Tercera Entrega

Sistemas complejos en máquinas paralelas:

OpenMP y Pthreads

2do Cuatrimestre 2014

Alumno: Ezequiel Darío Gambaccini

LU: 715/13

Realizar los siguientes ejercicios:

1. Implementar la ecuación logística vista en las clases teóricas en forma serial y paralela mediante OpenMP.

Uno de los ejemplos más sencillos y utilizados para ilustrar los sistemas dinámicos no lineales y el caos es lo que se conoce como la ecuación logística:

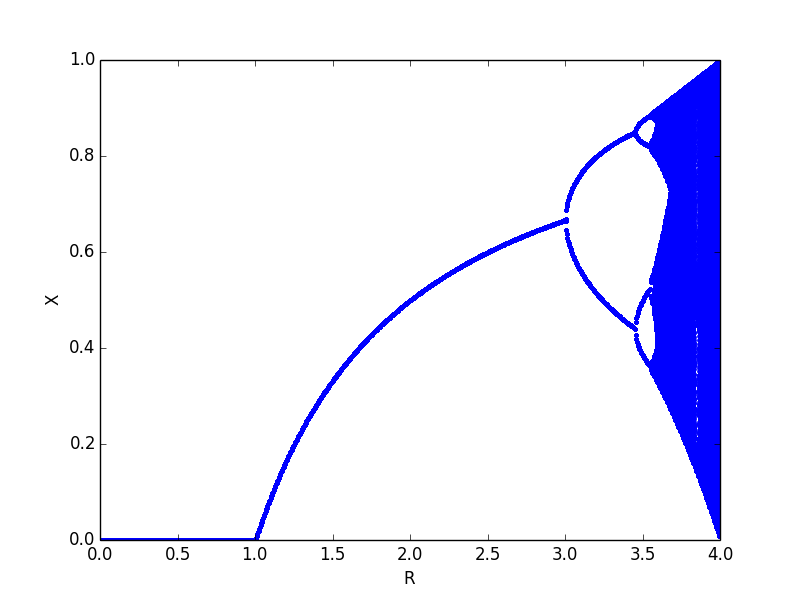
donde 0 ≤ r ≤ 4, y X0 = 0,5.

2. Implementar el algoritmo DLA (Diffusion-limited aggregation) visto en las clases teóricas de forma serial y paralela mediante Pthreads.

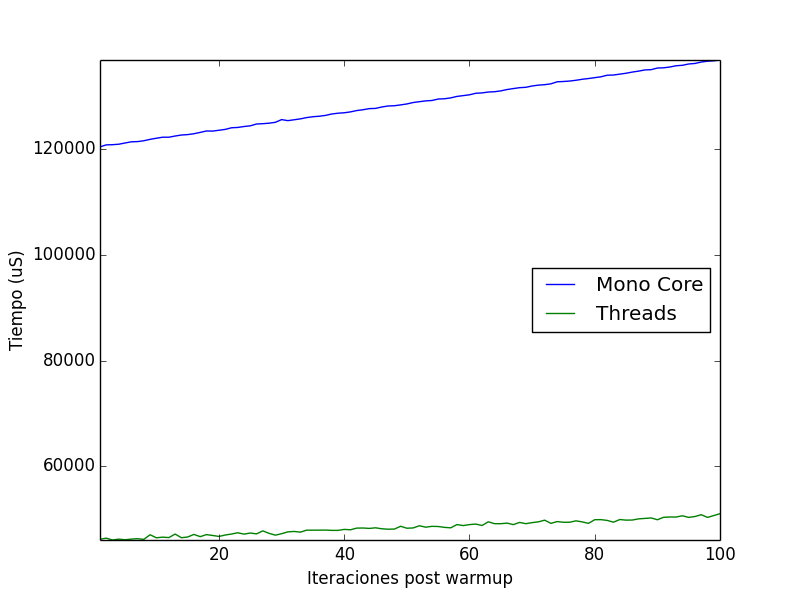
3. Elegir e implementar alguno de los algoritmos vistos en las clases teóricas o prácticas (excluidos los ya pedidos en las entregas hasta ahora) de forma serial y paralela. Para la versión paralela use OpenMP o bien Pthreads.

