Práctica SIMD + OpenMP

Se han probado todos los códigos utilizando N de tamaño 1000, 131072 (valor por defecto) y 100000000. Con terms = 12.

sinx.c

Tiempos con la versión por defecto.

N	1000	Defecto	100000000
sinx	0,00108	0,02891	19,64262

sinx-v.c

En este código optimizamos la versión original utilizando intrínsecas del procesador.

```
void sinx (int N, int terms, float *x, float *y)
{
  int i,j;

  for (i=0; i<N; i+=4)
  {
    __m128 value, numer, denom, sign;

    value = _mm_load_ps(&x[i]);
    __m128 valuex2 = _mm_mul_ps(value, value);
    numer = _mm_mul_ps(value, valuex2);
    long long int denominador = 6;
    int signo = -1;
    denom = _mm_set1_ps((float) denominador);
    sign = _mm_set1_ps ((float) signo);

    for (j=1; j<=terms; j++)
    {
      value = _mm_add_ps(value, _mm_mul_ps(sign, _mm_div_ps(numer, denom)));
      numer = _mm_mul_ps(numer, valuex2);
      denom = _mm_mul_ps(denom, _mm_set1_ps((float) ((2*j+2)*(2*j+3))));
      sign = _mm_mul_ps(sign, _mm_set1_ps ((float) -1));
    }

    _mm_store_ps(&y[i], value);
  }
}
```

Como podemos ver, el código es prácticamente el mismo pero traduciendo las instrucciones a las intrínsecas que se explican en la primera práctica.

N	1000	Defecto	100000000
sinx-v	0,00003	0,00972	7,13278

Si la comparamos con la versión original nos queda un el siguiente speedup.

N	1000	Defecto	100000000
speedUp	36	2,97427984	2,75385193

Como podemos apreciar, según el speedup podemos ver que la versión con intrínsecas es mucho mejor, especialmente para valores pequeños de N y que la mejora se empieza a estabilizar conforme subimos este valor.

sinx-p.c

```
void sinx(int N, int terms, float *x, float *y) {
  int i, j;
#pragma omp parallel for private (i, j)
  for (i = 0; i < N; i++) {
    float value = x[i];
    float numer = x[i] * x[i] * x[i];
    long long int denom = 6;
    int sign = -1;

    for (j = 1; j <= terms; j++) {
      value += sign * numer / denom;
      numer *= x[i] * x[i];
      denom *= (2 * j + 2) * (2 * j + 3);
      sign *= -1;
    }
    y[i] = value;
  }
}
```

Para ejecutar el programa utilizando OpenMP, simplemente añadimos la directiva para el bucle for.

N/Nº Threads	1000	Defecto	100000000
1	0,04697	0,02906	19,56506
4	0,00352	0,01922	6,96216
8	0,00784	0,01587	4,51616
16	0,01754	0,02505	3,34579
32	0,03522	0,02804	2,98013
64	0,0393	0,04643	2,35709

SpeedUp	1000	Defecto	100000000
1	0,0229934	0,99483827	1,00396421
4	0,30681818	1,50416233	2,82133993
8	0,1377551	1,82167612	4,34940746
16	0,06157355	1,15409182	5,87084665
32	0,0306644	1,0310271	6,59119569
64	0,02748092	0,62265776	8,3334196

Como podemos observar, empezamos a obtener rendimiento a partir de la versión por defecto debido al overhead, aunque si aumentamos el número de threads a partir de 8 va disminuyendo también por el overhead de OpenMP.

Al aumentar N vemos como la mejora es incluso mejor pero el rendimiento a la larga también bajaría si aumentamos el número de threads a uno mucho mayor, cosa improbable.

sinx-pv.c

```
void sinx (int N, int terms, float *x, float *y)
{
  int i,j;
  #pragma omp parallel for private (i,j)
  for (i=0; i<N; i+=4)
  {
    __m128 value, numer, denom, sign;

    value = _mm_load_ps(&x[i]);
    __m128 valuex2 = _mm_mul_ps(value, value);
    numer = _mm_mul_ps(value, valuex2);
    long long int denominador = 6;
    int signo = -1;
    denom = _mm_set1_ps((float) denominador);
    sign = _mm_set1_ps ((float) signo);

    for (j=1; j<=terms; j++)
    {
      value = _mm_add_ps(value, _mm_mul_ps(sign, _mm_div_ps(numer, denom)));
      numer = _mm_mul_ps(numer, valuex2);
      denom = _mm_mul_ps(denom, _mm_set1_ps((float) ((2*j+2)*(2*j+3))));
      sign = _mm_mul_ps(sign, _mm_set1_ps ((float) -1));
    }

    _mm_store_ps(&y[i], value);
  }
}
```

