Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ciencias de la Computación

Programación Multicore

Paralelización del Algoritmo Lempel-Ziv Relativo

Alumno: Joaquín Torres P.
Profesor: José Fuentes
Fecha de entrega: 24 de Junio de 2017
Santiago, Chile

Índice de Contenidos

1. Introducción

Este proyecto busca la paralelización del algoritmo Relativo Lempel-Ziv [1]. Para entender de que se habla en este proyectos se agregará una introducción a como comprime este algoritmo.

1.1. Relative Lempel-Ziv

Relative Lempel-Ziv (RLZ) es una modificación del conocido algoritmo LZ77 [2]. 40 años después de su publicación sigue siendo uno de los compresores más efectivos hasta ahora. Es el núcleo de diferentes compresores en la actualidad, tales como zip, gzip, 7zip, y el formato de imagen GIF.

El algoritmo de compresión LZ77 consiste en dos fases: parseo y codificación. Dado un texto $T[1, N]$, un parseo LZ77 válido es una partición de T en z subcadenas T^i tales que $T = T^1T^2...T^z$, y para todo i en el intervalo $[1, z]$ o puede haber una solo ocurrencia de T^i partiendo estrictamente en una posición menor a $|T^1T^2...T^{i\text{-}1}|$, o T^i es la primera ocurrencia de un caracter simple.

El proceso de codificación representa en cada frase T^i usando un par (p_i, l_i), donde p_i es la posición de la ocurrencia previa a T^i y $l_i = |T^i|$, para frases que no son un solo caracter. Cuando $T^i = \alpha \in \Sigma$, entonces es codificado en un par $(\alpha, 0)$. Llamamos a estas últimas *literal phrases* y a las primeras *copying phrases*. Para encontrar la posición de p_i en una *copying phrases* T^i, el algoritmo usa el diccionario del prefijo del texto $|T^1T^2...T^{i\text{-}1}|$. El dicccionario puede ser implementado usando árboles de sufijos o arreglos de sufijos.

Decodificar el texto comprimido en LZ77 es particularmente simple y rápido: los pares (p_i, l_i) se lee desde la izquierda hacia la derecha, si $l_i = 0$, entonces p_i es interpretado como un caracter y es agregado a la salida. Si $l_i \neq 0$, entonces l_i caracteres son copiados desde la posición p_i y son agregados a la salida.

El algoritmo RLZ usa un diccionario que no es un prefijo del texto. Al contrario, el diccionario es un texto diferente entregado separadamente. Este texto entregado es llamado *referencia* y es más pequeño que el texto comprimido. Por lo que el tamaño del diccionario va a ser menor que el diccionario del algoritmo original LZ77. En el algoritmo RLZ todas las posiciones de p_i de los factores son posiciones de la referencia. Por lo que la elección de la referencia es sumamente importante.

En este proyecto nos encargaremos de paralelizar el algoritmo RLZ [1]. Notar que la referencia y la salida estan concadenadas para entregar el resultado final.

1.2. Conceptos

- Falta del cache: es un estado donde la data pedida por procesar no es encontrada en la memoria cache.
- Speedup: es la proporción entre la ejecución de un programa secuencial y el mismo programa ejecutado con N procesadores paralelos.

2. Soluciones Propuestas

En esta sección se explicarán las tres soluciones propuestas para aumentar el tiempo de ejecución del algoritmo planteado en la introducción.

Para optimizar en el rendimiento del algoritmo, se decidió paralelizar la ejecución de los bloques en los que se divide el texto, paralelizar la ejecución de las particiones de cada bloque y además en cada iteración de cada partición no usar variables globales dentro de la ejecución del algoritmo. En la siguiente figura se puede observar las divisiones del algoritmo, la cual se paralelizarán los bloques y particiones.

