**INFORME FINAL DE RESULTADOS**

JESUS GARCÉS.

JUAN FERNANDO MARTINEZ.

JUAN PABLO ACEVEDO.

ARIEL PABÓN.

JUAN JOSÉ OSORIO.

**IMPLEMENTACIÓN MONOLÍTICA.**

La versión monolítica del programa de ordenamiento se basa en una implementación concurrente de MSD Radix Sort.1 Inicialmente, de manera secuencial se distribuyen los elementos del arreglo en su correspondiente “bucket” de acuerdo a la primera letra de la palabra. Seguidamente se hace un llamado recursivo sobre cada una de los “buckets” de manera concurrente para la posición de letra siguiente. La solución se realizó usando los patrones Fork-Join y Thread Pool haciendo uso de una implementación proveída por Java que combina ambos patrones (ForkJoinPool) y ofrece la interfaz ‘RecursiveAction’ para definir y ejecutar las tareas. Se hicieron experimentos con archivos de diferentes tamaños que contenían la lista de cadenas a ordenar (Ver Tabla 1). Basados en los resultados de las pruebas monolíticas presentadas, se pueden extraer las siguientes observaciones:

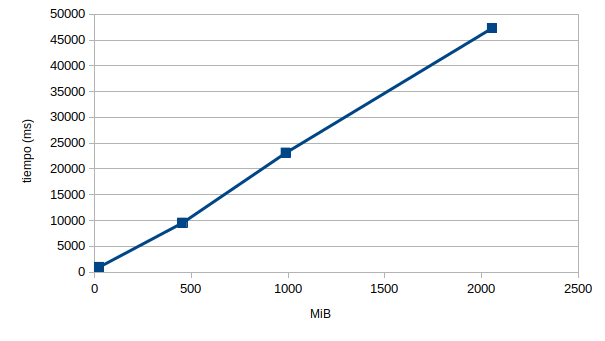
Tabla 1. Resultados de experimentación con la implementación monolítica de radix sort concurrente.

| 23 MiB | Time to read file: 277 ms.  Time to convert list to array: 5 ms.  Time to clone list: 1 ms.  Time to sort parallel: 225 ms.  Finished Writing to “./ordered-array.txt” 453 ms.  Total read + toArray + sort + write: **968 ms**. |
| --- | --- |
| 454 MiB | Time to read file: 1382 ms.  Time to convert list to array: 104 ms.  Time to clone list: 33 ms.  Time to sort parallel: 1305 ms.  Finished Writing to “./ordered-array.txt” 6746 ms.  Total read + toArray + sort + write: **9548 ms**. |
| 998 MiB | Time to convert list to array: 231 ms.  Time to clone list: 89 ms.  Time to sort parallel: 3991 ms.  Finished Writing to ./ordered-array.txt 14589 ms.  Total read + toArray + sort + write: **23130 ms**. |
| 2 GiB | Time to read file: 8224 ms.  Time to convert list to array: 471 ms.  Time to clone list: 132 ms.  Time to sort parallel: 11440 ms.  Finished Writing to ./ordered-array.txt 27151 ms.  Total read + toArray + sort + write: **47295 ms**. |
| 2.24 GiB | Time to read file: 9285 ms.  Time to convert list to array: 4021 ms.  Exception in thread "main" java.lang.OutOfMemoryError: Java heap space. |

**RESUMEN DE LA IMPLEMENTACIÓN MONOLÍTICA.**

Tabla 2. Resultados resumidos de la implementación monolítica de radix sort concurrent.

| TAMAÑO (MiB) | RESULTADO (ms) |
| --- | --- |
| 23 | 968 |
| 454 | 9548 |
| 988 | 23130 |
| 2054 | 47295 |
| 2254 | N/A |



Gráfica 1. Tiempo de ejecución en función del tamaño del archivo.

**CONCLUSIONES DE LA IMPLEMENTACIÓN MONOLÍTICA.**

1. Tiempo de lectura de archivo: A medida de que aumenta el tamaño del archivo de entrada, el tiempo de lectura aumenta linealmente. Por ejemplo, para 23 MB tomó 277 ms, para 454 MB tomó 1382 ms, y para 2GB tomó 8224 ms.

