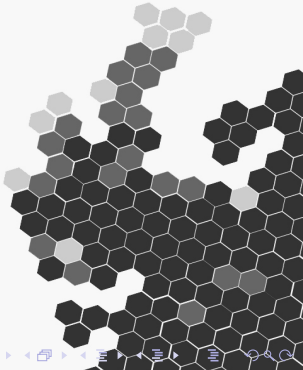
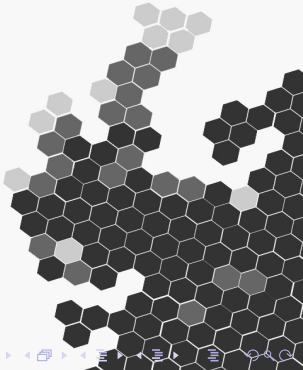
Breve introducción a la programación en paralelo

José Carlos García

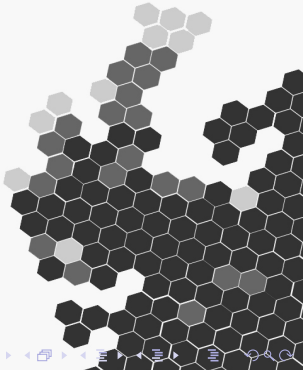


Introducción



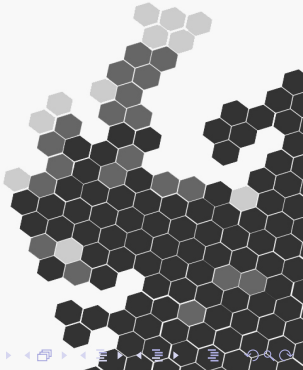
Ejemplos en la vida cotidiana

- Supongamos que **tenemos un restaurante.**



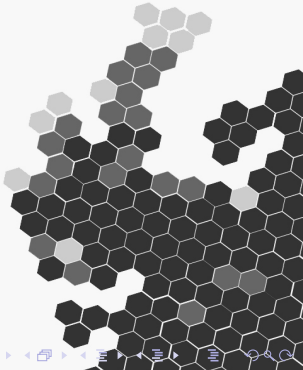
Ejemplos en la vida cotidiana

- Supongamos que **tenemos un restaurante.**
- Si tenemos **una persona trabajando a una velocidad media**, y nuestro restaurante **hay** $n = 10$ **mesas ocupadas**, ¿Qué sucede?



Ejemplos en la vida cotidiana

- Supongamos que **tenemos un restaurante.**
- Si tenemos **una persona trabajando a una velocidad media**, y nuestro restaurante **hay** $n = 10$ **mesas ocupadas**, ¿Qué sucede?
- Dado que tenemos una persona, y no un pulpo, **sólo puede atender a una persona a la vez.**



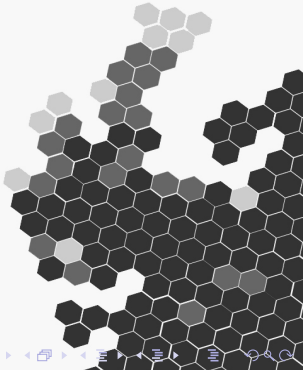
Ejemplos en la vida cotidiana

- Supongamos que **tenemos un restaurante.**
- Si tenemos **una persona trabajando a una velocidad media**, y nuestro restaurante **hay** $n = 10$ **mesas ocupadas**, ¿Qué sucede?
- Dado que tenemos una persona, y no un pulpo, **sólo puede atender a una persona a la vez.**
- En este caso, tendríamos un ejemplo de **un trabajo que no se está realizando en paralelo (o de manera secuencial)**, y nuestro solitario camarero estará realizando n operaciones él solito

Ejemplos en la vida cotidiana

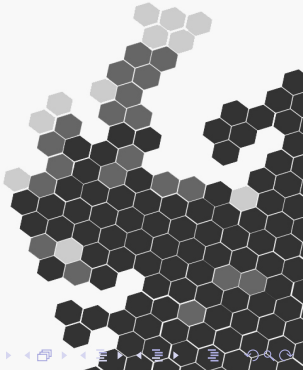
- Supongamos que **tenemos un restaurante.**
- Si tenemos **una persona trabajando a una velocidad media**, y nuestro restaurante **hay** $n = 10$ **mesas ocupadas**, ¿Qué sucede?
- Dado que tenemos una persona, y no un pulpo, **sólo puede atender a una persona a la vez.**
- En este caso, tendríamos un ejemplo de **un trabajo que no se está realizando en paralelo (o de manera secuencial)**, y nuestro solitario camarero estará realizando n operaciones él solito
- **¿Cual es la mejor solución de nuestro problema?**
  ¿Contratamos más camareros de velocidad media?
  ¿Contratamos el mejor camarero del país?

Aplicaciones en el mundo matemático



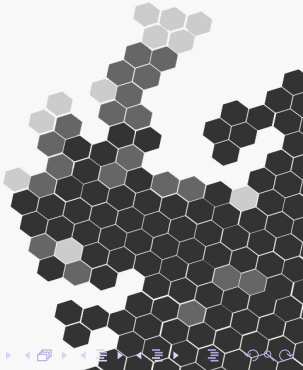
Operaciones con vectores

- Sean $v, u \in \mathbb{R}^n$ si queremos calcular $w = u + v$ debemos calcular la suma de cada componente, esto es:



Operaciones con vectores

- Sean $v, u \in \mathbb{R}^n$ si queremos calcular $w = u + v$ debemos calcular la suma de cada componente, esto es:
- $w_i = v_i + u_i$ con $i = 1, 2, 3, ..., n$ (Observamos que cada w_i es independiente del resto)



Operaciones con vectores

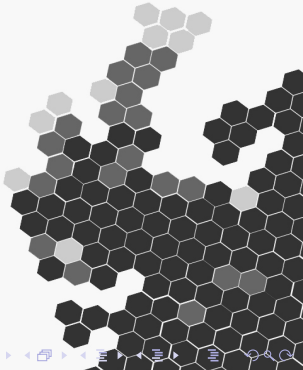
- Sean $v, u \in \mathbb{R}^n$ si queremos calcular $w = u + v$ debemos calcular la suma de cada componente, esto es:
- $w_i = v_i + u_i$ con $i = 1, 2, 3, ..., n$ (Observamos que cada w_i es independiente del resto)
- Por tanto, tenemos que hacer n operaciones para obtener el valor de w. Sin embargo, si ejecutamos estas n operaciones en paralelo, nos costaría casi lo mismo que hacer la suma de dos números.

