Odenación de números.

Comparativa de rendimiento.

Alejandro Marrero Díaz

14 de mayo de 2018

Universidad de La Laguna

Contenido

1. Introducción
2. Desarrollo
3. Experimento
4. Resultados
5. Conclusiones

Introducción

Introducción

Cuando tenemos un gran número de elementos que ordenar

¿Cuál es la mejor alternativa?

Por ejemplo:

- Ordenar el ratio *peso*/*beneficio* en KP.
- Ordenar lugares en VRP por estrellas.

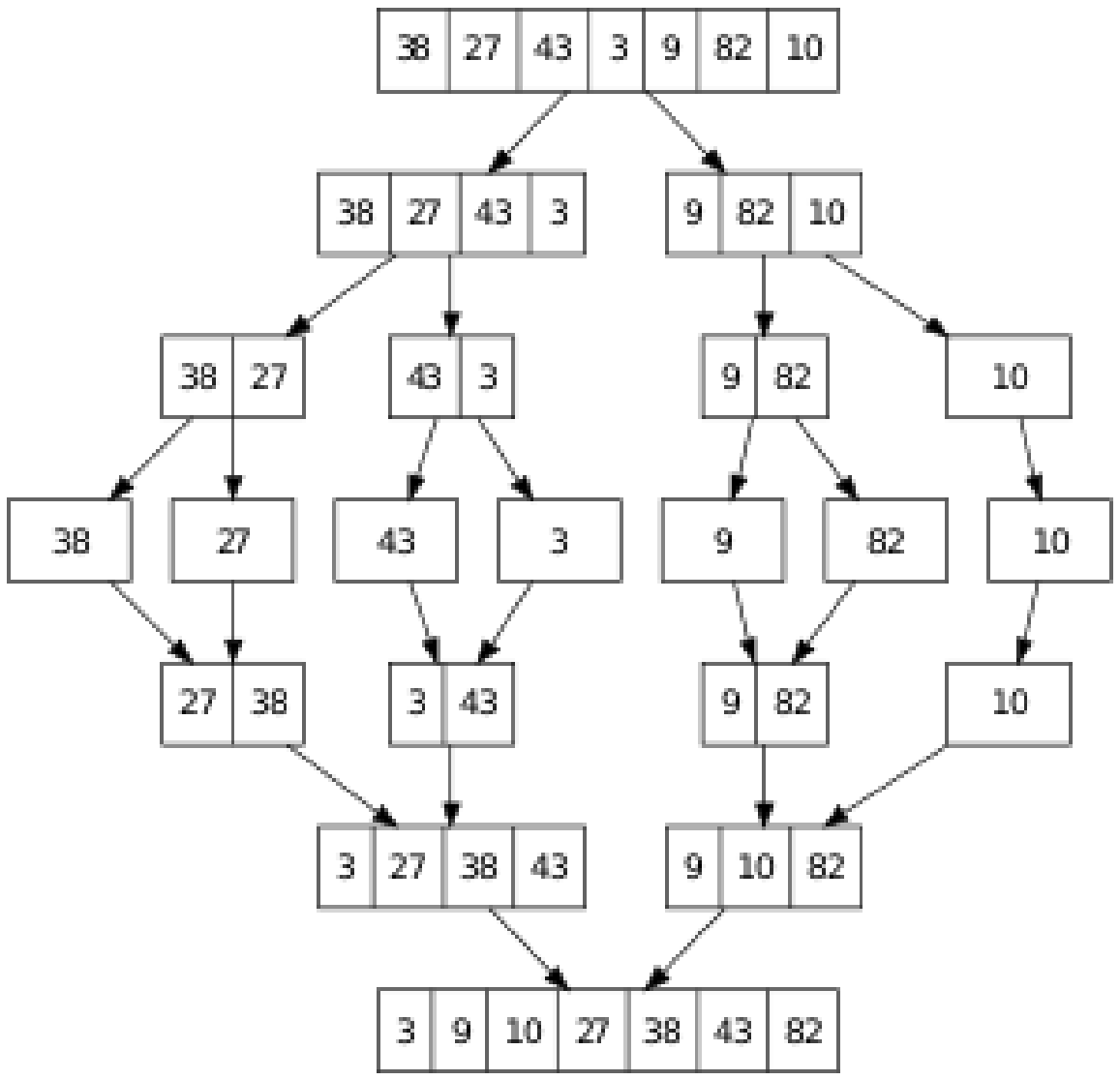
Desarrollo

Desarrollo

Para comprobar que opción es la más eficiente o rentable a la hora de ordenar una gran cantidad de números he decidio implementar los siguientes algoritmos:

- Secuencial: Merge Sort.
- Paralelo: Merge Sort con MPI.
- Cuda: Librería Thrust.