2. Tiempo de conversión de lista a array: Similar al punto anterior, a medida que crece el tamaño, aumenta el tiempo de conversión. De 5ms para 23MB a 471 ms para 2GB.

3. Tiempo de clonación de la lista: Nuevamente se observa un crecimiento lineal con el tamaño. De 1ms a 132 ms.

4. Tiempo de ordenamiento paralelo: Para 23MB fue de 225ms, para 454 MB 1305 ms, y para 2GB 11440 ms. El aumento es casi lineal.

5. Tiempo de escritura del archivo de salida: Para 23MB 453 ms, para 454 MB 6746 ms, y para 2GB se omitió el dato. Pero claramente crece con el tamaño del archivo.

6. Tiempo total (lectura + conversiones + ordenamiento + escritura): Para 23MB fue 968 ms, para 454 MB 9548 ms, y para 2GB 47295ms. Crecimiento aproximadamente lineal.

En resumen, todos los pasos crecen en tiempo casi linealmente a medida que aumenta el tamaño del archivo de entrada. Esto limita el escalamiento para archivos muy grandes, dado que los tiempos se vuelven prohibitivos. Adicionalmente, para 48 millones de strings se observó un error OutOfMemory, indicando que la aplicación monolítica no puede manejar archivos tan grandes debido a limitaciones de memoria.

**SOLUCIONES ALTERNATIVAS.**

Dados los problemas de escalabilidad y manejo de memoria evidenciados, se proponen las siguientes soluciones alternativas:

1. Una arquitectura distribuida: Una opción es desarrollar una arquitectura distribuida, donde la aplicación corra en múltiples nodos. Esto permitiría:

* Balancear la carga de lectura del archivo original sobre varios nodos. Distribuir la conversión de lista a array en varios nodos para paralelizar.
* Implementar el ordenamiento utilizando algoritmos de clasificación distribuida.
* Escritura del archivo final en paralelo o coordinada desde varios nodos. Con este enfoque se esperaría reducir significativamente los tiempos de procesamiento, así como eliminar las limitaciones de memoria de una aplicación monolítica.

2. Procesamiento por lotes: Otra opción es mantener la aplicación monolítica, pero permitiendo lectura y procesamiento por lotes del archivo original. En lugar de cargar todo el archivo a memoria, se puede configurar para procesar incrementalmente subconjuntos del mismo. Por ejemplo, se podrían definir tamaños de lote de 50MB. La aplicación iría leyendo de a 50MB, preprocesado ese lote, escribiendo el resultado intermedio, y así sucesivamente hasta finalizar todo el archivo. Si bien no es tan eficiente como una solución distribuida, al menos atenúa los problemas de memoria y de tiempos.

3. Algoritmos y estructuras de datos más eficientes: Otro aspecto a considerar es si existen opciones de algoritmos de ordenamiento o estructuras de datos más óptimos para manejar archivos grandes. Por ejemplo, algoritmos basados en división y conquista podrían tener mejor performance. Así mismo, en lugar de utilizar listas dinámicas, tal vez haya estructuras más eficientes para acceder a los datos en archivos grandes (árboles, tablas hash distribuidas, etc).

Se recomienda investigar estas alternativas para mejorar el rendimiento de la aplicación monolítica actual.

En definitiva, el informe de pruebas expone claramente las limitaciones de escalabilidad de la aplicación en su enfoque actual. Para llegar a manejar archivos más grandes (cientos GB o TB) se requerirá una solución distribuida basada en clúster. Por lo tanto, se concluye que la aplicación monolítica funciona bien para archivos pequeños y medianos, pero presenta problemas de escalabilidad y manejo de memoria en archivos muy grandes. Se recomienda evaluar alternativas distribuidas y/o de procesamiento por lotes para mejorar estos aspectos.

