@BritoAlv

Fastest way to fill an array with 1's

Dado un array, digamos que necesitamos rellenarlo con un valor prefijado como 1. Se obtuvieron los siguientes resultados:

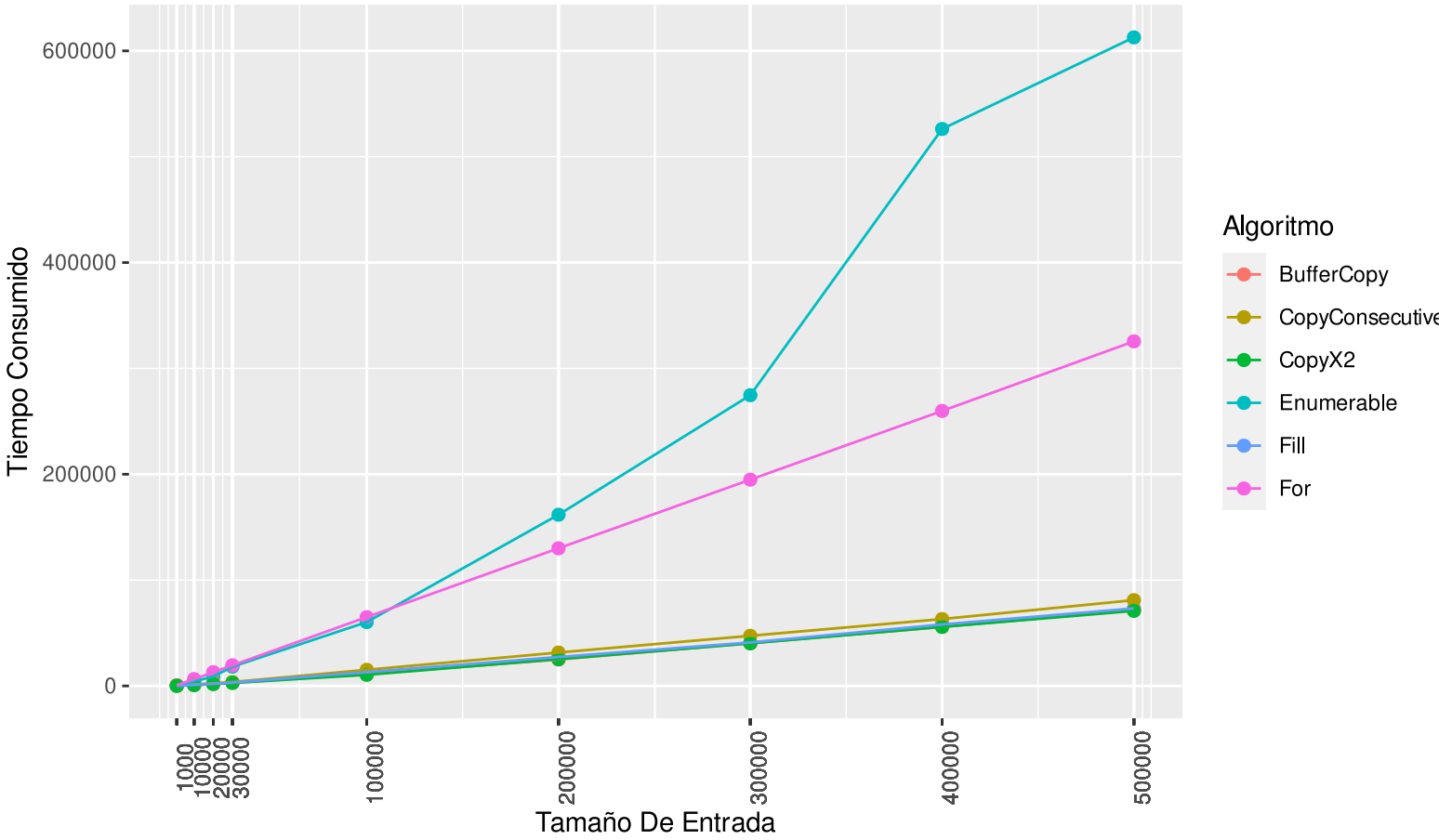

Figura 1: Plot de los resultados del bench-mark para distintas entradas

Primero se usaron los tres siguientes algoritmos para hacer un benchmark:

```
// la peor idea
data = Enumerable.Repeat(1, N).ToArray();

// lo que se le ocurriría a una persona en su sano juicio
for (int i = 0; i < data.Length; i++)
{
```

```
        data[i] = 1;
    }

    // librería de c#
    Array.Fill(data, 1);
```

