ordenamos

queremos ordenar unos simples números dentro de un rango

- ¿Cómo procedemos?
- ¿Cuáles son los aspectos/problemas a tratar?
- ¿Cómo lo trasladamos a un entorno programable?

desafios

¿Con qué desafios nos enfrentamos?

- selección del algoritmo
  (RAE: Conjunto (?!) ordenado (?!) y finito (?!) de operaciones que permite hallar la solución de un problema.)
  ¡menos mal que en tercero ya sabemos que es un algoritmo!

ahora para tal algoritmo de forma concurrente:

- división del trabajo
- distribución de los datos
- sincronización necesaria
- comunicación de los datos entre participantes
- comunicación de los resultados

desafios

Y también con:

- medición de características
- depuración del programa
- (fiabilidad de los componentes)
- (fiabilidad de la comunicación)
- (detección de la terminación)

Programar aplicaciones concurrentes puede ser divertido y frustante a la vez :-)

medición de tiempo: la carrera de *San Martiño*

explicado en pizarra en clase

medición de tiempo de ejecución (ejemplo en Java)

```
class Timer {
  private long[] startTime;
  private long[] stopTime;

  Timer(int n) {
    startTime=new long[n];
    stopTime=new long[n];
  }
  public void Start(int i) {
    startTime[i]=System.nanoTime();
  }
  public void Stop(int i) {
    stopTime[i]=System.nanoTime();
  }
  public double Elapsed() {
    long minTime=startTime[0];
    for(int i=1; i<startTime.length; ++i)
      if(startTime[i] < minTime) minTime=startTime[i];
    long maxTime=stopTime[0];
    for(int i=1; i<stopTime.length; ++i)
      if(stopTime[i] > maxTime) maxTime=stopTime[i];
    return (maxTime-minTime)/1000000.0;
  }
}
```

resumen del uso de OpenMP

- OpenMP es una API abierta para la programación paralela bastante cómoda para usar (`www.openmp.org`) en C++ y Fortran.
- La versión actual es la 5.1 (noviembre 2020), usamos la que viene con el compilador (no hacemos nada ni sofisticado ni específico).
- OpenMP usa en C++ la técnica de las `#pragma` para introducir instrucciones OpenMP (que se ignoran cuando compilamos sin OpenMP).
- Se incluye el fichero `omp.h` y se compila (en g++) con la opción `-fopenmp`.

Bucle de cálculo (compara con prácticas)

```
#pragma omp parallel for
    for(unsigned i=0; i<nIter; ++i) {
      double d(std::tan(std::atan(
        std::tan(std::atan(
          std::tan(std::atan(
            std::tan(std::atan(
              std::tan(std::atan(123456789.123456789))
            ))
          ))
        ))
      ))));
      d=std::cbrt(d);
    }
```

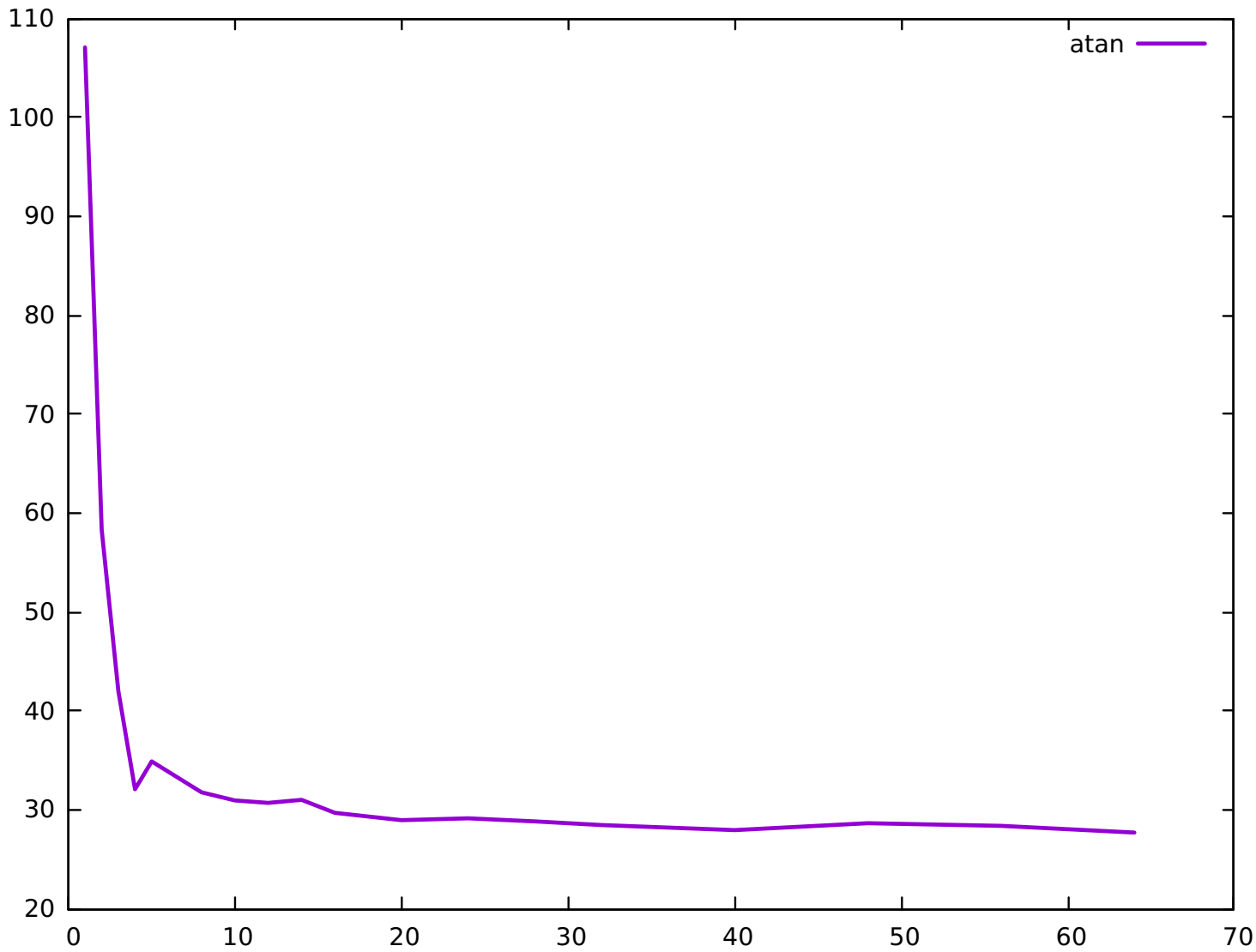
Optimizaciones del compilador

- si compilamos el código con optimización activado
- medimos un tiempo de ejecución casi 0 en caso secuencial
- dado que el compilador es capaz de detectar que no se realiza ninguna operación con la variable `d` fuera del bucle
- y elimina básicamente todo el bucle
  (*dead code elimination*, eliminación de código innecesario)

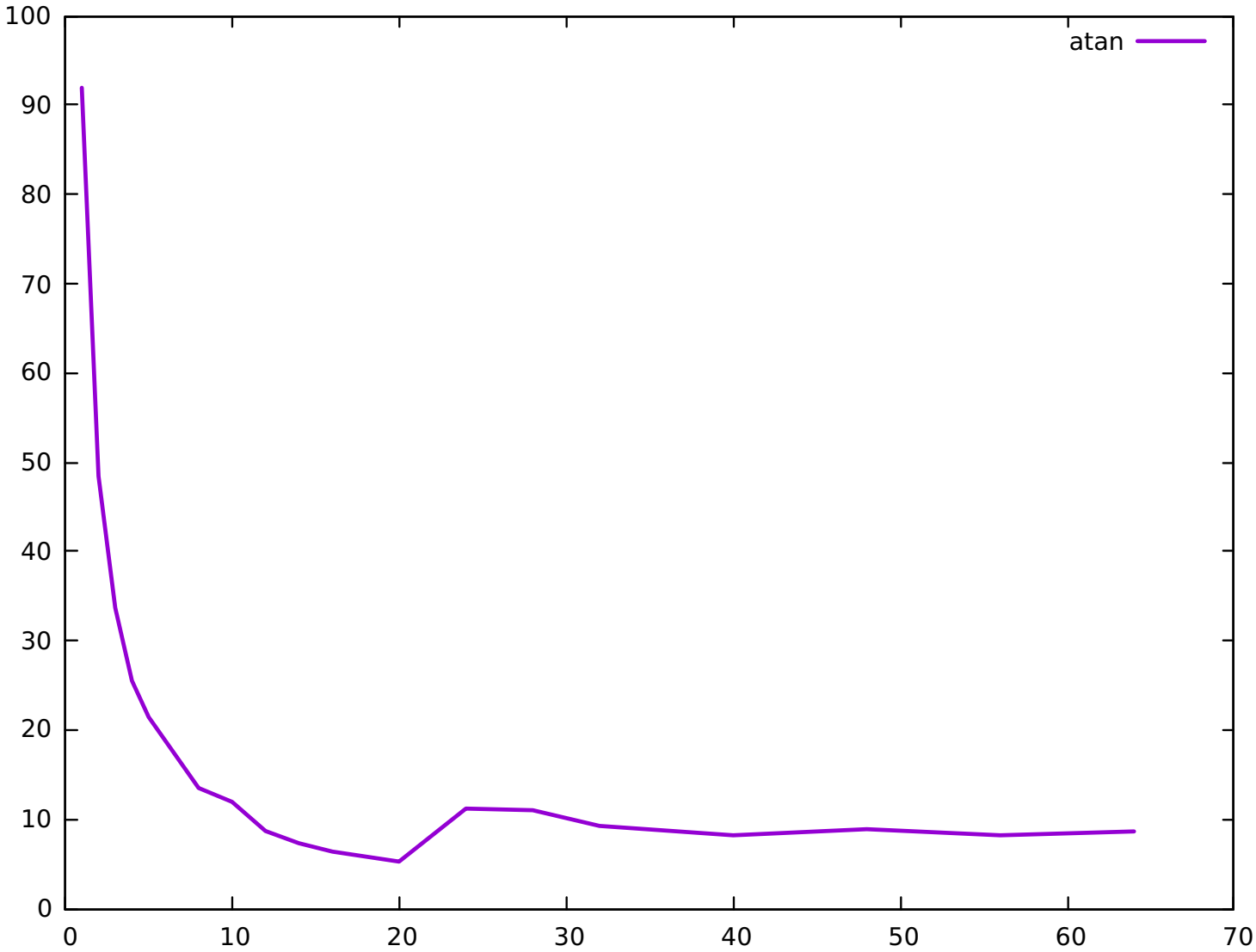
Bucle de cálculo con variable NO eliminable

```
#pragma omp parallel for \
    reduction(+:result)
    for(unsigned i=0; i<nIter; ++i) {
      double d(std::tan(std::atan(
        std::tan(std::atan(
          std::tan(std::atan(
            std::tan(std::atan(
              std::tan(std::atan(123456789.123456789))
            ))
          ))
        ))
      ))));
      result+=std::cbrt(d);
    }
```

tiempo de ejecución observado en i7



tiempo de ejecución observado en i9



ordenación

- La ordenación de datos es una tarea muy usada en muchas aplicaciones.
- ¿Cómo se ordena de forma concurrente (y eficiente)?
- En general no es nada trivial desarrollar algoritmos de ordenación paralelos, eficientes, escalables, etc.
- Miramos un caso especial (*counting sort*):
  ordenación de muchos datos (aquí enteros positivos) en cierto rango no demasiado grande, pero más grande que el número de procesos disponibles.

pizarra...

explicado el counting sort secuencial en pizarra en clase

pizarra...

explicado el counting sort paralelo en pizarra en clase

ordenación

- parallel counting sort (ordenación contando en paralelo)
  (mira: counting sort, radix sort, bucket sort)
- implementamos en C++ con OpenMP
- medimos el tiempo de ejecución en diferentes sistemas
- realizamos primero todo lo necesario alrededor,
  luego el propio algoritmo

comprobación de la ordenación

```
  // Checks whether data is sorted.
  // Note the standard library provides such a function,
  // but we do our own...
bool IsSorted(
  const std::vector<int>& data
) {
  const unsigned data_size(data.size());
  bool sorted(true);
  for(unsigned i=1; i<data_size; ++i) {
    if(data[i-1]>data[i]) sorted=false;
  }
  return sorted;
}
```

medición de tiempo de ejecución (ejemplo en C++)

```
  // Shortcut for complex timer type.
typedef std::chrono::high_resolution_clock::time_point
  TimePoint;

  // Starts our timer clock.
TimePoint StartClock(
) {
  return std::chrono::high_resolution_clock::now();
}

  // Stops our timer clock. Returns the elapsed time in [ms].
double StopClock(
  TimePoint start
) {
  const auto duration(
    std::chrono::duration_cast<std::chrono::milliseconds>(
      std::chrono::high_resolution_clock::now()-start
  ));
  return static_cast<double>(duration.count());
}
```

bucle principal de medición

```
  // Measure the runtime of our counting sort algorithm
  // in a measuring loop to smooth measuring fluctuations.
double time_countingsort(0.0);
for(unsigned i(0); i<measuringLoops; ++i) {
  W=V;
  start=StartClock();
  CountingSort(W);
  time_countingsort+=StopClock(start);
  if(!IsSorted(W)) std::cerr<<"Error: not sorted?\n";
}
```

lo que falta...