Operaciones con vectores

- Sean $v, u \in \mathbb{R}^n$ si queremos calcular $w = u + v$ debemos calcular la suma de cada componente, esto es:
- $w_i = v_i + u_i$ con $i = 1, 2, 3, ..., n$ (Observamos que cada w_i es independiente del resto)
- Por tanto, tenemos que hacer n operaciones para obtener el valor de w. Sin embargo, si ejecutamos estas n operaciones en paralelo, nos costaría casi lo mismo que hacer la suma de dos números.
- Esta misma idea es aplicable también para calcular el producto de un número λ y un vector u.

Producto escalar: No es paralelizable

- En este ejemplo, veremos que **no siempre se puede paralelizar un problema**, y veremos uno de los problemas más comunes a la hora de paralelizar; **Las condiciones de carrera**



Producto escalar: No es paralelizable

- En este ejemplo, veremos que **no siempre se puede paralelizar un problema**, y veremos uno de los problemas más comunes a la hora de paralelizar; **Las condiciones de carrera**
- Sea $v, u \in \mathbb{R}^n$ **dos vectores a los que queremos calcular el producto escalar**, $s = u \cdot v$. Recordemos que el producto escalar se puede calcular como la suma del producto de las componentes.

Producto escalar: No es paralelizable

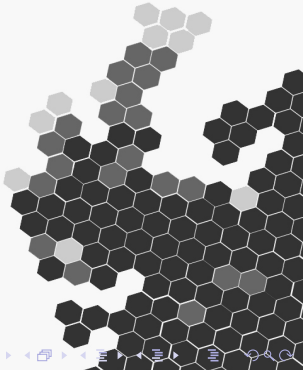
- En este ejemplo, veremos que **no siempre se puede paralelizar un problema**, y veremos uno de los problemas más comunes a la hora de paralelizar; **Las condiciones de carrera**
- Sea $v, u \in \mathbb{R}^n$ **dos vectores a los que queremos calcular el producto escalar**, $s = u \cdot v$. Recordemos que el producto escalar se puede calcular como la suma del producto de las componentes.
- Para programar esto, tenemos que ir sumando a una variable real s el producto de todos los componentes, $v_i * u_i$.

Producto escalar: No es paralelizable

- En este ejemplo, veremos que **no siempre se puede paralelizar un problema**, y veremos uno de los problemas más comunes a la hora de paralelizar; **Las condiciones de carrera**
- Sea $v, u \in \mathbb{R}^n$ **dos vectores a los que queremos calcular el producto escalar**, $s = u \cdot v$. Recordemos que el producto escalar se puede calcular como la suma del producto de las componentes.
- Para programar esto, tenemos que ir sumando a una variable real s el producto de todos los componentes, $v_i * u_i$.
- Supongamos que queremos hacerlo en paralelo, y sucede que las componentes i, j se ejecutan al mismo tiempo.

Producto escalar: No es paralelizable

- Por tanto, al mismo tiempo se ejecutará $s = s + v_i * u_i$ y $s = s + v_j * u_j$.



Producto escalar: No es paralelizable

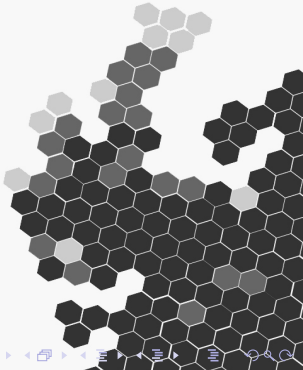
- Por tanto, al mismo tiempo se ejecutará $s = s + v_i * u_i$ y $s = s + v_j * u_j$.
- Al realizarse al mismo tiempo, el valor s que se suma al producto, será el mismo, por tanto, dado que sólo se asignará un valor de s (Supongamos que es i) el valor $v_j * u_j$ no se sumará a s.

Producto escalar: No es paralelizable

- Por tanto, al mismo tiempo se ejecutará $s = s + v_i * u_i$ y $s = s + v_j * u_j$.
- Al realizarse al mismo tiempo, el valor s que se suma al producto, será el mismo, por tanto, dado que sólo se asignará un valor de s (Supongamos que es i) el valor $v_j * u_j$ no se sumará a s.
- Por tanto, el producto escalar no podemos paralelizarlo, pues **tenemos una condición de carrera.**

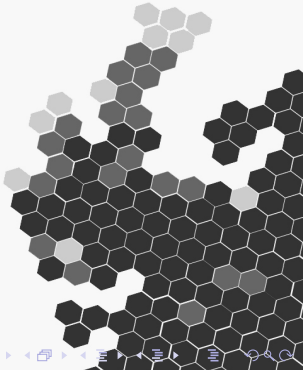
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.



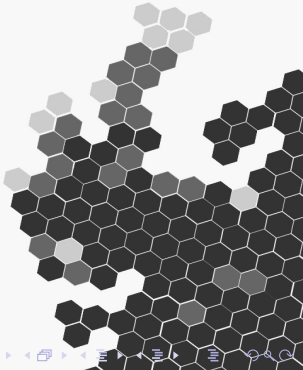
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...



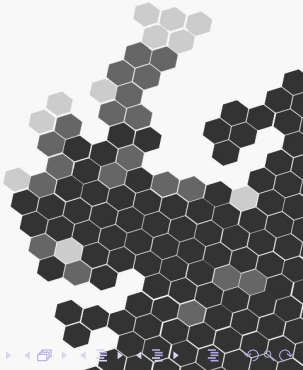
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.