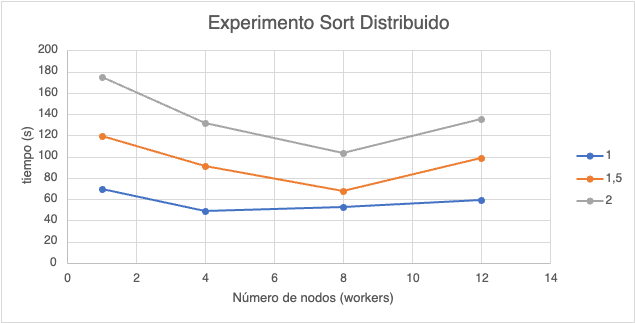
**IMPLEMENTACIÓN NO-MONOLÍTICA.**

Para la versión no monolítica se implementó el patrón Separable Dependencies. Esto implica que se empleó NORMA para el acceso de datos, ya que este patrón los copia en cada instancia de ‘Task’ y se los pasa a los Workers de esta forma. Además, se implementaron dos estrategias distintas:

* Estrategia 1: Primero se parte el arreglo en partes iguales, luego se mandan a los Workers para que los ordenen con Radix-Sort. Los resultados se mandan al Máster y finalmente se mezclan para obtener el resultado final empleando el algoritmo de Parallel-Merge.
* Estrategia 2: Primero se parte el arreglo en partes iguales, luego se mandan a los Workers para que se dividan los elementos en grupos de acuerdo con sus primeros N caracteres. Después se mandan estos grupos al Master, consolidando los grupos y mandándolos de vuelta a los Workers para ordenarlos. El resultado se genera con el resultado de estos ordenamientos, y dado que las listas empiezan con N caracteres, entonces la mezcla es trivial.

**RESULTADOS ESTRATEGIA 1.**

| Workers | Tamaño (GiB) | Tiempo 1 (ms) | Tiempo 2 (ms) | Promedio (ms) |
| --- | --- | --- | --- | --- |
| 1 | 1 | 70404 | 69552 | 69978 |
| 4 | 1 | 47475 | 51117 | 49296 |
| 8 | 1 | 46812 | 57814 | 52313 |
| 12 | 1 | 60291 | 57509 | 58900 |
| 1 | 1,5 | 125340 | 113840 | 119590 |
| 4 | 1,5 | 91758 | 91487 | 91622,5 |
| 8 | 1,5 | 72721 | 63385 | 68053 |
| 12 | 1,5 | 100256 | 96982 | 98619 |
| 1 | 2 | 198534 | 152180 | 175357 |
| 4 | 2 | 128502 | 134664 | 131583 |
| 8 | 2 | 101830 | 104871 | 103350,5 |
| 12 | 2 | 127044 | 143445 | 135244,5 |



Gráfica 2. Tiempo de ejecución en función del tamaño del archivo y la cantidad de Workers.

Se puede observar un comportamiento similar en las gráficas de tiempo de ejecución en función del número de Workers. Todas tienden a tener una cantidad que minimiza el tiempo de ejecución. Siendo específicos, para 1.5 GiB y 2 GiB este punto son 8 Workers, mientras que para 1 GiB es de 4 (es probable que este último resultado haya sido afectado por factores externos).