Como se puede observar en el plot, la solución usando `IEnumerable` no es la más optimizada, se pensaría que el `For` es la mejor opción, pués no, resulta que el método `Fill` se las arregla para ser más rápido. ¿ A qué se debe esto?.

1. IL-For

Primero el `For` es linear, y bastante estable como se puede observar en los plots.

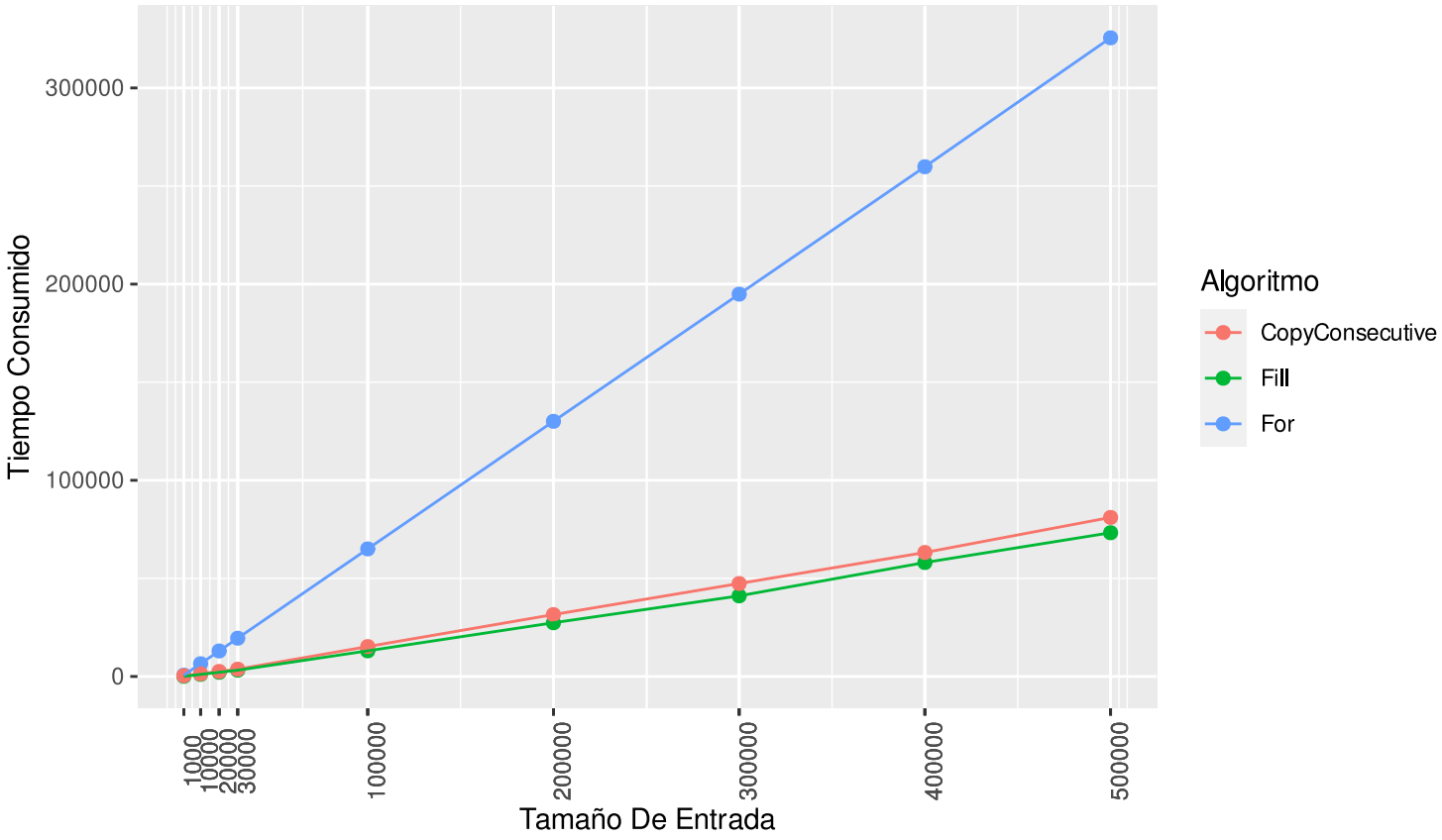

Figura 2: Plot de los resultados del bench-mark para distintas entradas

El siguiente for es traducido a `IL` por C# obteniéndose:

```
int[] data = new int[5];
for (int l = 0; l < data.Length; l++)
{
    data[l] = 3;
}
```

```
    // IL CODE.
    // push 5 en el stack como int32.
    IL_0000: ldc.i4.5

    // pop del stack el tamaño del array, crear el array, y
    // push en el stack una referencia al array.
    IL_0001: newarr

    // pop lo que hay en el stack y guardarlo en la variable local 0.
    IL_0006: stloc.0

    // push 5 en el stack como int32.
    IL_0007: ldc.i4.0 // poner 0 en el stack como int32.

    // pop lo que hay en el stack y guardarlo en la variable local 1.
    IL_0008: stloc.1
    // sequence point: hidden
    IL_0009: br.s IL_0013 // instrucción de branch.
    // loop start (head: IL_0013)

    // push variable local 0 en el stack.
    IL_000b: ldloc.0

    // push variable local 1 en el stack.
    IL_000c: ldloc.1

    // push 3 en el stack como int32.
    IL_000d: ldc.i4.3

    // pop el 3, pop el l, pop el array y asignar el valor.
    IL_000e: stelem.i4
```

```
    // push variable local 1 en el stack.
    IL_000f: ldloc.1

    // push 3 en el stack como int32.
