorios inicialmente obtenidos de la CPU, manteniendo las propiedades estadísticas de los mismos. K2 será ejecutado cada vez que el área de números aleatorios sea utilizada como flujo de entrada.

El kernel de evaluación (K1) obtiene la aptitud (*fitness*) de cada individuo. En este caso, el *frame buffer* es mapeado sobre la zona de aptitud de la textura. Como flujo de entrada toma una columna de la Población A.

Las características paralelas de la arquitectura objetivo imponen el uso de un método de selección altamente paralelo, nosotros optamos por el torneo. El proceso se describe en la Figura 5.

La zona de escritura es en este caso la Población B, sobre la que debe mapearse el *frame buffer*. El kernel K3 recibe como flujos de entrada las áreas que contienen la aptitud de cada individuo y los parámetros de probabilidades del AG, los números aleatorios, y la Población A.

Para cada individuo de la Población B, se escogen dos individuos de la Población A (mediante los números aleatorios) y se hacen competir mediante torneo.

El individuo ganador es seleccionado y copiado en el *frame buffer*, en su lugar correspondiente de la Población B. Debemos remarcar que la selección de los N nuevos individuos se realiza en paralelo.

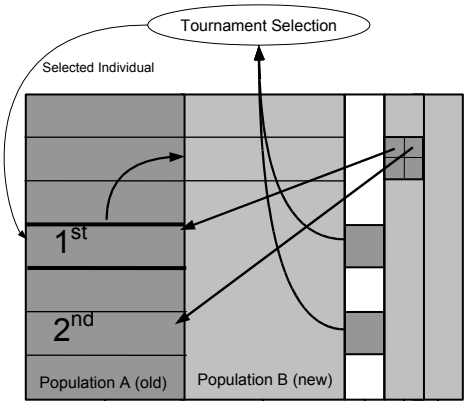

Figura 5: Descripción del kernel de selección (K3).

El kernel K4 se computa a continuación, realizando el cruce de los individuos seleccionados. La zona de escritura, sobre la que se mapea el *frame buffer* corresponde al área marcada como Población A. Como entrada se toman los flujos de datos que contienen los individuos previamente seleccionados (Población B), y los conjuntos de números aleatorios y probabilidades. Como se describe en la Figura 6, K4 cruza los individuos en una posición par dentro de la población con los de una impar. Se selecciona para los dos individuos un punto de cruce común aleatorio, utilizando para ello el flujo de píxeles aleatorios. Se decide si se cruzan usando la probabilidad de cruce y otro aleatorio. Los dos nuevos individuos se copian al *frame buffer; si se realiza el cruce*, cada uno de los hijos tiene una parte de cada padre; si no se cruzan, se copian los individuos originales.

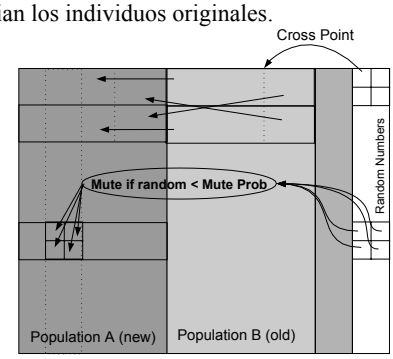

Figura 6: Descripción de los kernels de cruce (K4) y mutación (K5).

Finalmente, los nuevos individuos se mutan (K5). En este caso el *frame buffer* se mapea sobre cada columna del área marcada como Población A, que se utiliza también como flujo de datos de entrada. Esto es posible por que sólo se requiere los datos de el píxel a mutar, es decir, no necesitamos los datos del píxel vecino. Se decide si los genes contenidos en cada píxel de una columna en la Población A deben ser mutados, en función de los números aleatorios encontrados en el flujo de entrada.

4. Resultados experimentales

Para probar que el AG es eficiente sobre la tarjeta se han implementado AGs secuenciales utilizando la tarjeta gráfica. En esta primera comparativa, se ha contrastado la ejecución del AG onemax secuencial en la CPU con la ejecución secuencial en la GPU.

El problema OneMax es un problema muy simple que trata de maximizar el número de *unos* en una cadena de bits. Formalmente, este problema puede describirse como la búsqueda de una cadena $x=\{x1,x2,\ldots,xN\}$ con $xi \in \{0,1\}$, que maximiza la siguiente ecuación:

$$F\left(\vec{x}\right) = \sum_{i=1}^{N} x_i$$

Para definir una instancia del problema necesitamos solamente decidir el tamaño *n* de la cadena. Dado este valor *n* el óptimo se encuentra cuando el número de *unos* en la cadena es *n*, es decir, cuando todos los bits de la cadena son unos. Al utilizar OneMax únicamente se trata de comparar tiempos de ejecución en ambas plataformas y por ello se utiliza este problema sencillo como habitualmente en la literatura se hace para probar nuevas implementaciones.

Los resultados experimentales se han obtenido utilizando una CPU Pentium 3 a 933 MHz y una tarjeta GPU NVidia GeForce FX 5950. La implementación del AG sobre la GPU se ha llevado a cabo mediante código Cg. Los resultados se muestran en las gráficas 7 y 8. La Figura 7 compara el tiempo de ejecución de la CPU con la CPU para distinto número de individuos en la población. Por su parte la Figura

8 compara los tiempos de ejecución de ambas plataformas para distintas instancias del problema onemax, en concreto para valores de n igual a 24, 40, 100 y 400 genes.

