



---

# Sandpile

## Laboratorio 1

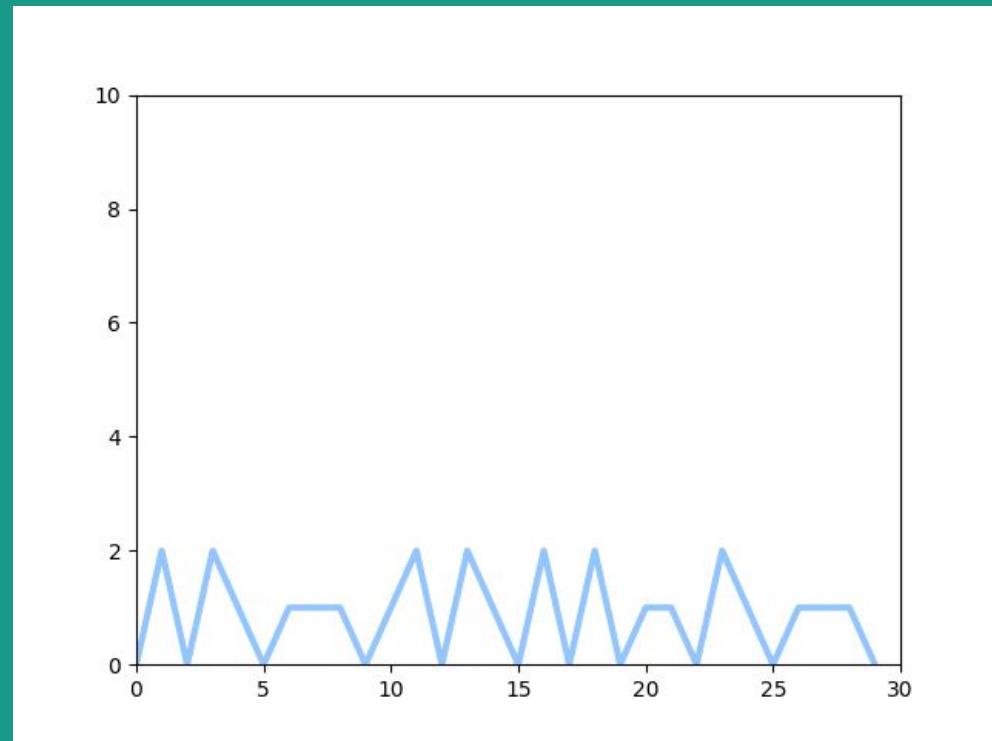
Federico Gonzalez Kriegel y Thomas Vadora



# Recordemos el Problema

---

- Sea  $h[i]$  el número de granitos en el sitio  $i$ ,  $0 < i < N - 1$ .
- Si  $h[i] > 1$  el sitio  $i$  esta "activo".
- Al tiempo  $t$ , un sitio "activo" se "descarga" completamente tirando cada uno de sus granitos aleatoriamente y con igual probabilidad a la izquierda o a la derecha





# CPUinfo

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	46 bits physical, 48 bits virtual
CPU(s):	24
On-line CPU(s) list:	0-11
Off-line CPU(s) list:	12-23
Thread(s) per core:	1
Core(s) per socket:	6
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	63
Model name:	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
Stepping:	2
CPU MHz:	1200.581
CPU max MHz:	3200.0000
CPU min MHz:	1200.0000
BogoMIPS:	4800.07
Virtualization:	VT-x
L1d cache:	384 KiB
L1i cache:	384 KiB
L2 cache:	3 MiB
L3 cache:	30 MiB

# Estructura

- Inicialización
- Desestabilización
- Descargar
  - 90% de la ejecución del código
  - Si hay que optimizar algo, seguro esta ahí

```
unsigned int descargar(Manna_Array &h, Manna_Array &dh)
{
    dh.fill(0);

    for (int i = 0; i < N; ++i) {
        // si es activo lo descargo aleatoriamente
        if (unlikely(h[i] > 1)) {
            for (int j = 0; j < h[i]; ++j) {
                // sitio receptor a la izquierda o derecha
                int k = (i+2*(randd()%2)-1+N)%N;
                ++dh[k];
            }
            inst += h[i];
            h[i] = 0;
        }
    }

    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        h[i] += dh[i];
        nroactivos +=(h[i]>1);
    }

    return nroactivos;
}
```



# Metrica

Llamémosle descargar a sumarle 1 a alguno de mis dos vecinos al azar y autorestarme 1. Entonces nuestra métrica que vamos a tratar de optimizar va a ser:

Score = Cantidad de descargas / tiempo de ejecución del programa

Queremos maximizar Score

# Metodología

- 1 Código**  
Versión del código con mejor resultado del momento
- 2 Compilar con distintas flags**  
Probar compilar con todas las versiones de cada compilador y con todas las combinaciones de flags ya fijadas
- 3 Medir la performance**  
Medimos con nuestra métrica la performance, si mejoró de la iteración anterior , se actualiza el código, si no, no se reemplaza el

---

**Compiladores** = [clang++-8, clang++-9, clang++-7,  
g++-10,g++-4.8, g++-4.9, g++-5, g++-6, g++-7, g++-8,  
g++-9]

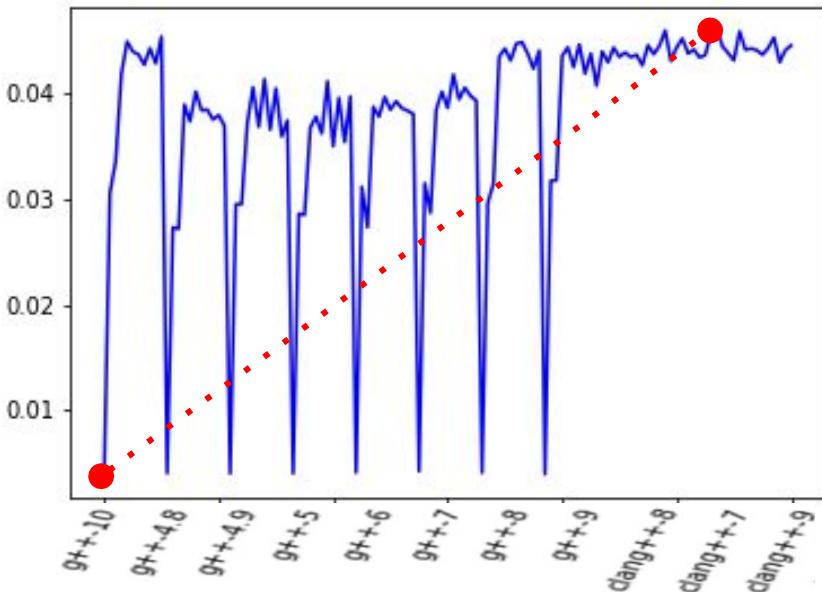
**Os** = [-O0, -O1, -O2, -O3, -Ofast]

**Flags** = [-march=native, -unroll-loops,  
-march=native -unroll-loops]

Observaciones:

- No hay cálculo de punto flotante casi
- No hay operaciones matemáticas intensas
- Mas “memorioso” que “calculoso”

# Versión 1



- Optimización: Ninguna

Compilar sin NINGUNA flag da un score de: **0.00366189**  
(llamemosle version 0)

Compilando de la siguiente manera:

```
clang++-7 -stdlib=libc++ -Wall -Wextra -std=c++0x -Ofast  
-march=native -o tiny_manna tiny_manna.cpp -lgomp
```

Obtenemos un Score de **0.046223833**, es decir que solamente compilando con cierto compilador particular y ciertas flags, el compilador de regalo nos da un **12.6X**

## Versión 2

Si vemos el código, podemos notar que hay una función que se llama que no sabemos qué hace exactamente, *rand()*. Corrimos varias veces el código y notamos que *rand()* se llama N/10 veces en una corrida del primer for. En particular para

$$\text{NSTEPS} = (1 \ll 14) \text{ y } N = (1 \ll 14)$$

simulando el procedimiento, se llama a *rand()* unas  $2.5e7$  veces.

```
int k = (i+2*(rand()%2)-1+N)%N;
```

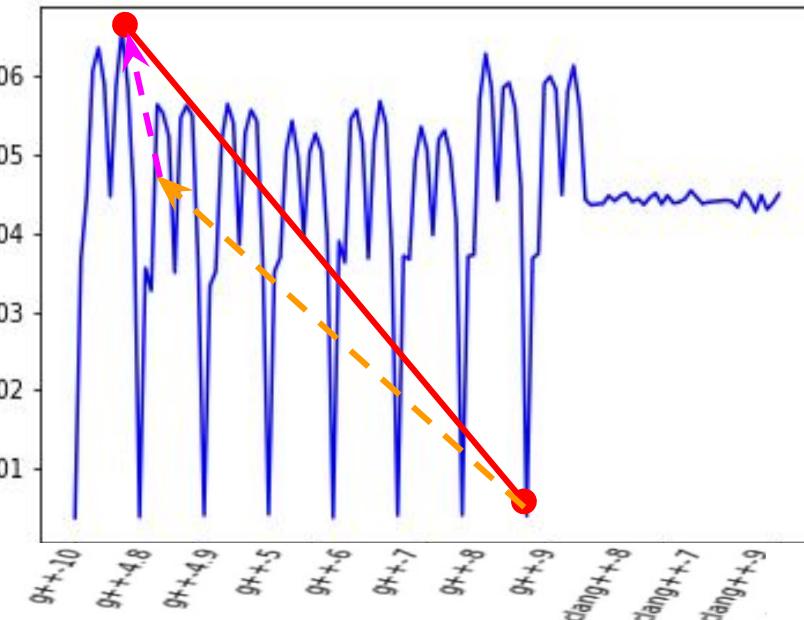
```
unsigned int descargar(Manna_Array &h, Manna_Array &dh)
{
    dh.fill(0);

    for (int i = 0; i < N; ++i) {
        // si es activo lo descargo aleatoriamente
        if (h[i] > 1) {
            for (int j = 0; j < h[i]; ++j) {
                // sitio receptor a la izquierda o derecha
                int k = (i+2*(rand()%2)-1+N)%N;
                ++dh[k];
            }
            inst += h[i];
            h[i] = 0;
        }
    }

    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        h[i] += dh[i];
        nroactivos +=(h[i]>1);
    }

    return nroactivos;
}
```

# Versión 2



- Optimización: Cambiamos la función rand() por una nueva variante del Mersenne Twister llamado SFMT (<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html>). Este algoritmo genera números pseudoaleatorios de 128-bits, SFMT está diseñado con técnicas de paralelismo en CPU como 'multi-stage pipelining' y SIMD.

Ahora usando SFMT + compilando con:

```
g++-10 -Wall -Wextra -std=c++0x -Ofast  
-march=native -DRAND -o tiny_manna  
Tiny_manna.cpp
```

Obtenemos un Score de: **0.0658847**  
Que es 18X comparado con la versión 0, y un aumento de 1.43X comparado de la versión 1

# Versión 3

Hicimos simulaciones de la ejecución del código y nos dimos cuenta que en promedio solo el 5% de las posiciones del arreglo *h* están activadas. Entonces es un desperdicio de tiempo recorrer también el otro 95% del arreglo en cada NSTEP. Modificamos el código de la función descargar, para que solo visite los nodos que están activados en cada iteración

```
unsigned int descargar2(Manna_Array &h, Manna_Array &dh)
{
    static Manna_Array pend;
    static int c=0;
    static bool b = 1;

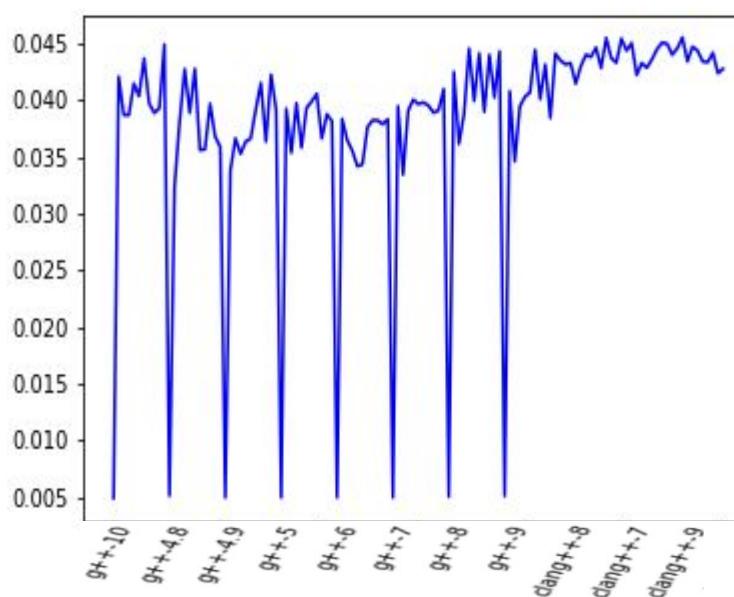
    dh.fill(0);

    if(b){
        for(int i = 0;i < N; i++){
            if(h[i]>1)pend[c++] = i;
        }
        b=0;
    }

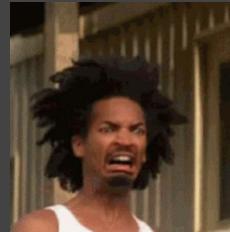
    int nc=0;
    for(int i = 0; i < c; ++i){
        int l = h[pend[i]];
        for (int j = 0; j < l; ++j) {
            int k = (pend[i]+2*(randd()%2)-1+N)%N;
            ++dh[k];
        }
        inst += h[pend[i]];
        h[pend[i]] = 0;
    }

    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        h[i] += dh[i];
        if(h[i]>1){
            pend[nc++]=i;
            ++nroactivos;
        }
    }
    c = nc;
    prom+=c;
    mn = min(mn,c);
    mx = max(mx,c);
    return nroactivos;
}
```

# Versión 3



El máximo Score lo da la línea de compilación: g++-10 -Wall -Wextra -std=c++0x -Ofast -march=native -DRAND -o tiny\_manna tiny\_manna.cpp, y es de 0.0455427. Es decir que el empeora la performance del programa.





Para poder hacer la optimización de la versión 3 necesitas más memoria, y además ya no haces saltos necesariamente consecutivos en memoria, agregas una recorrida, por lo tanto...

- +309 LLC misses
- Bajar hasta LLC +16558
- Aumento de branch-misses

Aparentemente versión 2 se queda más en L1 L2, mientras que versión 3 se mueve más entre los 3 niveles

#### Performance counter stats for './tiny\_manna opt':

515.47 msec	task-clock	# 0.999 CPUs utilized	
0	context-switches	# 0.000 K/sec	
0	cpu-migrations	# 0.000 K/sec	
211	page-faults	# 0.409 K/sec	
1434956076	cycles	# 2.784 GHz	(50.05%)
2537078518	instructions	# 1.77 insn per cycle	(62.75%)
645383813	branches	# 1252.033 M/sec	(62.76%)
18429621	branch-misses	# 2.86% of all branches	(62.75%)
583025472	L1-dcache-loads	# 1131.059 M/sec	(60.81%)
74213770	L1-dcache-load-misses	# 12.73% of all L1-dcache hits	(24.83%)
14027	LLC-loads	# 0.027 M/sec	(24.83%)
1055	LLC-load-misses	# 7.52% of all LL-cache hits	(37.25%)

#### Performance counter stats for './tiny\_manna a':

920.28 msec	task-clock	# 0.999 CPUs utilized	
3	context-switches	# 0.003 K/sec	
0	cpu-migrations	# 0.000 K/sec	
214	page-faults	# 0.233 K/sec	
2769163962	cycles	# 3.009 GHz	(49.60%)
5608872090	instructions	# 2.03 insn per cycle	(62.20%)
1375548621	branches	# 1494.704 M/sec	(62.19%)
25454688	branch-misses	# 1.85% of all branches	(62.47%)
1258332032	L1-dcache-loads	# 1367.334 M/sec	(61.70%)
65548717	L1-dcache-load-misses	# 5.21% of all L1-dcache hits	(25.21%)
30584	LLC-loads	# 0.033 M/sec	(24.92%)
1364	LLC-load-misses	# 4.46% of all LL-cache hits	(37.09%)

# Versión 4

**Teorema.** Supongamos que  $X_1, \dots, X_n$  son variables aleatorias independientes con misma distribución Bernoulli parámetro  $p \in [0, 1]$ , definidas sobre un mismo espacio de probabilidad  $(\Omega, \mathcal{F}, \mathbb{P})$ . Entonces la función de probabilidades de  $X := \sum_{i=1}^n X_i$  está dada por

$$p_X(k) := \mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k},$$

para  $k = 0, \dots, n$ .