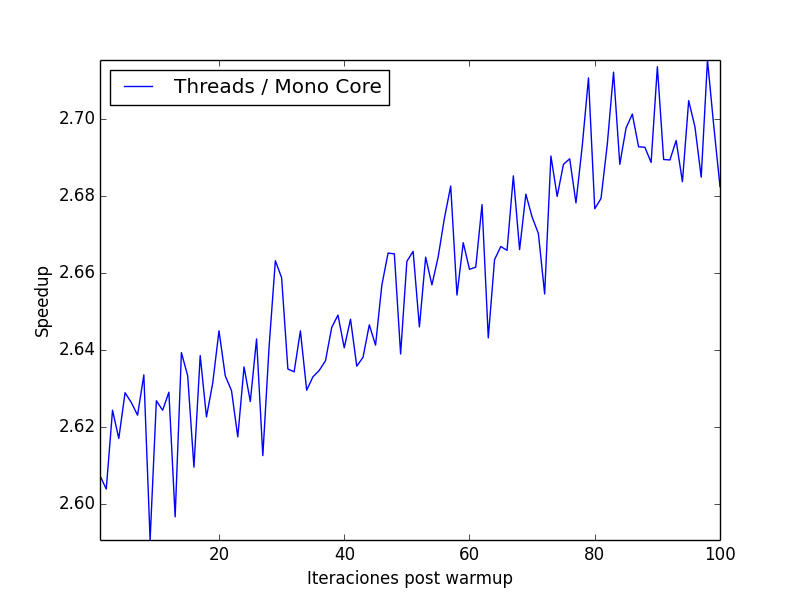
1) Mapa logístico generado con pasos de 4/1000 en r, con un warmup inicial de 10000 iteraciones para cada paso, guardando las siguientes 1000 iteraciones.

Se puede ver claramente el comportamiento caótico a partir de r = 3 hasta r = 4.



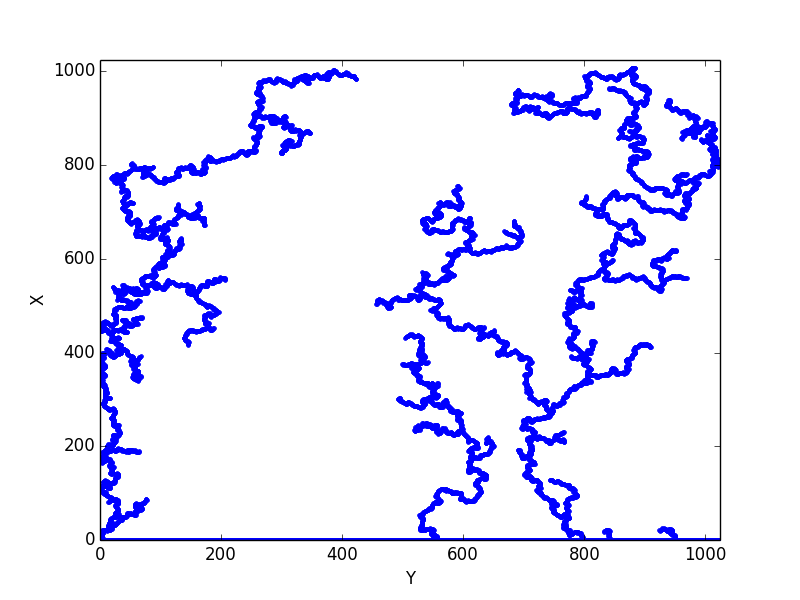
Comparación de tiempos entre monocore y openmp.





