Universidad Nacional Autónoma de México
Facultad de Ciencias
Algoritmos Paralelos

Proyecto final: Implementación de Run-length enconding (RLE)

Autor: Medina Peralta Joaquín

Problema

Run-Length encoding es una forma de compresión de datos sin pérdida, donde la secuencia de símbolos de la entrada se almacenan como una simple ocurrencia con el número simbolos consecutivos.

Definimos, para una cadena $w \in \Sigma^*$, una serie de pares (r, l, s) de datos consecutivos dadas por:

- El carácter de control $r \notin \Sigma$.
- La longitud de la secuencia $l \in \mathbb{N}$
- El símbolo del alfabeto $s \in \Sigma$

Por ejemplo, tenemos una cadena perteneciente al alfabeto en inglés, `ABABBBC` y la lista de tuplas con el caracter de control α, se tiene:

$$(\alpha, 1, \texttt{A}), (\alpha, 1, \texttt{B}), (\alpha, 1, \texttt{A})$$
$$(\alpha, 3, \texttt{B}), (\alpha, 1, \texttt{C})$$

Para poder representarlo en un archivo, los símbolos con un solo elemento consecutivo, es decir que no se repiten, se representarán como `s`. Con el ejemplo anterior, la cadena comprimida sería `ABA(`α`,3,B)C`.

Por lo tanto, el algoritmo es eficiente para cadenas o contenido donde hay muchas secuencias consecutivas, es decir, archivos de texto, binarios representación de una imagen por medio de pixeles o componentes de bloques largos de archivos de sonido. (Pu, 2005).

Para este trabajo, se consideran únicamente archivos de texto para comprimir que estén en `utf-8`, debido a que, se usará una combinación para representar las tuplas. Para los símbolos sin repetición o secuencia muy cortas, , como se menciona en (Pu, 2005), se implementará un modo literal donde los bytes se escriben directamente sin tupla. Esto evita la expansión del archivo, permitiendo que la compresión se reserve únicamente para las secuencias que si generen un ahorro significativo.

Algoritmo secuencial

Como se menciona en (Pu, 2005), se define caracteres de control de repetición $r_3, r_4, \ldots$, los cuales para r_i representa la cantidad i de símbolos consecutivos. De esta manera, se usará el caracter de control únicamente cuando la longitud sea mayor a 3.

Asi, definimos el algoritmo de encriptación como:

- Repetir hasta llegar al final del archivo:
  - Leer el símbolo S de la secuencia:
    - Contar el número de símbolos consecutivos iguales i hasta encontrar uno diferente.
    - Guardar en el archivo de salida el par r_iS solo si $i > 2$, en oto caso guardar el símbolo i

Para la implementación, se debe de considerar tres apuntadores i, j donde $j = i + 1$ donde los usaremos para recorrer el arreglo de símbolos S, para lo cual, mientras que $S[i] == S[j]$ entonces podemos incrementar el contador, de esta manera verificamos si es mayor al umbral, de esta manera podemos representarlo en forma de tupla y en otro caso, se escribe el caracter $S[q]$ y $S[p]$ en la cadena comprimida. (El pseudocódigo se encuentra en el anexo 1)

De esta manera, definimos el algoritmo de desencriptación como:

- Repetir hasta llegar al final del archivo:
  - Leer el símbolo S de la secuencia:
    - Si el símbolo $S = r_i$, se debe de leer el símbolo consecutivo y escribir i veces en la salida.
    - En otro caso, escribir el símbolo actual.

De la misma manera, es necesario un apuntador p para recorrer el archivo comprimido, donde si $S[p]$ es el caracter de control definido r, se toma el valor de $S[p+1] = n$ como un número y $S[p+2]$ como el símbolo a repetir n veces, en otro caso se guarda $S[p]$ en la salida. (El pseudocódigo se encuentra en el anexo 2)

A su vez, se considera el uso de `FLAG_RLE` para iniciar la tupla definida en la sección anterior y en dado caso que el símbolo s sea igual a `FLAG_RLE`, se considera a `FLAG_LITERAL` para escapar y escribir el valor de `FLAG_RLE`.