    IL_0010: ldc.i4.1

    // pop 3, pop local 1 y push el resultado en el stack.
    IL_0011: add

    // pop lo que hay en el stack y lo pone en variable local 1.
    IL_0012: stloc.1

    // push varible local 1 (l) en el stack
    IL_0013: ldloc.1

    // push el array en el stack
    IL_0014: ldloc.0

    // pop el array,calcula la longitud y la push en el stack
    IL_0015: ldlen

    // pop lo que hay en el stack, lo convierte a int32 y lo push nuevamente.
    IL_0016: conv.i4

    // push los dos primeros valores del stack y jump to IL_000b
    // si es verdadero, si no acaba el loop.
    IL_0017: blt.s IL_000b
// end loop
```

Entender lo que hace este `For` es importante púes como se observa, inicializa la variable l y después repetidamente asigna el valor de l y comprueba si puede continuar. Esto es $O(n)$, en fin si hay alguna implementación mejor deberá realizar menos instrucciones, **eso es lo que hace Fill**.

2. SIMD

La explicación está en el uso de **SIMD**, que significa Single Instruction Multiple Data, esto consiste en nuevas instrucciones que poseen los procesadores, cuya idea es aplicar la misma instrucción a various bloques de memoria de un tiro, por ejemplo en 256 bits se pueden guardar 8 floats ocupando 32 bits cada uno, si con una instrucción accedemos a ese bloque de 256 bits, y con otra indicamos que queremos asignarle el valor de 1 a cada valor, realizamos las instrucciones de 8 en 8 vs a 1 a 1, he aquí la optimización. En esto se basan los otros métodos `Copy` también usados, que prácticamente son iguales de rápido que *Fill*.