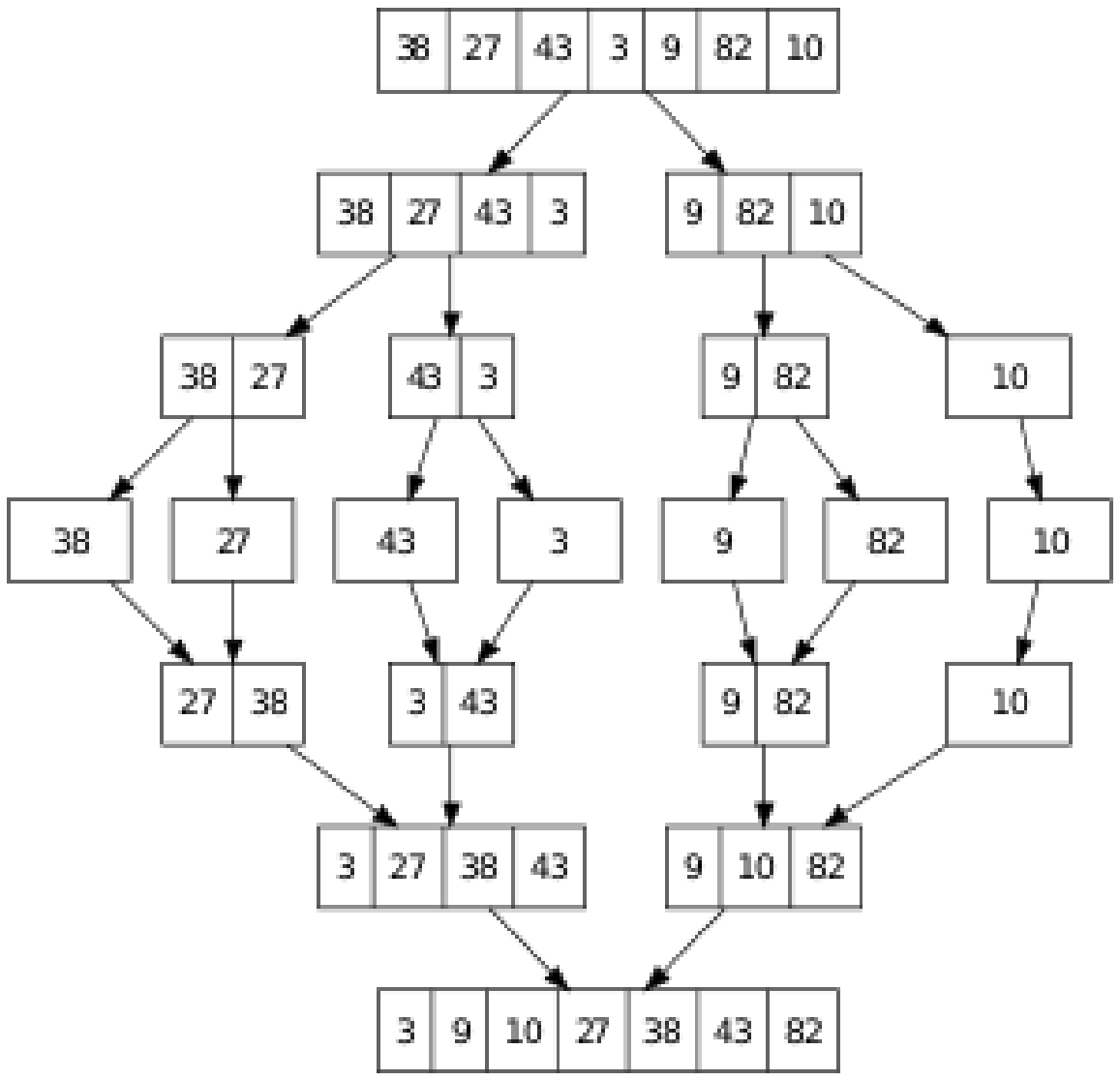
Secuencial - Merge Sort



MPI - Merge Sort

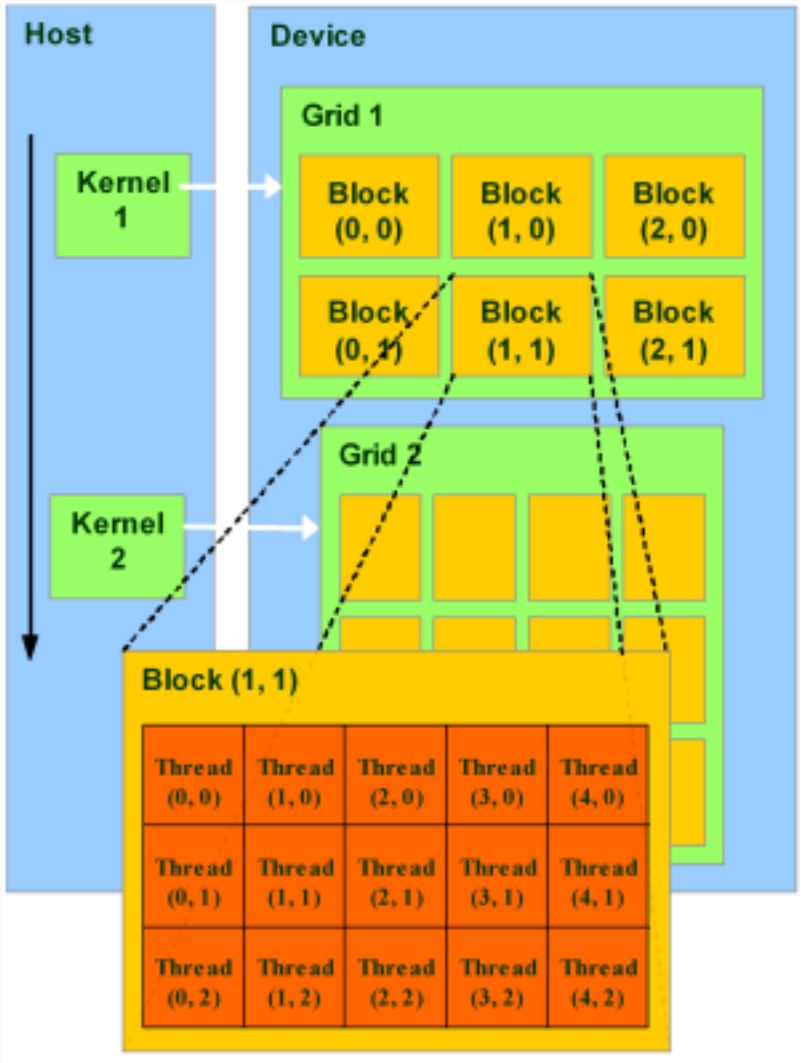
- El array inicial se divide entre *n procesos*.
- Cada proceso ordena (recursivamente) su *sub-array*.
- Cada proceso *i* envia sus resultados parciales al master.
- El proceso master ejecuta una útlima pasada sobre los resultados parciales.

MPI - Merge Sort



Cuda

Modelo de arquitectura en Cuda.



Cuda

- Asignar memoria en el Host(CPU) y en el Dispositivo(GPU).
  - cudaMemcpy*
  - cudaMallocManaged
- Gestión de bloques e hilos.
- No permite usar contenedores de la librería STL.

Librería Thrust

- Se asemeja a la librería STL de C++.
- Dos tipos de vectores:
  - Vector de Host: host_vector.
  - Vector de Dispositivo: device_vector.
- Al igual que en el caso anterior, hay que copiar los vectores.