Entonces en vez de hacer un for hasta  $h[i]$  que cada iteración al azar decide si descarga a la derecha o a la izquierda, intentemos sacar la variable  $X$  directamente de una binomial( $h[i], 0.5$ ), y luego descargar  $X$  a la derecha y luego  $h[i]-X$  a la izquierda.

```

REAL fp[N];

void bin_init(){
    REAL c=1;
    for(int i = 0; i < N; ++i){
        fp[i]=c;
        c*=0.5;
    }
}

int dist_bin(int n){
    REAL u = ((REAL)rand() / INT_MAX);
    REAL f = fp[n];
    REAL p = f;
    int i = 0;
    while(u >= f){
        p = p*((REAL)(n-i)/(REAL)(i+1));
        f+=p;
        i++;
    }
    return i;
}

unsigned int descargar3(Manna_Array &h, Manna_Array &dh)
{
    dh.fill(0);

    for (int i = 0; i < N; ++i) {
        // si es activo lo descargo aleatoriamente
        if (h[i] > 1) {
            int k = dist_bin(h[i]);
            dh[(i+1)%N] += k;
            dh[(i-1+N)%N] += h[i]-k;
            inst += h[i];
            h[i] = 0;
        }
    }

    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        h[i] += dh[i];
        nroactivos +=(h[i]>1);
    }

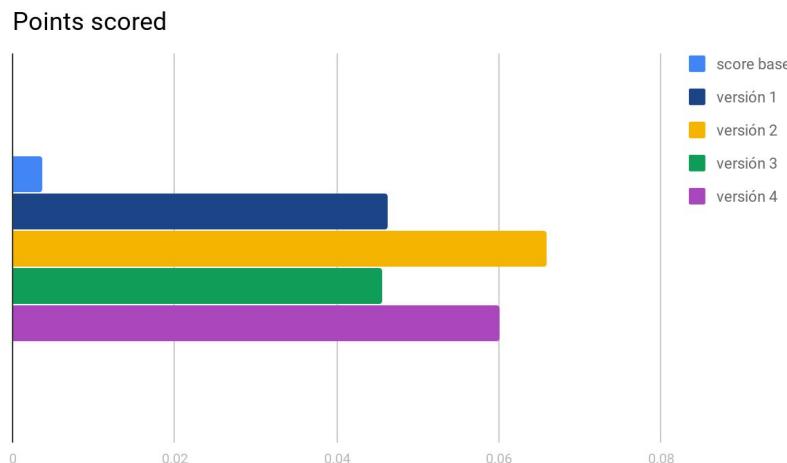
    return nroactivos;
}

```



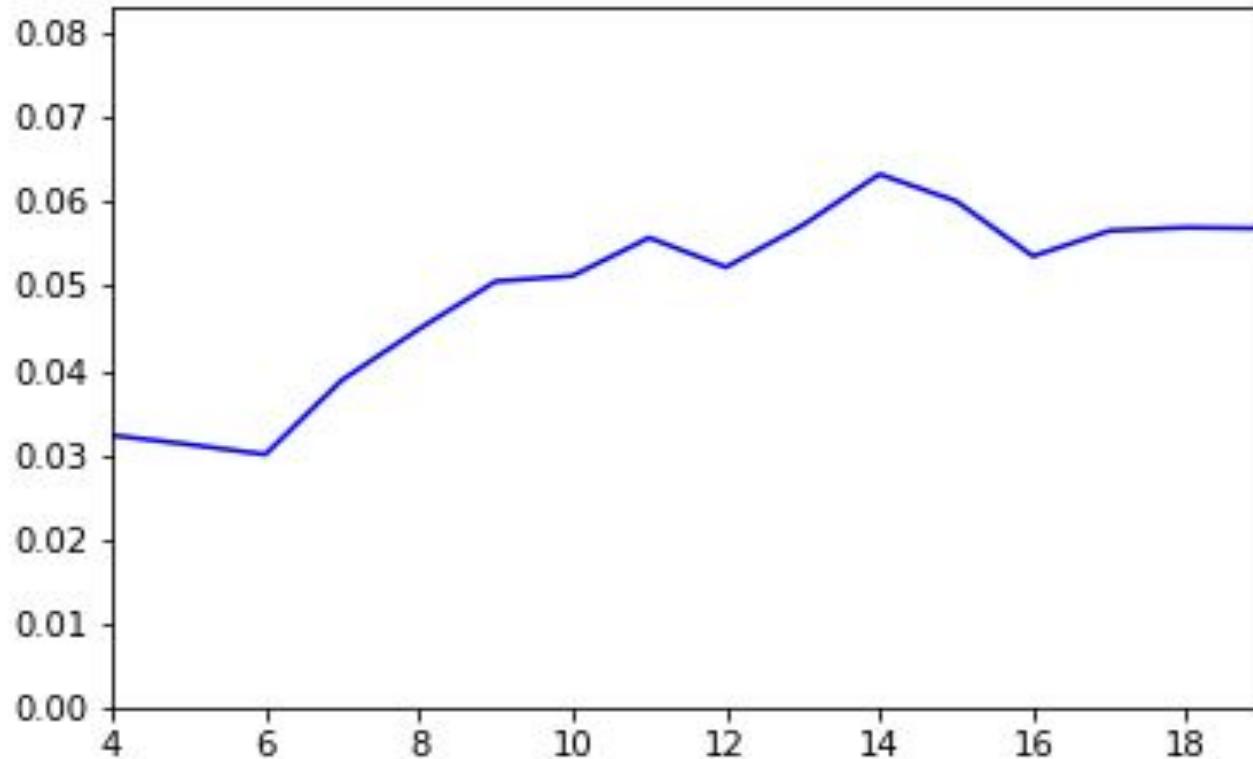
Lamentablemente la versión 4 arroja resultados muy similares e incluso un poco por debajo de la versión 2 (SFMT only)

Por lo tanto nuestro mejor resultado es un score de 0.0658847 que lo da la versión 2.



18x

## Flags + SFMT





**It ain't much, but it's honest work**



# Sandpile

## Laboratorio 2

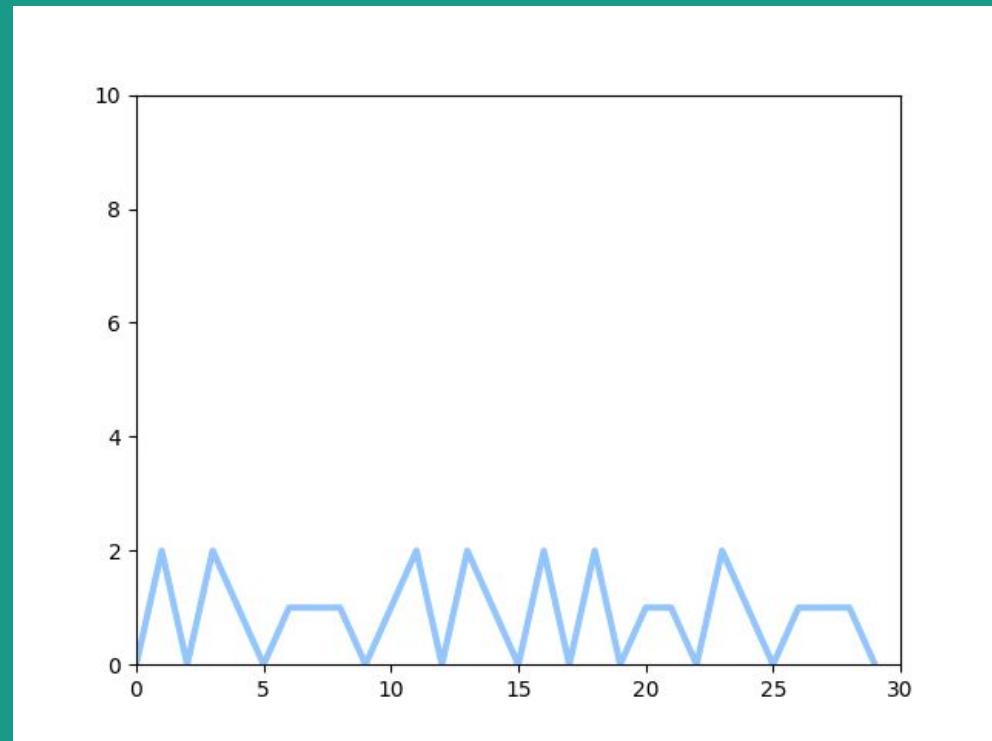
Federico Gonzalez Kriegel y Thomas Vadora



# Recordemos el Problema

---

- Sea  $h[i]$  el número de granitos en el sitio  $i$ ,  $0 < i < N - 1$ .
- Si  $h[i] > 1$  el sitio  $i$  esta "activo".
- Al tiempo  $t$ , un sitio "activo" se "descarga" completamente tirando cada uno de sus granitos aleatoriamente y con igual probabilidad a la izquierda o a la derecha





---

# Recordemos la Metrica

Llamémosle descargar a sumarle 1 a alguno de mis dos vecinos al azar y autorestarme 1. Entonces nuestra métrica que vamos a tratar de optimizar va a ser:

Score = Cantidad de descargas / tiempo de ejecución del programa

Queremos maximizar Score



En el laboratorio 1 habíamos logrado un score de 0.065

Que era una mejora de 18x comparado con la version base

## Recordemos Observaciones Importantes:

- El 85% de la ejecución del código es en la función “descargar”
- En promedio solo el 5% de los granos están activos, es decir solo el 5% de los  $h[i]$  son  $> 1$
- Simulando muchas veces te das cuenta que  $h[i] < 64$  siempre y además la mayoría del tiempo se mantiene en valores muy chicos ( $h$  puede ser de 8 bits)

# Auto Vectorización

```
unsigned int descargar(Manna_Array &h, Manna_Array &dh)
{
    dh.fill(0);
    for (int i = 0; i < N; ++i) {
        if (h[i] > 1) {
            for (int j = 0; j < h[i]; ++j) {
                int k = (i+2*(rand()&1)-1+N)%N;
                ++dh[k];
            }
            inst += h[i]+1;
            h[i] = 0;
        }
    }
    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        h[i] += dh[i];
        nroactivos += (h[i]>1);
    }
    return nroactivos;
}
```

No lo logra auto vectorizar, y es entendible

Tampoco logra auto vectorizarlo, pero si lo ayudas un poquito separando el for en dos fors si lo hace. Score = 0.08 (1.2x contra el lab1) (usando SFMT)

```
unsigned int nroactivos=0;
for (int i = 0; i < N; ++i) {
    h[i] += dh[i];
}
for (int i = 0; i < N; ++i) {
    nroactivos += (h[i]>1);
}
```

# Idea general

---

- Primero vamos a calcular para cada  $h[i]$  cuantos granos van a la izquierda (entonces a la derecha es  $h[i]-izq[i]$ )
- Luego vamos a descargar los que tiran para la izquierda, y luego los que tiran hacia la derecha
- Cuando generamos un random uniforme de 64 bits, llamémosle  $X$ , cada bit de  $X$  tiene la misma probabilidad de ser 0 o 1, por lo tanto podemos ver a los bits de  $X$  como 64 bernoullis, donde para nosotros 1 significa izquierda y 0 significa derecha. Como nosotros queremos  $h[i]$  bernoullis podemos hacer:

```
rand = random(); rand = rand & ((1<<h[i])-1); left = popcount(rand);
```

```

export uniform int descargar(uniform int8 * uniform h,
                            uniform int8 * uniform dh,
                            uniform int8 * uniform a,
                            uniform int n){
    varying int res = 0;
    foreach(i = 0 ... n){
        if(h[i] > 1){
            res += h[i];
        }
    }
    foreach(j = 0 ... n-1){
        if(h[j] > 1){
            dh[j+1] += a[j];
        }
    }
    if(h[n-1] > 1){
        dh[0] += a[n-1];
    }
    foreach(j = 1 ... n){
        if(h[j] > 1){
            dh[j-1] += h[j]-a[j];
        }
    }
    if(h[0] > 1){
        dh[n-1] += h[0]-a[0];
    }
    foreach(j = 0 ... n){
        if(h[j]>1){
            h[j]=0;
        }
    }
    return reduce_add(res);
}

```

```

unsigned int descargar(Manna_Array &h, Manna_Array &dh, Manna_Array &randoms)
{
    dh.fill(0);

    ispc::calcDescargar(h.data(), randoms.data(), N);

    inst += ispc::descargar(h.data(), dh.data(), randoms.data(), N);
    ispc::actH(h.data(), dh.data(), N);
    unsigned int nroactivos = ispc::nreactivos(h.data(), N);

    return nroactivos;
}

```

```

export void calcDescargar(uniform int64 * uniform h,
                           uniform int64 * uniform a,
                           uniform int n){

    uniform int64 b[N],c[N];

    foreach(i = 0...n){
        if(h[i] > 1){
            b[i] = random(&rng);
        }
    }
    foreach(i = 0...n){
        if(h[i] > 1){
            b[i]<=c[i];
        }
    }
    foreach(i = 0...n){
        if(h[i] > 1){
            c[i] = (32-(h[i]&31));
        }
    }
    foreach(i = 0 ... n){
        varying int sum = 0;
        for(varying int j = 0; j < (h[i]/32); ++j){
            sum += popcnt((varying int)random(&rng2));
        }
        varying int r = popcnt(b[i] );
        a[i] = (sum+r);
    }
}

```

```

export uniform int nreactivos(uniform int64 * uniform h,
                               uniform int n){

    varying int sum = 0;
    foreach(i = 0 ... n){
        sum += (h[i]>1);
    }
    return reduce_add(sum);
}

export void actH(uniform int64 * uniform h,
                 uniform int64 * uniform dh,
                 uniform int n){

    foreach(i = 0 ... n){
        h[i] += dh[i];
    }
}

```

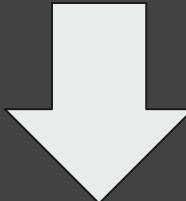
No sabemos porque, pero solo mejora un 2x (0.12) contra el lab1. (+ que tuvimos que trabajar con 64 bits)

Hipotesis: popcorn? 64bits?



Unas de las observaciones que hicimos era que  $h[i] < 64$  (por eso mismo con los bits de un random de 64 bits alcanza para descargar), sino que también al simular se puede ver que la mayoría de los  $h[i]$  se mantienen en números enteros positivos muy pequeños, por lo que generar siempre randoms de 64 puede ser un desperdicio, entonces vamos a generar randoms de 8 bits mientras necesite.