Algoritmo en paralelo

Para esto, se toma en inspiración del artículo (Manchev, 2006), donde se propone el dividir la secuencia de entrada en bloques procesados en paralelo donde cada proceso identifica las corridas locales dentro de su bloque, para posteriormente realizar una fase de combinación donde se revisa si los límites entre bloques contiguos se cruzan con corridas de símbolos, asegurando que las secuencias iguales se representen correctamente.

De esta manera, se debe de utilizar el uso de memoria compartidad y distribuida para poder ejecutar el algoritmo y por lo tanto, nos esta danto la pista de usar `MPI`.

Antes de pasar a la implementación, debemos mencionar que esta estrategía de paralelizar el algoritmo secuencial `RLE` sigue el principio de `Fork Join` donde se divide el trabajo entre los distintos procesadores y al final, juntar los resultados de cada procesador en uno global.

Se puede observar la técnica de `Fork Join` de manera explicita al dividir la secuencia de entrada en bloques para que cada procesador realice `RLE` de manera local y al final se junta cada bloque revisando si no hay cruzes entre cada uno.

De esta manera, proponemos el pseudocódigo (3) donde se puede observar la compresión en `RLE` en paralelo y en (4) la manera de descomprimir en paralelo.

Uno de los detalles al momento de realizar la implementación, es al momento de recopilar el resultado de cada proceso, debido a la siguientes consideraciones para sea un proceso P_i, P_{i-1}:

- Al momento de comprimir, si dos chunks de memoria contenian la misma secuencia de símbolos, el proceso P_i debe sumar los caracteres de la frontera de P_{i-1} y P_{i-1} debe de borrar del vector de resultado la información de la representación compactada de la secuencia de símbolos.
- Al momento de descomprimir, si la representación compactada de una secuencia de símbolos es dividida entre la frontera de dos chunks, el proceso P_i debe de tomar la información faltante de P_{i-1} para que P_i pueda descomprimir c orrectamentela información. Para esto, se toma a lo más 2 bytes de del chunk anterior continuo y se elimina la información tomada del chunk anterior.

Experimentación

Para esto, trabajaremos con 3 archivos de 100 MB con una alta, media y baja repetitividad los cuales se calculara el tiempo de comprimir de manera secuencial y paralela con $2, 4, 6, 20, 50$ procesadores.

De este modo, se realizará la prueba en el siguiente equipo:

```
---------------------
Sistema operativo: Arch Linux
Kernel: 6.17.9-arch1-1
Arquitectura: x86_64
CPU: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Nucleos fisicos: 4
Nucleos logicos: 8
Memoria RAM total: 16,00 GB
-----------------------------------------
```

Para esto, se creo un script `generate_test_data.sh`, el cual produce los siguientes archivos de `100 MB`.

1. Un archivo de alta repetitividad (solo contiene un único tipo de byte, `A = 0x41`).

2. Un archivo aleatorio que representa el peor caso para este algoritmo (tomando elementos aleatorios de `/dev/urandom`).
3. Un archivo de repetitividad media tipo malla, donde se repite el patrón `0123456789`.

Por último, se creo un script para comprimir archivos por medio de N procesos y de manera secuencial para medir la eficiencia y el `Speedup`.

Resultados

Se realizó la experimentación conforme a lo descrito en la sección anterior, teniendo los siguientes tiempos para cada uno de los archivos y con los siguientes detalles por cada caso (usando las gráficas 7, 6 y 5):

- Para `data_plana.bin`, por la tabla 1 se puede observar que el mejor tiempo promedio fue cuando se utilizo $N = 6$ procesos con 0.0701 y con un Speedup de $S = 1.9$ lo cual está lejos de lo ideal ya que tuve que mejorar 6 veces. Por último, su eficiencia, disminuyo cerca del 30 % pero en el caso de $N = 4$ su eficiencia es de 46 % lo cual es optimo.
- Para `data_malla.bin`,por la tabla 2 llega a su menor tiempo promedio con $N = 4$ con 0.2123 y un Speedup alrededor de $S = 2.5$, aunque está un poco más de lo idea ya que tiene que mejorar 4 veces pero es suficiente. De esta manera, con respecto a su eficiencia se tiene que es de 62 %, lo cual no dismninuye mucho.
- Para `data_aleatoria.bin`, por la tabla 3 llega con su mejor tiempo promedio con $N = 4$ con 0.2324 y un Speedup alrededor de 2.4200, lo cual como en los dos casos anteriores, es más de la mitad al esperarse que mejore 4 veces. De esta manera, con respecto a su eficiencia es del 60 %, lo cual no disminuye mucho.