**RESULTADO ESTRATEGIA 2.**

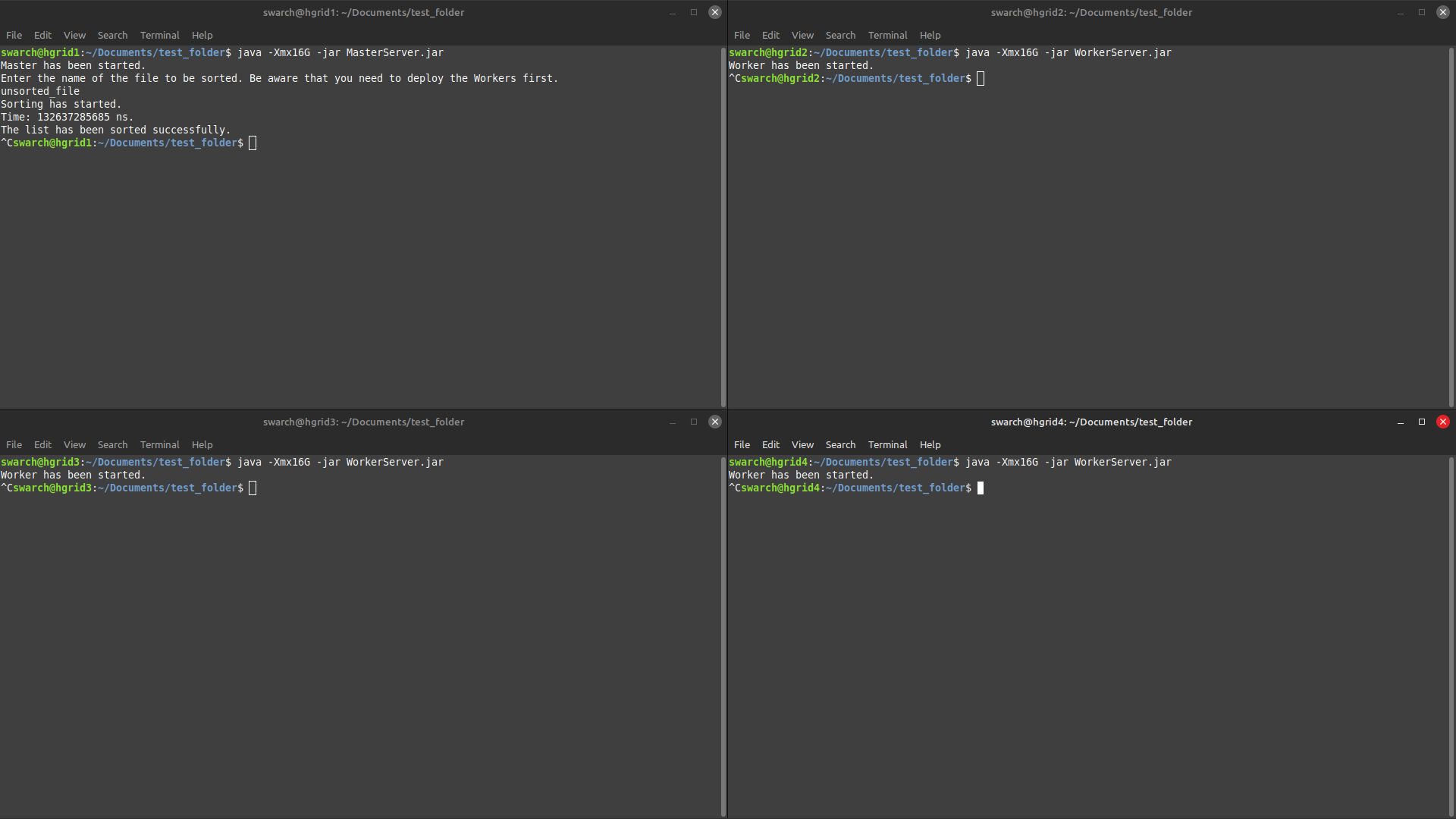


Figura 4. Tiempo de ejecución al ordenar 2 GiB con grupos de 128 MiBs (1 Master y 3 Workers).

A simple vista parece ser que la estrategia 2 tuvo peor rendimiento que la otra, razón por la cual se decidió enfocarse en la primera (a excepción de la búsqueda de tamaño óptimo, es decir, la siguiente sección de este documento). Sin embargo, puede que este resultado en específico haya sido afectado por factores externos y que esta decisión no haya sido contundente.