La función descargar, ahora se va a modularizar en 7 funciones:





\_mm256\_popcount\_epi8

Función que toma un vector `__m256i` y devuelve otro con los popcounts de cada elemento de 8 bits. (AVX2 no tiene instrucción de popcount, recien AVX512 lo tiene) La sacamos de un paper de la Universidad de Quebec: <https://arxiv.org/pdf/1611.07612.pdf>

get\_mask\_8b

Toma un vector `__m256i v` y devuelve una máscara donde cada elemento tiene el  $(1 << v[i]) - 1$  para  $i = 0, \dots, 32$

get\_rand

Genera con ispc 4 randoms de 64 bits, que es lo mismo que 32 randoms de 8 bits

calcDescargar

Calcula un arreglo en donde cada posición  $i$  en la que  $h[i] > 1$  guarda cuantos granos se tienen que descargar a la izquierda

descargarIzq

Descarga todos los granos que se tiran a la izquierda

descargarDer

Descarga todos los granos que se tiran a la derecha

actH

Actualiza los valores en `h`



```
__m256i __mm256_popcount_epi8(const __m256i v) {
    __m256i lookup = __mm256_setr_epi8(
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4);
    __m256i low_mask = __mm256_set1_epi8(0x0f);
    __m256i lo = __mm256_and_si256(v, low_mask);
    __m256i hi = __mm256_and_si256(__mm256_srli_epi32(v, 4), low_mask);
    __m256i popcnt1 = __mm256_shuffle_epi8(lookup, lo);
    __m256i popcnt2 = __mm256_shuffle_epi8(lookup, hi);
    __m256i total = __mm256_add_epi8(popcnt1, popcnt2);
    return total;
}
```

Es más rápido generar de primera varios randoms y luego si es necesario generar más. Generas pocos también porque el 5% están activos

```
unsigned long long* get_rand(){
    const int n = 64;
    static unsigned long long int a[n];
    static int i = n;
    if(i == n){
        i = 0;
        ispc::rand_array((int64_t*)a,n);
    }
    return &a[i+=4];
}
```

```
__m256i get_mask_8b(const __m256i v) {
    __m256i lookup = __mm256_setr_epi8(
        0, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff,
        0, 0, 0, 0, 0, 0, 0, 0,
        0, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f, 0xff,
        0, 0, 0, 0, 0, 0, 0, 0);
    return __mm256_shuffle_epi8(lookup, v);
}
```

```
export void rand_array(uniform int64 * uniform h,
                      uniform int n){
    foreach(i = 0...n){
        h[i] = (((int64)random(&rng))<<32);
    }
    foreach(i = 0...n){
        h[i] |= (random(&rng));
    }
}
```

# Descargar lab2

```
int descargar_intrinsics(Manna_Array &h, Manna_Array &dh){  
    uint8_t a[N];  
  
    dh.fill(0);  
    for(int i =0; i < N; i++){  
        inst+=h[i]*(h[i]>1);  
    }  
  
    calcDescargar(h.data(),a);  
  
    descargarIzq(dh.data(),a);  
    descargarDer(h.data(), dh.data(),a);  
  
    actH(h.data(),dh.data());  
  
    unsigned int nroactivos=0;  
    for (int i = 0; i < N; ++i) {  
        nroactivos +=(h[i]>1);  
    }  
  
    return nroactivos;  
}
```

dh.fill(0);

for(int i =0; i < N; i++){  
 inst+=h[i]\*(h[i]>1);  
}

calcDescargar(h.data(),a);

descargarIzq(dh.data(),a);  
descargarDer(h.data(), dh.data(),a);

actH(h.data(),dh.data());

unsigned int nroactivos=0;  
for (int i = 0; i < N; ++i) {  
 nroactivos +=(h[i]>1);  
}

return nroactivos;

A[i] es cuantos se descargan a la izquierda

Primero descargo hacia izquierda, y luego a derecha

Actualizo el arreglo h

Autovectoriza

```
void calcDescargar(uint8_t *h, uint8_t *a){  
    for(int i = 0; i < N; i += 32){  
        __m256i res = _mm256_set1_epi8(0);  
        __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);  
        __m256i activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));  
        hv = _mm256_and_si256(hv, activos);  
        while(_mm256_testz_si256(activos, activos) == 0){  
            __m256i tirar = _mm256_min_epi8(hv, _mm256_set1_epi8(8));  
            __m256i rand = _mm256_loadu_si256((__m256i*) get_rand());  
            __m256i mask_8b = get_mask_8b(tirar);  
            rand = _mm256_and_si256(rand,mask_8b);  
            __m256i aux = _mm256_popcount_epi8(rand);  
            hv = _mm256_sub_epi8(hv,tirar);  
            res = _mm256_add_epi8(res,aux);  
            activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(0));  
        }  
        _mm256_storeu_si256((__m256i *) &a[i], res);  
    }  
}
```

```
void calcDescargar(uint8_t *h, uint8_t *a){  
    for(int i = 0; i < N; i++){  
        uint8_t res = 0;  
        uint8_t hv = h[i];  
        uint8_t activos = (hv > 1)  
        hv *= activos;  
        // hv es h[i] si estamos en una posicion activa  
        // y es 0 en caso contrario  
        while(activos){  
            uint8_t tirar = min(hv, 8); // cuantos voy a tirar en esta pasada  
            // asegurarnos de no descargar de mas, como vamos a ir descargando  
            // de a 8, descargo 8 h[i]/8 veces, pero al ultimo descargo  
            // h[i] % 8 granos nomas  
            int8_t rand = get_rand(); // un random de 8 bits  
            int8_t mask_8b = (1<<tirar)-1;  
            rand &= mask_8b; // solo quiero tener en cuenta los primeros h[i] bits  
            int8_t aux = __builtin_popcount(rand); // cuantos a la izquierda  
            hv -= tirar // tiro  
            res += aux; // acumulo todos los que voy tirando  
            activos = hv > 0;  
        }  
        a[i] = res;  
    }  
}
```

```

void descargarIzq(uint8_t *dh, uint8_t *a){
    dh[N-1] += a[0];
    int p;
    for(int i = 0; i+32 < (N-1); i+=32){
        p = i;
        __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);
        __m256i av = _mm256_loadu_si256((__m256i*) &a[i+1]);
        dhv = _mm256_add_epi8(dhv,av);
        _mm256_storeu_si256((__m256i *) &dh[i], dhv);
    }
    for(int i = p+32; i < N-1; i++){
        dh[i] += a[i+1];
    }
}

void descargarDer(uint8_t *h, uint8_t *dh, uint8_t *a){
    dh[0] += (h[N-1]-a[N-1])*(h[N-1]>1);
    int p;
    for(int i = 1; i+32 < N; i+=32){
        p = i;
        __m256i hv = _mm256_loadu_si256((__m256i*) &h[i-1]);
        __m256i act = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
        __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);
        __m256i av = _mm256_loadu_si256((__m256i*) &a[i-1]);
        hv = _mm256_and_si256(hv,act);
        av = _mm256_and_si256(av,act);
        dhv = _mm256_add_epi8(dhv,hv);
        dhv = _mm256_sub_epi8(dhv,av);
        _mm256_storeu_si256((__m256i *) &dh[i], dhv);
    }
    for(int i = p+32; i < N; i++){
        dh[i] += (h[i-1]- a[i-1])*(h[i-1]>1);
    }
}

```

```

void descargarIzq(uint8_t *dh, uint8_t *a){
    dh[N-1] += a[0] //descargo el ultimo
    for(int i=0; i < (N-1); i++){
        uint8_t dhv = dh[i];
        uint8_t av = a[i+1]; // a mi derecha tengo cuantos me van a descargar
        dhv += av;
        dh[i] = dhv;
    }
}

void descargarDer(uint8_t *h, uint8_t *dh, uint8_t *a){
    dh[0] += (h[N-1]-a[N-1])*(h[N-1]>1); // Descargar el primera
    for(int i = 1; i < N; i++){
        uint8_t hv = h[i-1];
        uint8_t act = (hv > 1) ? 0xff : 0x00; // esta activa a mi izquierda?
        uint8_t dhv = dh[i];
        uint8_t av = a[i-1]; // cuanto se me va a descargar
        hv &= act; // hv != 0 si y solo si se va a descargar
        av &= act; // av != 0 si y solo si se va a descargar
        // el descargo a derecha es (h[i-1]-a[i-1]) pues es lo que no se
        // descargo a la izquierda
        dhv += hv;
        dhv -= av;
        dh[i] = dhv; // dh[i] = (h[i-1]-a[i-1])
    }
}

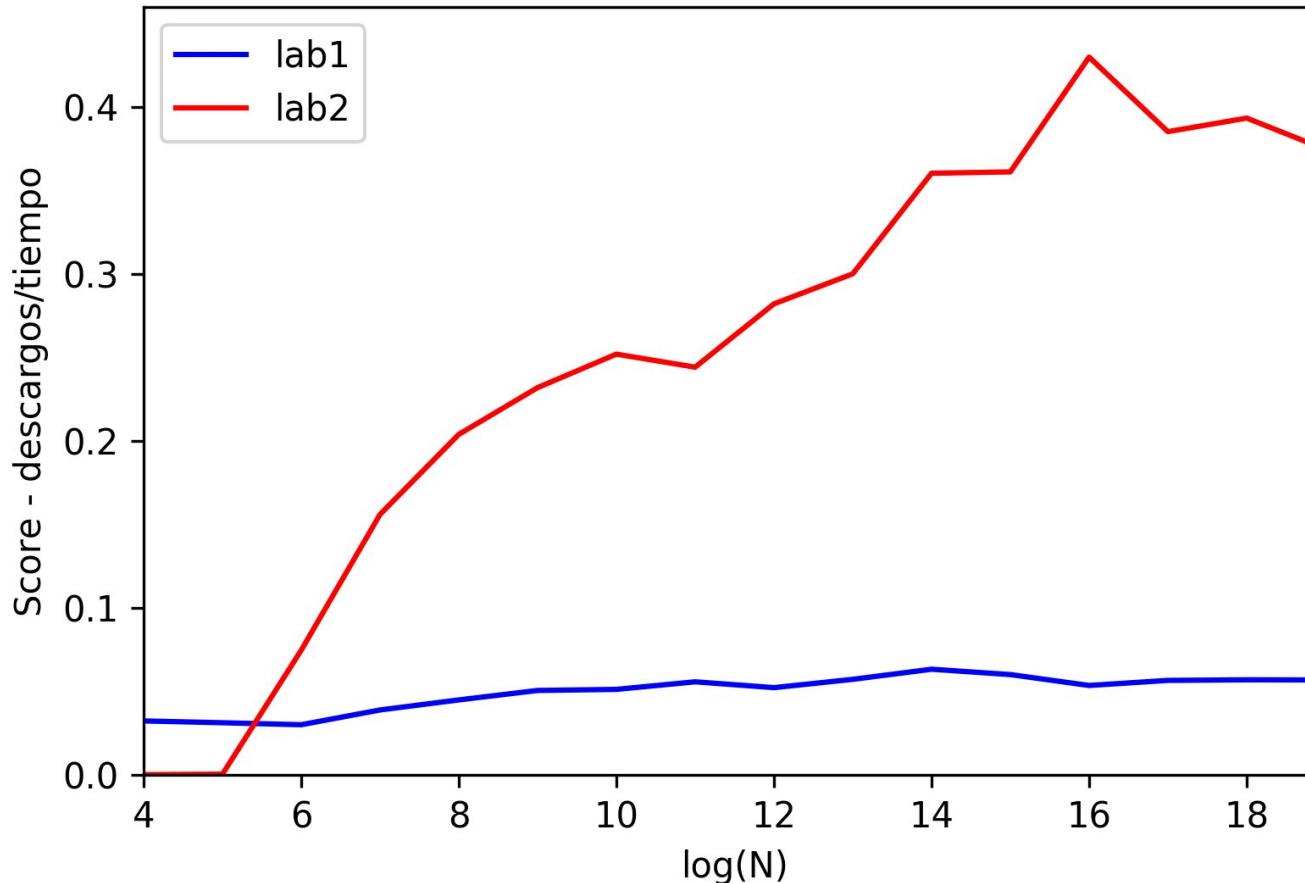
```

```
void actH(uint8_t *h, uint8_t *dh){  
    for(int i = 0; i < N; i+=32){  
        __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);  
        __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);  
        __m256i notact = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));  
        notact = _mm256_andnot_si256(notact, _mm256_set1_epi8(0xff));  
        hv = _mm256_and_si256(notact, hv);  
        hv = _mm256_add_epi8(hv, dhv);  
        _mm256_storeu_si256((__m256i *) &h[i], hv);  
    }  
}
```

```
void actH(uint8_t *h, uint8_t *dh){  
    for(int i = 0; i < N; i++){  
        uint8_t hv = h[i];  
        uint8_t dhv = dh[i];  
        uint8_t noact = (hv > 1) ? 0xff : 0x00;  
        noact = !noact & 0xff;  
        hv &= noact;  
        // si estaba activo lo pongo en 0  
        hv += dhv;  
        h[i] = hv;  
    }  
}
```

Con esta implementación con Intrinsics logramos un score de 0.43. Es decir un 6.6x contra el mejor del lab1, y un 143.3x contra la versión base.

## Scaling Tamaño-Score. Lab1 vs Lab2





# Sandpile

## Laboratorio 3

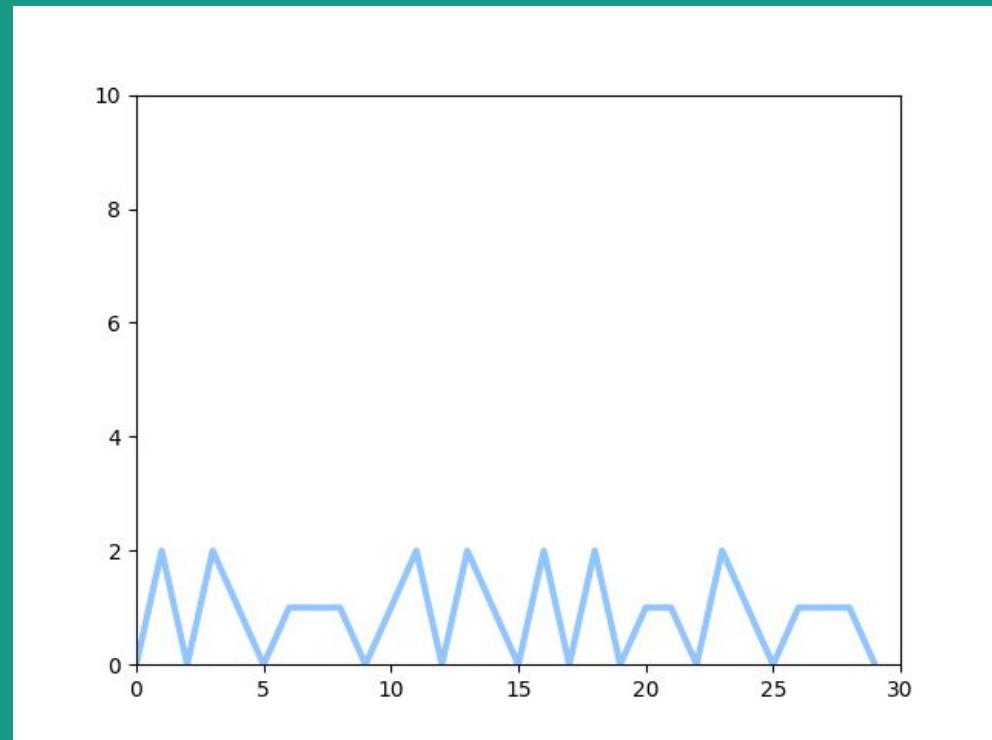
Federico Gonzalez Kriegel y Thomas Vadora



# Recordemos el Problema

---

- Sea  $h[i]$  el número de granitos en el sitio  $i$ ,  $0 < i < N - 1$ .
- Si  $h[i] > 1$  el sitio  $i$  esta "activo".
- Al tiempo  $t$ , un sitio "activo" se "descarga" completamente tirando cada uno de sus granitos aleatoriamente y con igual probabilidad a la izquierda o a la derecha





---

# Recordemos la Metrica

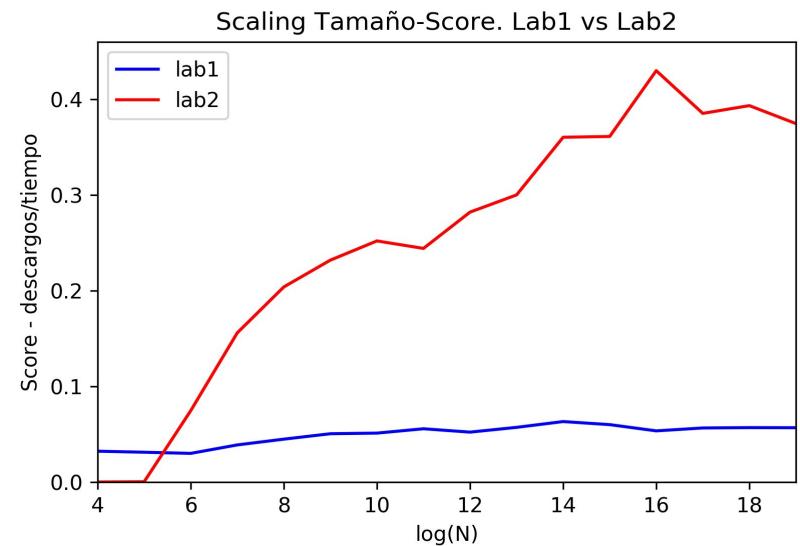
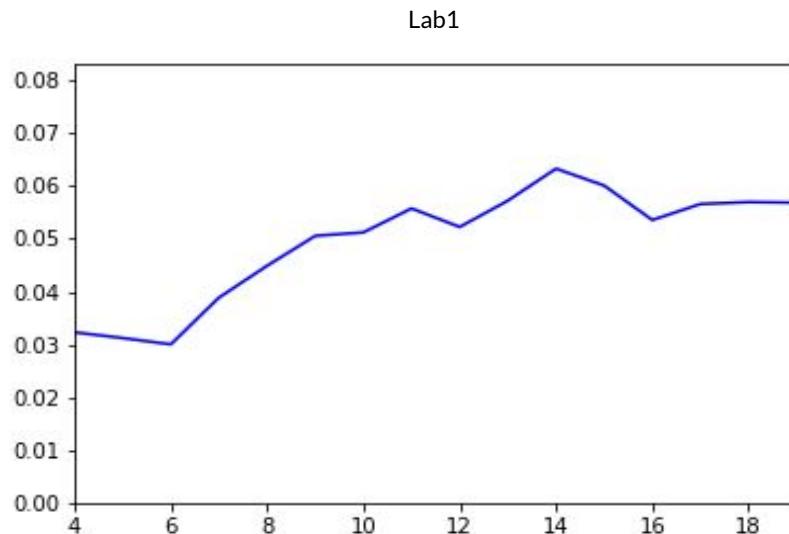
Llamémosle descargar a sumarle 1 a alguno de mis dos vecinos al azar y autorestarme 1. Entonces nuestra métrica que vamos a tratar de optimizar va a ser:

Score = Cantidad de descargas / tiempo de ejecución del programa

Queremos maximizar Score

# Scalings Anteriores

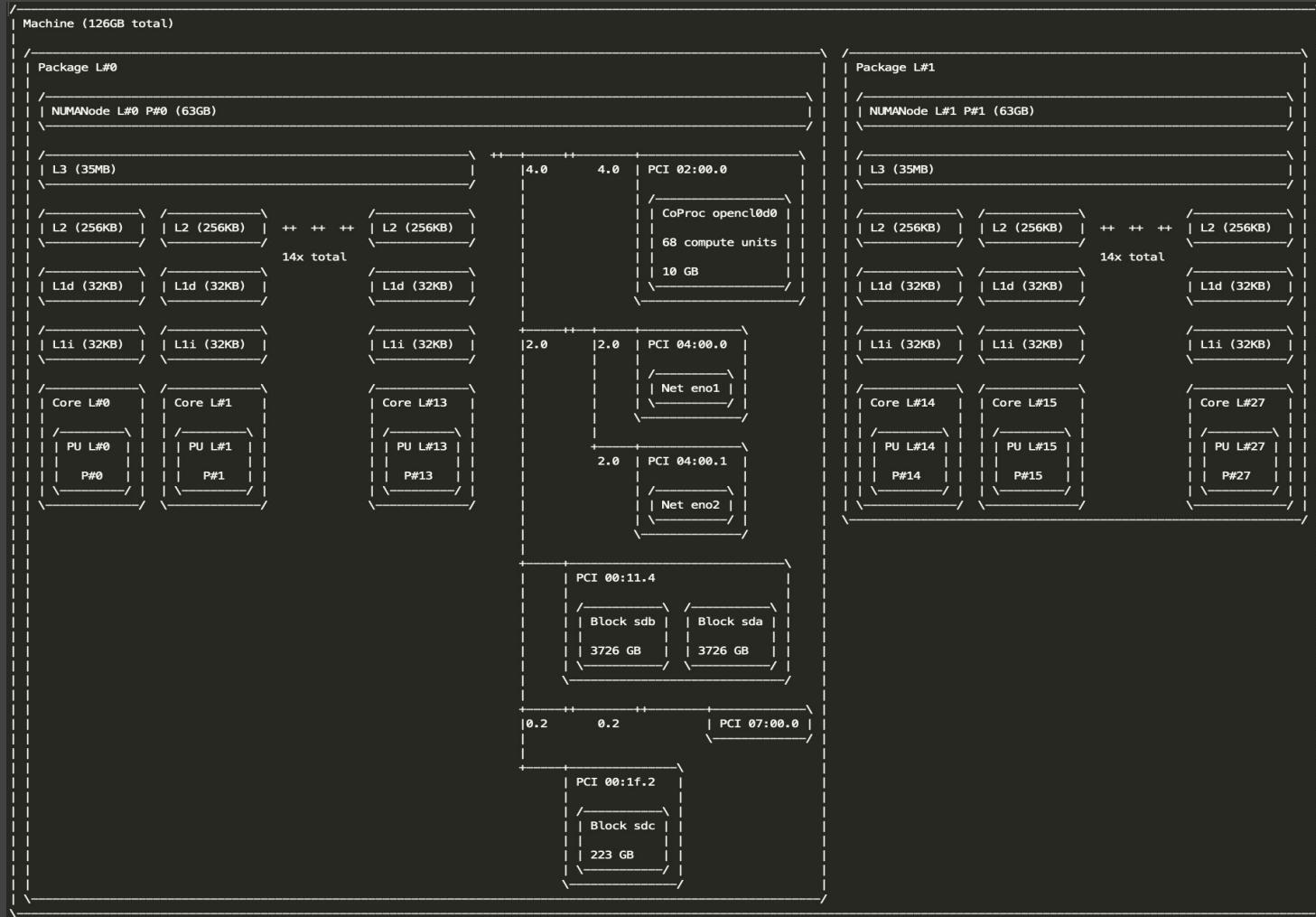
---





---

Estábamos muy entusiasmados con este Lab porque veníamos bien sacando bastante performance en los anteriores. En nuestras cabezas sonaban los números “Ahora le sacamos un x10 más y estaríamos llegando a un 1433x a la version base”. No fue así, y lamentamos el día en el que elegimos este problema.





# Estructura heredada lab2

---

- Calcular descargar: calcula un arreglo que tiene cuantos granitos se deben descargar a la izquierda
- Descargar Izquierda: Realiza el descargo a la izquierda
- Descargar Derecha: Realiza el descargo a la derecha (total - los que se descargaron a la izquierda)
- Actualizar H: Actualizar realmente el arreglo H



# Lo que intentamos

---

- Opción A: Paralelizar horizontalmente el lab2
- Opción B: Paralelizar verticalmente el lab2
- Opción C: Paralelizar verticalmente el lab1

# Horizontal Lab2

H =



Cada core computa la función "calcular descargar" para el rango que le tocó dependiendo de la cantidad de cores

WAIT

Más cuidado con los bordes

<lo mismo para Descargar izquierda>

Más cuidado con los bordes

<lo mismo para Descargar derecha>

Más cuidado con los bordes

<lo mismo para Actualizar H>

```
int descargar_desmodulado_omp(Manna_Array &h, Manna_Array &dh){  
    uint8_t a[N];  
  
    int nroactivos=0;  
    int n_33 = N-33;  
    int n_32 = N-32;  
  
#pragma omp parallel  
{  
    //inicializo dh con 0s  
    #pragma omp for  
    for(int i=0; i<N; i++){  
        dh[i]=0;  
        a[i]=0;  
    }  
  
    //calculo cantidad de instrucciones que voy a hacer (descargas)  
    #pragma omp for reduction(+:inst)  
    for(int i=0; i<N; i++) inst+=h[i]*(h[i]>1);
```

```
//calcular descargar
#pragma omp for
for(int i = 0; i < N; i += 32){
    __m256i res = _mm256_set1_epi8(0);
    __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);
    __m256i activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
    hv = _mm256_and_si256(hv, activos);
    while(_mm256_testz_si256(activos, activos) == 0){
        __m256i tirar = _mm256_min_epi8(hv, _mm256_set1_epi8(8));
        __m256i rand;
        #pragma omp critical
        {
            rand = _mm256_loadu_si256((__m256i*) get_rand());
        }
        __m256i mask_8b = get_mask_8b(tirar);
        rand = _mm256_and_si256(rand,mask_8b);
        __m256i aux = _mm256_popcount_epi8(rand);
        hv = _mm256_sub_epi8(hv,tirar);
        res = _mm256_add_epi8(res,aux);
        activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(0));
    }
    _mm256_storeu_si256((__m256i *) &a[i], res);
}
```

```
//descargo izquierda
#pragma omp single
{
    dh[N-1] += a[0];
}

#pragma omp for
for(int i = 0; i < n_33; i+=32){
    __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);
    __m256i av = _mm256_loadu_si256((__m256i*) &a[i+1]);
    dhv = _mm256_add_epi8(dhv,av);
    _mm256_storeu_si256((__m256i *) &dh[i], dhv);
}

#pragma omp single
{
    for(int i = (N-64)+32; i < N-1; i++){
        dh[i] += a[i+1];
    }
}
```

```
//descargo derecha
#pragma omp single
{
    dh[0] += (h[N-1]-a[N-1])*(h[N-1]>1);
}

#pragma omp for
for(int i = 1; i < n_32; i+=32){
    __m256i hv = _mm256_loadu_si256((__m256i*) &h[i-1]);
    __m256i act = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
    __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);
    __m256i av = _mm256_loadu_si256((__m256i*) &a[i-1]);
    hv = _mm256_and_si256(hv,act);
    av = _mm256_and_si256(av,act);
    dhv = _mm256_add_epi8(dhv,hv);
    dhv = _mm256_sub_epi8(dhv,av);
    _mm256_storeu_si256((__m256i *) &dh[i], dhv);
}

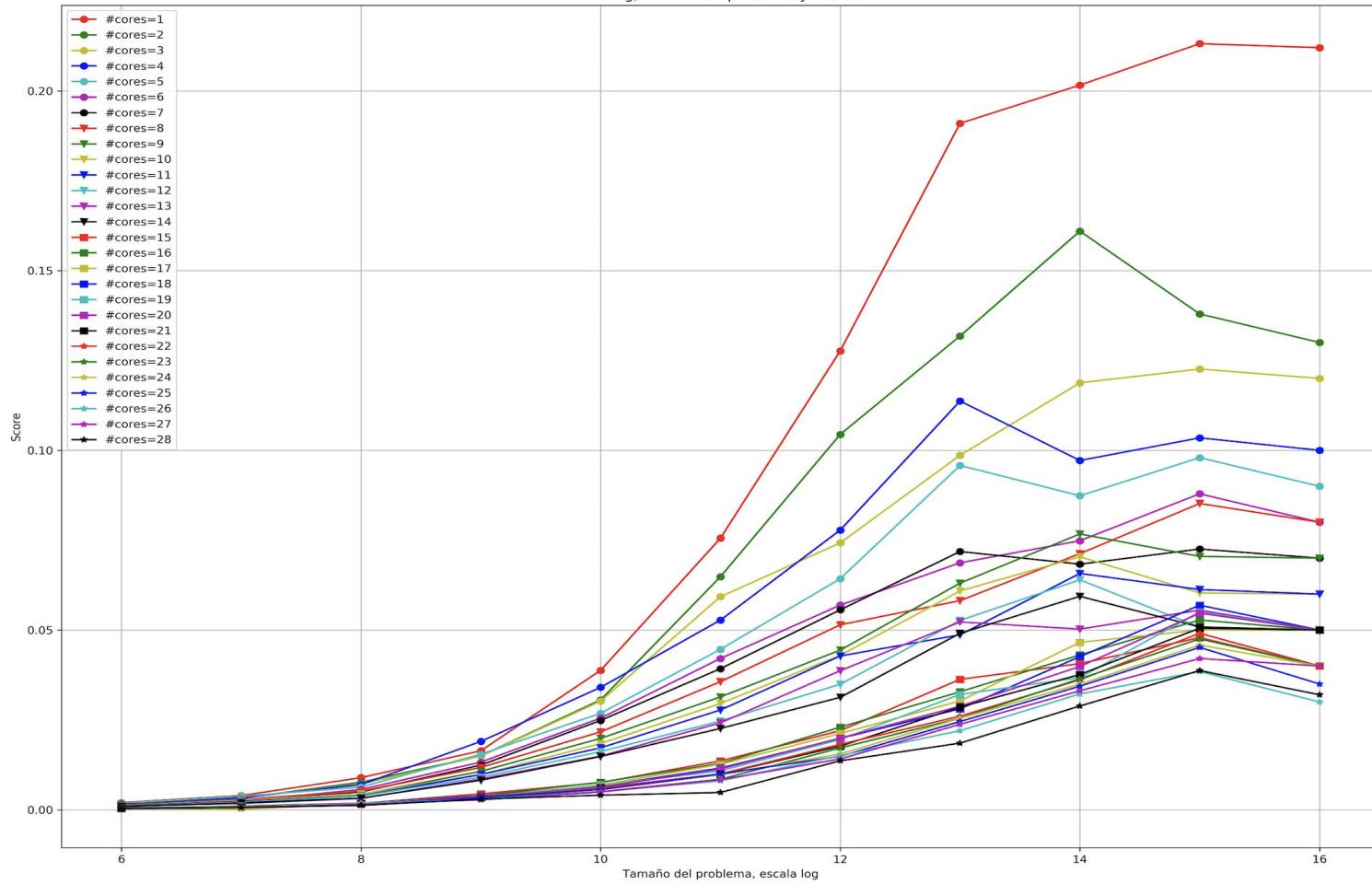
#pragma omp single
{
    for(int i = (N-63)+32; i < N; i++){
        dh[i] += (h[i-1]- a[i-1])*(h[i-1]>1);
    }
}
```

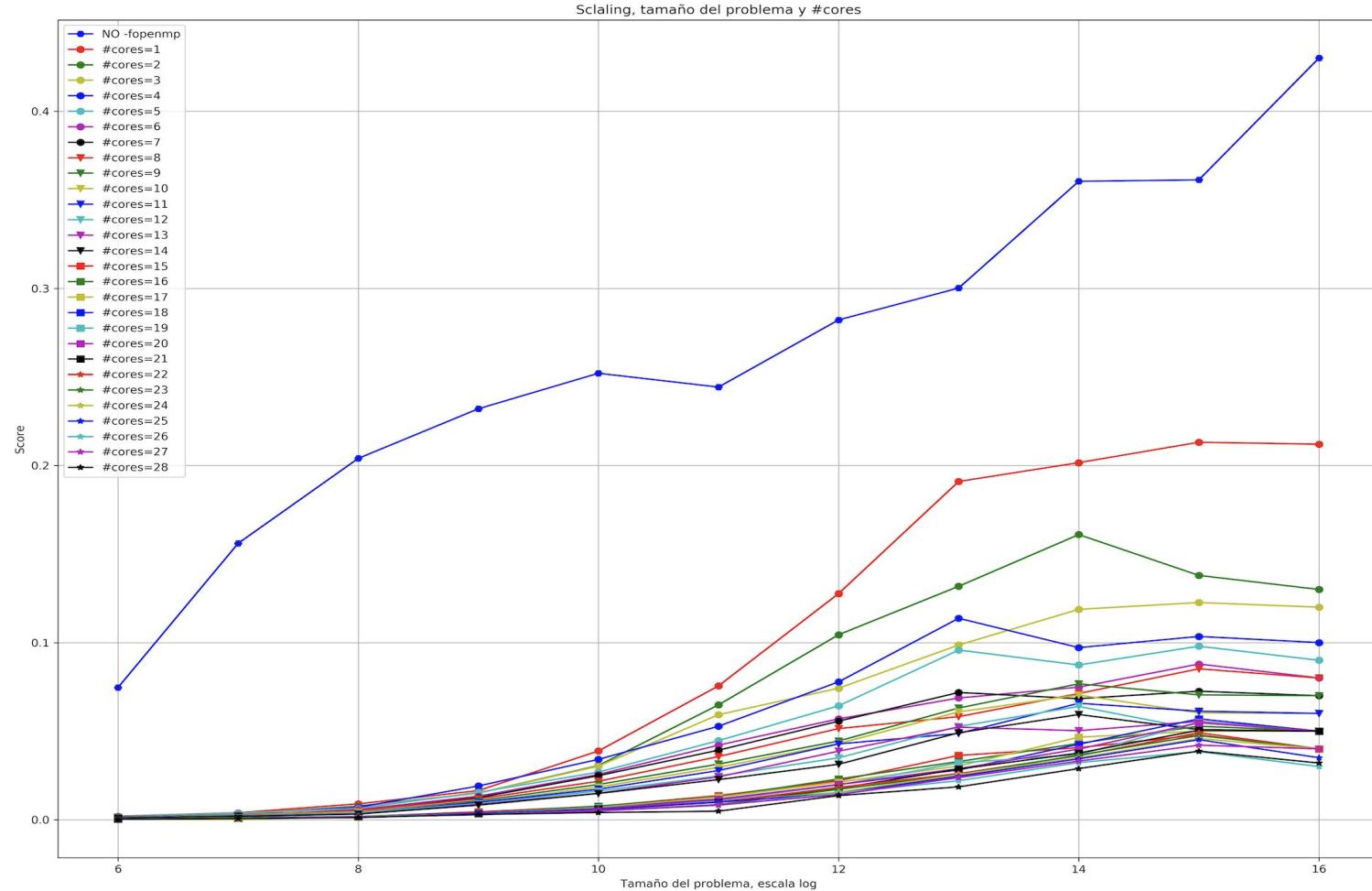
```
//actualizo H
#pragma omp for
for(int i = 0; i < N; i+=32){
    __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);
    __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i]);
    __m256i notact = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
    notact = _mm256_andnot_si256(notact,_mm256_set1_epi8(0xff));
    hv = _mm256_and_si256(notact,hv);
    hv = _mm256_add_epi8(hv,dhv);
    _mm256_storeu_si256((__m256i *) &h[i], hv);
}

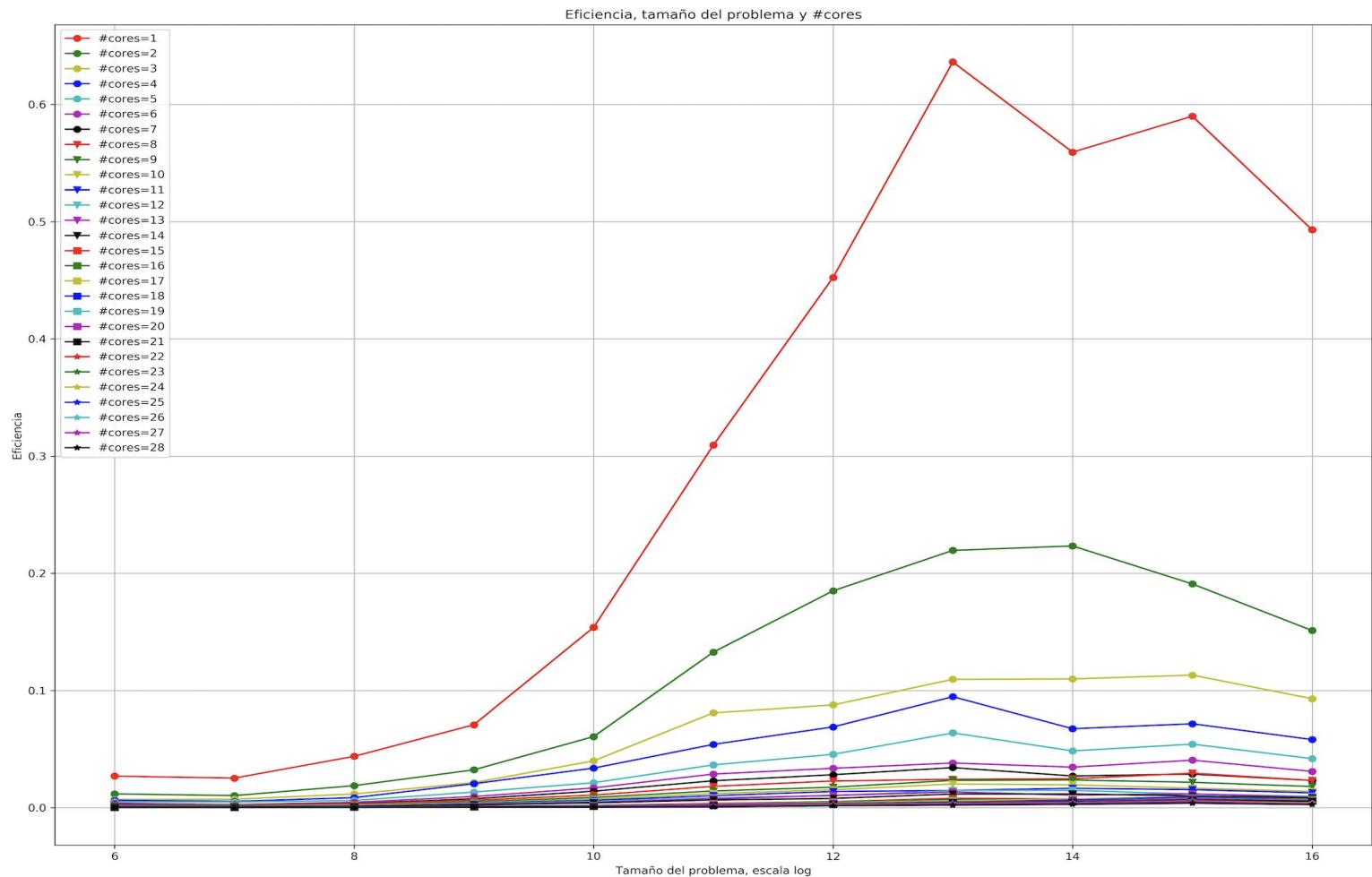
//calculo numero de activos
#pragma omp for reduction(+:nroactivos)
for (int i = 0; i < N; ++i) {
    nroactivos +=(h[i]>1);
}

return nroactivos;
}
```