Librería Thrust

Copiar los datos a la GPU.

```
// generate 32M random numbers serially
thrust::host_vector<int> h_vec(32 << 20);
std::generate(h_vec.begin(), h_vec.end(), rand);

// transfer data to the device
thrust::device_vector<int> d_vec = h_vec;
```

Copiar los datos a la CPU.

```
// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Experimento

Experimento

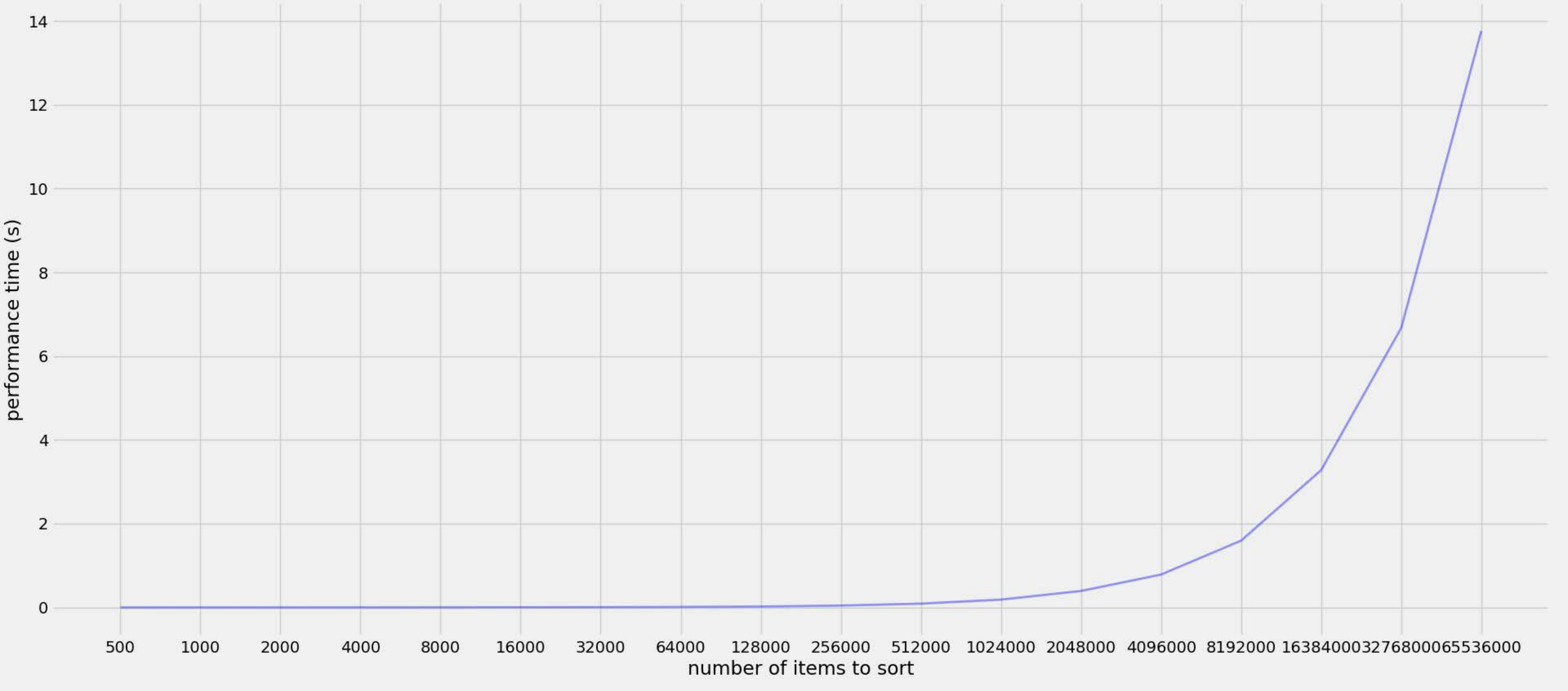
Con el objetivo de medir el tiempo de cómputo, cada implementación se ha probado con la siguiente configuración:

- Números aleatorios en el rango [0, 1].
- Vectores de diferentes tamaños:
  - Tamaño inicial: $size = 500$.
  - Incremento: $size = size * 2$
  - Tamaño máximo: 65536000.

Resultados

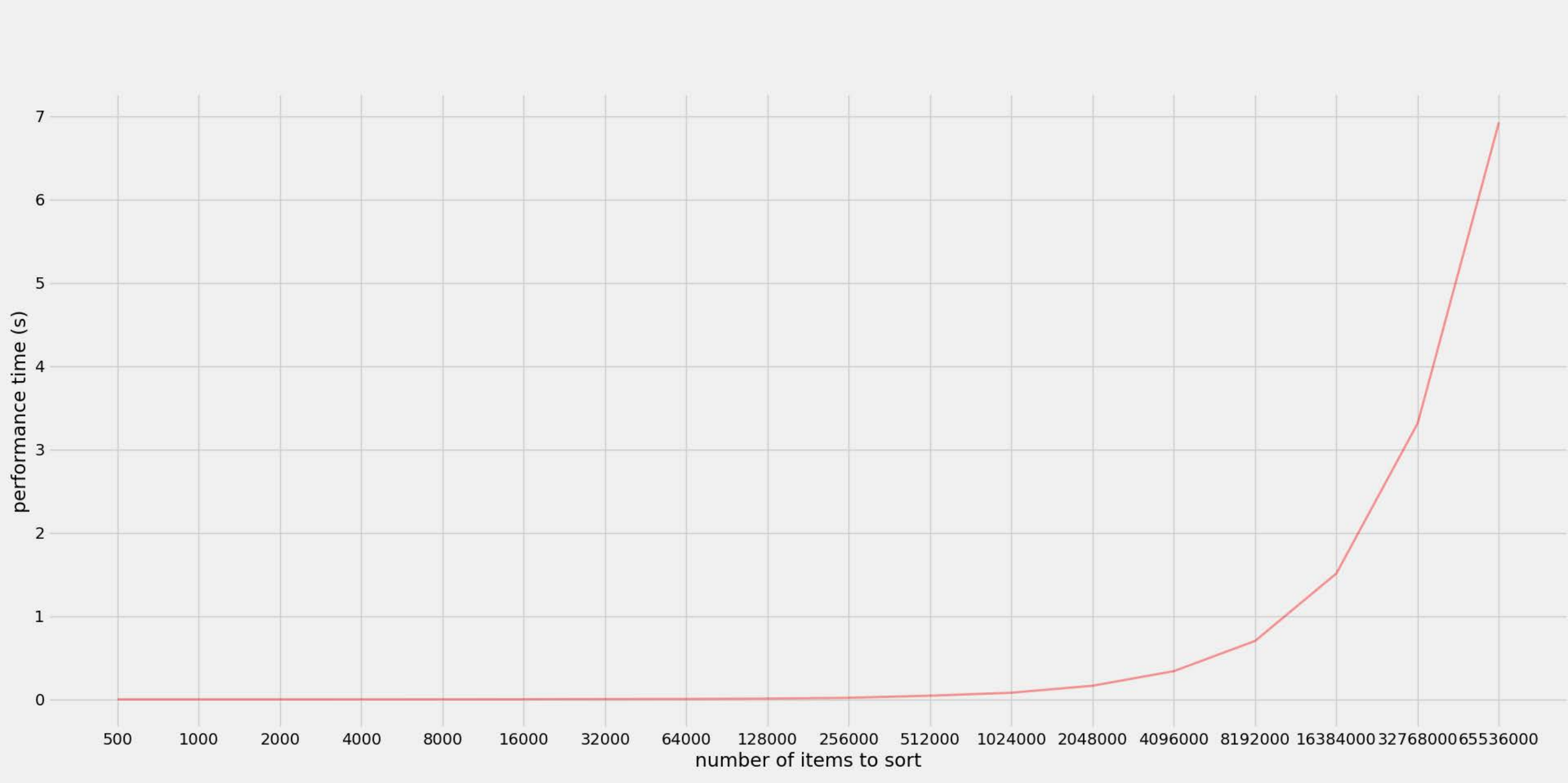
Resultados

Rendimiento de la implementación secuencial.


14
12
10
8
6
4
2
0
performance time (s)
500
1000
2000
4000
8000
16000
32000
64000
128000
256000
512000
1024000
2048000
4096000
8192000
16384000
32768000
65536000
number of items to sort

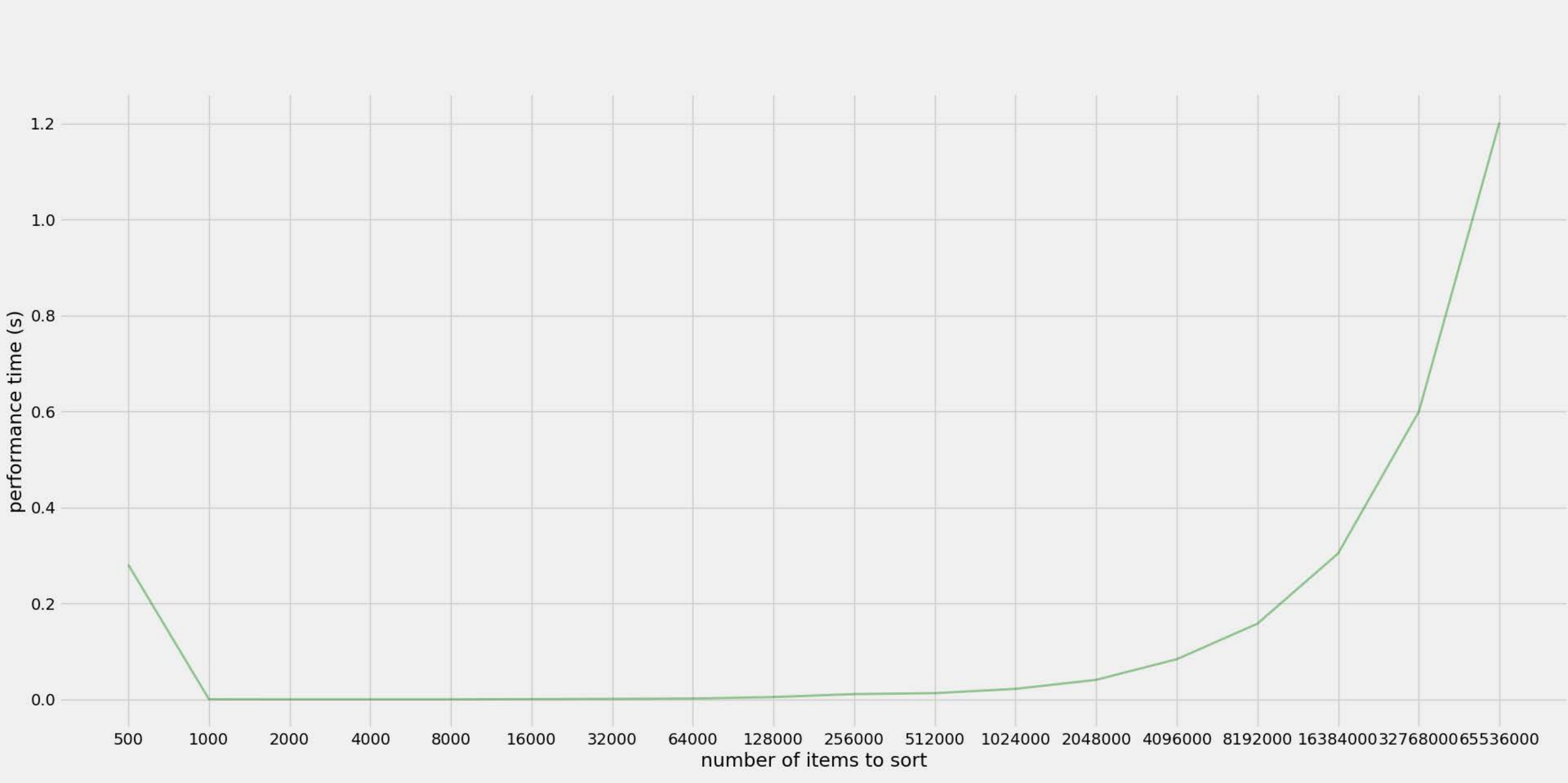
Resultados