Con lo anterior dicho y revisando las gráficas, podemos observar que el número de proceso optimos para el equipo en que se realizaron las pruebas es de $N = 4$ al tener en su mayría de los casos el mejor tiempo proemdio, Speedup y eficiencia.

De similar manera, podemos ver que al incrementar el número de procesos mayor a 4, empieza a incrementar el tiempo promedio entonces no se beneficia del incremento de los procesos, ya que al realizarse cambio de contexto se ralentiza el proyecto.

Otra observación, es al momento de comunicar los resultados y verificando las fronteras entre procesos continuos para evitar que la compresión no sea correcta al no considerar secuencias continuas entre dos chunks o se pierda información en la descompresión al dividir el archivo en chunks y que se pierda información como tuplas de información.

De esta manera, el uso de computo distribuido para trabajar el algoritmo paralelo por medio de la técnica de `Fork Join` aunque mejora los tiempos, a la larga no se beneficia por la cantidad de procesos. Por último, la parte más complicada de la implementación de esta versión de `RLE` fue la unión de los datos y la comunicación entre procesos.

Conclusión

Como podemos observar durante el reporte, la realización de un algoritmo secuencial a paralelo aunque en ciertas ocasiones es fácil como en `RLE`, su implementación es de las más criticas porque se debe de considerar condiciones para combinar el resultado de cada proceso y obtener la salida correctamente como el algoritmo secuencial.

De esta manera, en lo personal se considero un reto bastante técnico y de implementación al considerar el uso de computo distribuido como uno de los ejes principales del proyecto junto con la técnica de `Fork Join` para paralelizar `RLE`.

A su vez, al trabajar con poco con `C++`, el aprender la sintaxis y la manera de usar `MPI` fue de lo más complicado y era de lo mejor para evitar el manejo de memoria que se debe d e considerar en una implementación en `C` y el uso de programación orientada a objetos ya incluida en `C++` facilitando la implementación.

Por último, con la visualización de los datos, podemos entender que la implementación mejoro significativamente el tiempo del algoritmo alcanzando su menor tiempo promedio con $N = 4$ procesos lo cual para datos enormes de memoria, se puede considerar mejor alternativa al la implementación secuencial.

Referencias

Manchev, N. (2006). Parallel Algorithm for Run Length Encoding. *Proceedings of the Third International Conference on Information Technology: New Generations (ITNG 2006).* https://doi.org/10.1109/ITNG.2006.106

Pu, I. M. (2005). *Fundamental Data Compression* [pp. 56–64]. Elsevier Butterworth–Heinemann.

Anexo

Algoritmo 1 Algoritmo para comprimir en `RLE` con modo literal (Umbral 3)

```
1: procedure COMPRIMIR(S)                                   ▷ Cadena de símbolos S
2:     N ← longitud(S)
3:     0 ← []                                               ▷ Cadena de salida comprimida
4:     p ← 0
5:     while not EOF do
6:         i ← 1
7:         q ← p + 1                                        ▷ Fase 1: Contar la secuencia
8:         while q < N y S[q] = S[p] y i < 255 do           ▷ Máximo conteo en 1 byte
9:             i ← i + 1
10:            q ← q + 1
11:        end while
                                                            ▷ Fase 2: Aplicar el umbral y codificar
12:        if i ≥ 3 then                                    ▷ Guardar como tupla RLE: ⟨FLAG_RLE, i, S_p⟩
13:            O ← O ‖ FLAG_RLE ‖ i ‖ S[p]
14:        else                                             ▷ Guardar como secuencia literal: S[p] se repite i veces
15:            for k ← 0 hasta i − 1 do
16:                O ← O ‖ S[p]
17:            end for
18:        end if
19:        p ← p + i        ▷ Mover el apuntador principal al inicio de la siguiente secuencia
20:    end while
21:    Salida: 0
22: end procedure
```