## Scaling, tamaño del problema y #cores







## Performance counter stats for './tiny\_manna':

```

 434.13 msec task-clock          #  0.999 CPUs utilized
    1 context-switches           #  0.002 K/sec
    0 cpu-migrations            #  0.000 K/sec
 189 page-faults                #  0.435 K/sec
1393606100 cycles              #  3.210 GHz          (30.23%)
4125497014 instructions         #  2.96  insn per cycle   (38.52%)
388842594 branches              # 895.679 M/sec        (39.44%)
1030344 branch-misses          #  0.26% of all branches (40.36%)
533583336 L1-dcache-loads      # 1229.083 M/sec        (41.04%)
29667083 L1-dcache-load-misses #  5.56% of all L1-dcache hits (40.30%)
 13794 LLC-loads                #  0.032 M/sec         (31.08%)
   666 LLC-load-misses          #  4.83% of all LLC-cache hits (30.16%)
<not supported> L1-icache-loads
 10387 L1-icache-load-misses    #  29.48%             (29.48%)
566129800 dTLB-loads            # 1304.052 M/sec        (29.48%)
  254 dTLB-load-misses          #  0.00% of all dTLB cache hits (29.48%)
 1198 iTLB-loads                #  0.003 M/sec         (29.48%)
  386 iTLB-load-misses          #  32.44% of all iTLB cache hits (29.48%)
<not supported> L1-dcache-prefetches
<not supported> L1-dcache-prefetch-misses

```

0.434675627 seconds time elapsed

## Performance counter stats for './tiny\_manna':

```

 2471.25 msec task-clock          #  1.995 CPUs utilized
    15 context-switches           #  0.006 K/sec
     3 cpu-migrations            #  0.001 K/sec
 248 page-faults                #  0.100 K/sec
7481443457 cycles              #  3.027 GHz          (30.12%)
4667025390 instructions         #  0.62  insn per cycle   (37.86%)
645798593 branches              # 261.322 M/sec        (37.96%)
3822434 branch-misses          #  0.59% of all branches (38.28%)
1046236996 L1-dcache-loads      # 423.361 M/sec        (38.61%)
20109081 L1-dcache-load-misses #  1.92% of all L1-dcache hits (38.81%)
 7715822 LLC-loads              #  3.122 M/sec         (31.07%)
 5543710 LLC-load-misses        #  71.85% of all LLC-cache hits (31.08%)
<not supported> L1-icache-loads
  85284 L1-icache-load-misses    #  31.07%             (31.07%)
1042464587 dTLB-loads            # 421.837 M/sec        (31.07%)
 226971 dTLB-load-misses          #  0.02% of all dTLB cache hits (30.97%)
 423588 iTLB-loads               #  0.171 M/sec         (30.64%)
  435900 iTLB-load-misses        # 102.91% of all iTLB cache hits (30.32%)
<not supported> L1-dcache-prefetches
<not supported> L1-dcache-prefetch-misses

```

1.238509364 seconds time elapsed

## Performance counter stats for './tiny\_manna':

```

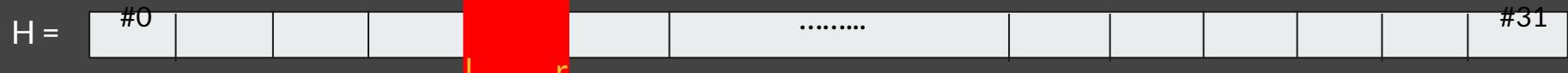
144436.58 msec task-clock          #  27.605 CPUs utilized
 11522 context-switches           #  0.080 K/sec
   480 cpu-migrations            #  0.003 K/sec
   748 page-faults                #  0.005 K/sec
404696929935 cycles              #  2.802 GHz          (30.66%)
66072535233 instructions         #  0.16  insn per cycle   (38.34%)
18199557246 branches              # 126.004 M/sec        (38.41%)
 31636747 branch-misses          #  0.17% of all branches (38.47%)
10041000009 L1-dcache-loads      # 69.518 M/sec         (38.53%)
 209633453 L1-dcache-load-misses #  2.09% of all L1-dcache hits (38.57%)
 145694953 LLC-loads              #  1.009 M/sec         (30.86%)
 27495105 LLC-load-misses        #  18.87% of all LLC-cache hits (30.82%)
<not supported> L1-icache-loads
  5120663 L1-icache-load-misses    #  30.79%             (30.79%)
9728959102 dTLB-loads            # 67.358 M/sec         (30.77%)
  446446 dTLB-load-misses          #  0.00% of all dTLB cache hits (30.73%)
  431512 iTLB-loads               #  0.003 M/sec         (30.71%)
  486546 iTLB-load-misses        # 112.75% of all iTLB cache hits (30.68%)
<not supported> L1-dcache-prefetches
<not supported> L1-dcache-prefetch-misses

```

5.232170305 seconds time elapsed

- +480 cpu-migrations
- +559 page faults
- +(numero grande) de fallas en caches L1
- + 27.5 MILLONES de LLC-Load-Misses

# Vertical Lab2



Dividimos en 32 bloques. (Múltiplo de 32 por los saltos de la vectorización)

- + Cambios en el código para que funcione en rangos
- + Cambio en el código para no depender de los costados de cada bloque

Paralelo cada bloque (tasks)