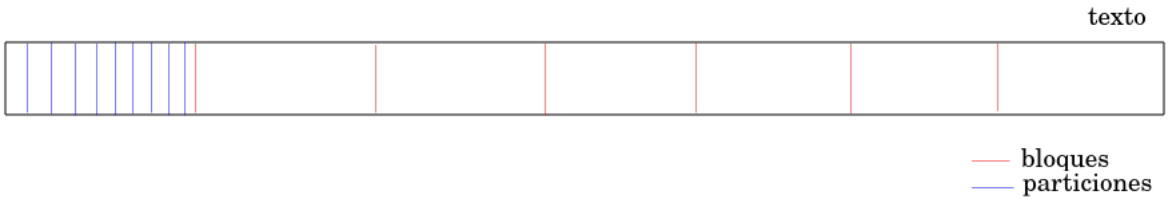

Figura 1: División del texto para paralelizarlo.

2.1. Análisis de complejidad

Para analizar este algoritmo se usará como supuesto que el orden del algoritmo sin paralelizar es conocido. Dado que se paraleliza en bloques y después en particiones los ordenes se suman, además notar que al final en cada rama se deberá ejecutar el algoritmo de compresión en cada partición. Por lo que el algoritmo planteado queda del siguiente orden:

$$\mathcal{O}(algoritmo\_paralelo) = \mathcal{O}(división\_bloques) + \mathcal{O}(división\_particiones) + \mathcal{O}(algoritmo\_secuencial)$$
$$\mathcal{O}(algoritmo\_paralelo) = \mathcal{O}(\log b) + \mathcal{O}(\log p) + \mathcal{O}(algoritmo\_secuencial)$$

Dado lo anterior podemos ver que el orden del algoritmo depende de la parte secuencial, pero dadas las divisiones en bloques y particiones podemos expresar el tamaño del algoritmo secuencial en tamaño del texto divido en bloques y particiones, acotando lo que se tiene que ejecutar de forma secuencial, obteniendo lo siguiente:

$$\mathcal{O}(algoritmo\_paralelo) = \mathcal{O}(\log b) + \mathcal{O}(\log p) + \mathcal{O}(RLZ(texto/(p * b)))$$

3. Implementación

Toda la implementación realizada para este proyecto se encuentra en el repositorio: `https://github.com/juakotorres/ParallelRelativeLempel-Ziv`. Este es una extensión de la implementación secuencial entregada.

Para la primera solución se paralelizó los bloques en los que se separa el texto, esto se explica en el pseudocodigo del Algoritmo 1, notar que el ciclo implementado es un ciclo for paralelo (se implementó usando CilkPlus [3])

Algorithm 1: Paralelización de bloques

Data: Texto a comprimir
Result: Texto comprimido
1 block_len ← se calcula el largo de cada bloque;
2 **for** bloques (en paralelo) hasta el largo del texto **do**
3 $buffer\_len$ ← inicializar largo del buffer en caso de pasar el tamaño del texto;
4 $buffer[block\_len + 1]$; `// inicializar buffer para cada bloque`
5 `ProcessBlock`$(i, ...)$; `// además de pasar el bloque pasar los demás input necesarios para procesar el bloque.`
6 **end**

Para la segunda solución se usó lo que estaba implementado y se paralelizó el procesamiento de cada partición dentro de un bloque. Esto se muestra en el pseudocódigo del Algoritmo 2.

Algorithm 2: Paralelización de particiones

Data: Buffer de un bloque
Result: Buffer comprimido
1 **for** particiones (en paralelo) hasta el largo del buffer **do**
2 $starting\_pos$ ← inicializar donde inicia cada partición;
3 $length$ ← inicializar largo de cada partición;
4 `Chunk_Parse`$(buffer + starting\_pos, length, dictionary, factor)$; `// Se procesa la partición y en factor se guarda un resultado de la compresión.`
5 **end**

Para la optimización en memoria se usaron variables auxiliares para no tener que acceder a variables globales en cada hilo de ejecución mejorando las faltas del cache. Ejemplo al Algoritmo 2 se agregó una variable auxiliar local llamada Factor para no tener que acceder a la lista de factores de las particiones.

4. Experimentos

Para los experimentos se usaron dos conjuntos de datos, el primero de ADN y el segundo de código fuente en Java. Para ambos conjuntos se cambiaron el número de bloques a usar y el número de particiones a usar. Se usó como tamaño de la referencia 10MB, esto fue para todos los experimentos.