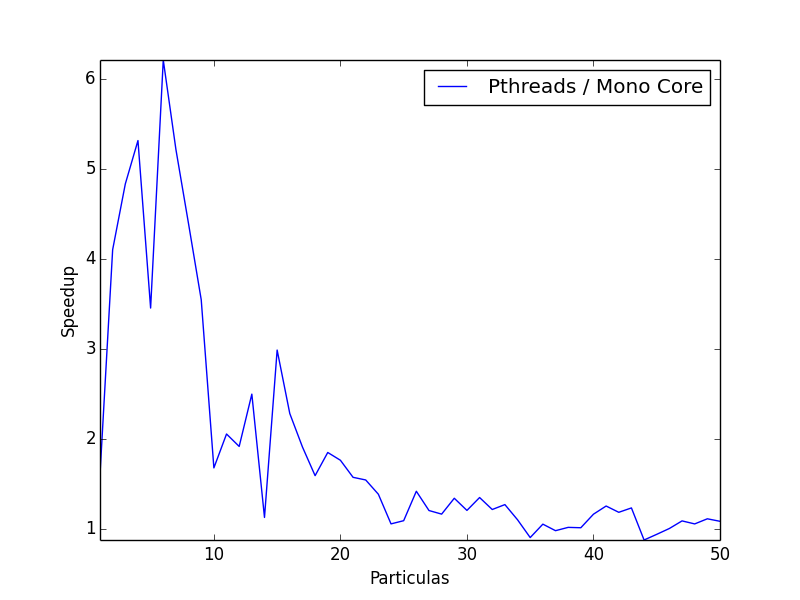
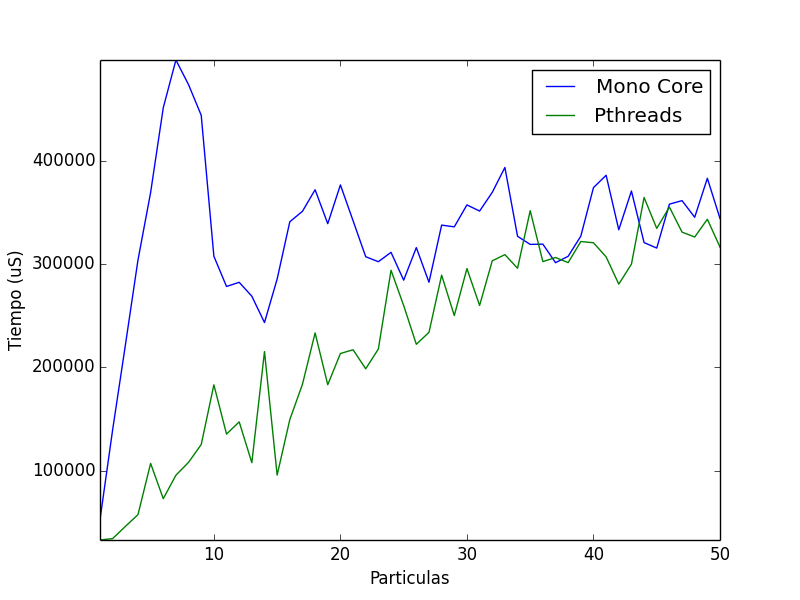
El código openmp fue corrido en una máquina quad core i5-3330 @ 3.00 GHz.

La mayor dificultad con este código fue entender como usar openmp, y crear las variables locales a cada thread.