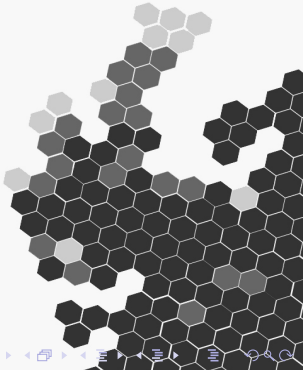
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.



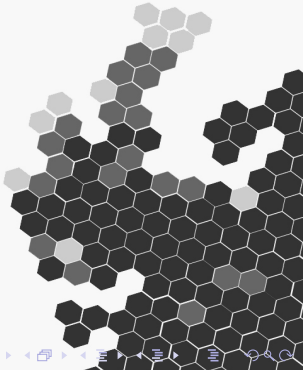
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.



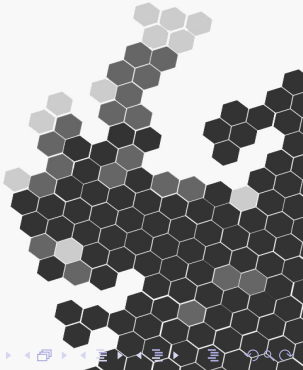
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.
- Acelerar método de diferencias finitas.



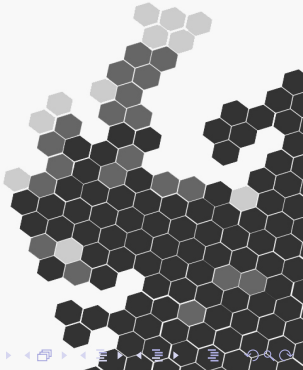
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.
- Acelerar método de diferencias finitas.
- Trabajar con Big Data.



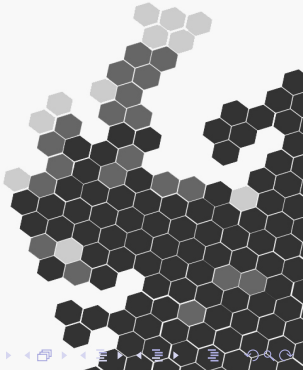
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.
- Acelerar método de diferencias finitas.
- Trabajar con Big Data.
  Podemos procesar muchos datos a la vez.



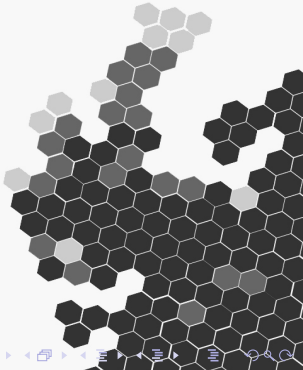
Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.
- Acelerar método de diferencias finitas.
- Trabajar con Big Data.
  Podemos procesar muchos datos a la vez.
- Acelerar todas las tareas que se hacen por fuerza bruta;
  Romper contraseñas, encontrar combinaciones, ...

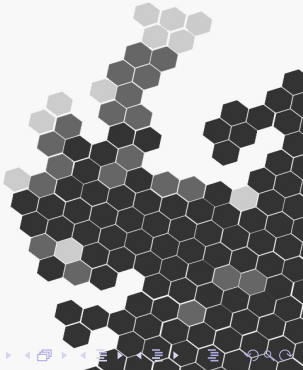


Otras aplicaciones

- Resolución de sistemas de ecuaciones; Jacobi es fácilmente paralizable.
- Criptografía; Encontrar primos, blockchain...
- Cálculos con matrices.
- Encontrar mínimos de funciones.
- Encontrar gran cantidad de raíces de una función a la vez.
- Acelerar método de diferencias finitas.
- Trabajar con Big Data.
  Podemos procesar muchos datos a la vez.
- Acelerar todas las tareas que se hacen por fuerza bruta;
  Romper contraseñas, encontrar combinaciones, ...
- ...

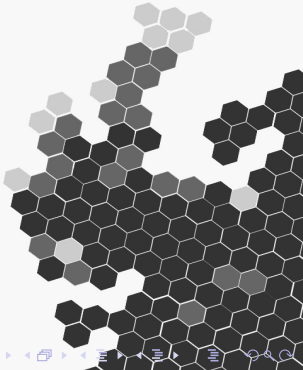


Algunos conceptos básicos



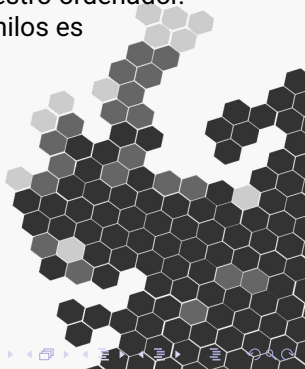
Conceptos básicos

- Decimos que **un algoritmo de *n* pasos se puede paralelizar** si cada *n* iteración no depende del resto de iteraciones.



Conceptos básicos

- Decimos que **un algoritmo de *n* pasos se puede paralelizar** si cada *n* iteración no depende del resto de iteraciones.
- Formas de paralelización:
  - **CPU:** Utilizando los hilos disponibles en el procesador de nuestro ordenador. Todos los hilos son de alto rendimiento, pero la cantidad de hilos es bastante pequeña.



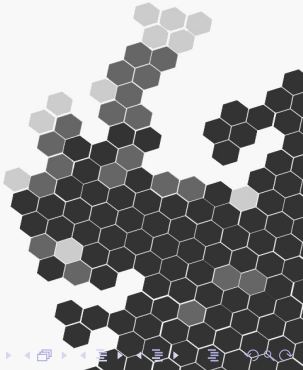
Conceptos básicos

- Decimos que **un algoritmo de *n* pasos se puede paralelizar** si cada *n* iteración no depende del resto de iteraciones.
- Formas de paralelización:
  - **CPU:** Utilizando los hilos disponibles en el procesador de nuestro ordenador. Todos los hilos son de alto rendimiento, pero la cantidad de hilos es bastante pequeña.
  - **GPU:** Utilizando los hilos disponibles en la tarjeta gráfica de nuestro ordenador. Los hilos tienen un menor desempeño que los de la CPU, sin embargo, contiene una gran cantidad de hilos.