A la vista de las gráficas, se comprueba que la GPU es tanto más eficiente cuanto mayor es la proporción entre el tamaño de la población y la longitud del cromosoma de cada individuo. Esto se debe a la imposición de la ejecución por columnas de los kernels K1 y K6, en vez de una ejecución completamente paralela sobre cada gen.

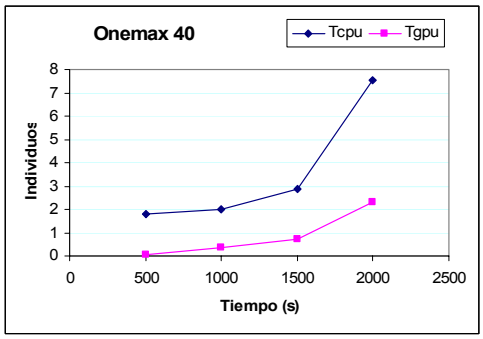

Figura 7: Tiempos comparados para diferentes poblaciones con un cromosoma de 40 genes.

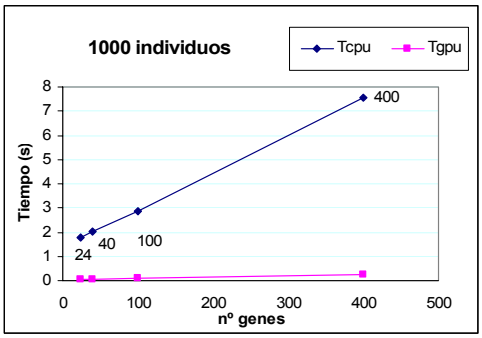

Figura 8: Tiempos comparados para diferentes longitudes de cromosoma.

Debido al mecanismo implementado para gestionar números aleatorios en la GPU (descrito en el apartado 3.2), los números aleatorios empleados difieren en el AG ejecutado sobre la CPU y el ejecutado sobre la GPU. Para corregir esta diferencia sería necesario emular la estructura de datos de los números aleatorios y el uso que de ella se hace en la GPU en el algoritmo que se corre sobre la CPU y, puesto que la CPU carece del paralelismo inherente a la GPU, se introduciría un coste excesivo e injusto en la comparativa en el algoritmo de la CPU.

Las poblaciones iniciales son idénticas en ambos algoritmos, pero los operadores genéticos funcionan con números aleatorios diferentes. Esto introduce diferencias en el número de operaciones (mutaciones, cruces, etc.) aplicadas en sendas versiones, que sin embargo no son significativas frente al coste que supone simular el mecanismo de los números aleatorios en la GPU sobre la CPU.

5. Conclusiones y trabajo futuro

Se han implementado una versión de un algoritmo genético simple sobre una tarjeta gráfica configurable. Este tipo de tarjeta está disponible en muchos de los equipos de sobremesa habituales sin que se haga ningún uso de ellos para ciertas aplicaciones. Los resultados muestran que es recomendable probar la ejecución de un algoritmo genético paralelo para aprovechar estos recursos. La GPU es más eficiente cuanto mayor es la dimensión del problema, aunque esta ganancia vendrá limitada por el tamaño de población máximo que se pueda procesar.

En la actualidad estamos trabajando en los AGPs propuestos en este trabajo y en sus distintas configuraciones.

6. Agradecimientos

Este artículo ha sido financiado por el proyecto del Ministerio de Ciencia y Tecnología TIC 2002-750 del Gobierno de España.

Referencias

[1] J. Holland. Adaptation in Natural and Artificial Systems. Univ. of Mitchigan Press, 1975.

[2] A E.-G. Talbi, A P. Bessi. A parallel genetic algorithm for the graph partitioning problem, Proceedings of the 5th international conference on Supercomputing, Cologne, West Germany, pp 312-320, 1991,ACM Press

[3] J.I. Hidalgo, M. Prieto, J. Lanchares, F.Tirado. A Parallel Genetic Algorithm for solving the Partitioning Problem in Multi-FPGA systems. Proceedings of 3rd international Meeting on vector and parallel processing. Oporto (Portugal), 21-23 de Junio de 1998. Páginas 717-722.

[4] J. Herrera, E. Huedo, R.S. Montero, and I.M. Llorente. A Grid-Oriented Genetic Algorithm, in P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 315–322, 2005. Springer-Verlag, 2005

[5] M. Prieto. Paralelizacion de Metodos Multimalla Robustos. PhD thesis, Universidad Complutense de Madrid, 2000.

[6] Mpi: A message passing interface standard. http://www.mpi-forum.org.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. PVM: Parallel Virtual Machine. A users guide and tutorial for network parallel Computers. MIT Press Books, Massachussets, 1994.

[8] H. Muhlenbein. Parallel genetic algorithms, population genetic and combinatorial optimization. In Parallel Problem Solving from Nature - Proceedings of 1st Workshop PPSN, volume 496, pages 407-417. Lecture Notes in Computer Science, 1991.

[9] E. Cantu-Paz. Migration policies and takeover times in parallel genetic algorithms. In Morgan Kaufmann, editor, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99, page 775, San Francisco, CA, July 1999.

[10] E. Cantu-Paz. Topologies, migration rates, and multi-population parallel genetic algorithms. In Morgan Kaufmann, editor, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-99, pages 91-98, San Francisco, CA, July 1999.

[11] E. Cantu-Paz. Efficient and Accurate Parallel Genetic Algorithms. Kluwer Academic Publishers, Boston, MA, 2000.

[12] E. Cantu-Paz and D.E. Goldberg. Efficient parallel genetic algorithms: theory and practice. Computer Methods in Applied Mechanics and Engineering, (186):221-238, 2000.