- paso de parámetros via línea de comando
- ayuda e información de copyright
- inicialización de los datos con valores aleatorios
- visualización de los datos (uso durante depuración)
- miramos estas funciones y,
  un poco más tarde, la implementación del propio algoritmo
  directamente en el fichero de código

las pragmas que usamos

- `#pragma omp parallel`
  comienza una región paralela, es decir, los hilos (el principal incluido) empiezan actuar en paralelo
- `#pragma omp for schedule(static)`
  realiza un bucle `for` que automaticamente se divide en trozos iguales para cada hilo participante
- `#pragma omp single`
  comienza una región secuencial dentro de una paralela, es decir, esta parte ejecuta solamente un hilo
- observa: tenemos sincronización implícita entre bucles, es decir, todos empiezan dentro de la región paralelo los bucles `for` al mismo tiempo (y en la región secuencial los que no actuan tampoco avanzan más allá)

comprobación de la ordenación en paralelo

```
  // Checks whether data is sorted.
  // Note the standard library provides such a function,
  // but we do our own...
bool IsSorted(
  const std::vector<int>& data
) {
  const unsigned data_size(data.size());
  bool sorted(true);
#pragma omp parallel \
  default(none) \
  shared(data) \
  shared(sorted)
  {
#pragma omp for schedule(static)
    for(unsigned i=1; i<data_size; ++i) {
      if(data[i-1]>data[i]) sorted=false;
    }
  }
  return sorted;
}
```

miramos el código en C++

mira el fichero `Sort.cpp` disponible

un breve análisis del tiempo de ejecución

- acordamos:
  n número de datos, v rango de valores, p número de procesos
- el algoritmo usa 8 bucles:
  1. copia y min/max-búsqueda (en paralelo): $O(n/p)$
  2. inicialización de tamaños (en paralelo): $O(v/p)$
  3. contar ocurrencias (en paralelo): $O(n/p)$
  4. cálculo de tamaños (en paralelo): $O(v/p*p) = O(v)$
  5. prefix-sum para tamaños (secuencial): $O(v)$
  6. inicialización de índices (en paralelo): $O(v/p)$
  7. prefix-sum para índices (en paralelo): $O(v/p*p) = O(v)$
  8. copia de datos (en paralelo): $O(n/p)$
- entonces tiempo de ejecución: $O(n/p+v)$
- observa: hay trabajo por hacer en la versión paralela (pasos 4,5,6, y 7) que no es necesario en la versión secuencial

un breve análisis del uso de memoria

- estructuras de datos principales:
  1. array de copia de datos: $O(n)$
  2. array para tamaños: $O(v)$
  3. arrays para contadores/índices: $O(v * p)$
- entonces uso de memoria: $O(n + v * p)$
- observa: la versión paralela necesita más memoria que la versión secuencial (aunque poco más en este caso)

modelación simple del tiempo de ejecución

$$T(p) \quad = \quad T_s + T_p/p$$

donde

- $T(p)$ es el tiempo de ejecución con p procesos
- T_s es el tiempo inherentemente secuencial
- T_p es el tiempo paralelizable

Gnuplot nos permite adecuar una curva basada en esta modelación (o en otras) de forma bastante simple.

herramientas de producción

- compilación para depuración:

```
# script that compiles with OpenMP (parallel version), DEBUG
g++ -std=gnu++11 -DDEBUG Sort.cpp -fopenmp -o cdi_sort_g
```

- compilación para pruebas:

```
# script that compiles with OpenMP (parallel sorting), OPTIMIZED
g++ -std=gnu++11 -O3 -DNDEBUG Sort.cpp -fopenmp -o cdi_sort
```

- visualización de resultados de medición con gnuplot:

```
am0(x)=a0+a1/x
am1(x)=a2+a3/x

a0=200
a1=200
fit am0(x) "sort_p_hpformella.gdt" u 2:3 via a0,a1
a2=100
a3=3000
fit am1(x) "sort_p_hpformella.gdt" u 2:4 via a2,a3

set key right top
plot \
  "sort_p_hpformella.gdt" u 2:4 w lp lw 2 t "std", \
  am1(x) t sprintf("%.0f+%.0f/p",a2,a3), \
  "sort_p_hpformella.gdt" u 2:3 w lp lw 2 t "count", \
  am0(x) t sprintf("%.0f+%.0f/p",a0,a1)
```

- todos estos comandos agrupamos en scripts correspondientes

herramientas de producción

- script para ejecución con parámetros en línea de comando: (aumentando tamaño de vector, número de procesos fijos)

```
# script that runs test instance of parallel sort
# fixed thread number, increasing problem size
./cdi_sort -i 0 -x 499 -n 10000 -p 20 > sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 40000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 400000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 1000000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 4000000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 10000000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 40000000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 20 >> sort_n_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 400000000 -p 20 >> sort_n_`hostname`.gdt
```

herramientas de producción

- script para ejecución con parámetros en línea de comando: (tamaño de vector fijo, aumentando número de procesos)

```
# script that runs test instance of parallel sort
# increasing thread number, fixed problem size
./cdi_sort -i 0 -x 499 -n 100000000 -p 1 > sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 2 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 3 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 4 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 5 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 8 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 10 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 16 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 20 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 32 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 40 >> sort_p_`hostname`.gdt
./cdi_sort -i 0 -x 499 -n 100000000 -p 64 >> sort_p_`hostname`.gdt
```

resultados de medición

- datos tal como el programa escribe el fichero (ejemplo):
  (aumentando tamaño de vector, número de procesos fijos)

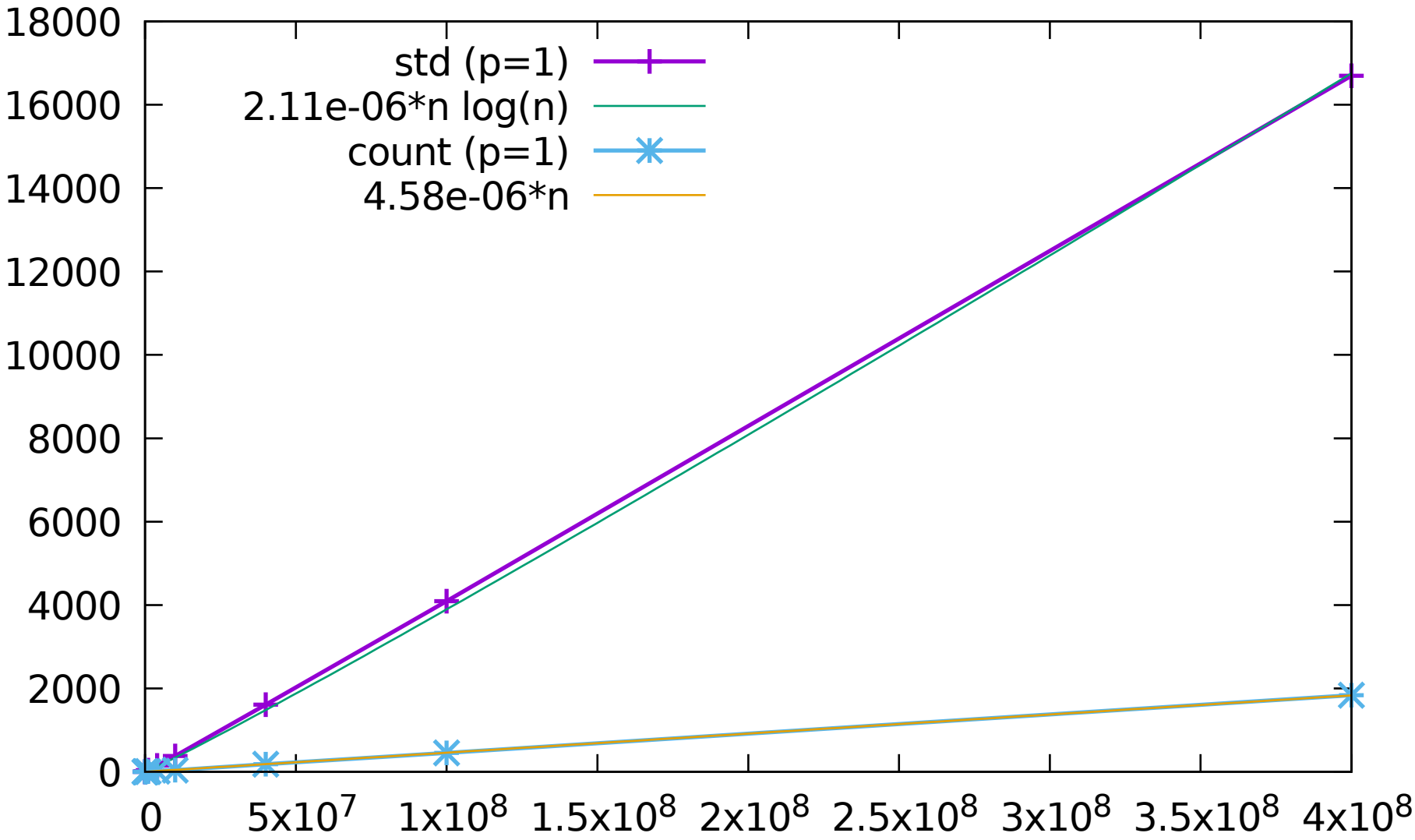
```
10000 20 0 0
40000 20 0.4 1
100000 20 0.6 2
400000 20 1.2 7.2
1000000 20 2.8 17.8
4000000 20 10 71.2
10000000 20 32.6 185.2
40000000 20 122.8 773.4
100000000 20 304 1975.4
400000000 20 1241.8 8259.4
```