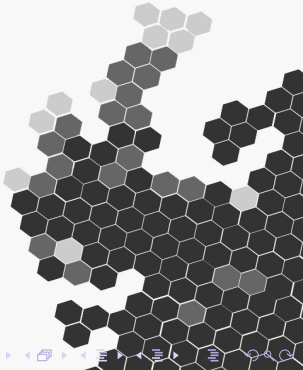
Conceptos básicos

- Decimos que **un algoritmo de *n* pasos se puede paralelizar** si cada *n* iteración no depende del resto de iteraciones.
- Formas de paralelización:
  - **CPU:** Utilizando los hilos disponibles en el procesador de nuestro ordenador. Todos los hilos son de alto rendimiento, pero la cantidad de hilos es bastante pequeña.
  - **GPU:** Utilizando los hilos disponibles en la tarjeta gráfica de nuestro ordenador. Los hilos tienen un menor desempeño que los de la CPU, sin embargo, contiene una gran cantidad de hilos.
  - **Red:** Distribuimos el trabajo entre varios ordenadores a través de una conexión de red.

Programar en paralelo

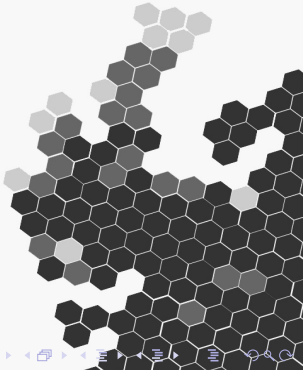


Trabajar con la CPU



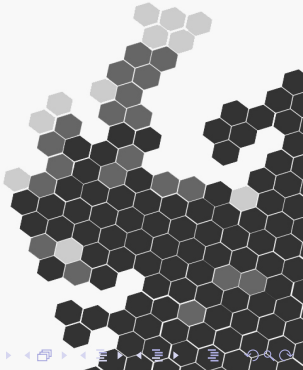
Wolfram Language

- Función *ParallelX[]*.



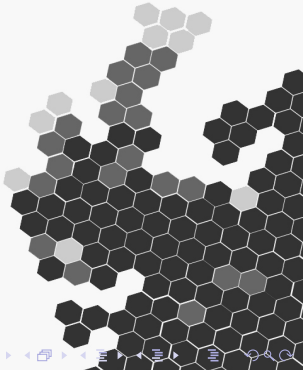
Wolfram Language

- Función *ParallelX[]*.



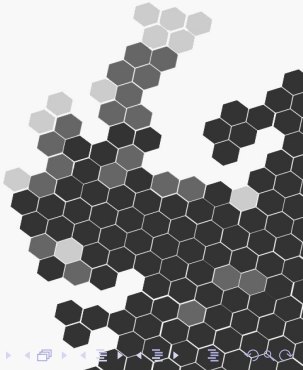
Wolfram Language

- Función *ParallelX[]*.
- **Demo de *ParallelX***



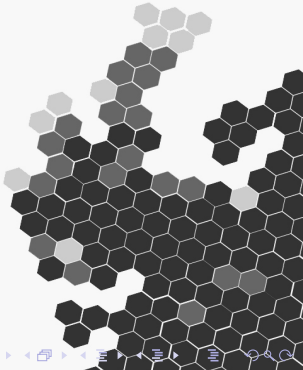
Wolfram Language

- Función *ParallelX[]*.
- **Demo de *ParallelX***
- Función *Parallelize[]*



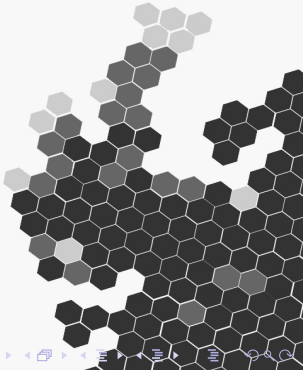
Wolfram Language

- Función *ParallelX[]*.
- **Demo de *ParallelX***
- Función *Parallelize[]*



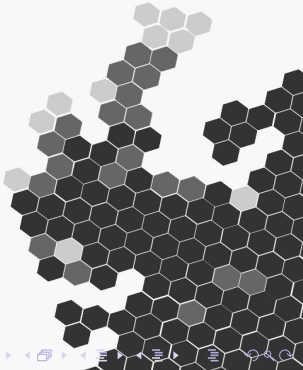
Wolfram Language

- Función *ParallelX[]*.
- **Demo de *ParallelX***
- Función *Parallelize[]*
- **Demo de *Parallelize***

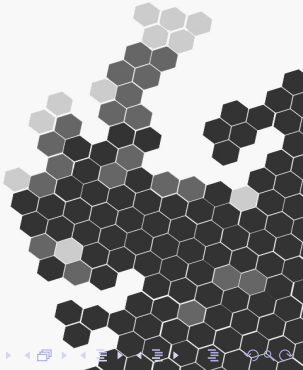


Wolfram Language

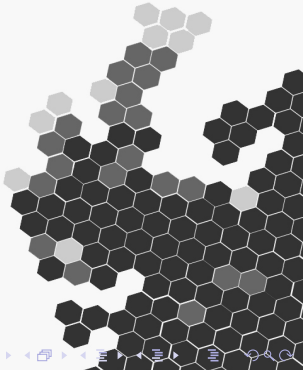
- Función *ParallelX[]*.
- **Demo de *ParallelX***
- Función *Parallelize[]*
- **Demo de *Parallelize***



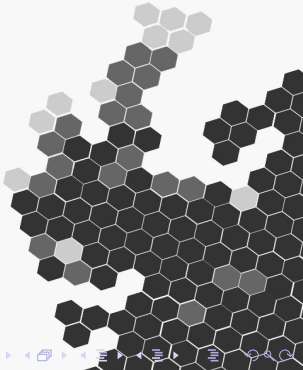
- *lapply*:



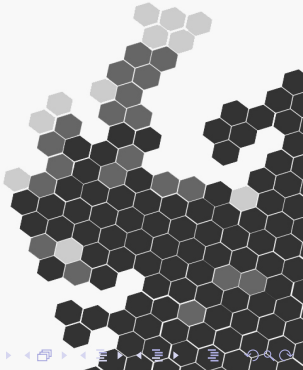
- *lapply*: Aplicar una lista a una función y obtener el resultado.