Rendimiento de la implementación en MPI con 4 procesos.


7
6
5
4
3
2
1
0
performance time (s)
500
1000
2000
4000
8000
16000
32000
64000
128000
256000
512000
1024000
2048000
4096000
8192000
16384000
32768000
65536000
number of items to sort

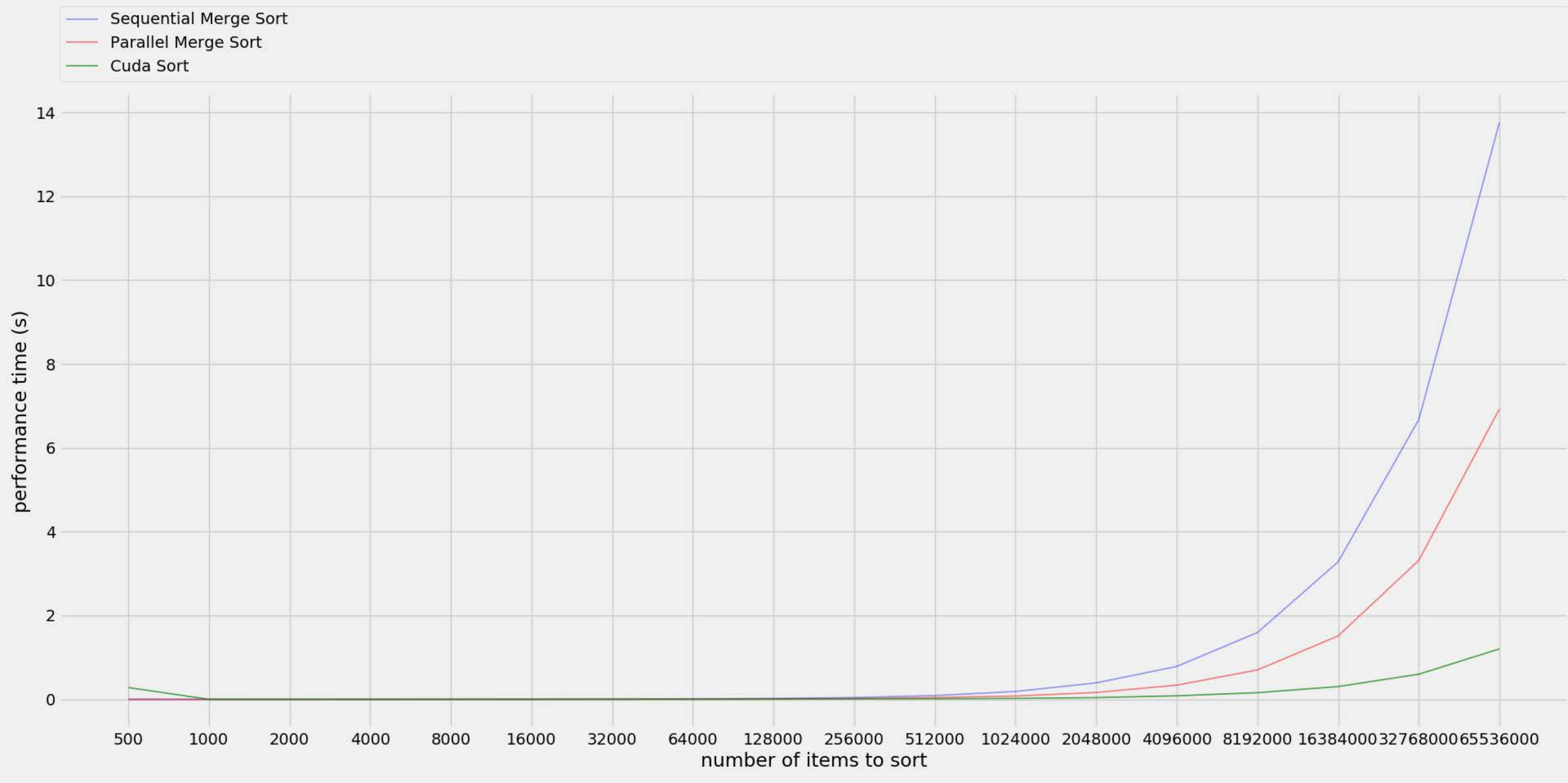
Resultados

Rendimiento de la implementación en Cuda.


1.2
1.0
0.8
0.6
0.4
0.2
0.0
performance time (s)
500
1000
2000
4000
8000
16000
32000
64000
128000
256000
512000
1024000
2048000
4096000
8192000
16384000
32768000
65536000
number of items to sort

Resultados

Comparativa de los resultados.


Sequential Merge Sort
Parallel Merge Sort
Cuda Sort
performance time (s)
0
2
4
6
8
10
12
14
500
1000
2000
4000
8000
16000
32000
64000
128000
256000
512000
1024000
2048000
4096000
8192000
16384000
32768000
65536000
number of items to sort

Conclusiones

Conclusiones

- La versión desarrollada en Cuda es muy superior al resto.
  - Conversión.
  - GPU NVIDIA y conocimientos (mínimos) de Cuda.
- Complejidad del desarrollo en MPI.
- La versión secuencial puede ser superada por el Quicksort.

¿Preguntas?