resultados de medición

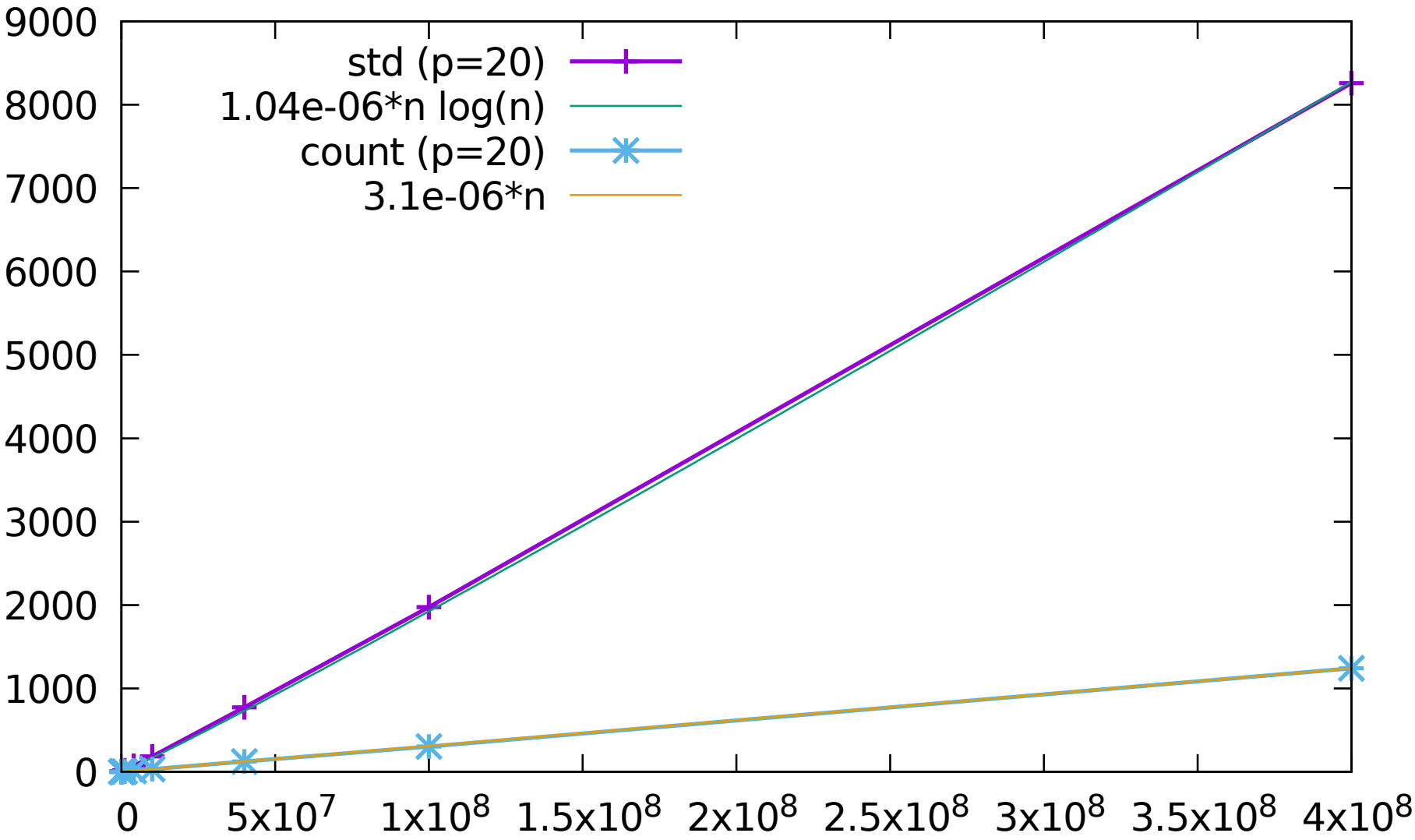
- datos tal como el programa escribe el fichero (ejemplo):
  (tamaño de vector fijo, aumentando número de procesos)

```
100000000 1 448.6 3961
100000000 2 399.8 2425.6
100000000 3 348 2131.6
100000000 4 354.2 1823.6
100000000 5 314.2 1986.4
100000000 8 318.2 1941.2
100000000 10 306.8 1982
100000000 16 301.2 1923.6
100000000 20 302.4 1959.6
100000000 32 301.6 1943.4
100000000 40 307.8 1985.2
100000000 64 300 1982
```

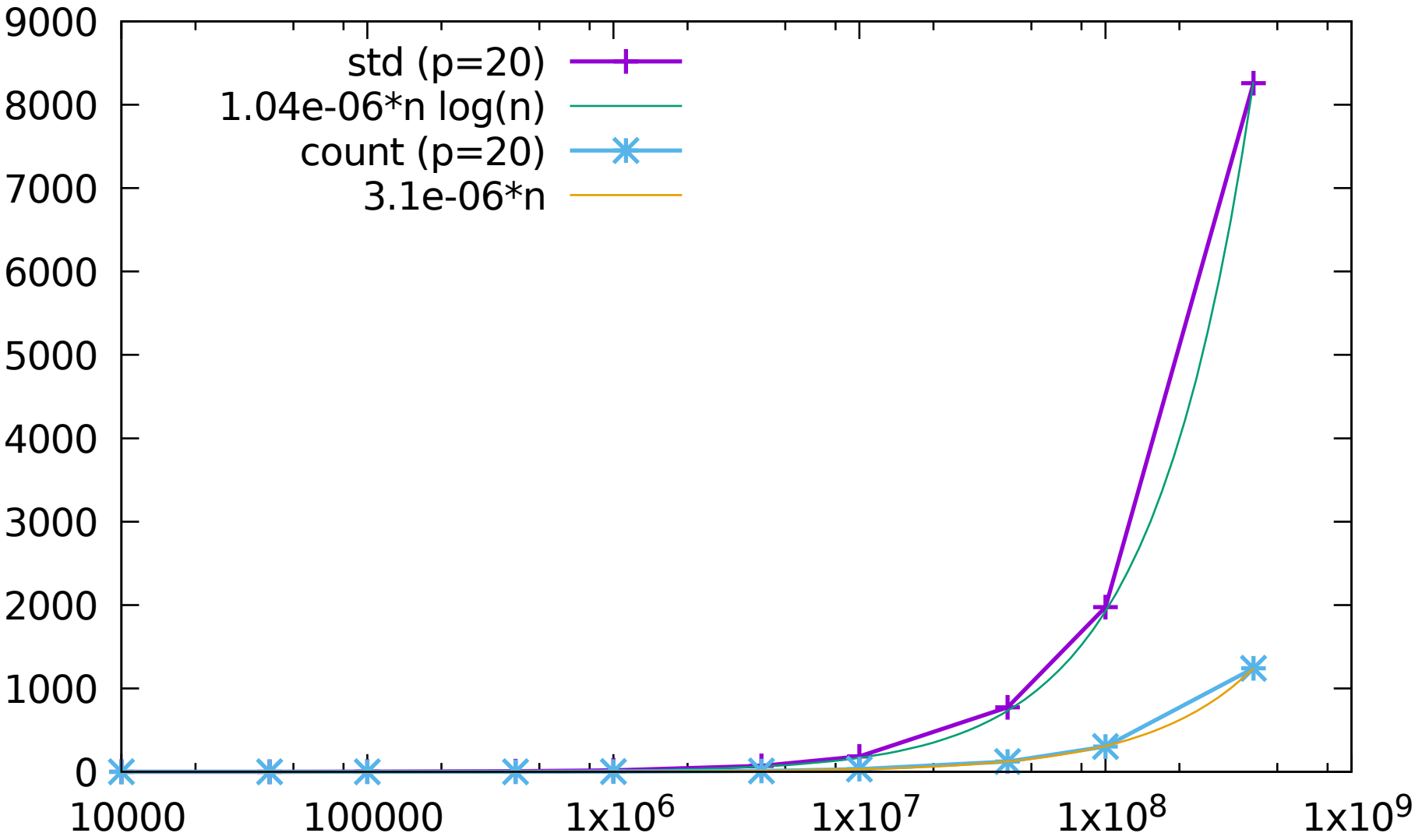
portátil i7: aumentando data, 1 proceso



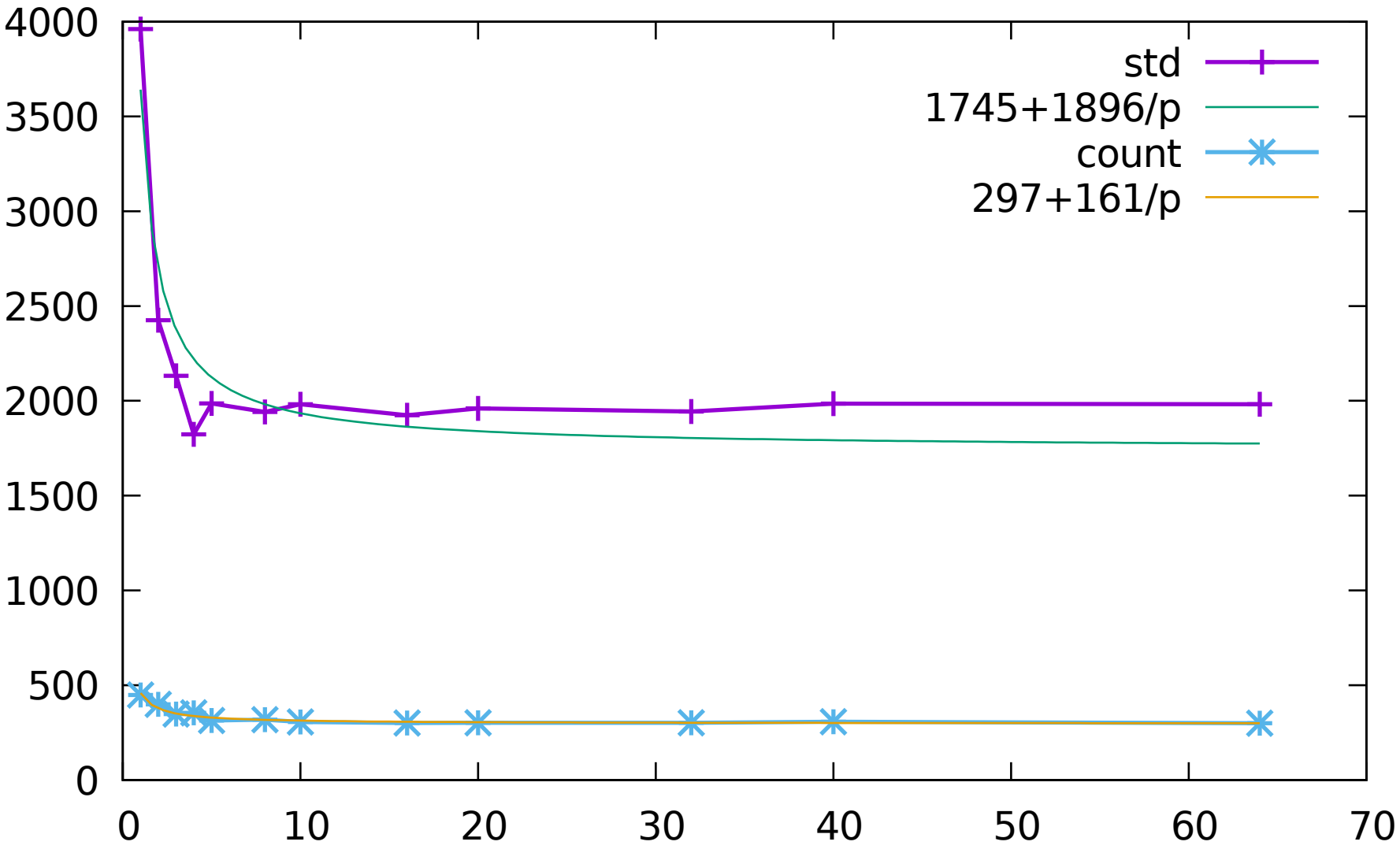
portátil i7: aumentando data, 20 proceso