Figura 1: Algoritmo secuencial para comprimir

Algoritmo 2 Algoritmo para Descomprimir en `RLE` (Modo Extendido)

```
1: procedure DESCOMPRIMIR(C)                          ▷ Cadena de símbolos comprimidos C
2:     N ← longitud(C)
3:     0 ← []                                          ▷ Cadena de salida descomprimida
4:     p ← 0
5:     while p < N_C do
6:         S ← C[p]                  ▷ Leer el símbolo actual (byte de control o literal)
7:         if S = FLAG_RLE then
8:             if p + 2 ≥ N_C then
9:                 break                                ▷ Error: Buffer incompleto
10:            end if
11:            i ← C[p + 1]                             ▷ Leer el conteo (longitud)
12:            V ← C[p + 2]                             ▷ Leer el valor del símbolo
13:            for k ← 1 hasta i do
14:                O ← O ‖ V                ▷ Escribir el valor V el número de veces i
15:            end for
16:            p ← p + 3        ▷ Avanzar el apuntador 3 posiciones (FLAG, Conteo, Valor)
17:        else if S = FLAG_LITERAL then                ▷ Símbolo FLAG escapado
18:            if p + 1 ≥ N_C then
19:                break                                ▷ Error: Buffer incompleto
20:            end if
21:            V ← C[p + 1]          ▷ Leer el siguiente byte (que es un FLAG real)
22:            O ← O ‖ V                                ▷ Escribir el byte escapado
23:            p ← p + 2          ▷ Avanzar 2 posiciones (FLAG_LITERAL, Valor)
24:        else                                         ▷ Símbolo sin repetición
25:            O ← O ‖ S                  ▷ Escribir el símbolo literal actual S
26:            p ← p + 1                                ▷ Avanzar 1 posición
27:        end if
28:    end while
29:    Salida: O
30: end procedure
```

Figura 2: Algoritmo secuencial para descomprimir

Algoritmo 3 Algoritmo para comprimir en `RLE` en paralelo

procedure COMPRIMIR_PARALELO(S,N) ▷ S entrada y N número de procesos
▷ Lógica del Coordinador (Pre-procesamiento)
    $Global\_Size \leftarrow longitud(S)$
    for $id \leftarrow 0 \dots N-1$ **do**
        OffsetStart
        $SizeChunk \leftarrow calculate\_chunk(id, Global\_Size, S)$
        $Chunkid \leftarrow S[OffsetStart \dots OffsetStart + SizeChunk + 1]$
▷ Incluye 1 byte de frontera
        Enviar($Chunkid$) a N_id
    end for
    $Sincronizar()$
▷ Lógica del Trabajador
    $\text{Input}_{\text{T}} \leftarrow$ **Recibir**(Chunk_{id})
    $\text{Input}_{\text{Main}} \leftarrow \text{Input}_{\text{T}}$[datos principales]
    $\text{Byte}_{\text{Next}} \leftarrow \text{Input}_{\text{T}}$[byte de frontera]
    $\text{Output}_{\text{local}} \leftarrow$ Comprimir_Secuencial($\text{Input}_{\text{Main}}$) ▷ Usa RLE Extendido
▷ Corrección de Fronteras (Comunicación/Sincronización)
    if $id < N-1$ **then**
        Enviar_Front($\text{Output}_{\text{local}}$.último_bloque) a $\text{Trabajador}_{\text{id+1}}$
        Recibir_Front($\text{Señal}_{\text{Fus}}$) de $\text{Trabajador}_{\text{id+1}}$
        if $\text{Señal}_{\text{Fus}}$ = FUSIONADO **then**
            $\text{Output}_{\text{local}}$.Eliminar_último_bloque()
        end if
    end if
    if $id > 0$ **then**
        Recibir_Front($\text{Bloque}_{\text{Prev}}$) de $\text{Trabajador}_{\text{id-1}}$
        Intentar_Fusion($\text{Output}_{\text{local}}$.primero, $\text{Bloque}_{\text{Prev}}$)
        Enviar_Front($\text{Señal}_{\text{Fus}}$) a $\text{Trabajador}_{\text{id-1}}$
    end if
    Enviar($\text{Output}_{\text{local}}$) al Coordinador
    Sincronizar()
▷ Lógica del Coordinador (Post-procesamiento)
    Recibir_Buffers(Todos los trabajadores)
    $\text{Len}_{\text{Acumulada}} \leftarrow$ Calcular_Suma_Prefijo(Longitudes_locales)
    $O_{\text{comp}} \leftarrow$ Concatenar(Buffers recibidos, $\text{Len}_{\text{Acumulada}}$)
    Salida: O_{comp}
end procedure