**TAMAÑO DE GRUPO Y RENDIMIENTO EN SEGUNDA ESTRATEGIA.**

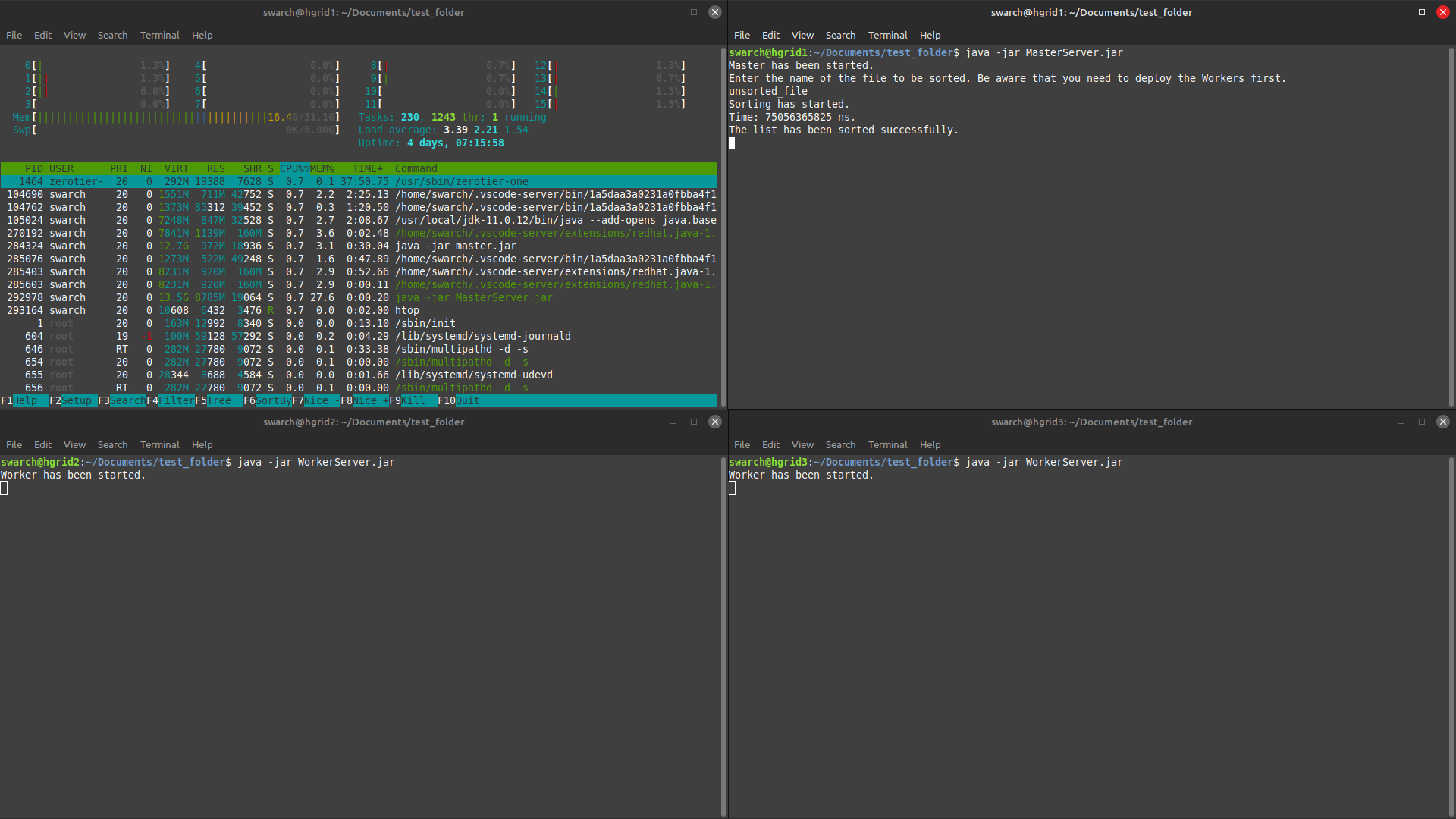


Figura 1. Tiempo de ejecución al ordenar 1 GiB con grupos de 64 MiBs (1 Master y 2 Workers).