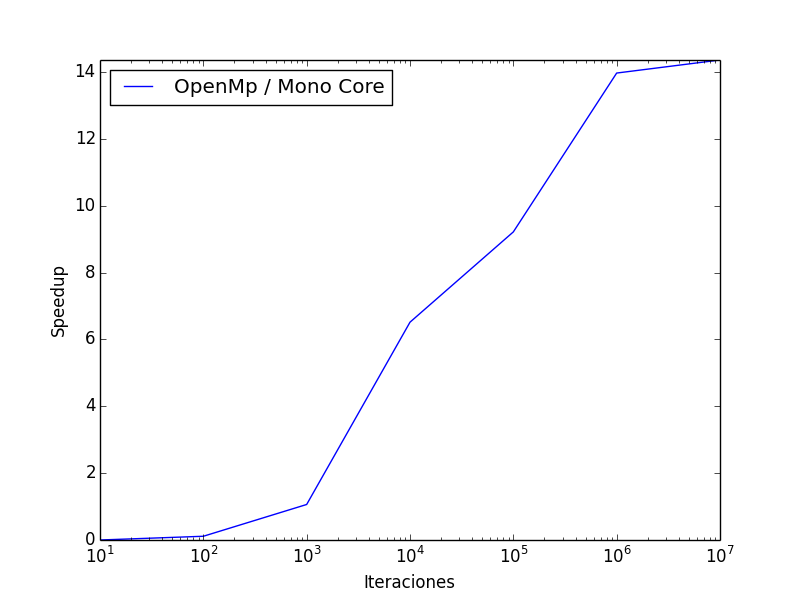
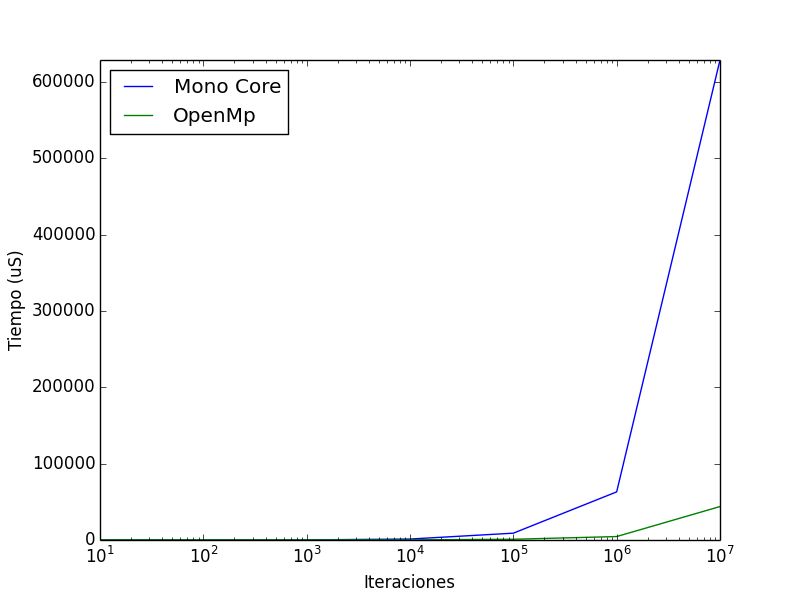
2) Fractal DLA generado con 10000 partículas, generando la acumulación en el borde inferior de la imagen.

El mayor problema con este algoritmo fue asegurarse que el comportamiento aleatorio de las partículas. Para lograr esto, se diseño desde un primer momento el programa serial usando funciones random thread-safe, por lo que portarlo para usar pthreads fue trivial. Los threads se ejecutan independientemente entre ellos, todos accediendo al mismo tiempo a la matriz binaria que representa a la imagen. No hay problema de data races ya que lo único que harían si sobreescriben posiciones al mismo tiempo sería escribir el mismo valor final (1), y además, agrega aleatoriedad al problema.

Las siguientes mediciones de performance no son precisas, debido a la aleatoriedad del problema, pero por pruebas para generar los fractales, con 10000 partículas, el código multithreadeado es mucho más rápido que el serial.

Para 50 partículas, cada partícula iterada 10 veces, y el tiempo promediado, se observó lo siguiente:

3) El algoritmo elegido para este punto fue calcular pi usando Montecarlo, y luego paralelizar el mismo. Este algoritmo tuvo los mismos inconvenientes que el del mapa logístico, es decir, aprender como usar openmp de manera correcta. Para lograr esto, se decidió usar un algoritmo de generador de números pseudoaleatorios en vez de las funciones de números aleatorios de la librería estándar.



**Programas extra:**

Para ejecutar los diferentes problemas y crear un archivo de salida con varias iteraciones, en cada carpeta hay un archivo \*\_run.py (map\_run.py, dla\_run.py, pi\_run.py). Se ejecutan de la siguiente manera:

python \*\_run.py ./exe salida.txt

Donde:

\*\_run.py es uno de map\_run.py, dla\_run.py, pi\_run.py.

exe es uno de logistic\_map\_serial, logistic\_map\_mp, dla\_serial, dla\_threads, pi\_monte\_carlo\_serial, pi\_monte\_carlo\_mp.

Para graficar las comparaciones, en cada carpeta hay un script plot\_speed.py.

Se ejecuta de la siguiente manera:

python plot\_speed.py mono.txt dual.txt

Donde mono.txt es la salida de la ejecución serial, y dual.txt es la salida de la ejecución paralela.

En el caso del mapa logístico y de los fractales dla, hay scripts específicos para graficar sus resultados.

Para el mapa logístico:

python plot\_map.py map.txt

Donde map.txt es la salida de ejecutar el mapa logístico.

Para dla:

python plot\_fractal.py fractal.txt

Donde fractal.txt es la salida de ejecutar dla.