Figura 3: Algoritmo paralelo para comprimir

Algoritmo 4 Algoritmo para descomprimir en `RLE` en paralelo

procedure DESCOMPRIMIR_PARALELO(C,N) ▷ C entrada comprimida y N número de procesos
    ▷ Lógica del Coordinador (Pre-procesamiento)
    $Par_{count} \leftarrow read\_Metadato(C)$ ▷ Número total de pares/bloques
    for $id \leftarrow 0 \dots N-1$ **do**
        $Offset_{par}, Size_{par} \leftarrow Calcular\_Particion\_Pares(id, Par_{count})$
        $Chunk_{id} \leftarrow C[Offset_{par} \dots Offset_{par} + Size_{par}]$ ▷ Lectura de porción de pares
        Enviar($Chunkid$) a N_id
    end for
    $Sincronizar()$
    ▷ Lógica del Trabajador
    $\text{Input}_{\text{T}} \leftarrow$ **Recibir**(Chunk_{id})
    $\text{Output}_{\text{local}} \leftarrow$ Descomprimir_Secuencial(Input_{T}) ▷ Genera bytes sin comprimir
    $\text{Len}_{\text{local}} \leftarrow$ longitud($\text{Output}_{\text{local}}$)
    ▷ Cálculo del Offset Global (Suma Prefijo)
    Enviar($\text{Len}_{\text{local}}$) a Concentrador
    Sincronizar()
    Recibir(Todas_$\text{Len}_{\text{local}}$)
    $\text{Offset}_{\text{Final}} \leftarrow$ Calcular_Suma_Prefijo(Todas_$\text{Len}_{\text{local}}$)[id]
    ▷ Cada trabajador conoce dónde escribir
    Escribir_en_Posicion_Global($O_{\text{descomp}}, \text{Offset}_{\text{Final}}, \text{Output}_{\text{local}}$)
    ▷ Escritura dispersa
    Sincronizar()
    ▷ Lógica del Coordinador (Post-procesamiento)
    Finalizar_Archivo(O_{descomp})
    Salida: O_{descomp}
end procedure

Figura 4: Algoritmo paralelo para descomprimir

Tabla 1: Resultados de rendimiento para el archivo de **Alta Repetitividad** (`data_plana.bin`).

Archivo	**Procesos**	**Tiempo Promedio (s)**	**Speedup (S)**
`data_plana.bin`	1	0.1341	1.0000
`data_plana.bin`	2	0.0929	1.4400
`data_plana.bin`	4	0.0723	1.8600
`data_plana.bin`	6	0.0701	1.9100
`data_plana.bin`	8	0.0848	1.5800
`data_plana.bin`	10	0.0892	1.5000
`data_plana.bin`	16	0.0973	1.3800

Tabla 2: Resultados de rendimiento para el archivo de **Media Repetitividad** (`data_malla.bin`).