4.1. Resultados datos ADN

Primero se mostrarán los resultados obtenidos con los datos de ADN, el tamaño del conjunto de datos es de 400MB.

Tabla 4.1: Datos obtenidos con 100 particiones y 20 bloques, algoritmo secuencial

Tiempo (s)	Mejor tiempo
284.742684	240.06467
246.259511	
240.06467	
285.773509	
285.432134	

Tabla 4.2: Datos obtenidos con 100 particiones y 20 bloques, algoritmo paralelo

	Tiempo (s)	Cache misses	Radio de Compresión	Speedup
	36.214325	9042045386	0.873362445414847	6.62899750305991
	40.415557	8786854840	0.873362445414847	5.9399074965118
	44.300266	8898303363	0.873362445414847	5.4190345042172
	42.921386	9103771575	0.873362445414847	5.59312483525113
	39.676313	8775327968	0.873362445414847	6.05057909488717
Mediana	40.415557	8898303363	0.873362445414847	5.9399074965118

Tabla 4.3: Datos obtenidos con 100 particiones y 10 bloques, algoritmo secuencial

Tiempo (s)	Mejor tiempo
286.510343	245.092304
288.479197	
245.092304	
246.550035	
245.683398	

Tabla 4.4: Datos obtenidos con 100 particiones y 10 bloques, algoritmo paralelo

	Tiempo (s)	Cache misses	Radio de Compresión	Speedup
	43.593543	9035118203	0.873362445414847	5.62221574878647
	44.523269	8877436100	0.873362445414847	5.5048137637872
	37.965025	9032722392	0.873362445414847	6.45573930216034
	39.685549	8994156638	0.873362445414847	6.17585771586529
	43.158951	8874263540	0.873362445414847	5.6788290336343
Mediana	43.158951	8994156638	0.873362445414847	5.6788290336343

Tabla 4.5: Datos obtenidos con 200 particiones y 20 bloques, algoritmo secuencial

Tiempo (s)	Mejor tiempo
245.464193	244.637318
244.637318	
248.154475	
253.798293	
247.681748	
248.681748	

Tabla 4.6: Datos obtenidos con 200 particiones y 20 bloques, algoritmo paralelo

	Tiempo (s)	Cache misses	Radio de Compresión	Speedup
	39.24455	8523465845	0.873362445414847	6.23366347683946
	39.442169	8409455287	0.873362445414847	6.2024306523305
	39.507749	8579904676	0.873362445414847	6.19213506697129
	40.15033	8740813000	0.873362445414847	6.09303380569973
	39.14626	8382810762	0.873362445414847	6.24931520916685
	39.848885	8581214164	0.873362445414847	6.13912579988123
Mediana	39.474959	8551685260	0.873362445414847	6.1972828596509

4.2. Resultados datos código fuente

Ahora se mostrarán los datos obtenidos son los código fuente, el tamaño del conjunto de datos es de 202MB.

Tabla 4.7: Datos obtenidos con 100 particiones y 20 bloques, algoritmo secuencial

Tiempo (s)	Mejor tiempo
115.211399	114.941896
127.830516	
116.684714	
114.941896	
115.644068	

Tabla 4.8: Datos obtenidos con 100 particiones y 20 bloques, algoritmo paralelo

	Tiempo (s)	Faltas del caché	Radio de Compresión	Speedup
	21.731207	4031571791	0.47196261682243	5.28925503309595
	21.031135	4023950900	0.47196261682243	5.46532063057938
	20.618367	4009483021	0.47196261682243	5.57473324633323
	20.564965	3989940070	0.47196261682243	5.58920941513881
	20.918137	4059825721	0.47196261682243	5.4948438285876
Mediana	20.918137	4023950900	0.47196261682243	5.4948438285876

Tabla 4.9: Datos obtenidos con 100 particiones y 10 bloques, algoritmo secuencial

Tiempo (s)	Mejor tiempo
115.051335	114.876583
116.120681	
114.876583	
115.806615	
115.065829	