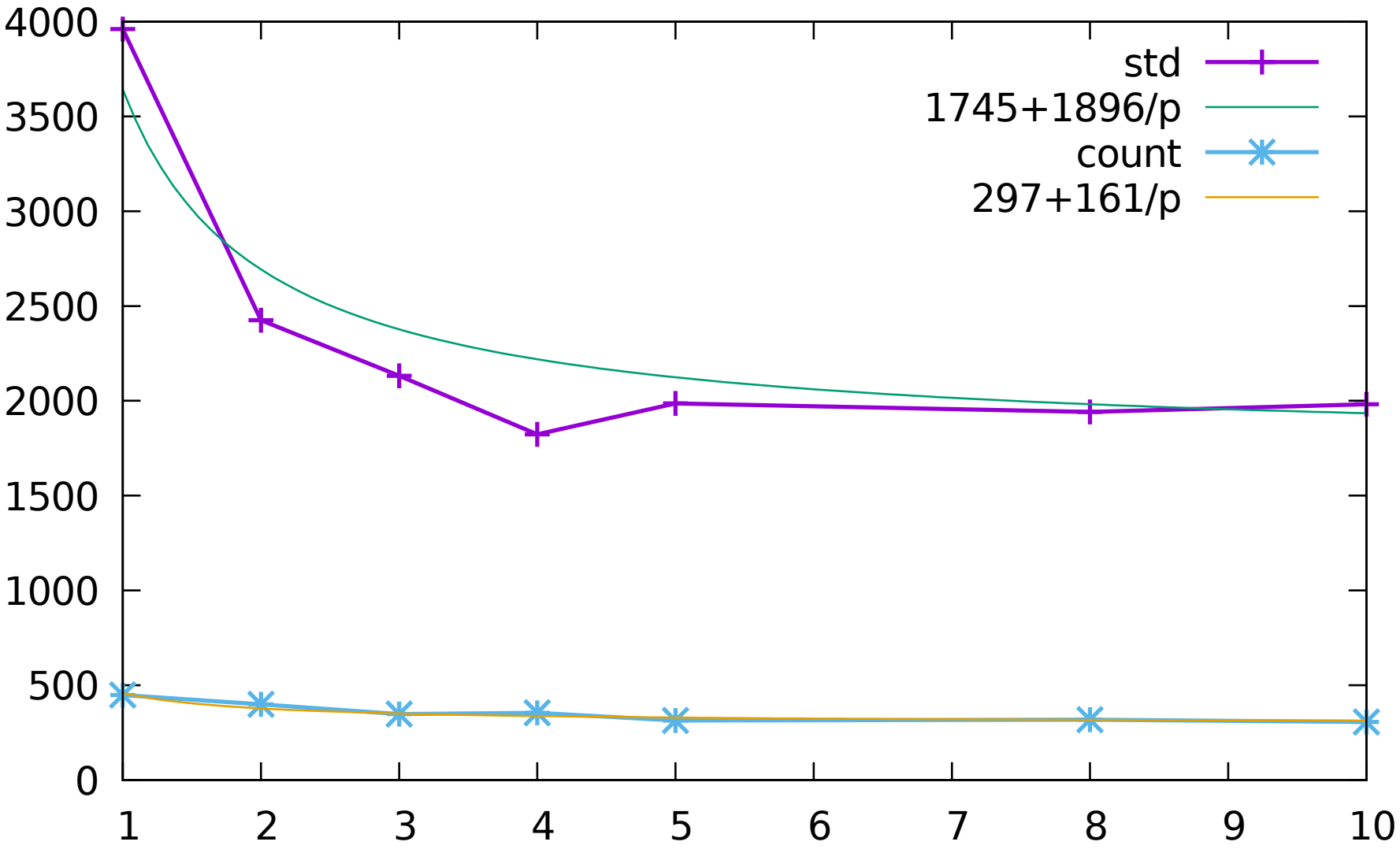
portátil i7: aumentando data, 20 proceso



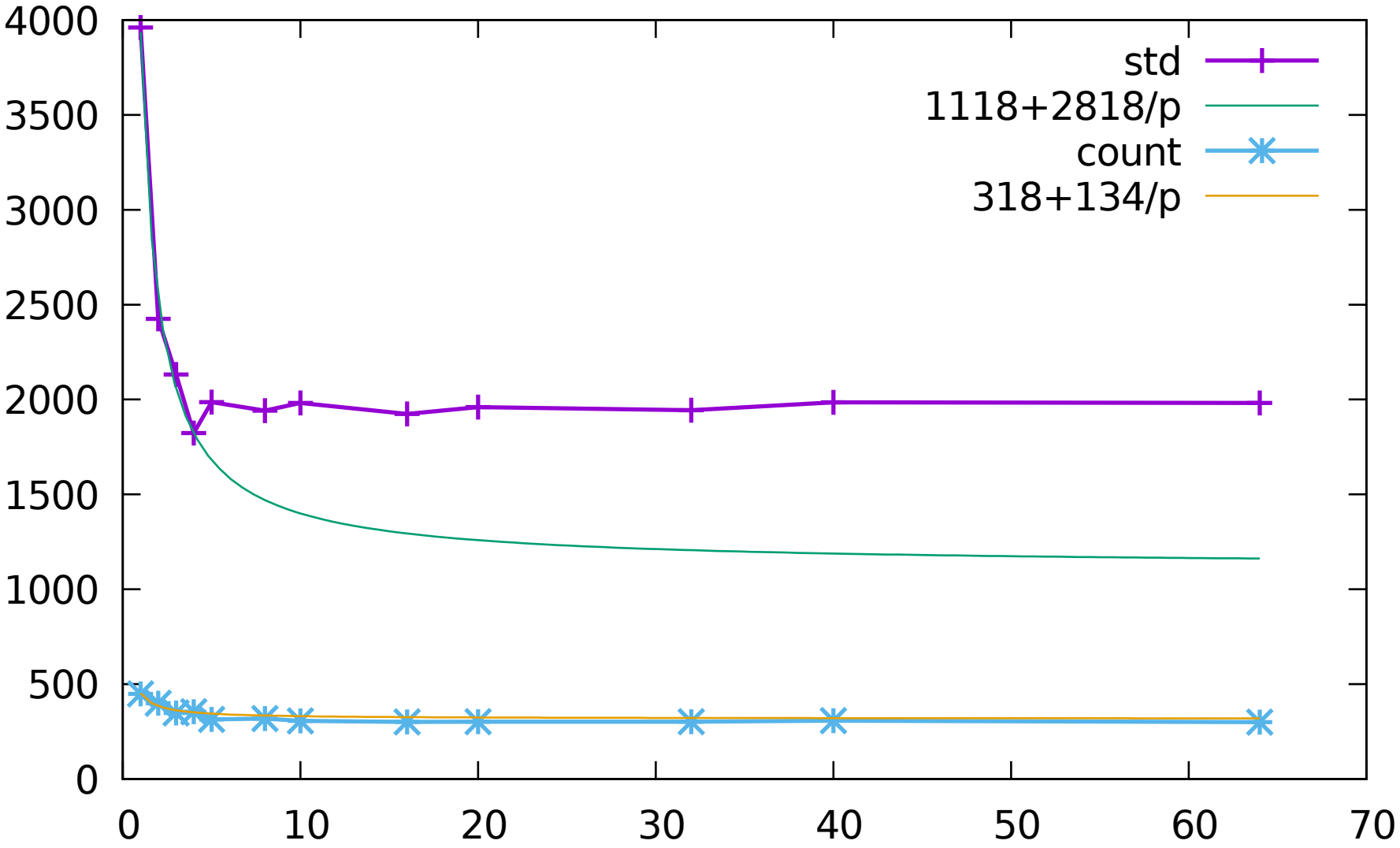
portátil i7: data fijo, aumentando procesos



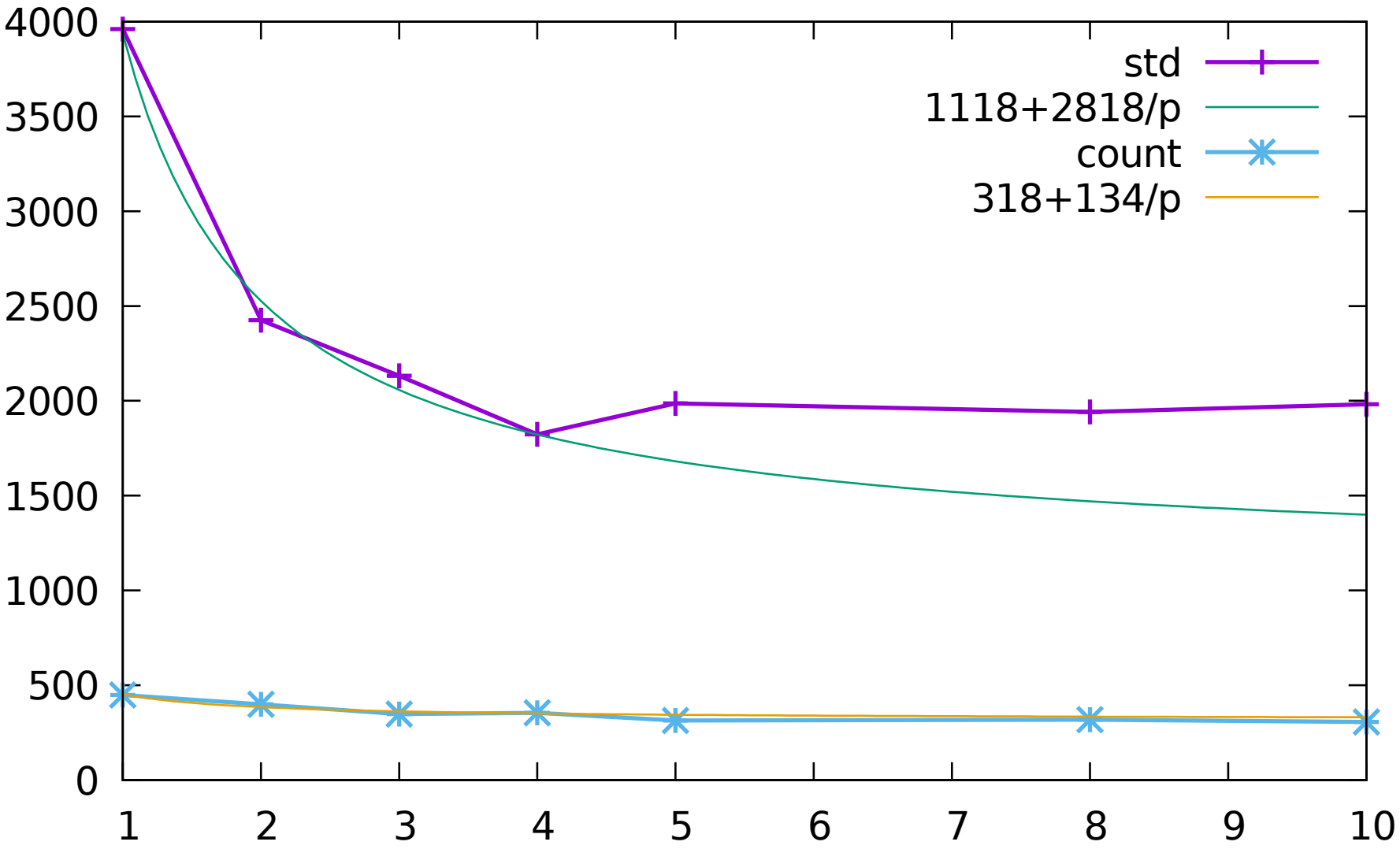
portátil i7: data fijo, aumentando procesos (ampliación)



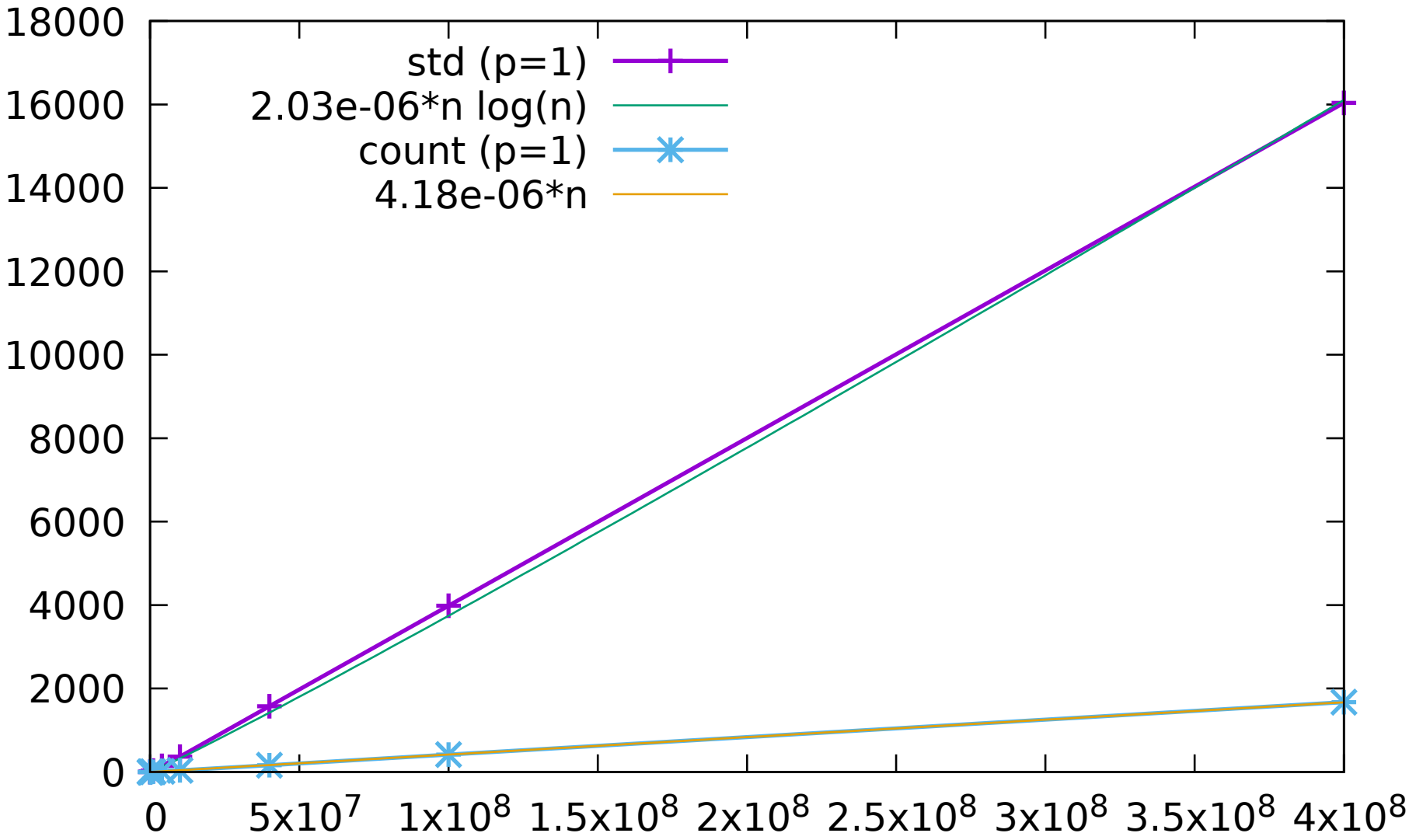
portátil i7: data fijo, aumentando procesos