Calcular Descargar rango l,r

Descargar Izquierda rango l,r

Descargar Derecha rango l,r

Wait los 32 bloques

Actualizar H rango 0,N

```
int descargar_omp_vertical(Manna_Array &h, Manna_Array &dh){
    uint8_t a[N];
    dh.fill(0);
    memset(a,0,sizeof(a));
    for(i,0,N)inst+=h[i]*(h[i]>1);

    const int tasks = 32;

    #pragma omp parallel for shared(h,dh,a) schedule(dynamic)
    for(int i = 0; i < tasks; i++){
        #pragma omp task firstprivate(i) shared(h,dh,a)
        {
            int l = N/tasks*i;
            int r = N/tasks*(i+1);
            calcDescargar(h.data(),a,l,r);
            descargarIzq(dh.data(),a,l,r);
            descargarDer(h.data(), dh.data(),a,l,r);
        }
    }

    #pragma omp parallel for shared(h,dh,a)
    for(int i = 0; i < tasks; i++){
        #pragma omp task firstprivate(i) shared(h,dh,a)
        {
            int l = N/tasks*i;
            int r = N/tasks*(i+1);
            actH(h.data(),dh.data(),l,r);
        }
    }

    unsigned int nroactivos=0;
    for (int i = 0; i < N; ++i) {
        nroactivos +=(h[i]>1);
    }

    return nroactivos;
}
```

```
void calcDescargar(uint8_t *h, uint8_t *a, int l, int r){
    for(int i = l; i < r; i+=32){
        __m256i res = _mm256_set1_epi8(0);
        __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);
        __m256i activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
        hv = _mm256_and_si256(hv, activos);
        while(_mm256_testz_si256(activos, activos) == 0){
            __m256i tirar = _mm256_min_epi8(hv, _mm256_set1_epi8(8));
            tirar = _mm256_and_si256(tirar, activos);
            __m256i rand = _mm256_loadu_si256((__m256i*) get_rand());
            __m256i mask_8b = get_mask_8b(tirar);
            rand = _mm256_and_si256(rand,mask_8b);
            __m256i aux = _mm256_popcount_epi8(rand);
            hv = _mm256_sub_epi8(hv,tirar);
            res = _mm256_add_epi8(res,aux);
            activos = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(0));
        }
        _mm256_storeu_si256((__m256i *) &a[i], res);
    }
}
```

```
void descargarIzq(uint8_t *dh, uint8_t *a, int l, int r){
    #pragma omp critical
    {
        dh[(l-1+N)%N] += a[l];
        dh[(l+N)%N] += a[l+1];
        dh[r-2] += a[r-1];
    }

    for(int i = l+2; i < l+32; ++i){
        dh[i-1] += a[i];
    }

    for(int i = r-32; i < r-1; i++){
        dh[i-1] += a[i];
    }

    for(int i = l+32; i < r-33; i+=32){
        __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i-1]);
        __m256i av = _mm256_loadu_si256((__m256i*) &a[i]);
        dhv = _mm256_add_epi8(dhv,av);
        _mm256_storeu_si256((__m256i *) &dh[i-1], dhv);
    }
}

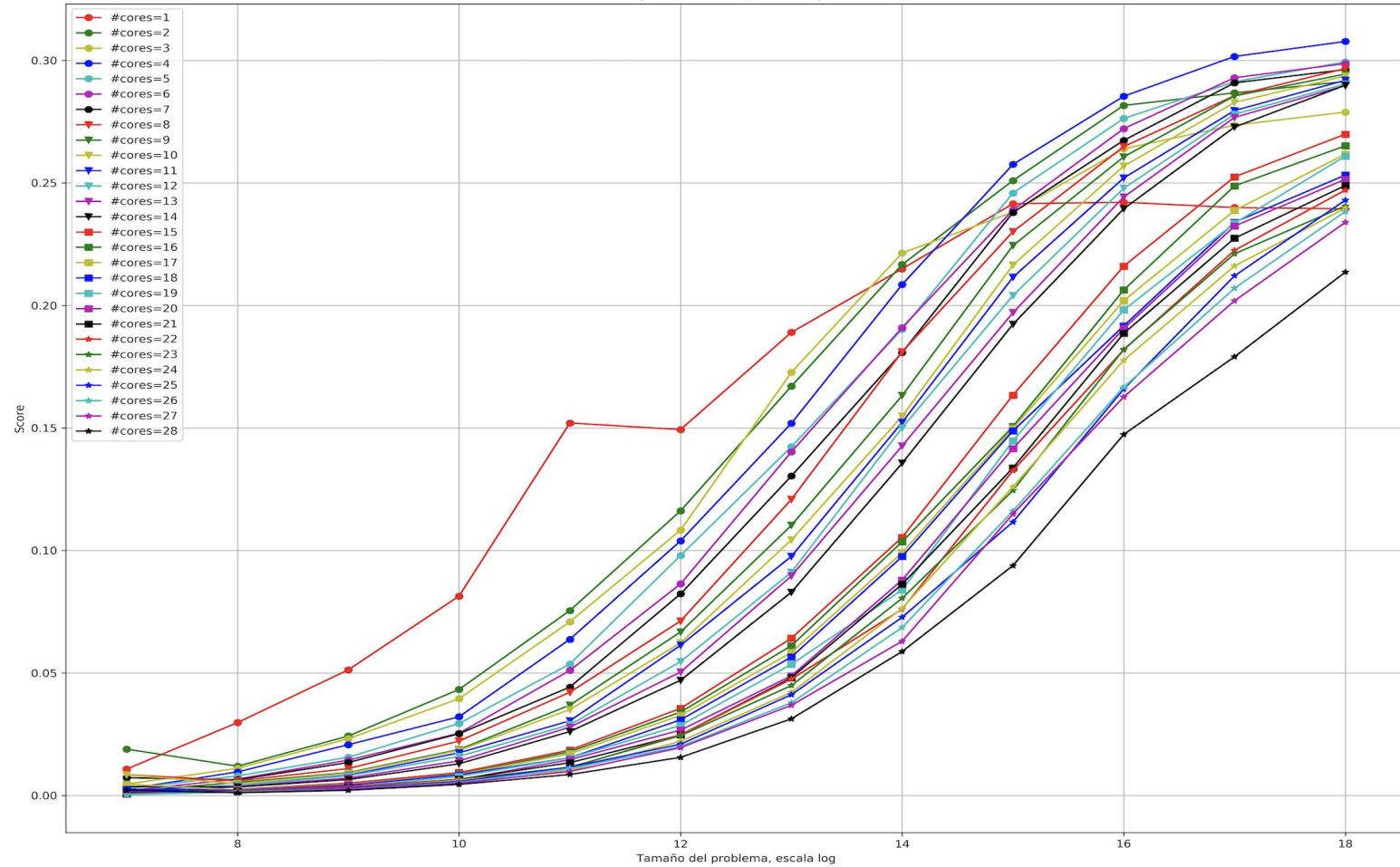
void descargarDer(uint8_t *h, uint8_t *dh, uint8_t *a, int l, int r){
    #pragma omp critical
    {
        dh[l+1] += (h[l] - a[l])*(h[l]>1);
        dh[(r-1)%N] += (h[r-2] - a[r-2])*(h[r-2]>1);
        dh[(r)%N] += (h[r-1] - a[r-1])*(h[r-1]>1);
    }

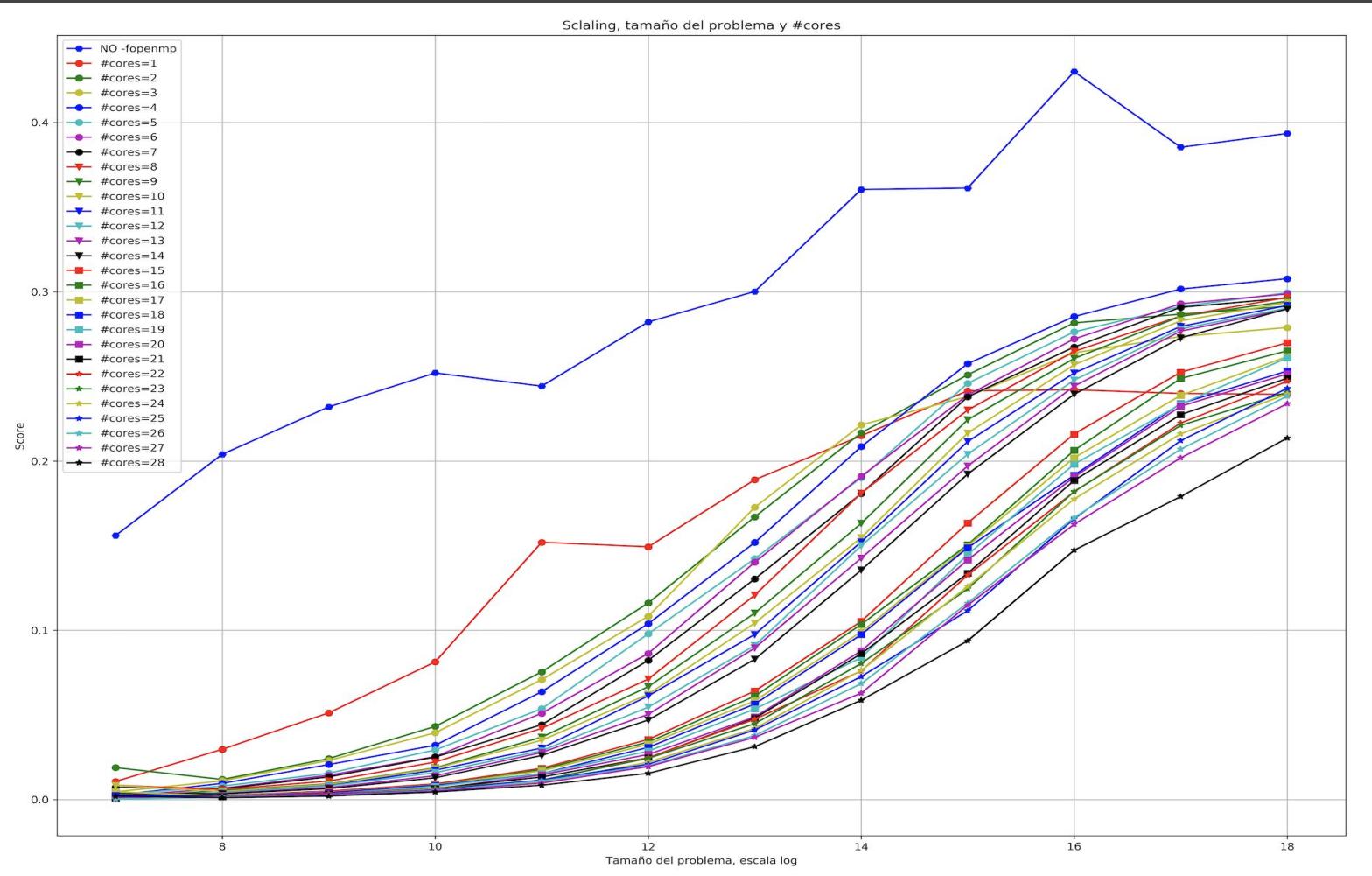
    for(int i = l+1; i < l+32; ++i){
        dh[i+1] += (h[i] - a[i])*(h[i]>1);
    }

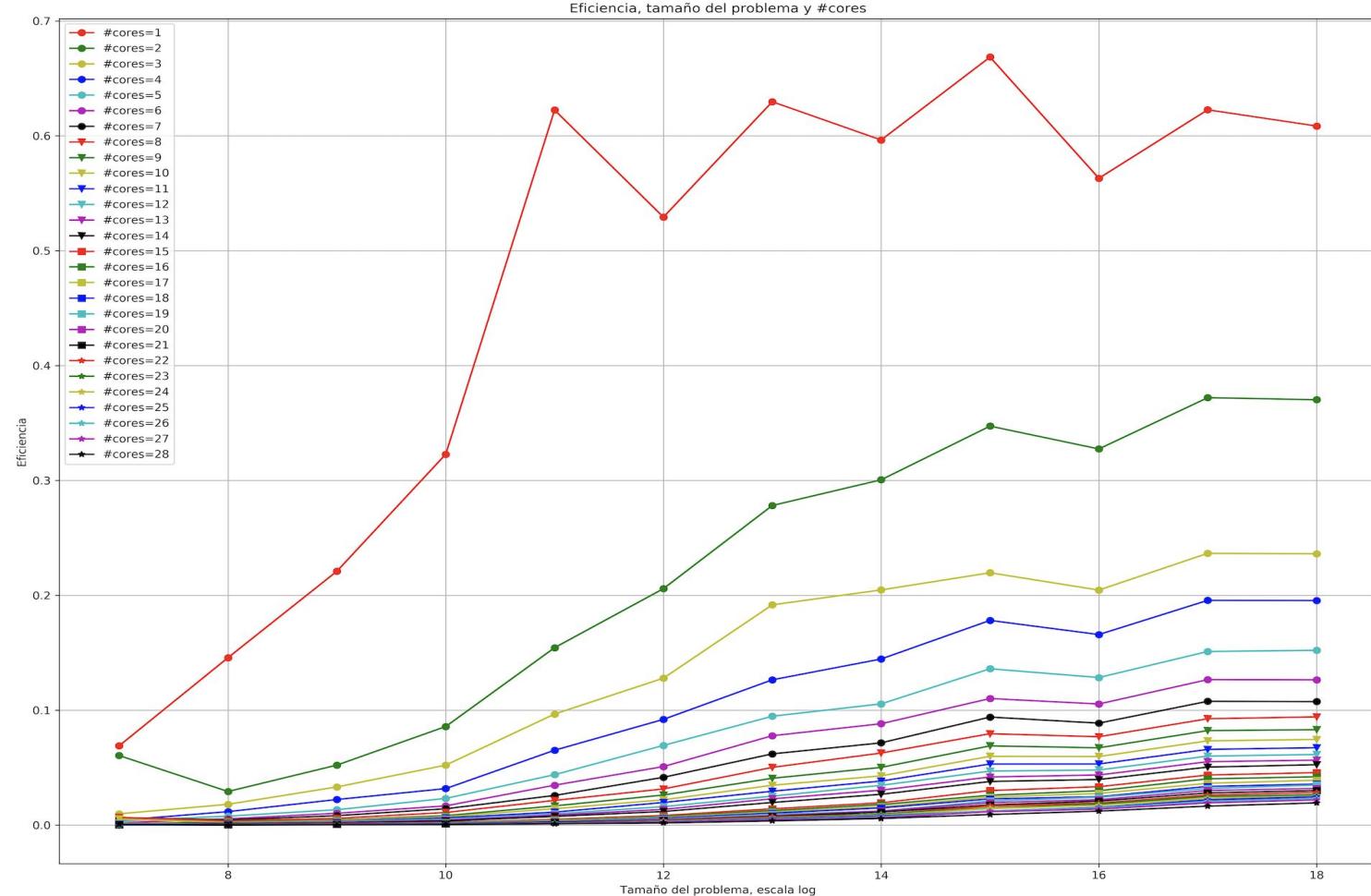
    for(int i = r-32; i < r-2; i++){
        dh[i+1] += (h[i] - a[i])*(h[i]>1);
    }

    for(int i = l+32; i < r-33; i+=32){
        __m256i hv = _mm256_loadu_si256((__m256i*) &h[i]);
        __m256i act = _mm256_cmpgt_epi8(hv, _mm256_set1_epi8(1));
        __m256i dhv = _mm256_loadu_si256((__m256i*) &dh[i+1]);
        __m256i av = _mm256_loadu_si256((__m256i*) &a[i]);
        hv = _mm256_and_si256(hv,act);
        av = _mm256_and_si256(av,act);
        dhv = _mm256_add_epi8(dhv,hv);
        dhv = _mm256_sub_epi8(dhv,av);
        _mm256_storeu_si256((__m256i *) &dh[i+1], dhv);
    }
}
```