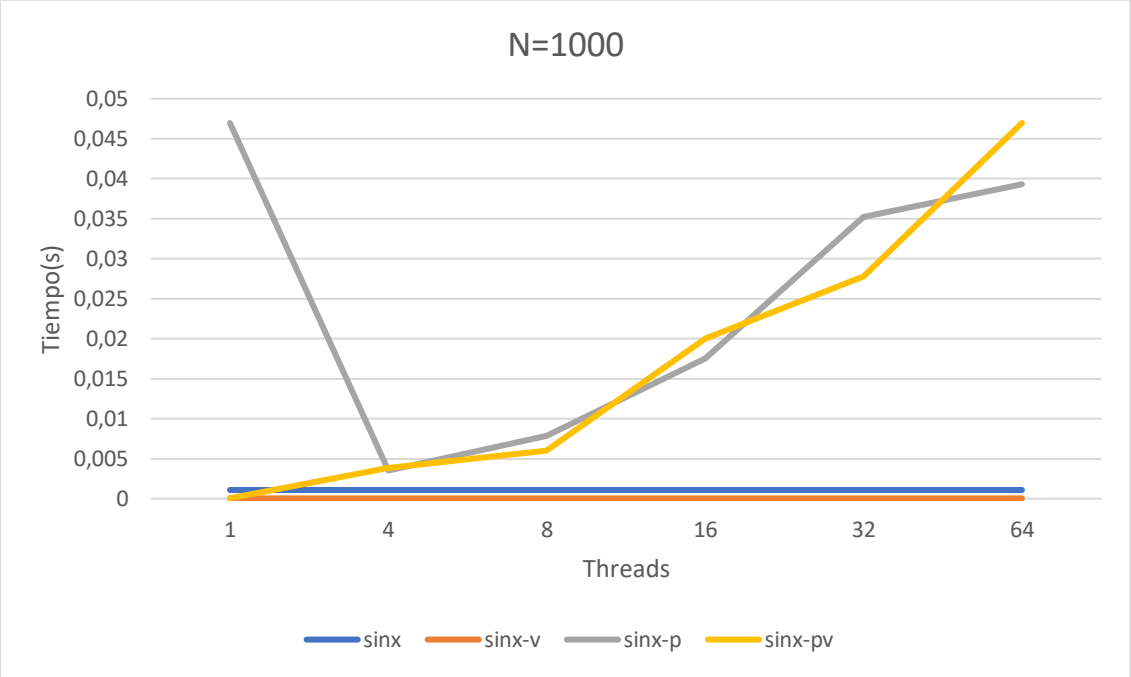
En esta versión utilizamos la misma que en la primera mejora con intrínsecas pero con la directiva de OpenMP utilizada en la segunda versión.

N/Nº Threads	1000	Defecto	100000000
1	0,00005	0,00969	7,23453
4	0,00385	0,01385	2,53786
8	0,00604	0,1329	1,66035
16	0,01999	0,01539	1,35561
32	0,02779	0,03513	1,19915
64	0,04697	0,05486	0,79609

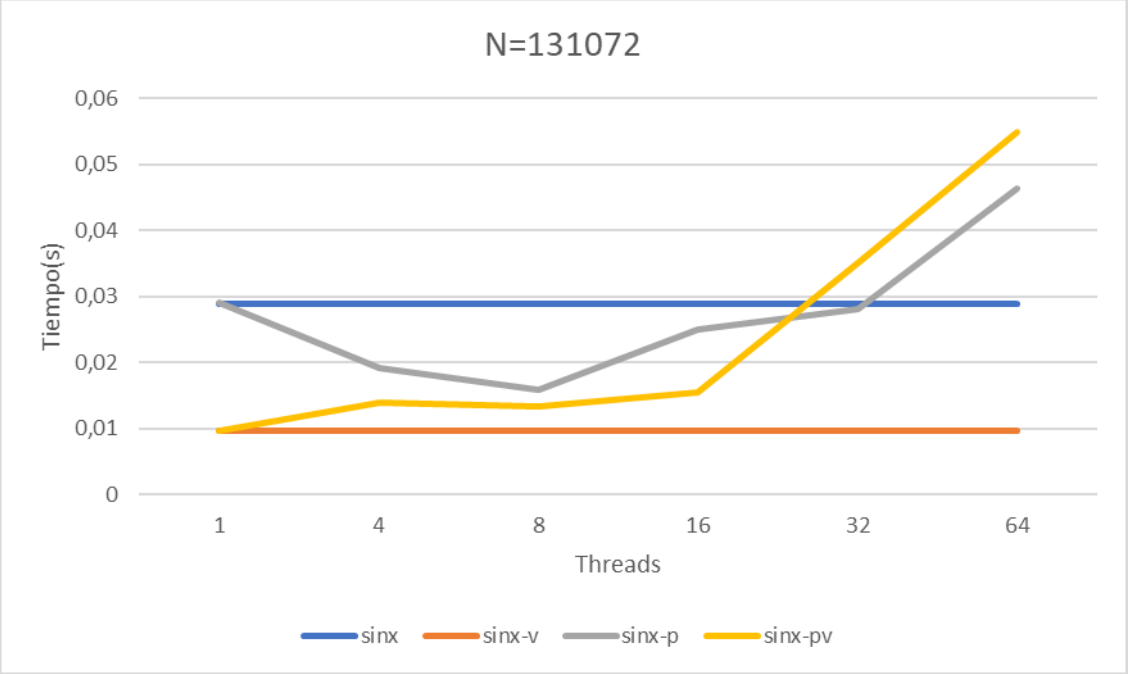
SpeedUp	1000	Defecto	100000000
1	21,6	2,98348813	2,7151204
4	0,28051948	2,08736462	7,73983592
8	0,17880795	0,21753198	11,8304093
16	0,05402701	1,87849253	14,4898754
32	0,0388629	0,82294335	16,3804528
64	0,0229934	0,52697776	24,6738685