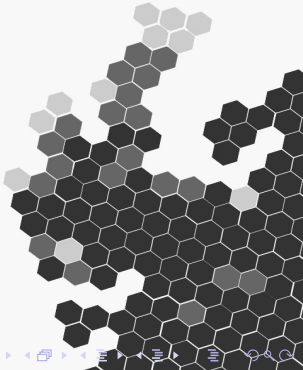
- *lapply*: Aplicar una lista a una función y obtener el resultado.
- *Librería paralell*:



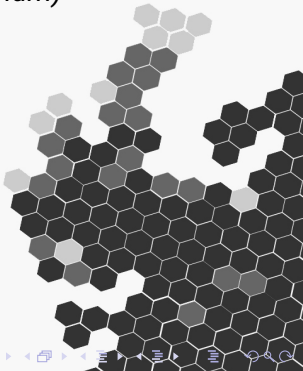
- *lapply*: Aplicar una lista a una función y obtener el resultado.
- *Librería paralell*: Esta librería nos permite paralelizar funciones en R.



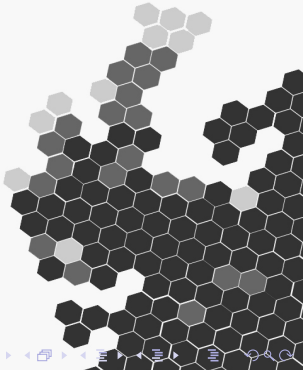
- *lapply*: Aplicar una lista a una función y obtener el resultado.
- *Librería paralell*: Esta librería nos permite paralelizar funciones en R. Tenemos dos funciones importantes; *detectCores()* nos detecta el número de núcleos de nuestro ordenador.



- *lapply*: Aplicar una lista a una función y obtener el resultado.
- *Librería paralell*: Esta librería nos permite paralelizar funciones en R. Tenemos dos funciones importantes; *detectCores()* nos detecta el número de núcleos de nuestro ordenador. *makeCluster(corenum)* inicializa una instancia en paralelo, donde corenum es el número de núcleos que quieres usar.
- *parLapply*: Igual que la función lapply, pero en paralelo.
- *Otras funciones interesantes:* Paquete foreach, nos permite ejecutar una acción en una lista de elementos.
- **Demo**

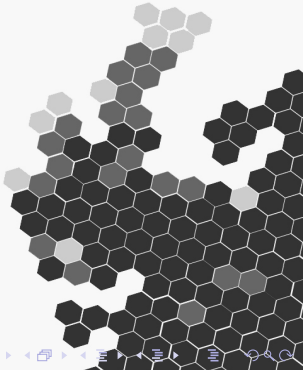


C: PThread & OpenMP (Unix)



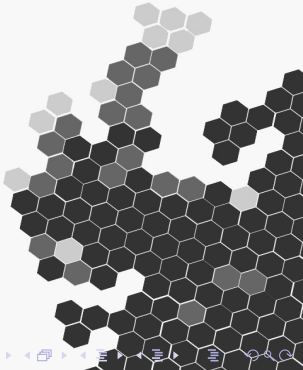
C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h



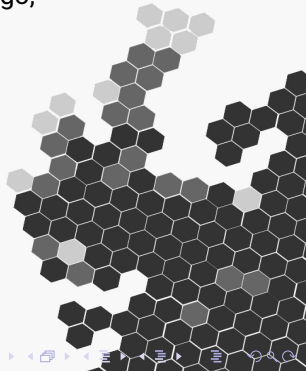
C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h
  - Tenemos que incluir *pthread.h*



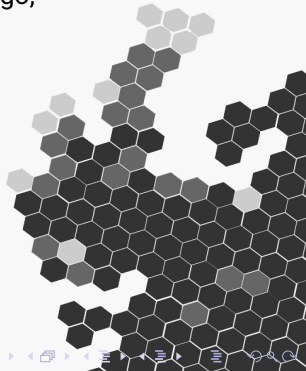
C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h
  - Tenemos que incluir *pthread.h*
  - Para ejecutar una función en paralelo, la llamamos con *phtread_create*.
  - Si queremos esperar que un hilo termine antes de ejecutar algo, llamamos *pthread_join*
- Con OpenMP todo esto se simplifica,



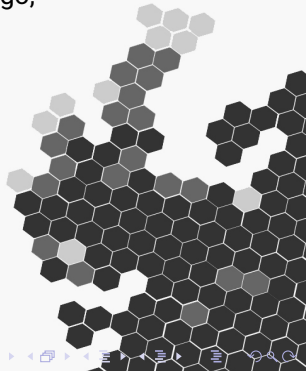
C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h
  - Tenemos que incluir *pthread.h*
  - Para ejecutar una función en paralelo, la llamamos con *phtread_create*.
  - Si queremos esperar que un hilo termine antes de ejecutar algo, llamamos *pthread_join*
- Con OpenMP todo esto se simplifica,
  - #pragma omp parallel for



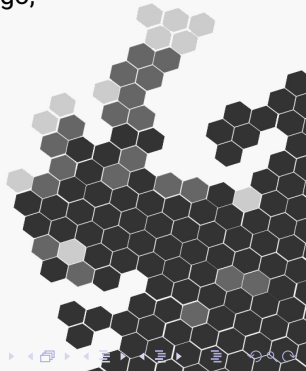
C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h
  - Tenemos que incluir *pthread.h*
  - Para ejecutar una función en paralelo, la llamamos con *phtread_create*.
  - Si queremos esperar que un hilo termine antes de ejecutar algo, llamamos *pthread_join*
- Con OpenMP todo esto se simplifica,
  - #pragma omp parallel for
  - #pragma omp parallel
  - ...