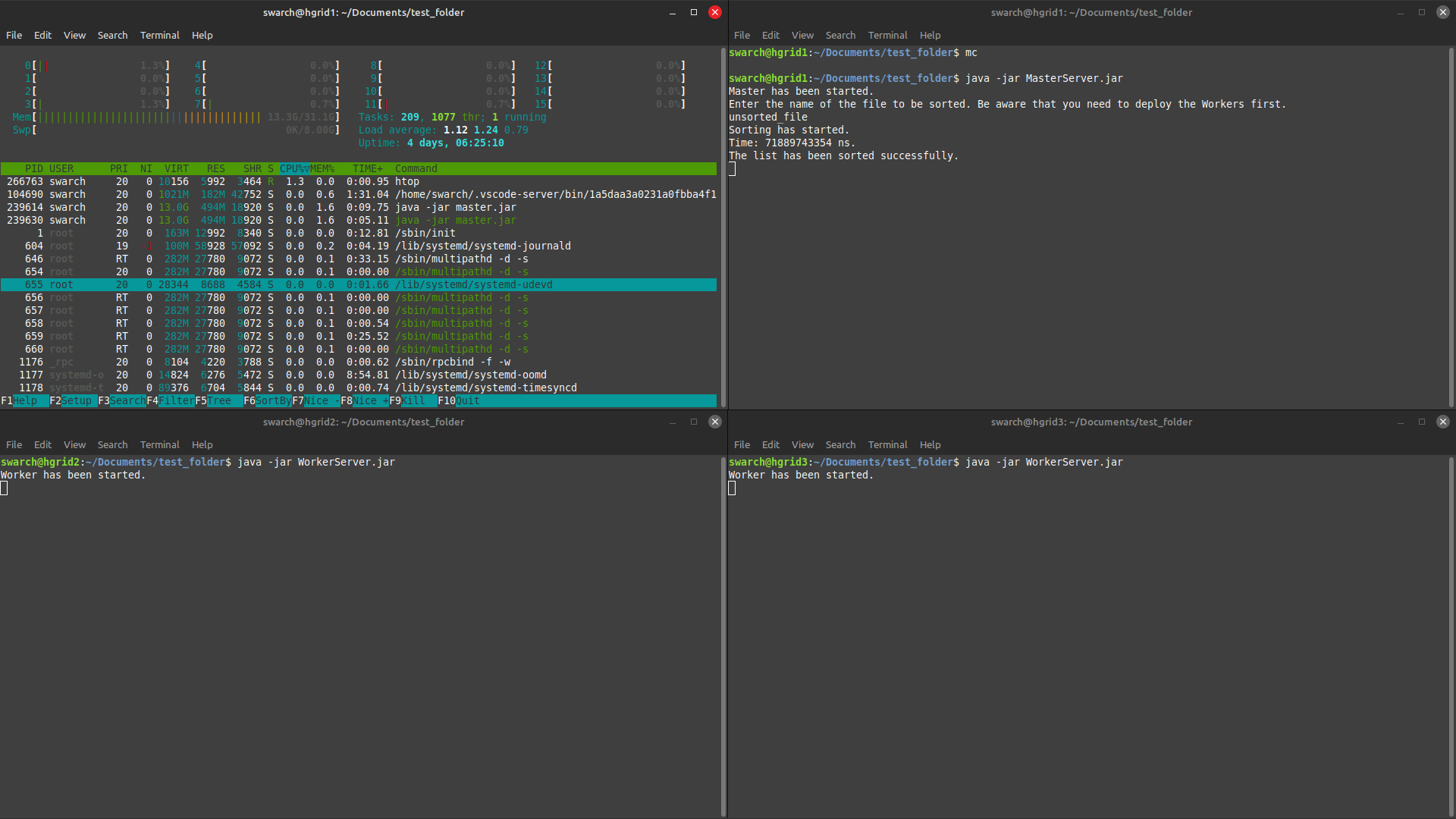


Figura 2. Tiempo de ejecución al ordenar 1 GiB con grupos de 128 MiBs (1 Master y 2 Workers).