Verificando la rapidez de las instrucciones de **SIMD** ,usando el lenguaje C++, a través de una implementación del DotProduct que es un ejemplo típico se obtuvo que:

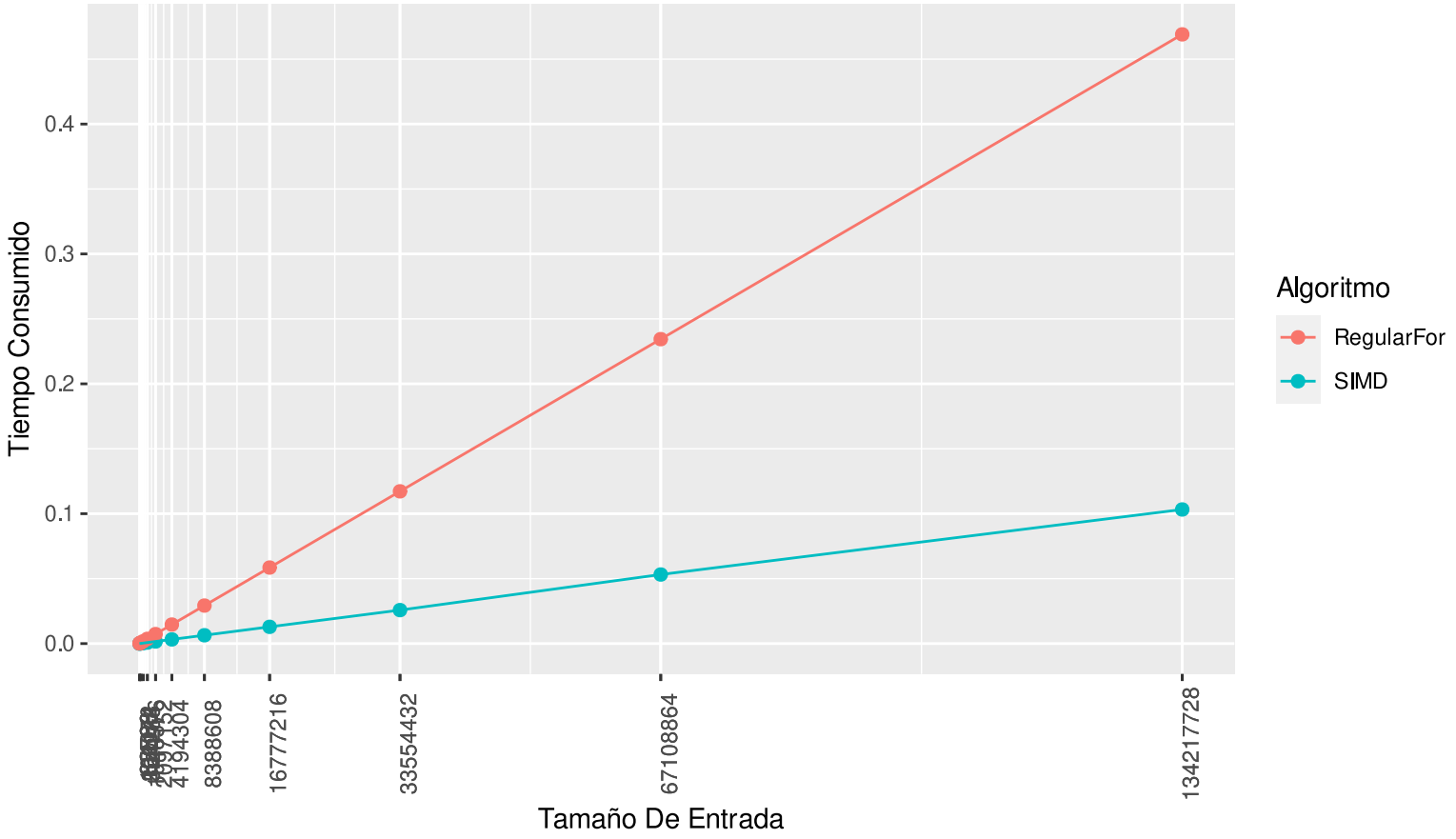

Figura 3: La entrada es potencias de 2, ahora en C++