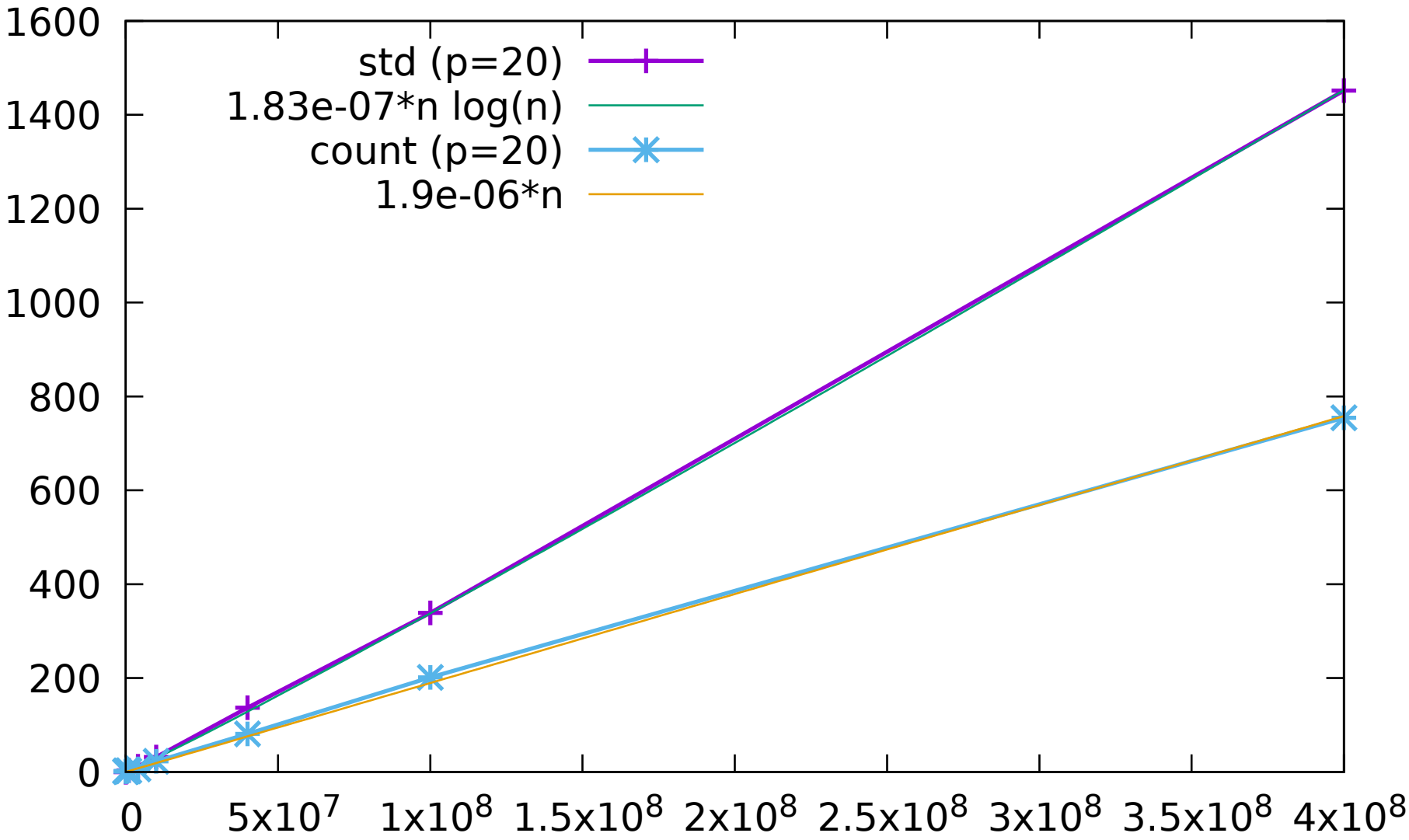
portátil i7: data fijo, aumentando procesos (ampliación)



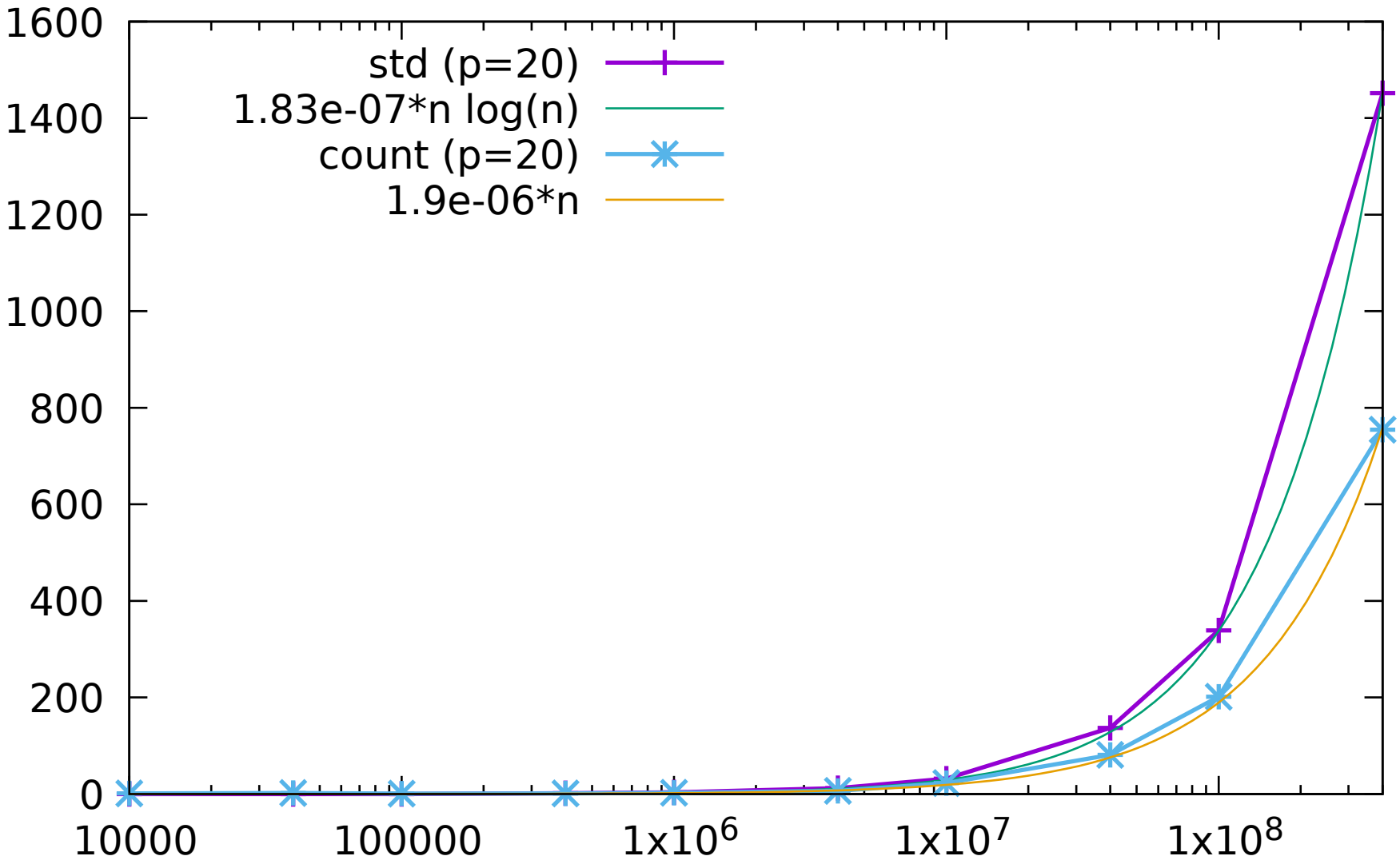
estación i9: aumentando data, 1 proceso



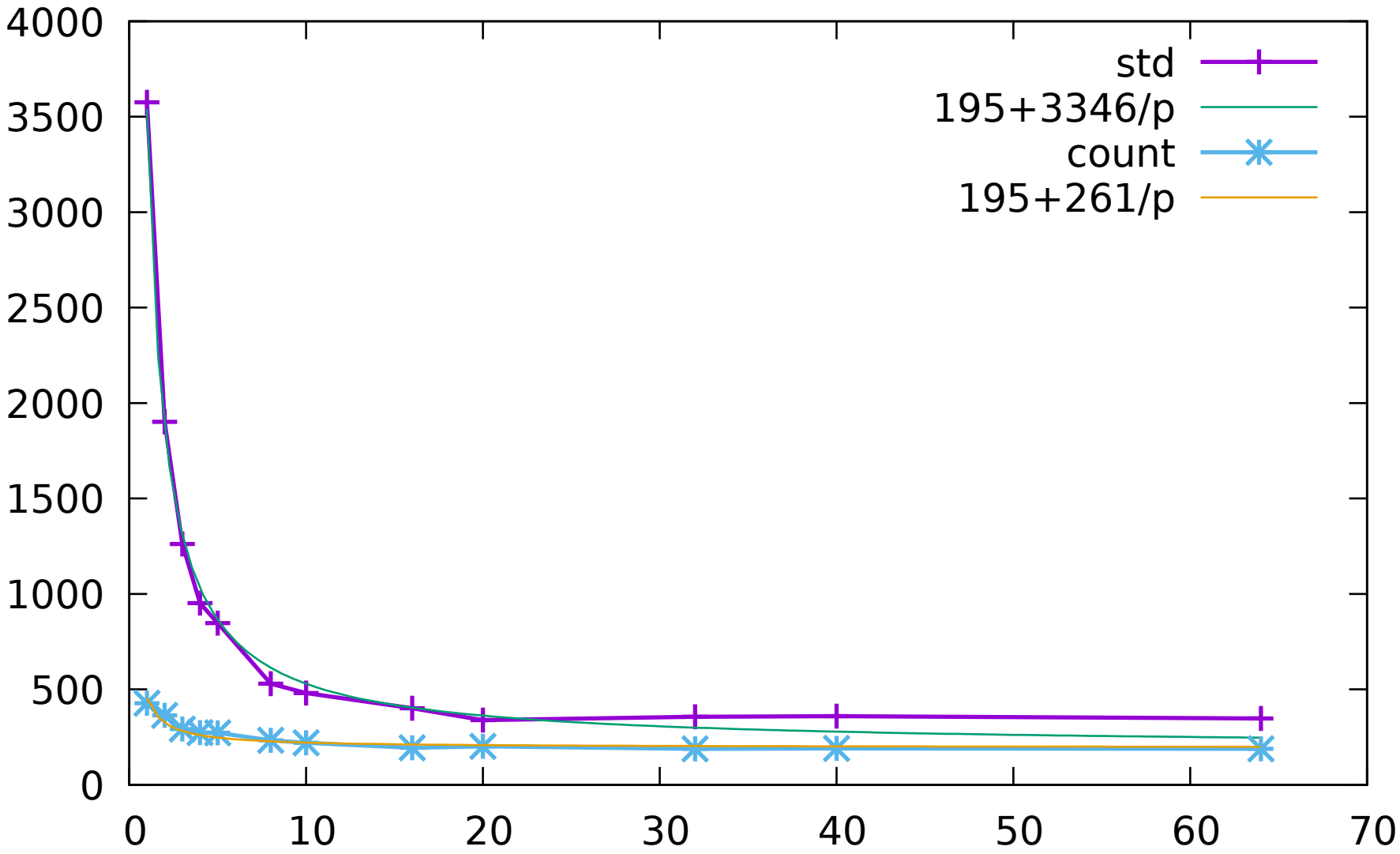
estación i9: aumentando data, 20 proceso