Archivo	**Procesos**	**Tiempo Promedio (s)**	**Speedup (S)**
`data_malla.bin`	1	0.5299	1.0000
`data_malla.bin`	2	0.3366	1.5700
`data_malla.bin`	4	0.2123	2.5000
`data_malla.bin`	6	0.2413	2.2000
`data_malla.bin`	8	0.2458	2.1600
`data_malla.bin`	10	0.3061	1.7300
`data_malla.bin`	16	0.3196	1.6600

Tabla 3: Resultados de rendimiento para el archivo **Aleatorio** (Peor Caso) (`data_aleatoria.bin`).

Archivo	**Procesos**	**Tiempo Promedio (s)**	**Speedup (S)**
`data_aleatoria.bin`	1	0.5613	1.0000
`data_aleatoria.bin`	2	0.3522	1.5900
`data_aleatoria.bin`	4	0.2324	2.4200
`data_aleatoria.bin`	6	0.2388	2.3500
`data_aleatoria.bin`	8	0.2399	2.3400
`data_aleatoria.bin`	10	0.3187	1.7600
`data_aleatoria.bin`	16	0.3505	1.6000

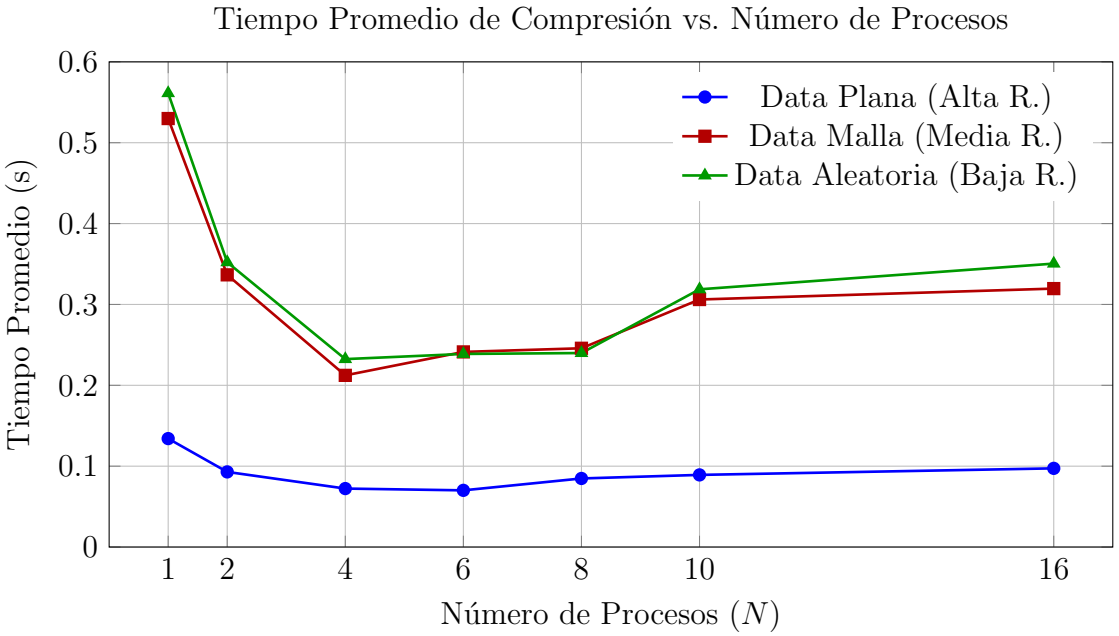

Figura 5: Gráfica del tiempo de compresión promedio en función del número de procesos para los tres tipos de archivos.