Scaling, tamaño del problema y #cores



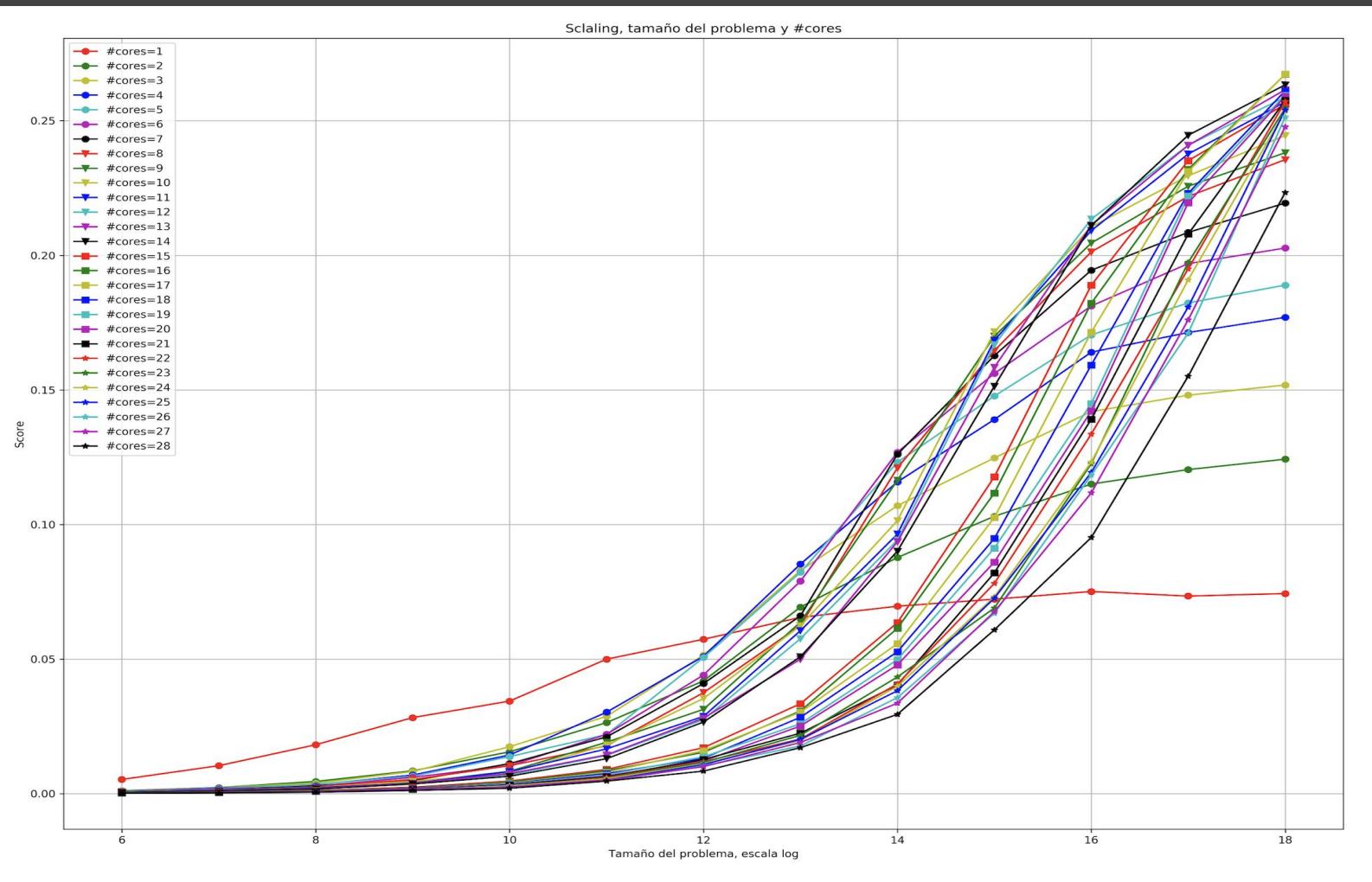


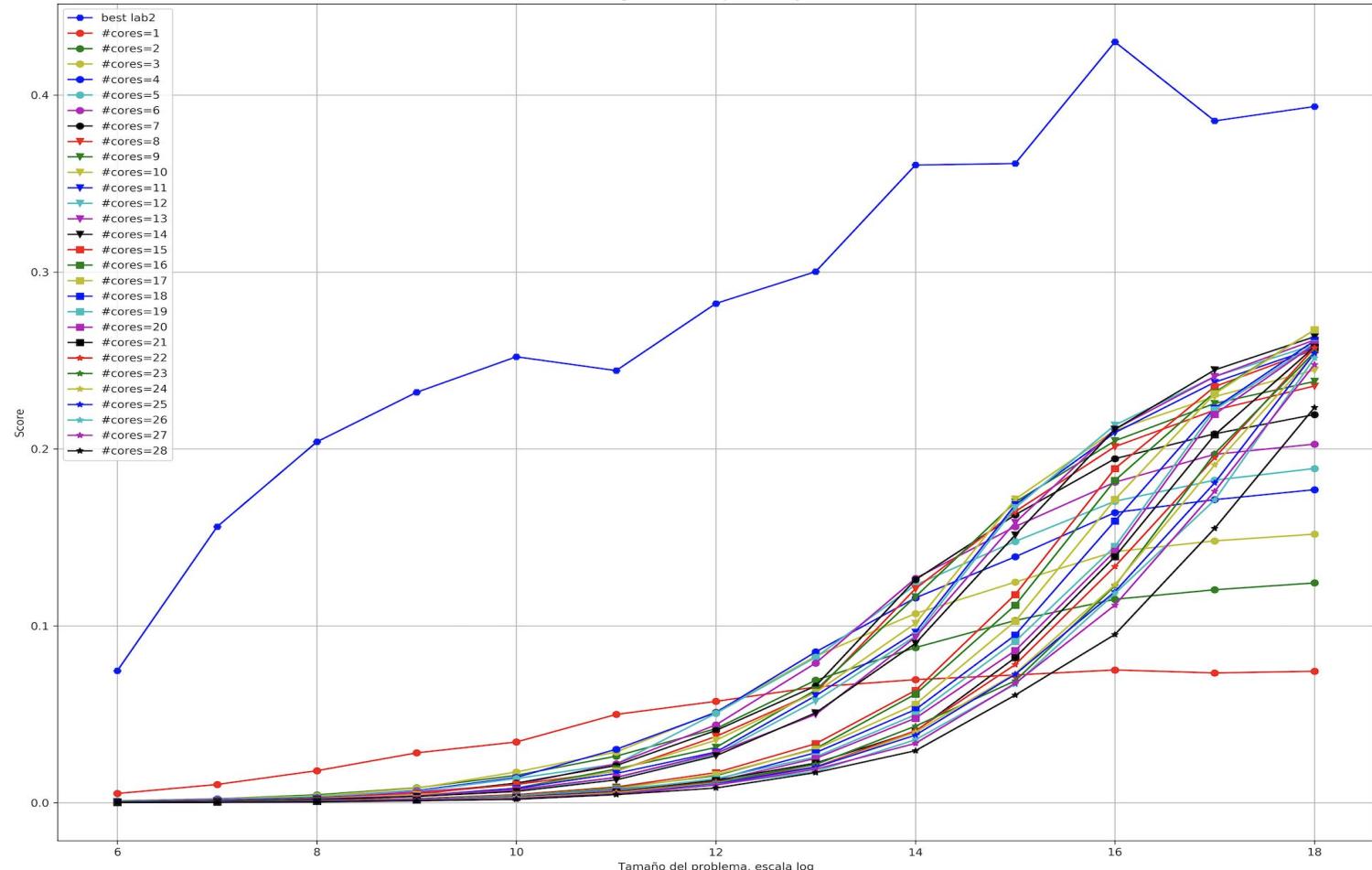


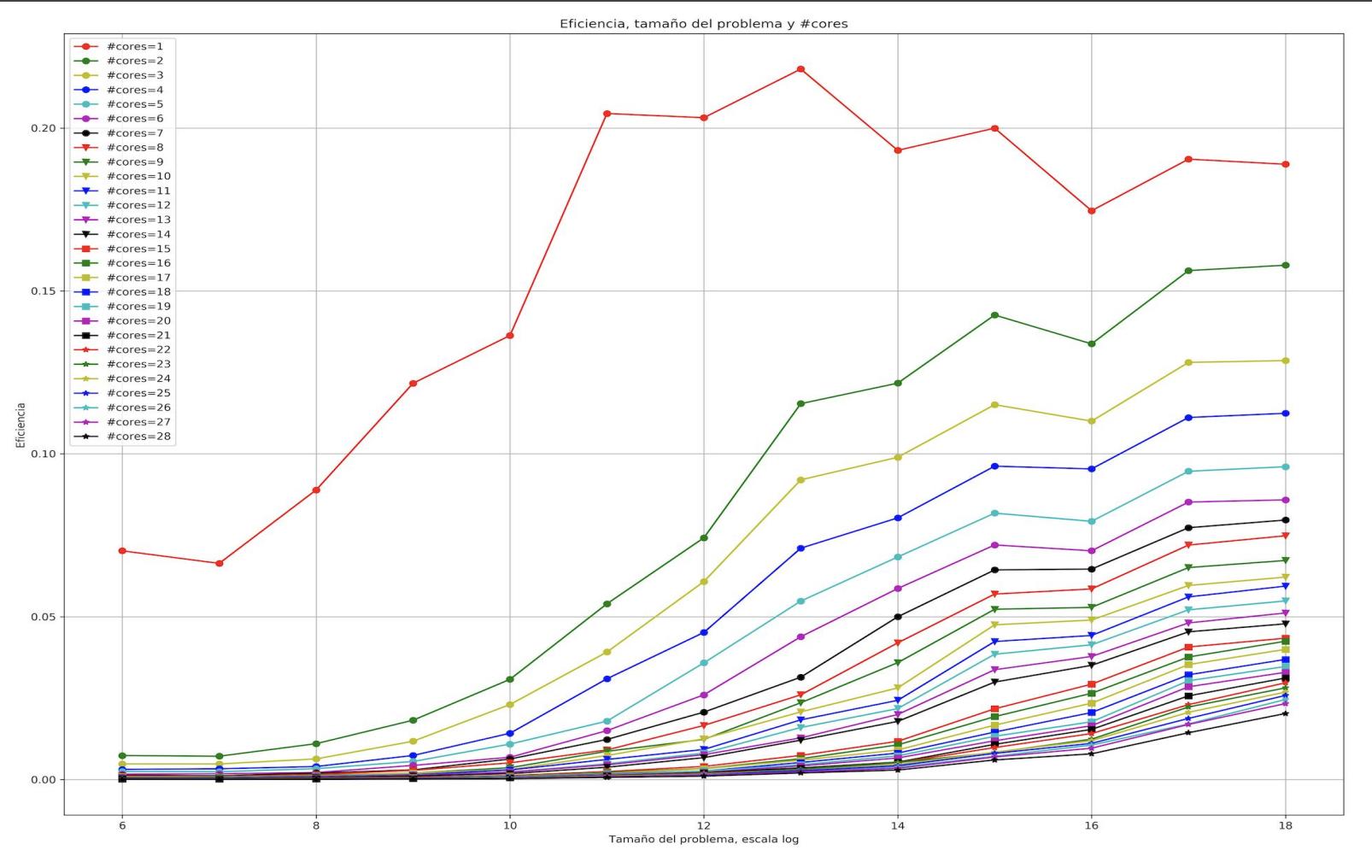
# Vertical Lab1

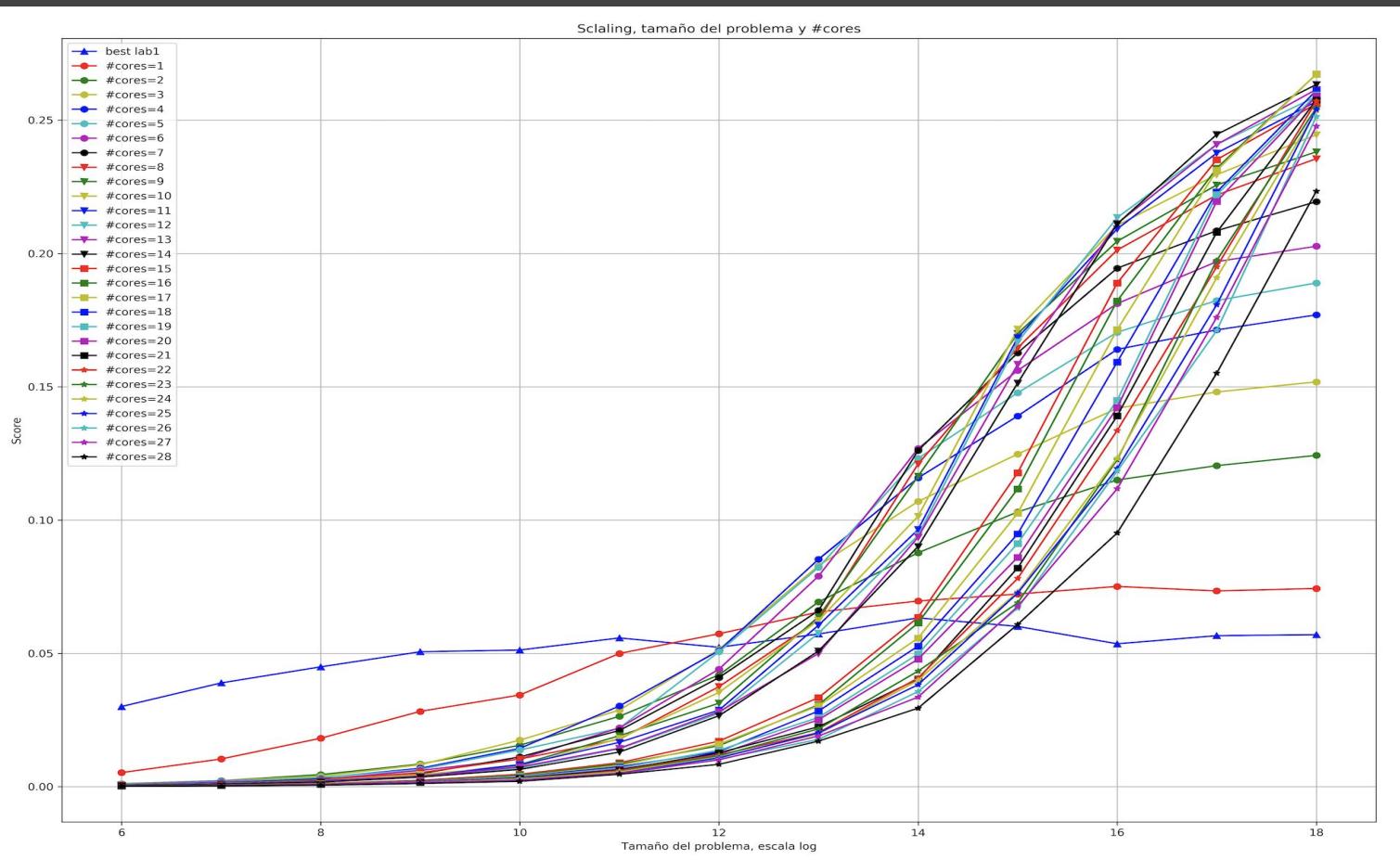
```
int descargar(Manna_Array &h, Manna_Array &dh){  
    dh.fill(0);  
    for(i,0,N)inst+=h[i](h[i]>1);  
  
    #pragma omp parallel for shared(h,dh) schedule (dynamic)  
    for (int i = 0; i < 32; ++i) {  
        #pragma omp task shared(h,dh,a,i)  
        {  
            lr(i*N/32,(i+1)*N/32,h,dh);  
        }  
    }  
    unsigned int nroactivos=0;  
    #pragma omp parallel for shared(h,dh) reduction(+:nroactivos)  
    for (int i = 0; i < N; ++i) {  
        h[i] += dh[i];  
        nroactivos+=(h[i]>1);  
    }  
    return nroactivos;  
}
```

```
void lr(int l, int r, Manna_Array &h, Manna_Array &dh){  
    #pragma omp critical  
    {  
        for (int i = l; i < l+2; ++i) {  
            // si es activo lo descargo aleatoriamente  
            if (h[i] > 1) {  
                int k = builtin_popcount(get_rand() & ((1ll<<h[i])-1));  
                dh[(i+1)%N]+=k;  
                dh[(i-1+N)%N]+=(h[i]-k);  
                h[i] = 0;  
            }  
        }  
    }  
    for (int i = l+2; i < r-2; ++i) {  
        // si es activo lo descargo aleatoriamente  
        if (h[i] > 1) {  
            int k = builtin_popcount(get_rand() & ((1ll<<h[i])-1));  
            dh[(i+1)%N]+=k;  
            dh[(i-1+N)%N]+=(h[i]-k);  
            h[i] = 0;  
        }  
    }  
    #pragma omp critical  
    {  
        for (int i = r-2; i < r; ++i) {  
            // si es activo lo descargo aleatoriamente  
            if (h[i] > 1) {  
                int k = __builtin_popcount(get_rand() & ((1ll<<h[i])-1));  
                dh[(i+1)%N]+=k;  
                dh[(i-1+N)%N]+=(h[i]-k);  
                h[i] = 0;  
            }  
        }  
    }  
}
```



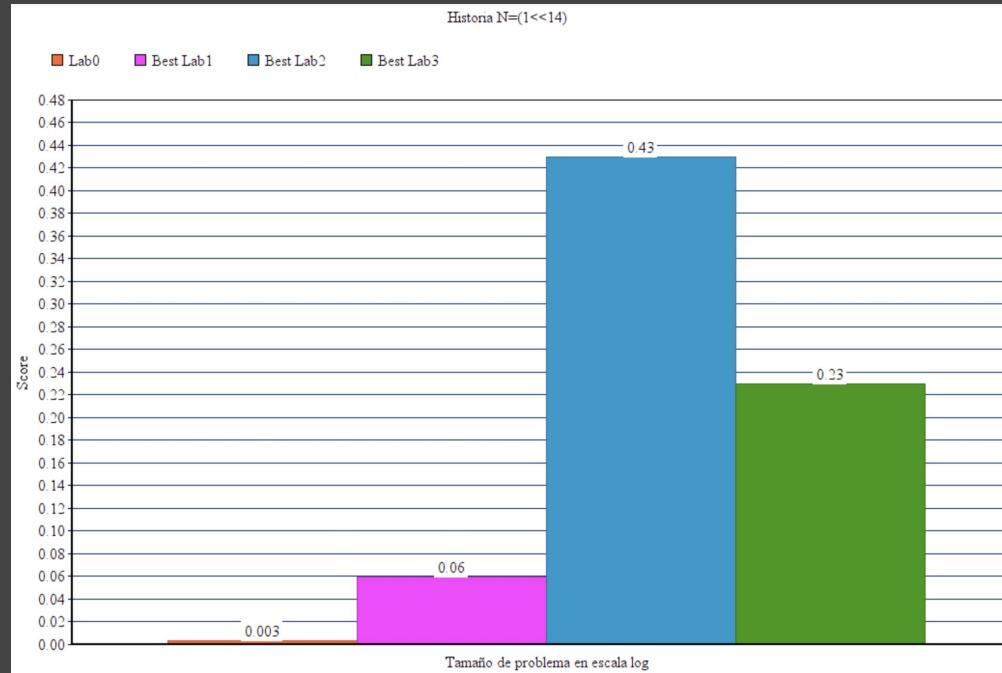






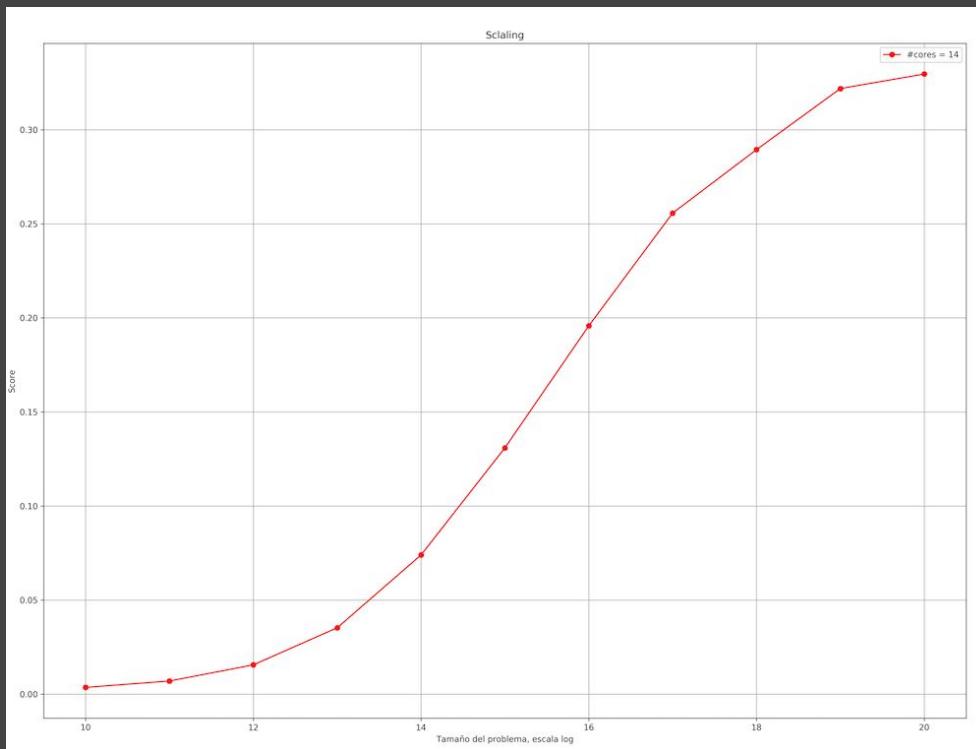
# Conclusiones

(se tomó como válido  
cuando #cores > 1, sino  
no tiene sentido)



PD para los de 2022: No  
elijan manna

(EDIT) Realizamos algunas modificaciones guiándonos por las observaciones que nos hizo el Profe BC, y mejoró un poco, a continuación vemos como escala `omp+avx2` con 14 cores en el Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz





---

# Sandpile

## Laboratorio 4

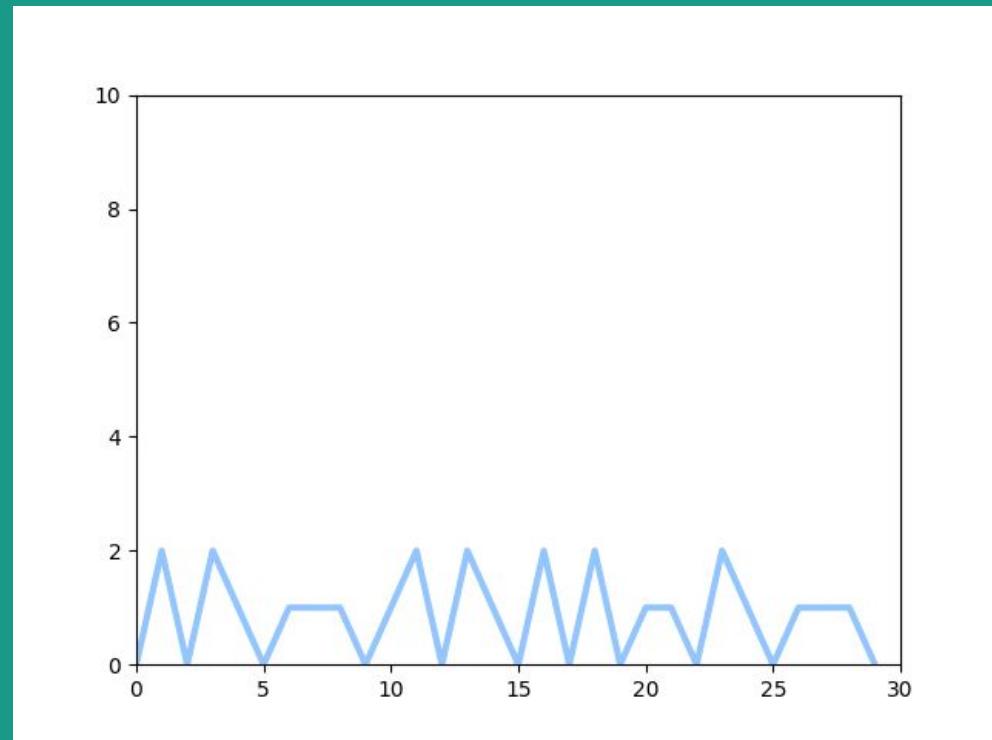
Federico Gonzalez Kriegel y Thomas Vadora