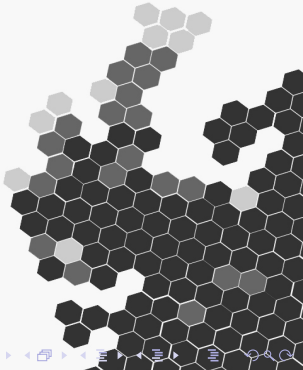


C: PThread & OpenMP (Unix)

- La manera más convencional es usando pthread.h
  - Tenemos que incluir *pthread.h*
  - Para ejecutar una función en paralelo, la llamamos con *phtread_create*.
  - Si queremos esperar que un hilo termine antes de ejecutar algo, llamamos *pthread_join*
- Con OpenMP todo esto se simplifica,
  - #pragma omp parallel for
  - #pragma omp parallel
  - ...
- **DEMO**

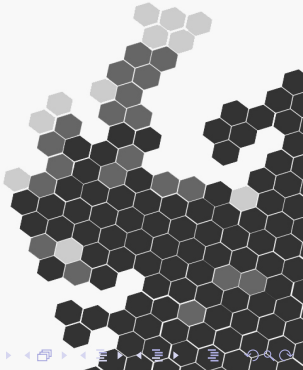


Trabajar con la GPU



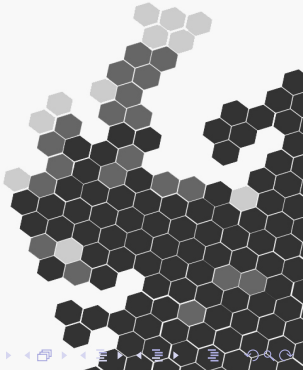
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.



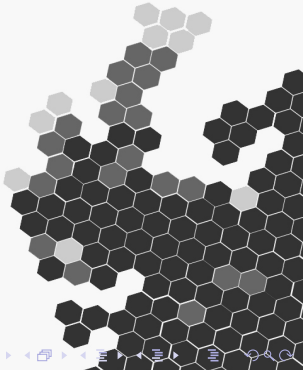
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.



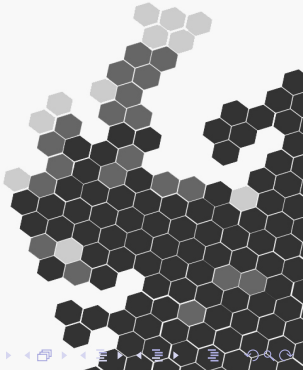
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.
- Por tanto, necesitamos unas funciones diferentes para reservar memoria para CPU y GPU.



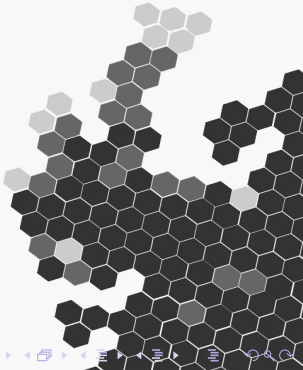
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.
- Por tanto, necesitamos unas funciones diferentes para reservar memoria para CPU y GPU.
- Las funciones que definamos en la CPU, en general, no son accesibles por la GPU.



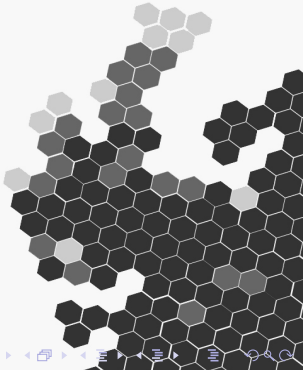
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.
- Por tanto, necesitamos unas funciones diferentes para reservar memoria para CPU y GPU.
- Las funciones que definamos en la CPU, en general, no son accesibles por la GPU.
- Las variables que definamos en la CPU, en general, no son accesibles desde la GPU.



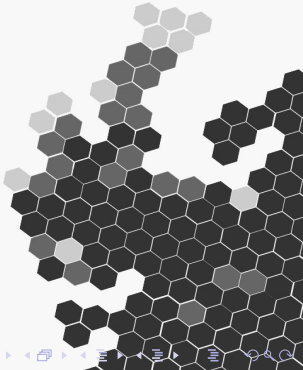
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.
- Por tanto, necesitamos unas funciones diferentes para reservar memoria para CPU y GPU.
- Las funciones que definamos en la CPU, en general, no son accesibles por la GPU.
- Las variables que definamos en la CPU, en general, no son accesibles desde la GPU.
- Existen compiladores libres, como OpenCL, sin embargo, el compilador propietario CUDA funciona mejor en gráficas NVIDIA.



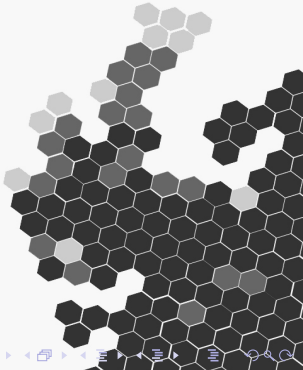
Problemas

- La GPU y la CPU por lo general, **NO COMPARTEN** memoria.
- Copiar memoria de la CPU a la GPU (y viceversa) es *lento*.
- Por tanto, necesitamos unas funciones diferentes para reservar memoria para CPU y GPU.
- Las funciones que definamos en la CPU, en general, no son accesibles por la GPU.
- Las variables que definamos en la CPU, en general, no son accesibles desde la GPU.
- Existen compiladores libres, como OpenCL, sin embargo, el compilador propietario CUDA funciona mejor en gráficas NVIDIA.
- Si queremos trabajar con matrices como en C, se complica.



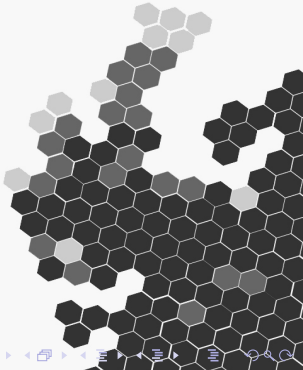
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.



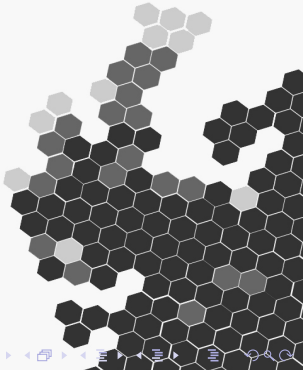
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.