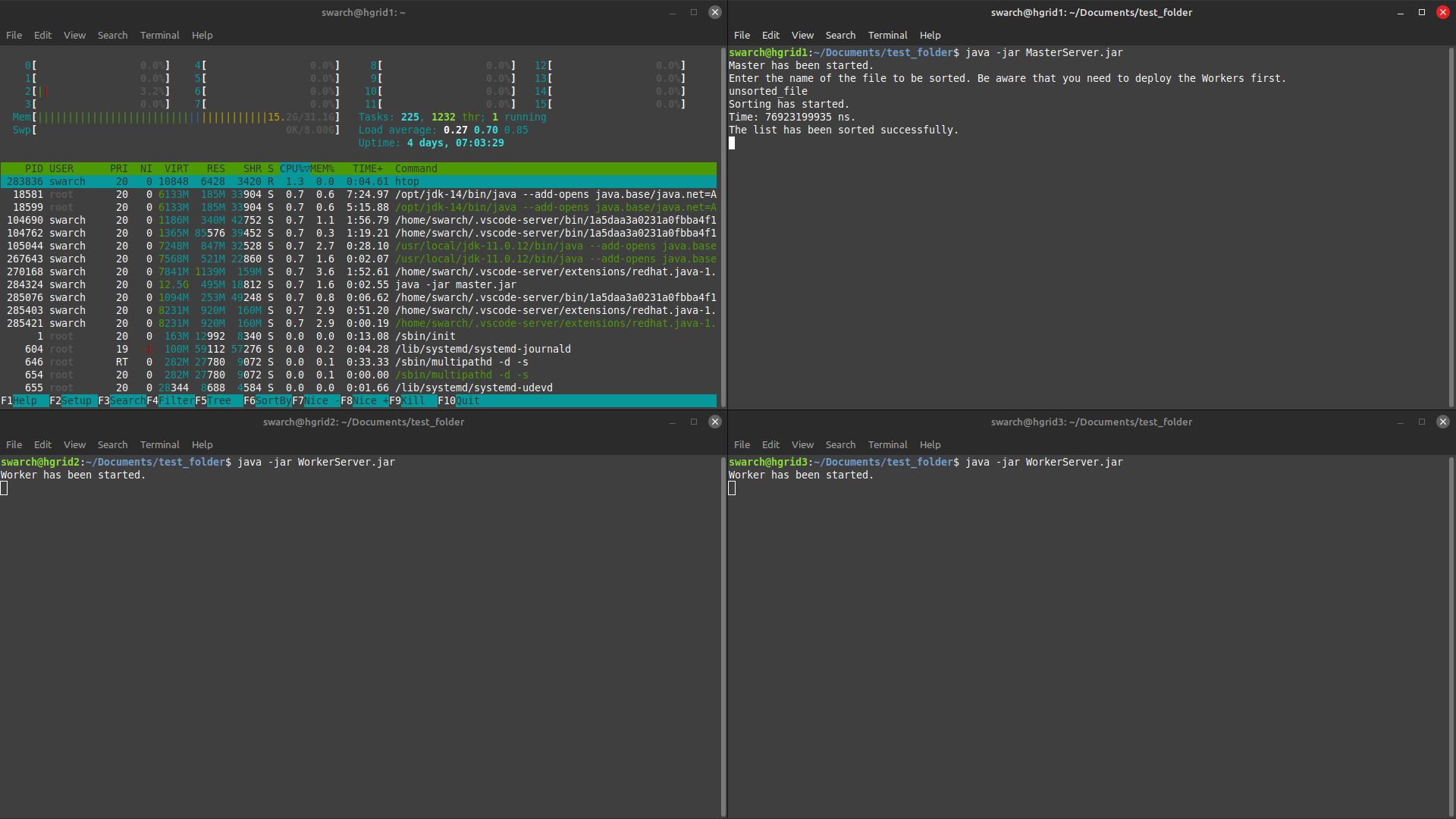


Figura 3. Tiempo de ejecución al ordenar 1 GiB con grupos de 256 MiBs (1 Master y 2 Workers).