# Recordemos el Problema

---

- Sea  $h[i]$  el número de granitos en el sitio  $i$ ,  $0 < i < N - 1$ .
- Si  $h[i] > 1$  el sitio  $i$  esta "activo".
- Al tiempo  $t$ , un sitio "activo" se "descarga" completamente tirando cada uno de sus granitos aleatoriamente y con igual probabilidad a la izquierda o a la derecha

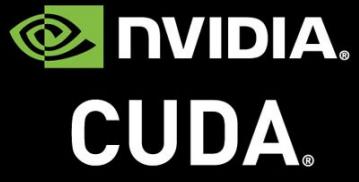




---

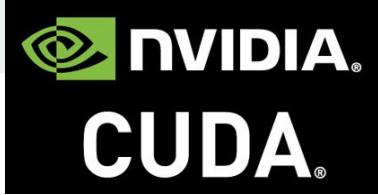
# Recordemos la Metrica

Llamémosle descargar a sumarle 1 a alguno de mis dos vecinos al azar y autorestarme 1. Entonces nuestra métrica que vamos a tratar de optimizar va a ser: Score = Cantidad de descargas / tiempo de ejecución del programa. Queremos maximizar Score



Idea:

- Calculamos la cantidad de instrucciones.  $\text{Sum}(h[i] * (h[i] > 1))$ , lo necesitamos para la métrica.
- Calculamos un arreglo de cuantos se van a tirar a la izquierda. (para los random usamos cuRAND)
- Descargamos a ambos lados
- Actualizamos h, y contamos la cantidad de activos



```
cudaError_t err;

curandState_t *d_state;
checkCudaErrors(cudaMalloc(&d_state, N*sizeof(curandState_t)));
initstuff<<<N/NBLOCKS,NBLOCKS>>>(d_state);
err = cudaGetLastError();
if(err!=cudaSuccess)cout<<cudaGetErrorString(err)<<endl;

Manna_Array h, cudah, dh, a, activos, nactivos, cp, in;

h = (Manna_Array)malloc(N*sizeof(int));
inicializacion(h);
desestabilizacion_inicial(h);

checkCudaErrors(cudaMalloc(&cudah, N*sizeof(int));
checkCudaErrors(cudaMemcpy(cudah,h,N*sizeof(int),cudaMemcpyHostToDevice));
checkCudaErrors(cudaMalloc(&dh, N*sizeof(int)));
checkCudaErrors(cudaMalloc(&a, N*sizeof(int)));
cp = (Manna_Array)malloc(1*sizeof(int));
nactivos = (Manna_Array)malloc(1*sizeof(int));
checkCudaErrors(cudaMalloc(&in,1*sizeof(int)));
checkCudaErrors(cudaMalloc(&activos,1*sizeof(int)));
```

```
__global__ void initstuff(curandState_t *state) {

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init(1337, idx, 0, &state[idx]);
}
```

Inicializamos y desestabilizamos en el host

**REGLA: MINIMIZAR LA CANTIDAD DE INFORMACIÓN QUE MOVEMOS DEL HOST AL DEVICE Y VICEVERSA.**

Copio h en cudah que vive en el device y a partir de ahora ese es nuestro h, luego declaro todos los arreglos en el device, excepto las variables que en cada iteracion van a traer un solo resultado necesario en el host (instrucciones y cantidad de activos)



# CUDA®

```
do{
    t++;

    calcinst<<<N/NBLOCKS,NBLOCKS>>>(cudah,dh,in);
    checkCudaErrors(cudaMemcpy(cp,in,1*sizeof(int),cudaMemcpyDeviceToHost));
    inst+=cp[0];

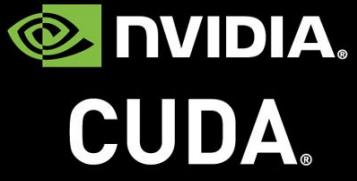
    calcdescargar<<<N/NBLOCKS,NBLOCKS>>>(cudah,dh,a,d_state);
    err = cudaGetLastError();
    if(err!=cudaSuccess)cout<<cudaGetErrorString(err)<<endl;

    descargarcuda<<<N/NBLOCKS,NBLOCKS>>>(cudah,dh,a);
    err = cudaGetLastError();
    if(err!=cudaSuccess)cout<<cudaGetErrorString(err)<<endl;

    calcactivosandact<<<N/NBLOCKS,NBLOCKS>>>(cudah,dh,activos);
    err = cudaGetLastError();
    if(err!=cudaSuccess)cout<<cudaGetErrorString(err)<<endl;

    checkCudaErrors(cudaMemcpy(nactivos,activos,1*sizeof(int),cudaMemcpyDeviceToHost));
    activity = nactivos[0];

}while(t<NSTEPS && activity>0);
```



```
__global__ void calcdescargar(int * h, int * dh, int * a, curandState_t * state){

    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    a[i]=0;
    if(h[i]>1){
        float r = curand_uniform(&state[i]);
        long long x = *((int *)(&r));
        r = curand_uniform(&state[i]);
        int y = *((int *)(&r));
        long long z = (long long)(x<<32)|y;
        z&=((1ll<<h[i])-1);
        a[i]=__popcll(z);
    }
}
```



# CUDA®

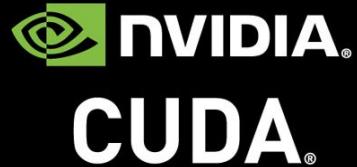
```
__global__ void descargarcuda(int *h, int *dh, int *a){
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int l = i-1, r = i+1;
    dh[i]=0;
    if(i==0){
        l = N-1;
        r = 1;
    } else if(i==N-1){
        r = 0;
        l = N-2;
    }
    dh[i]+=a[r];
    dh[i]+=((h[l]-a[l])*(h[l]>r));
    h[l]*=(h[l]<=r);
}
```



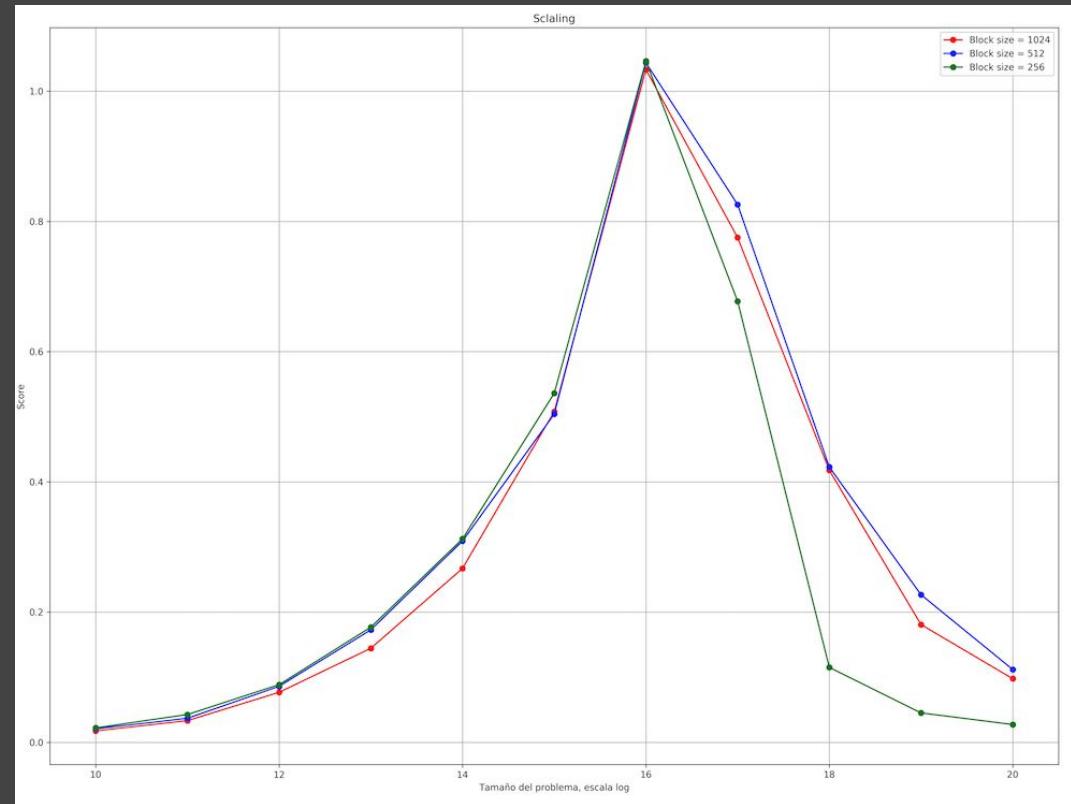
Calcular la cantidad de instrucciones y la cantidad de activos se puede ver como una suma de un arreglo con condiciones en cada elemento. Entonces utilizamos un reduce modificado y que hace uso de todos los niveles de memoria de forma eficiente (osea el código de nwlolvick con un par de cambios)

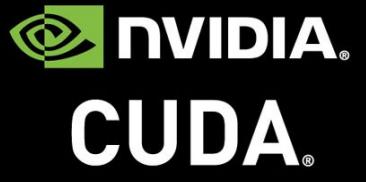
```
__global__ void calcinst(int *h, int *dh, int *s){  
    s[0]=0;  
    __shared__ int block_reduce;  
  
    unsigned int gtid = blockIdx.x * blockDim.x + threadIdx.x;  
    unsigned int tid = threadIdx.x;  
    int lane = tid & 31;  
  
    block_reduce = 0;  
  
    __syncthreads();  
  
    int warp_reduce = h[gtid]*(h[gtid]>1);  
    int FULL_MASK = 0xffffffff;  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 16);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 8);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 4);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 2);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 1);  
  
    if(0==lane){  
        atomicAdd(&block_reduce, warp_reduce);  
    }  
  
    __syncthreads();  
  
    if(0==tid){  
        atomicAdd(&s[0], block_reduce);  
    }  
}
```

```
__global__ void calcactivosandact(int *h, int *dh, int *s){  
    s[0]=0;  
    __shared__ int block_reduce;  
  
    unsigned int gtid = blockIdx.x * blockDim.x + threadIdx.x;  
    unsigned int tid = threadIdx.x;  
    int lane = tid & 31;  
  
    block_reduce = 0;  
  
    h[gtid]+=dh[gtid];  
  
    __syncthreads();  
  
    int warp_reduce = (h[gtid]>1);  
    int FULL_MASK = 0xffffffff;  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 16);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 8);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 4);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 2);  
    warp_reduce += __shfl_down_sync(FULL_MASK, warp_reduce, 1);  
  
    if(0==lane){  
        atomicAdd(&block_reduce, warp_reduce);  
    }  
  
    __syncthreads();  
  
    if(0==tid){  
        atomicAdd(&s[0], block_reduce);  
    }  
}
```

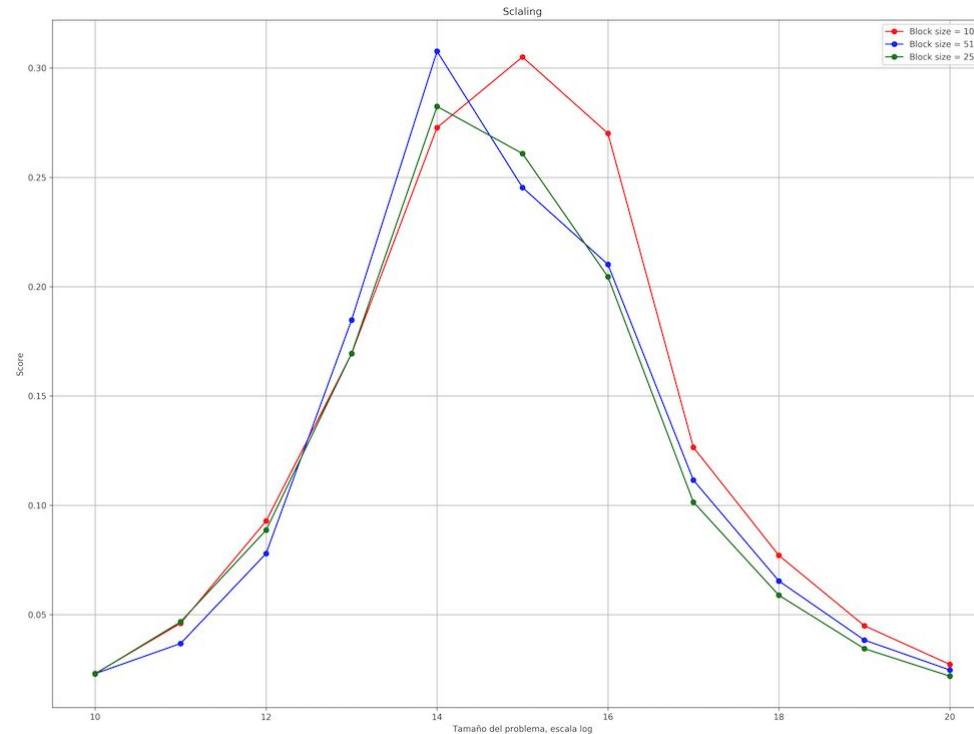


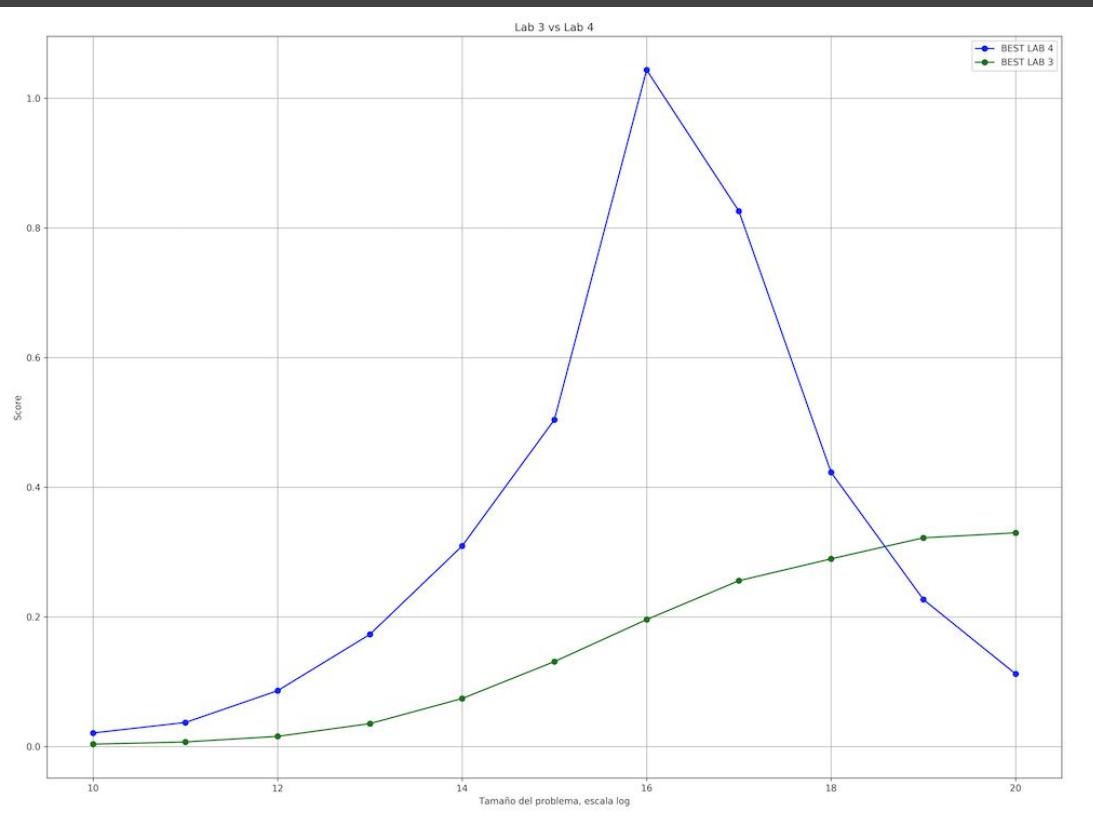
# Resultados NVIDIA RTX 2080 Ti





# Resultados NVIDIA GTX 1070





## Hipótesis:

- El problema es muy memory bound
- No hay mucho cálculo flotante ni operaciones costosas, el micro no se puede lucir
- Puro asignaciones a memoria
- La GPU se topa con la memoria rápido, pero escala mejor en tamaño de problema mediano y chico



VS



(No es tan justo  
comparar, pero si  
podemos comparar  
\$\$\$\$)

Asumamos:



600 USD



1000 USD



2000 USD

Veamos para cada tamaño de problema el cantidad de descargas por segundo por dolar (USD) para cada hardware.  
(bestscore/preciohardware)



Relación performance/precio