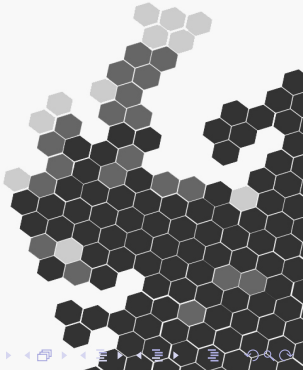
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)



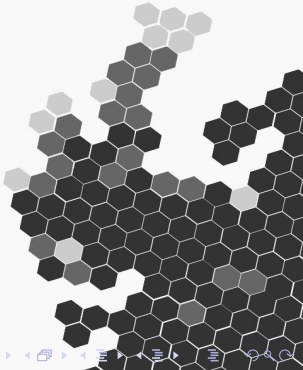
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)
- Para definir una función que se llame en la GPU, debemos poner delante *__global__* ;



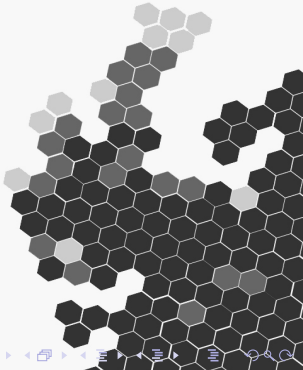
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)
- Para definir una función que se llame en la GPU, debemos poner delante *__global__* ; *__global__ void funcion(...)*



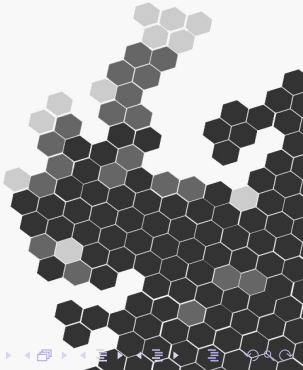
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)
- Para definir una función que se llame en la GPU, debemos poner delante *__global__* ; *__global__ void funcion(...)*
- Para llamarla, escribimos *funcion«bloques, hilos»(args)*



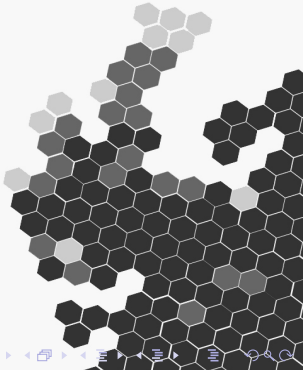
CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)
- Para definir una función que se llame en la GPU, debemos poner delante *__global__* ; *__global__ void funcion(...)*
- Para llamarla, escribimos *funcion«bloques, hilos»(args)*
- Wolfram Language, permite ejecutar código en CUDA.

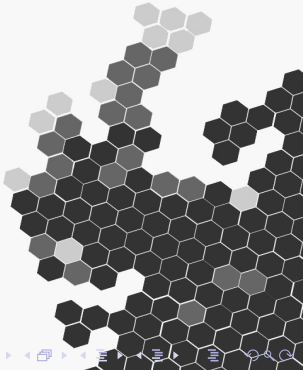


CUDA

- Es un lenguaje desarrollado por NVIDIA basado en C++, orientado a programar en GPU.
- *cudaMalloc:* Equivalente a malloc, pero para reservar memoria en la GPU. También tenemos *cudaFree*.
- *cudaMemcpy:* Nos permite copiar memoria de la CPU a la GPU (y viceversa)
- Para definir una función que se llame en la GPU, debemos poner delante *__global__* ; *__global__ void funcion(...)*
- Para llamarla, escribimos *funcion«bloques, hilos»(args)*
- Wolfram Language, permite ejecutar código en CUDA.
- **DEMO**

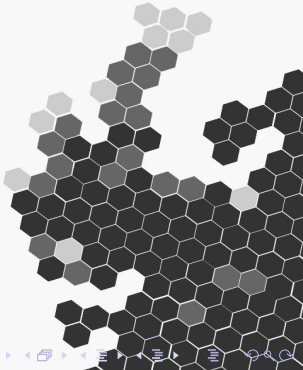


Conclusiones



Entorno de pruebas

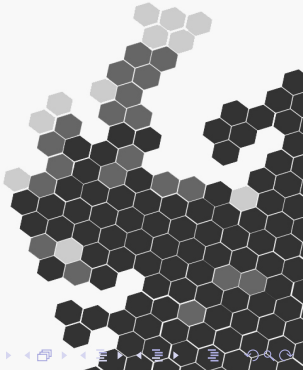
Todas las pruebas se han realizado se han realizado en un ordenador con las siguientes características:



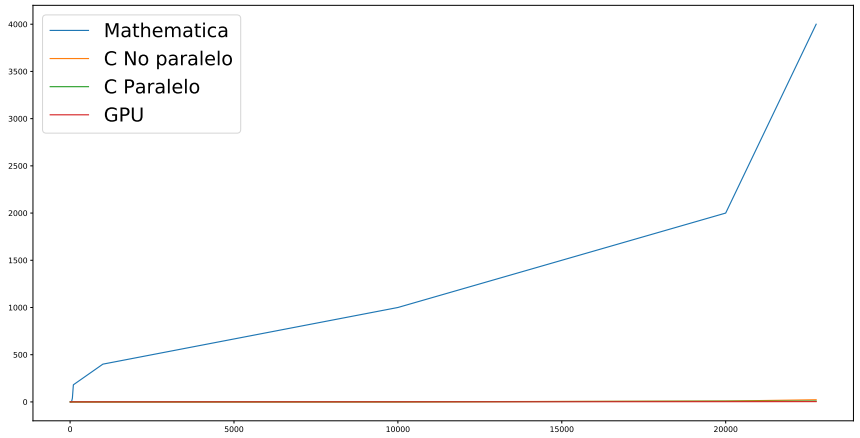
Entorno de pruebas

Todas las pruebas se han realizado se han realizado en un ordenador con las siguientes características:

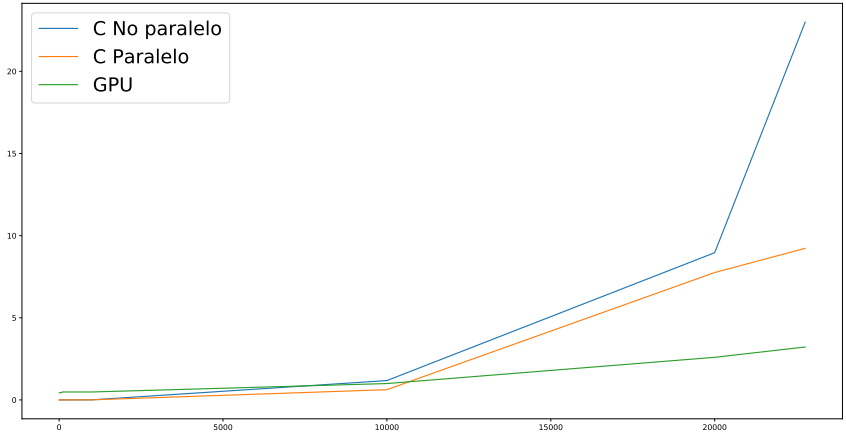
- **Procesador:** Intel i7-5500U (4) @ 3.0GHz
- **Sistema operativo:** Debian GNU/Linux 9.4 (stretch) x86_64
- **Núcleo:** 4.9.0-6-amd64
- **Tarjeta gráfica:** NVIDIA GeForce 930M (2GB RAM)
- **Memoria:** 12GB RAM



Mathematica VS C VS CUDA (Segundos)



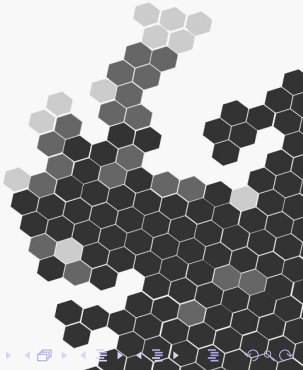
C VS CUDA (Segundos)



Consumo energético

¿Cómo hemos realizado las pruebas?

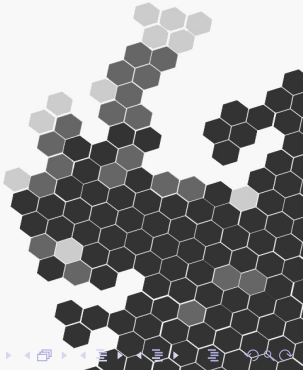
1. Hemos medido los amperios de la batería durante un minuto.



Consumo energético

¿Cómo hemos realizado las pruebas?

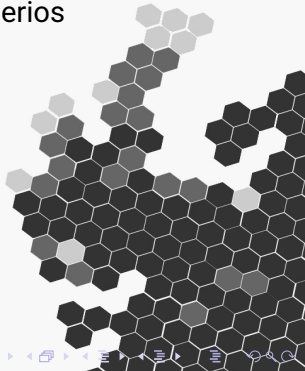
1. Hemos medido los amperios de la batería durante un minuto.
2. Hemos calculado el tiempo de descarga media en $\mu A/s$



Consumo energético

¿Cómo hemos realizado las pruebas?

1. Hemos medido los amperios de la batería durante un minuto.
2. Hemos calculado el tiempo de descarga media en $\mu A/s$
3. Hemos abierto nuestra aplicación, y hemos medido los amperios durante un minuto.



Consumo energético

¿Cómo hemos realizado las pruebas?

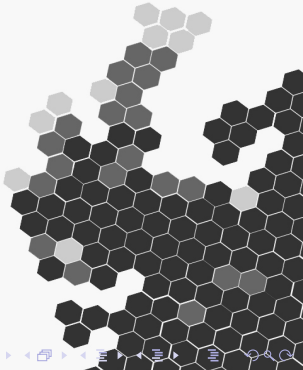
1. Hemos medido los amperios de la batería durante un minuto.
2. Hemos calculado el tiempo de descarga media en $\mu A/s$
3. Hemos abierto nuestra aplicación, y hemos medido los amperios durante un minuto.
4. Hemos calculado la descarga media con la aplicación abierta en $\mu A/s$

Consumo energético

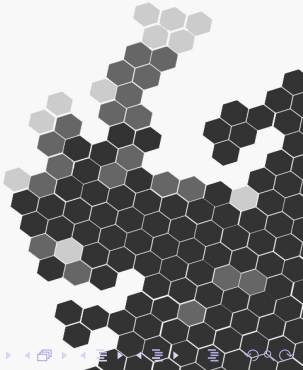
¿Cómo hemos realizado las pruebas?

1. Hemos medido los amperios de la batería durante un minuto.
2. Hemos calculado el tiempo de descarga media en $\mu A/s$
3. Hemos abierto nuestra aplicación, y hemos medido los amperios durante un minuto.
4. Hemos calculado la descarga media con la aplicación abierta en $\mu A/s$
5. El consumo energético de nuestra aplicación lo calculamos como la diferencia de las dos velocidades.

Consumo energético

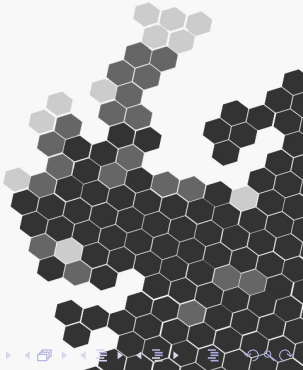


Referencias



Referencias

- **Programación paralela en R** - https://goo.gl/i7nTNw
- **Documentación OpenMP** - https://goo.gl/xZ5b4X
- **CUDA** - https://goo.gl/avaPcC
- **Pthread** - https://goo.gl/xG5eGy
- Aplicación *man* de Debian



Código Fuente (Licencia GPLv3)

- **Presentación** - https://git.io/vpjVl
- **Aplicación para medir la energía** - https://git.io/vpjVo
- **Experimentos numéricos** - https://git.io/vpjVS