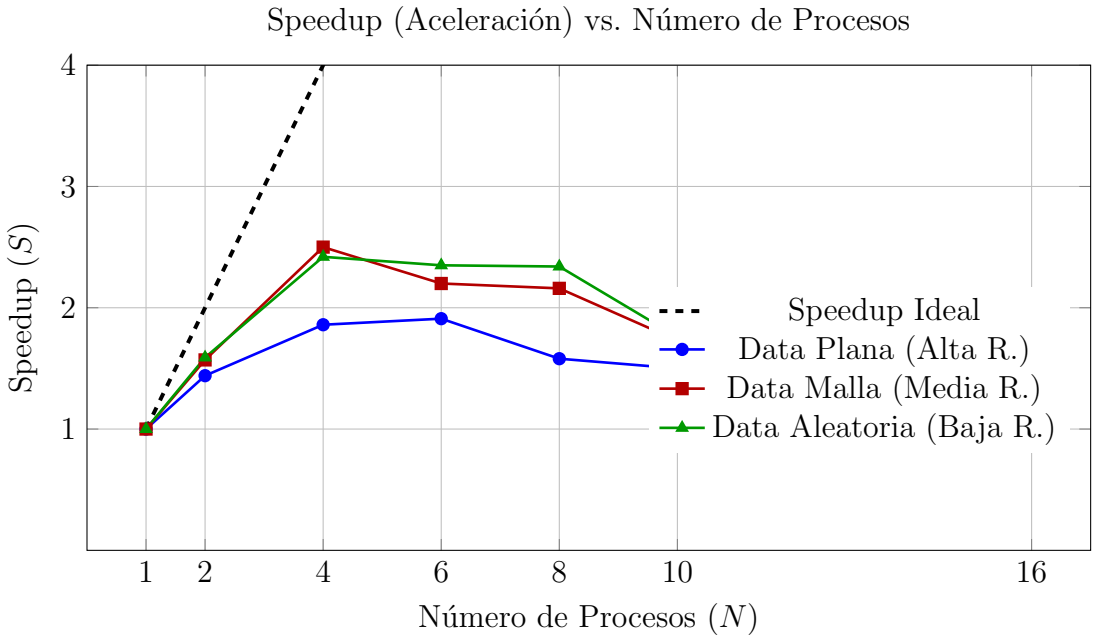

Figura 6: Gráfica de Speedup obtenido. La aceleración ideal es la línea de referencia.

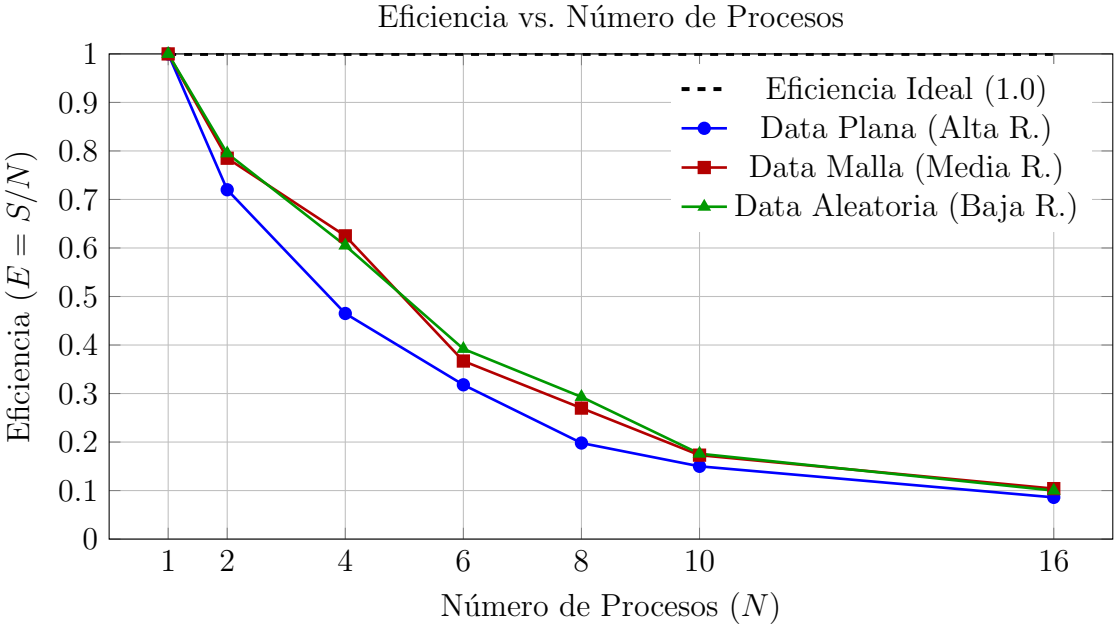

Figura 7: Gráfica de Eficiencia en función del número de procesos.