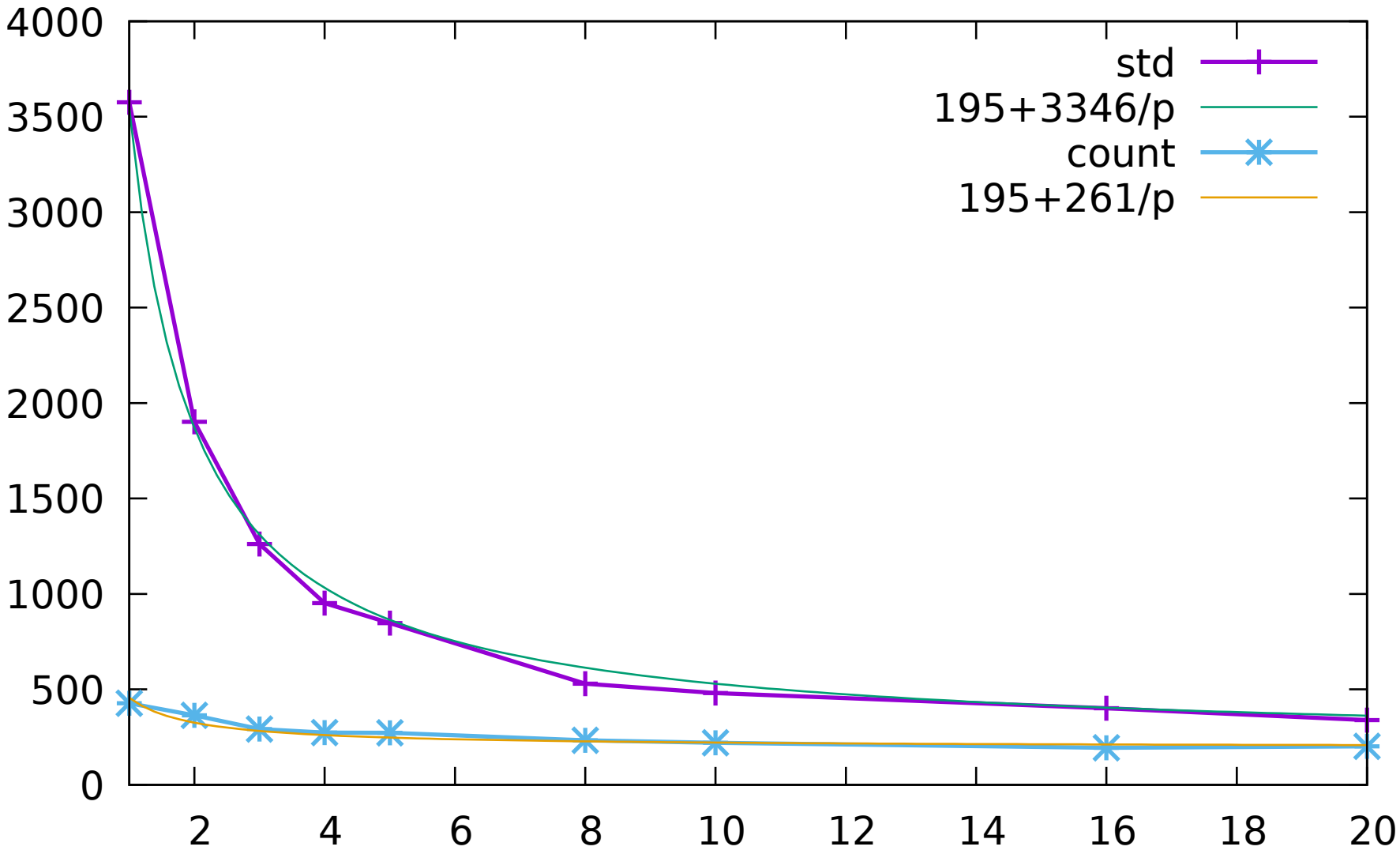
estación i9: aumentando data, 20 proceso



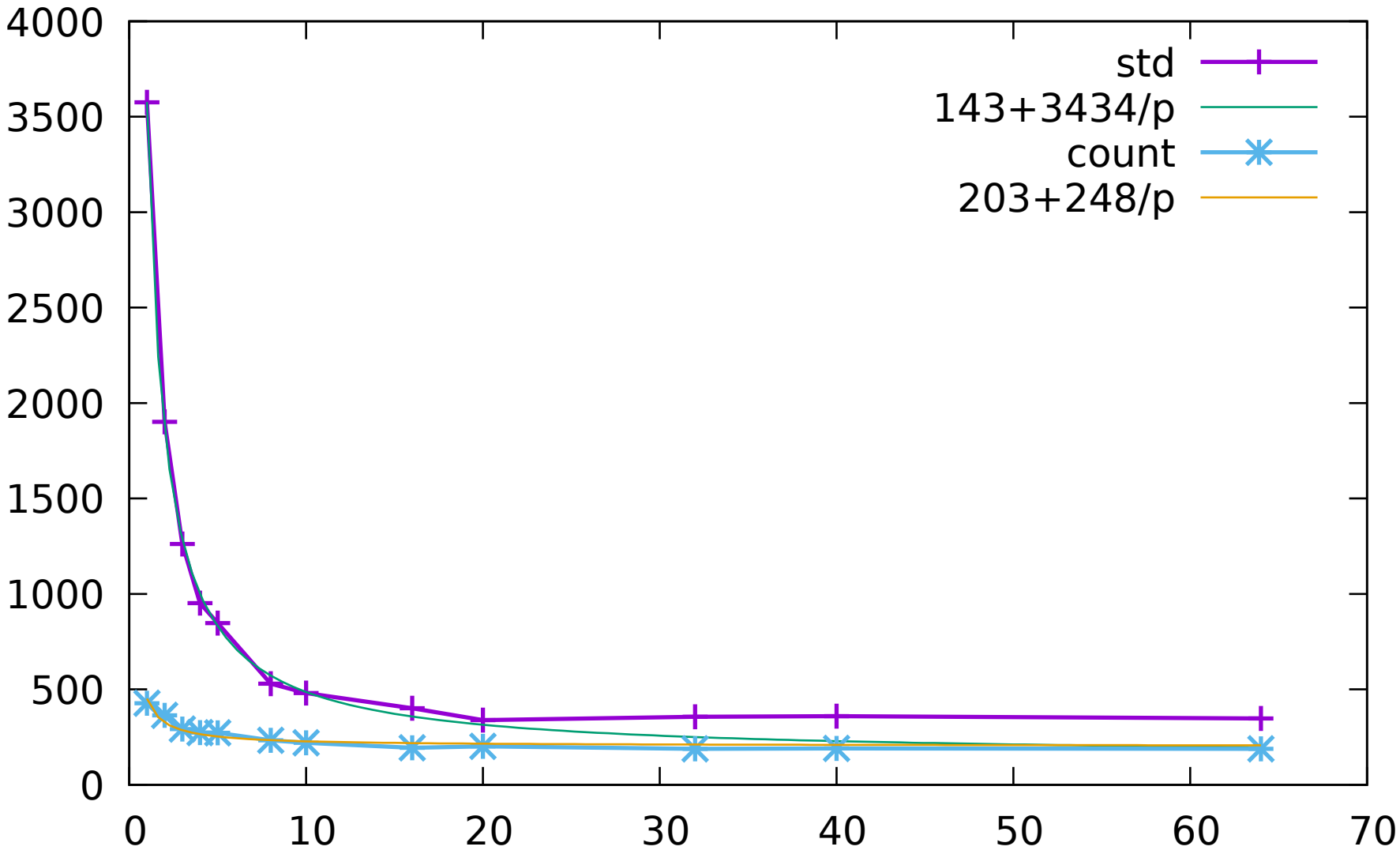
estación i9: data fijo, aumentando procesos



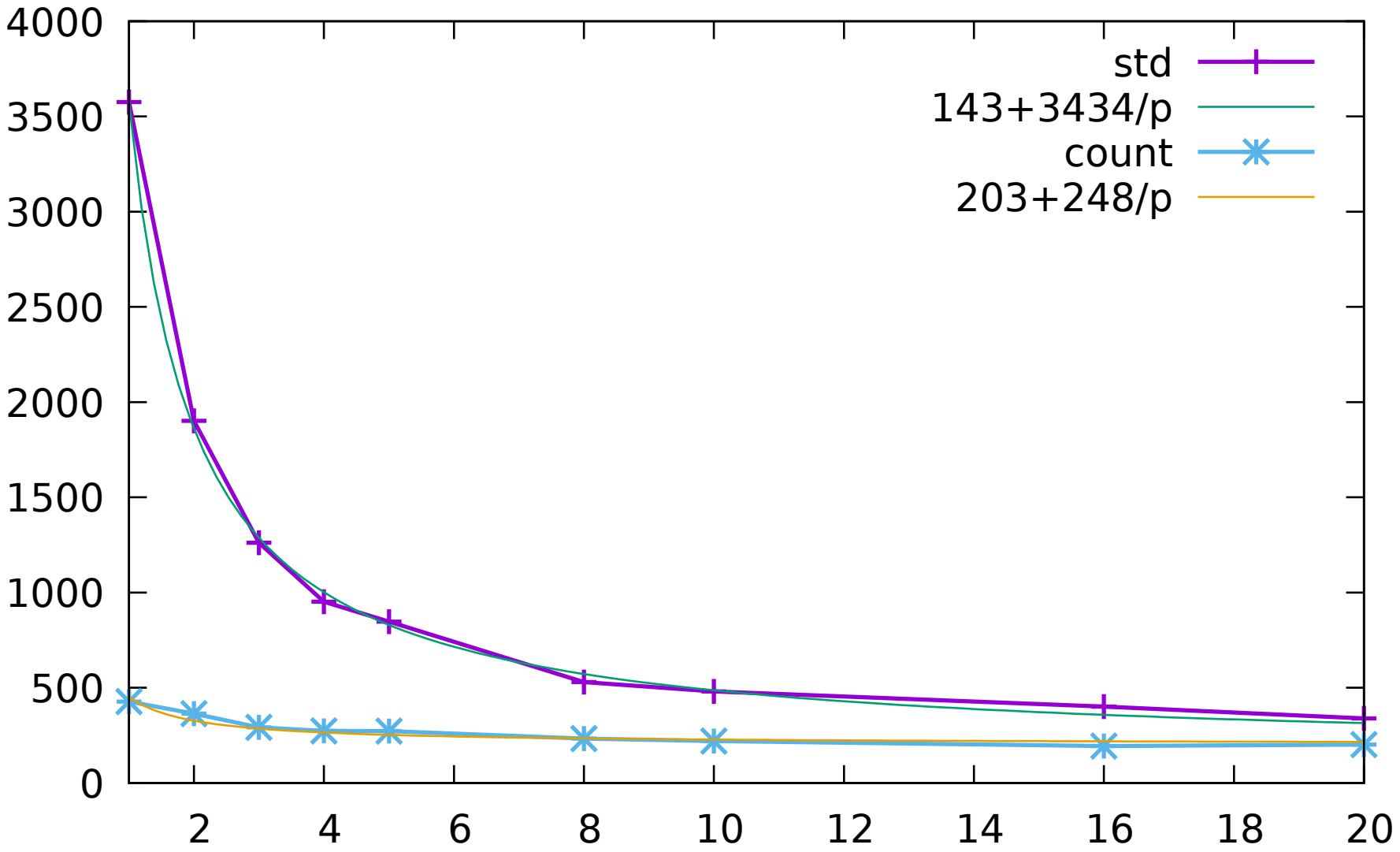
estación i9: data fijo, aumentando procesos (ampliación)