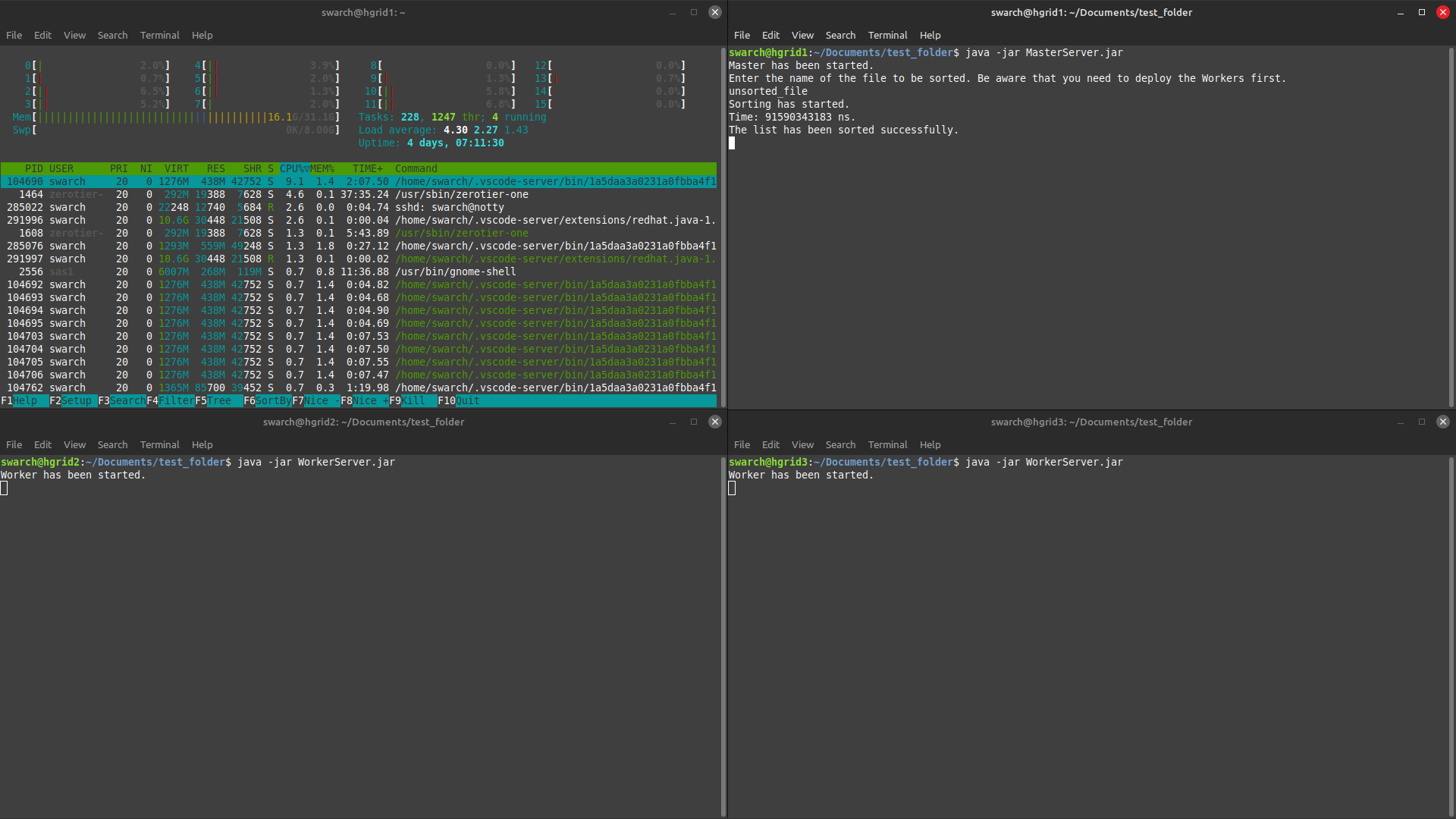


Figura 4. Tiempo de ejecución al ordenar 1 GiB con grupos de 512 MiBs (1 Master y 2 Workers).

| TAMAÑO DE BATCH (MiB) | TIEMPO (S) |
| --- | --- |
| 64 | 75.05 |
| 128 | 71.89 |
| 256 | 76.92 |
| 512 | 91.59 |

Se puede observar que el tamaño de los arreglos que se agrupan y ordenan incide en el tiempo de ejecución. Según lo experimentado, se deduce que tamaños muy grandes o muy pequeños son perjudiciales para el rendimiento. Esto es porque la transferencia de archivos tiene un impacto grande en el tiempo de ejecución, por lo que mandar mucho se puede tardar demasiado, pero mandar poco implica procesar, valga la redundancia, muy poco. Además, hay un tamaño que tiende a minimizar el tiempo de ejecución, el cual es 128 MiB para este caso.

**CONCLUSIONES DE LA IMPLEMENTACIÓN NO-MONOLÍTICA.**

Se observa que las implementaciones distribuidas tienen peor rendimiento que la monolítica, lo cual se asume que se debe al tiempo de transmisión y de comunicación de los datos. Se puede deducir entonces que el ordenamiento distribuido sólamente vale la pena cuando se tienen grandes cantidades de datos (y con grandes nos referimos a MUY grandes).

También se puede deducir que el manejo de la memoria es esencial en la implementación. Si no se hace correctamente, entonces los tamaños de archivo que se podrán organizar serán muy pequeños; en nuestro caso las implementaciones no soportaban más de 2 GiB.

Remediar esto probablemente tenga que ver con qué tantas ‘Task’ se cargan a la vez, así como el uso de una memoria secundaria a la DRAM para guardar los resultados parciales. También es importante considerar el tamaño de los arreglos de cada Task, ya que esto también incide en el tiempo de procesamiento y de transferencia. Por otro lado, se podrían emplear herramientas más adecuadas para el gran manejo de datos, ya que ‘ICE’ junto con Java tiene limitaciones en cuanto a la tasa de transferencia de datos.

**BIBLIOGRAFÍA.**

1. Sedgewick, R. & Wayne, K. (2011) 5.1 String Sorts. Tomado de: <https://algs4.cs.princeton.edu/51radix/>