En esta última versión, se puede apreciar como la mejora es mucho mejor al aumentar N, ya que combinamos paralelismo con intrínsecas y potenciamos el rendimiento.

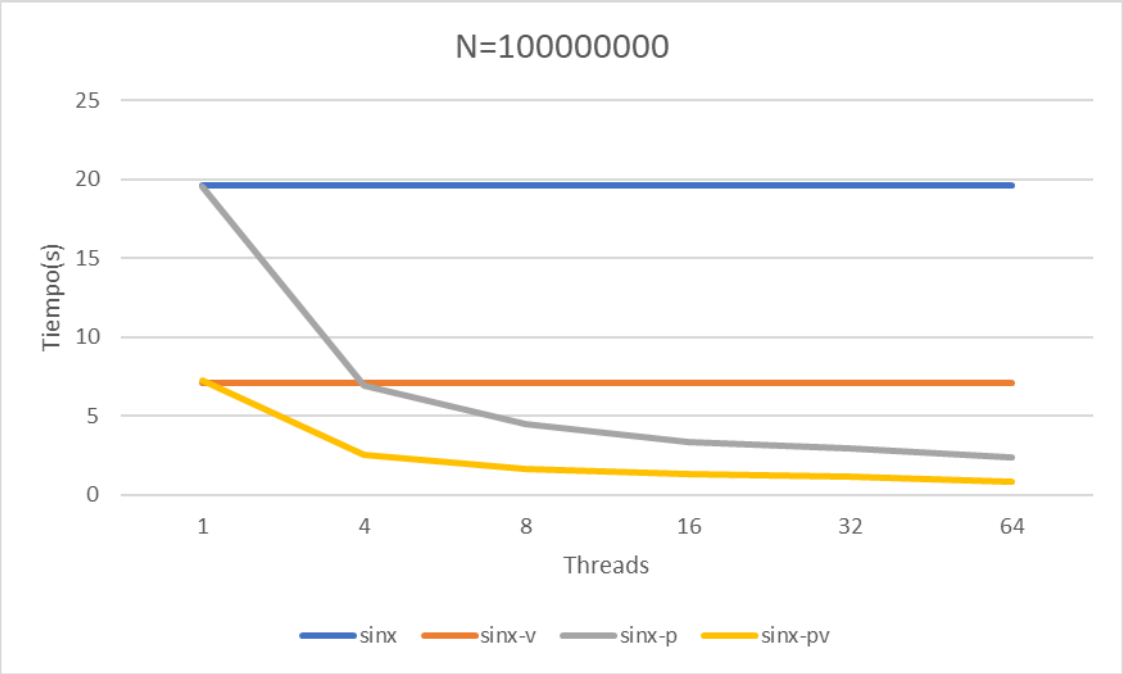
Con N=1000 obtenemos mejora solamente en la versión sin paralelizar debido al overhead de las directivas de openMP, por lo que para valores de N pequeños sería recomendable utilizar sólo la versión con intrínsecas.



Se puede observar que la versión que mejor escala es la versión con intrínsecas ya que el resto, debido a OpenMP, tienen un peor rendimiento al no merecer la pena paralelizarlo con un valor tan bajo de N.



Para el valor por defecto del programa, vemos como la versión con intrínsecas sigue dominando, aunque ya podemos ver como para un número de threads menor que 32 para sinx-p y entre 16 y 32 el rendimiento vuelve a bajar debido al overhead de OpenMP, por lo que nuestro número de N no es suficiente para ver que nos de una mejora positiva en rendimiento sin sacrificar el paralelismo.



Para valores altos de N, podemos ver como la escalabilidad de sinx-pv y sinx-p no tiene rival, ya que el overhead es mínimo debido a la cantidad de cómputo que el código requiere. Por lo que podríamos concluir que, para valores de N muy grandes, sí que merece la pena utilizar una versión paralelizada en la que podemos combinar tanto intrínsecas como directivas paralelas. También en la tabla de aceleración se observa como la versión de 64 threads del código sinx-pv es 24 veces más rápida que la versión original, y aún sigue escalando el rendimiento, por lo que tiene aún más margen te mejora. Si queremos hacer un algoritmo más genérico que sea más compatible con todo tipo de CPUs, la versión sinx-p nos puede servir para este cometido ya que no utiliza las intrínsecas propias del conjunto de instrucciones de la CPU.