Tabla 4.10: Datos obtenidos con 100 particiones y 10 bloques, algoritmo paralelo

	Tiempo (s)	Cache misses	Radio de Compresión	Speedup
	21.309138	4025265533	0.47196261682243	5.39095401231153
	21.030081	4054919669	0.47196261682243	5.46248885108907
	20.724409	3997008992	0.47196261682243	5.54305712650238
	20.911063	4052995394	0.47196261682243	5.49357930775686
	20.84145	4029487717	0.47196261682243	5.51192853664212
Mediana	20.911063	4029487717	0.47196261682243	5.49357930775686

Tabla 4.11: Datos obtenidos con 200 particiones y 20 bloques, algoritmo paralelo

Tiempo (s)	Mejor tiempo
115.515028	114.984652
114.984652	
115.20431	
116.382001	
115.416972	

Tabla 4.12: Datos obtenidos con 200 particiones y 20 bloques, algoritmo paralelo

	Tiempo (s)	Cache misses	Radio de Compresión	Speedup
	20.7687	4037220274	0.47196261682243	5.53643954604766
	20.473259	4045125462	0.47196261682243	5.61633357932902
	21.092907	4053114165	0.47196261682243	5.45134210282158
	20.695533	4044601021	0.47196261682243	5.556013077798
	20.283789	4040825008	0.47196261682243	5.66879550955692
Mediana	20.695533	4044601021	0.47196261682243	5.556013077798

5. Discusión

De los resultados obtenidos a partir de los datos de ADN y código fuente se puede notar que los radios de compresión son malos, ya que no comprimen en menor tamaño del archivo original, se probó si aumentar la referencia aumentaba el radio de compresión y comprobó la teoría disminuyendo hasta un 50 % el tamaño original.

Para la eficiencia lograda a partir del paralelismo, se observar de los datos obtenidos que se alcanza un *Speedup* de hasta 6, lo que nos dice que con 12 cores usados se logró que la compresión sea 6 veces más rápida, lo que nos da un buen indicio de que paralelizar no siempre aumenta con respecto a los cores, pero se logra un aumento en el tiempo considerable.

No sé logró identificar porque las faltas del caché eran tan grandes, son del orden de 10 mil millones, podría ser que como al paralelizar sin ordenar las tareas por hilo las particiones que se usaban eran cercanas en memoria lo que producía muchas faltas de cache al pedir la siguiente partición que podría estar lejos en memoria. Se intentó dividir los bloques por hilo, pero se logró analizar de que si se hacer esa optimización las particiones deberían ser secuenciales para que no se generaran tantas faltas del caché.

6. Conclusiones

De lo aprendido en este proyecto fue lo siguiente, paralelizar no es una tarea simple, pero se puede lograr una eficiencia notable en ciertos problemas. En este caso se logró disminuir el tiempo de una compresión de datos 6 veces con respecto a la original.

También no se pudo lograr la escritura de la compresión paralela estuviera en el orden correcto, pero al comprimir y descomprimir se pudo ver que el tamaño era el mismo y que los archivos no eran tan diferentes, y las diferencias eran en bloques, lo cual es lógico siendo que se paralelizó y las escrituras se hacen en las particiones.

Se aprendió de que los experimentos son importantes a la hora de probar los algoritmos, sirvió para saber si el algoritmo funcionaba correctamente. Además un aprendizaje importante fue que no se debió probar con archivos muy grandes para ver si funcionaba correctamente el algoritmo, al final se probó con archivos pequeños y se logró hacer los experimentos que se esperaban para este proyecto.

Finalmente, CilkPlus es una herramienta valiosisima que servirá para oprimizar problemas a futuro que sean paralelizables.

7. Referencias

1. Daniel Valenzuela. Chico: A compressed hybrid index for repetitive collections. In Proceedings of the 15th International Symposium on Experimental Algorithms - Volume 9685, SEA 2016, pages 326–338, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

2. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, 23(3):337–343, May 1977.

3. Herramienta para paralelizar código. `https://www.cilkplus.org/`