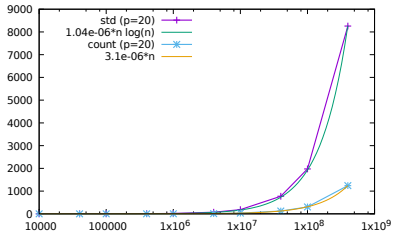
estación i9: data fijo, aumentando procesos

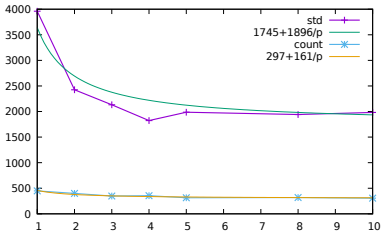


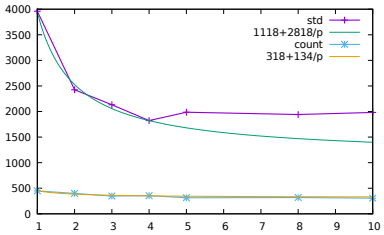
estación i9: data fijo, aumentando procesos (ampliación)

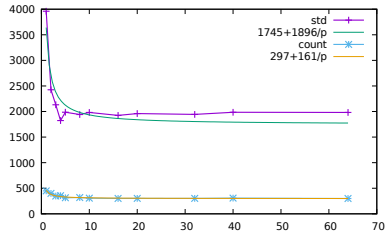


interpretamos lo que vemos en las gráficas del i7



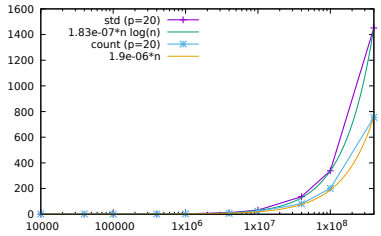


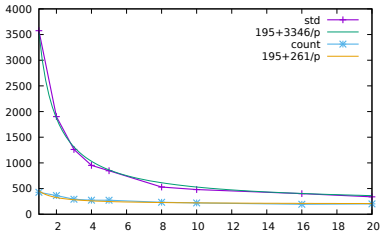


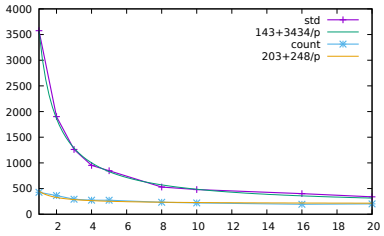


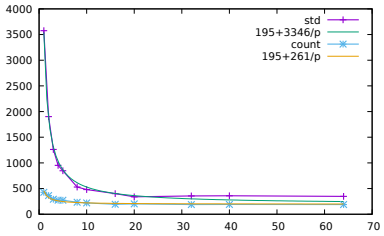
- el algoritmo de counting sort es más rápido que el algoritmo de la biblioteca estándar que implementa una ordenación en tiempo $O(n/p * log(n) + ...)$
- el mínimo de la curva está en 4, eso sugiere que hay 4 procesadores en el sistema

interpretamos lo que vemos en las gráficas del i9









- el algoritmo de *counting sort* aquí es más rápido que el algoritmo de la biblioteca estándar
- pero parece que no se gane tanto
- el mínimo de la curva está en 20, eso sugiere que hay 20 procesadores en el sistema

factor de acceleración (*speedup*)

- sea $T(1)$ el tiempo de ejecución del programa secuencial
- sea $T(p)$ el tiempo de ejecución del programa paralelo/concurrente
- la fracción $s_p = T(1)/T(p)$ se llama factor de acceleración o *speedup*

Notas:

- para medir $T(1)$ se debe usar el mejor programa secuencial conocido para caracterizar el algoritmo paralelo
- si se usa para $T(1)$ el programa paralelo con un solo proceso, solamente se caracteriza la paralelizabilidad del algoritmo
- el tamaño del problema por resolver (n, en nuestro caso) influye en la acceleración alcanzable
- un factor de acceleración de p es lo más deseable, es decir, con p procesos el cálculo se realiza p-veces más rápido

Ley de Amdahl

- con el modelado que usamos, tenemos $T(1) = T_s + T_p$
- podemos usar una fracción f para separar secuencial/paralelo:

$$T(1) = f \cdot T(1) + (1-f) \cdot T(1)$$

- y escribir la accelacion:

$$s_p = \frac{T(1)}{T(p)} = \frac{f \cdot T(1) + (1-f) \cdot T(1)}{f \cdot T(1) + (1-f) \cdot T(1)/p}$$

- que simplifica a:

$$s_p = \frac{1}{f + (1-f)/p}$$

- que la ley de Amdahl, con $p \longrightarrow \infty$:

$$s_\infty = \frac{1}{f}$$

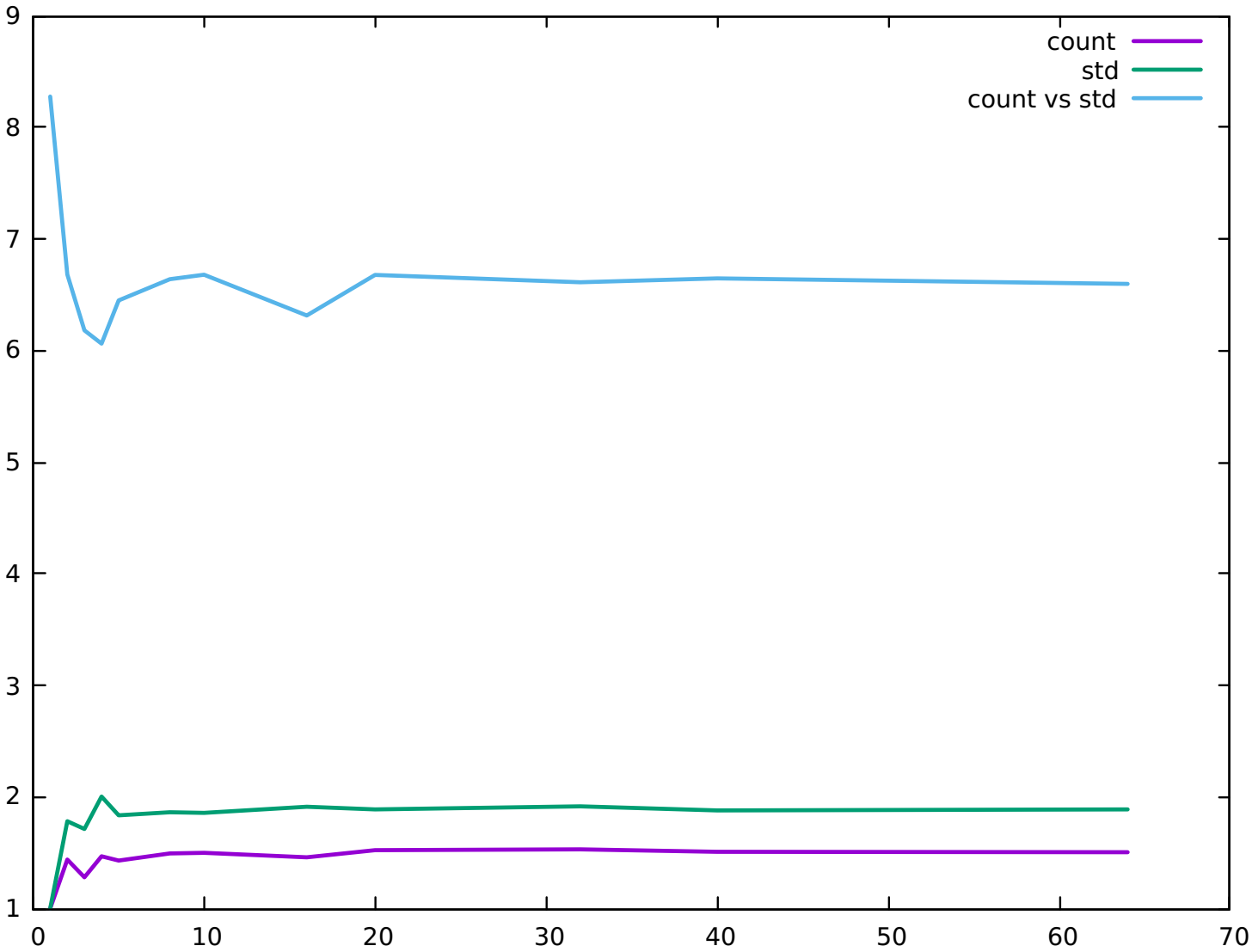
factor de ganancia entre sistemas

- sea T_{S1} el tiempo de ejecución en un sistema S1
- sea T_{S2} el tiempo de ejecución en un sistema S2 (más potente)
- la fracción $g = T_{S2}/T_{S1}$ se llama factor de acceleración entre sistemas (o factor de ganancia)

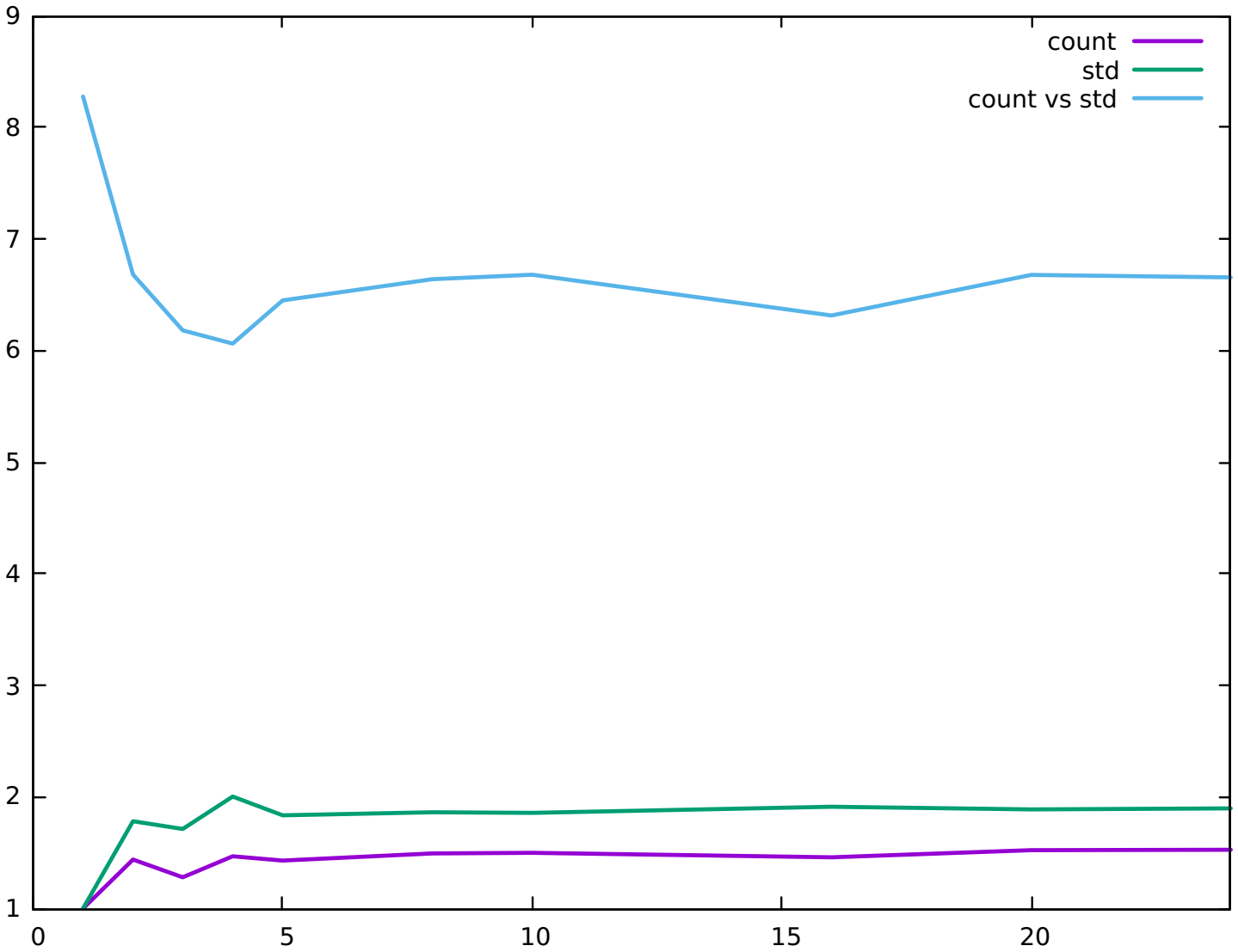
Notas:

- para medir T_{S1} y T_{S2} se debe fijar el tamaño del problema por resolver (n, en nuestro caso)
- se debe usar el mejor algoritmo para cada sistema para caracterizar la ganancia de un sistema respecto al otro
- si se usa para ambos sistemas el mismo algoritmo, solamente se caracteriza la ganancia para un algoritmo en concreto

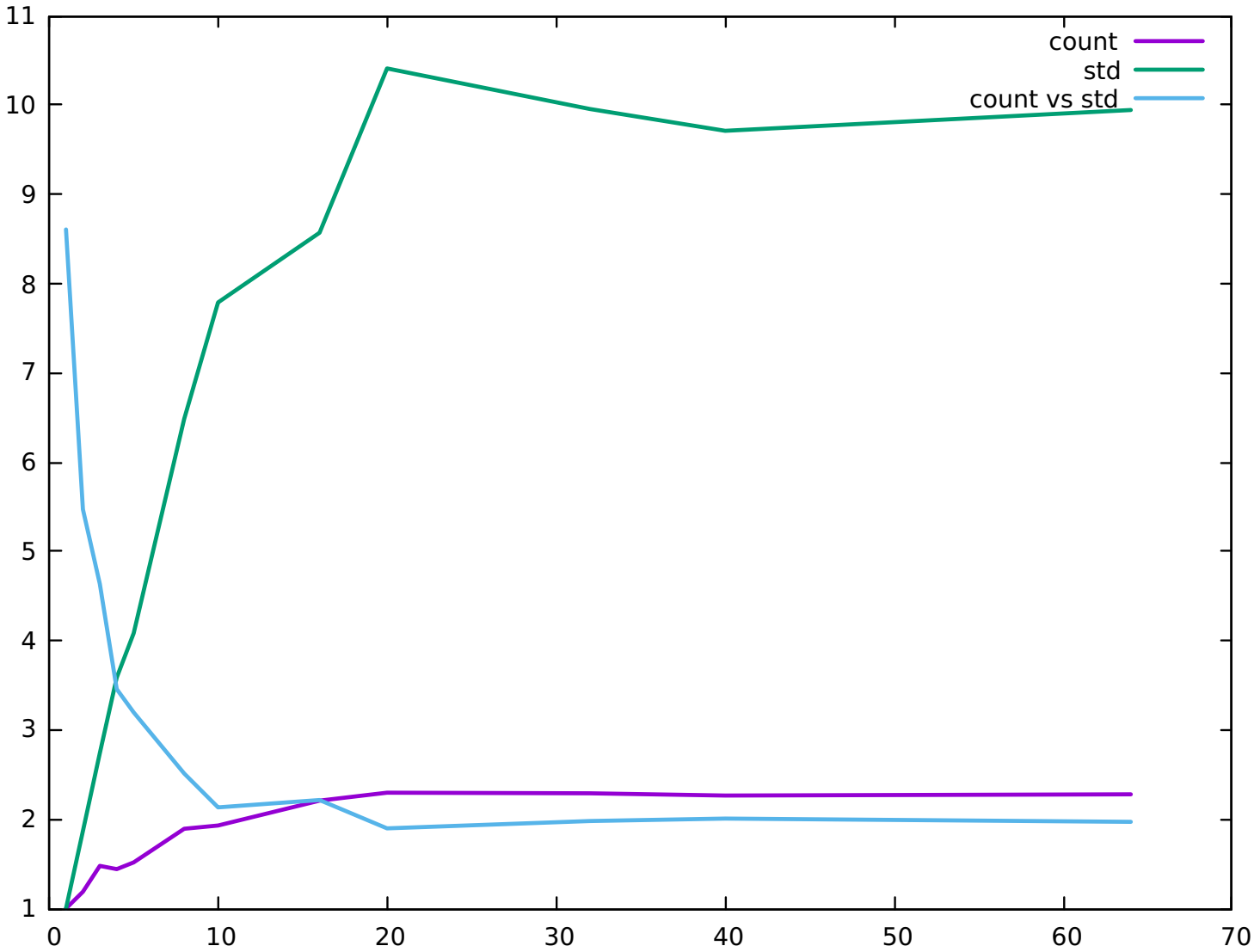
análisis de aceleración (*speedup*) (i7)



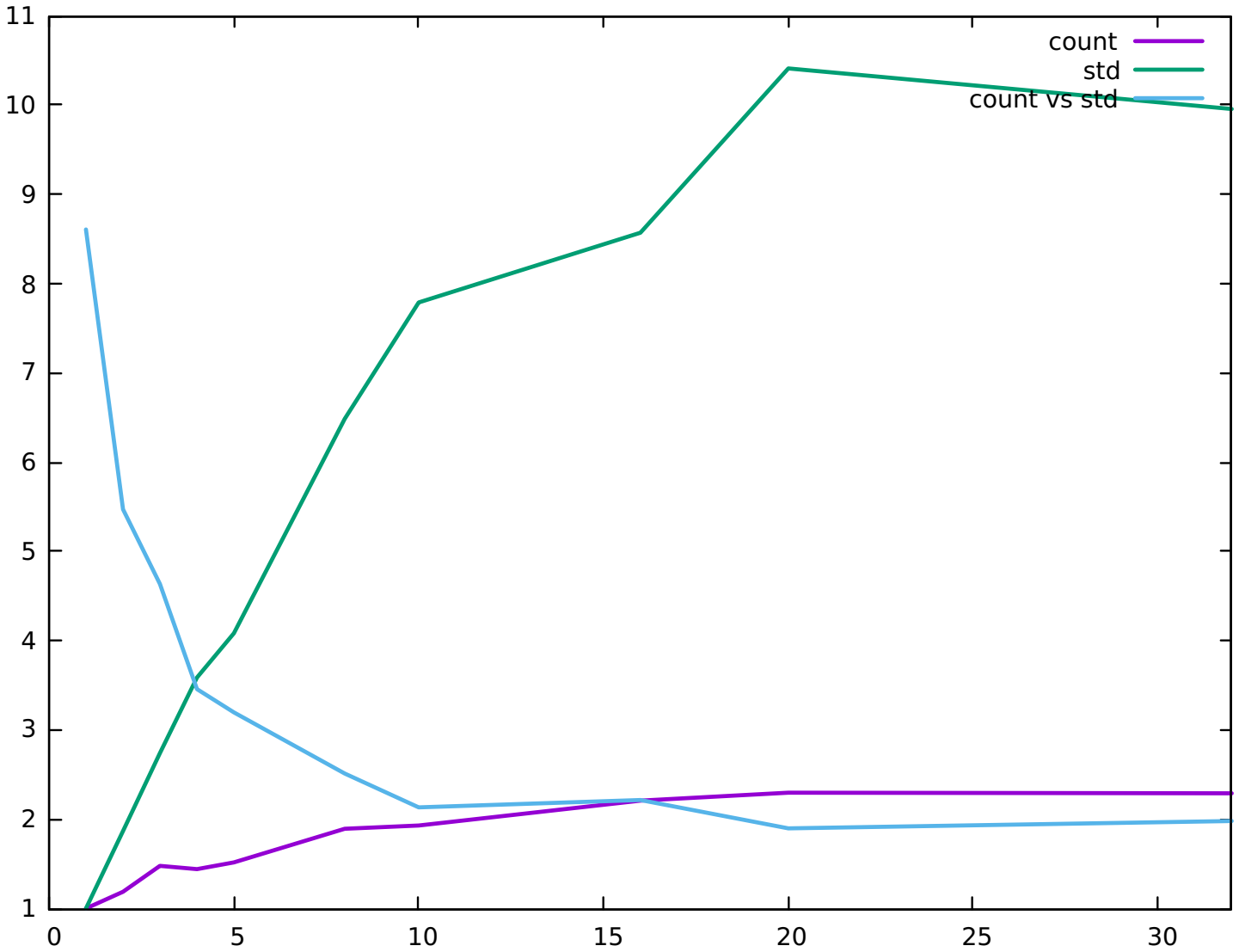
análisis de aceleración (*speedup*) (i7) ampliación



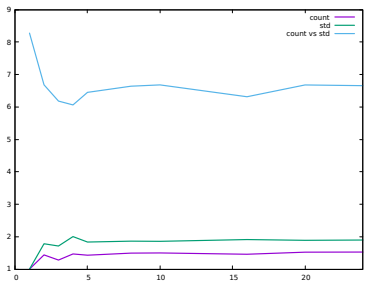
análisis de acceleración (*speedup*) (i9)



análisis de aceleración (*speedup*) (i9) ampliación

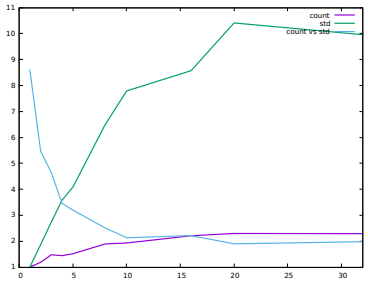


interpretamos lo que vemos en la gráfica del i7



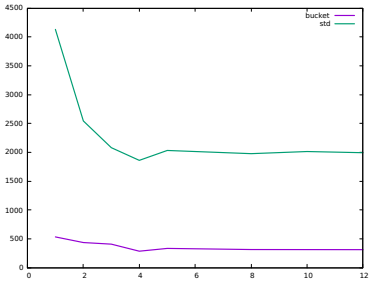
- el sistema alcanza un *speedup* de 2 para el algoritmo de la biblioteca estándar
- el sistema alcanza un *speedup* de 1.5 para el algoritmo de *counting sort*
- en ambos casos se alcanza el máximo a partir de 4 procesos
- el sistema alcanza una acceleración de por lo menos 6 comparando los dos algoritmos de ordenación
- este mínimo (o peor ganancia) se da justamente en el momento de mejor *speedup*

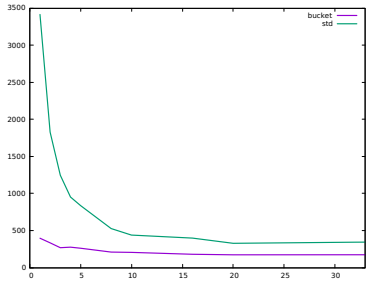
interpretamos lo que vemos en la gráfica del i9



- el sistema alcanza un *speedup* de 10 para el algoritmo de la biblioteca estándar
- el sistema alcanza un *speedup* de 2 (y pico) para el algoritmo de *counting sort*
- en ambos casos se alcanza el máximo a partir de 20 procesos
- el sistema alcanza una acceleración de por lo menos 2 comparando los dos algoritmos de ordenación
- este mínimo (o peor ganancia) se da justamente en el momento de mejor *speedup*

interpretamos lo que vemos comparando los dos sistemas





- fijamos un punto en el cual ambos sistemas son lo más rápido (+o-)
- es cuando se usan unos 20 procesos
- el *speedup* del i9 respecto al i7 en este punto es $307/172 \approx 1.8$ para *counting sort* y $1985/328 \approx 6$ para el algoritmo de biblioteca
- es decir, el sistema del i9 es solamente dos veces mejor que el sistema del i7 si usamos el mejor algoritmo para resolver nuestro problema específico

¿y ...?

- ¿hemos entendido todo?
- pensamos que el i7 tiene 4 núcleos y el i9 tiene 20 núcleos, ¿por qué no vemos un speedup de 4 (respectivamente 20) en el algoritmo de *counting sort* cuyo comportamiento es de $O(n/p+v)$ con el v muy pequeño respecto a n/p?
- ¿por qué vemos un speedup de 2 (respectivamente 10) en el algoritmo de la biblioteca estándar, pero un 1.5 (respectivamente solo un 2) en el algoritmo de *counting sort*?
- os dejo pensando...
- precisamos ciertas cosas más en adelante...