

Crusher Simulator: simulatore del framework CRUSHER per la generazione di algoritmi

Raffaele D'Arco, Sabato De Gregorio, Egidio Giacoia
Dipartimento di Informatica
Università degli Studi di Salerno

Reference: SPDP - An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data

Steven Claggett, Sahar Azimi, and Martin Burtscher

Department of Computer Science, Texas State University - San Marcos, TX 78666, USA

1. Introduzione

Le applicazioni scientifiche producono, memorizzano e trasferiscono grandi mole di dati in virgola mobile a precisione singola o doppia. Per questo motivo, la compressione gioca un ruolo fondamentale in questo contesto per ridurre la mole dei dati.

Molti utenti di informatica scientifica si avvalgono di framework come HDF5 per la gestione dei propri dati. Alcuni di questi framework supportano "filtri" di compressione; tuttavia, tali filtri possono essere impiegati solo se sono efficaci e devono essere lossless, in modo che possano essere tranquillamente utilizzati in qualsiasi contesto. A seconda dell'applicazione, i dati scientifici vengono in genere archiviati in formato a virgola mobile IEEE 754 a precisione singola o doppia. Gli utenti non vogliono dover selezionare filtri diversi a seconda della precisione dei dati, piuttosto, vorrebbero avere un unico algoritmo che comprima in maniera lossless e ottimale i dati in virgola mobile indipendentemente dalla precisione utilizzata.

Per ottenere tale algoritmo, è stato utilizzato il framework **CRUSHER**^[1] per generare sistematicamente tanti algoritmi a partire da un insieme di componenti. Ogni componente implementa una trasformazione dei dati e può operare a granularità di word e byte. CRUSHER ha quindi eseguito una ricerca esaustiva per determinare il miglior algoritmo a quattro componenti testando un insieme di 13 dataset a singola precisione e 13 a doppia precisione. L'algoritmo risultante è stato **SPDP**^[2], che è un'abbreviazione di "Singola Precisione Doppia Precisione". È nuovo di zecca e non è simile agli algoritmi di compressione precedenti: SPDP infatti offre il più alto rapporto di compressione su undici dei 26 set di dati testati, solo Zstd risulta essere migliore. In media, SPDP supera Blosc, bzip2, FastLZ, LZ4, LZO e Snappy di almeno il 30% in termini di rapporto di compressione; tuttavia, tende ad essere più lento.

L'obiettivo del nostro lavoro è stato quello di simulare il framework CRUSHER e aggiungere altre componenti per tentare di aumentare il rapporto di compressione.

La struttura del documento è la seguente:

- Nella sezione 2 verrà spiegato il simulatore Crusher Simulator del framework CRUSHER con le relative componenti utilizzate.
- Nella sezione 3 presentiamo e discutiamo i risultati sperimentali.
- La sezione 4 si conclude con una sintesi e i possibili sviluppi futuri.

2. Crusher Simulator

Nella seguente sezione spieghiamo il tool *Crusher Simulator* che, come può suggerire il nome, consiste nel simulare il framework *CRUSHER* di M. Burtsher, S. Claggett ed S. Azimi.



Banner iniziale mostrato all'avvio dello script CrusherSimulator.py

2.1 DESCRIZIONE

Il tool Crusher Simulator, in accordo alla logica del framework CRUSHER, è in grado di fornire il migliore algoritmo di compressione lossless possibile, dato un set di dati; infatti, tramite una ricerca esaustiva, è in grado di identificare delle componenti algoritmiche predeterminate all'interno di uno spazio di ricerca e le concatena, dando in output l'algoritmo migliore.

Lo spazio di ricerca viene determinato a partire da algoritmi di compressioni esistenti, i quali vengono rotti nelle loro parti costitutive e generalizzate.

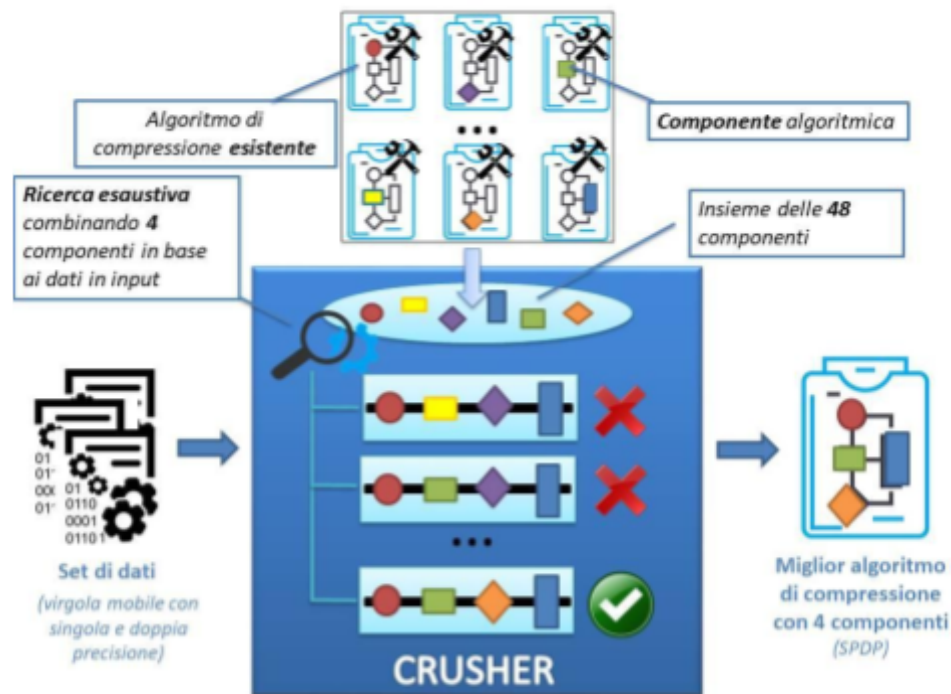


Illustrazione logica del funzionamento di CRUSHER

Le componenti ottenute vengono implementate utilizzando un'interfaccia comune, in modo che ogni componente possa ricevere un blocco di dati come input, che lo trasforma in un blocco di output di dati. In questo modo si possono combinare le componenti in qualsiasi modo, formando quindi una catena, consentendo la generazione di un numero elevato di candidati all'algoritmo di compressione da un piccolo insieme di componenti.

Il framework CRUSHER esegue una ricerca esaustiva per determinare il miglior algoritmo a quattro componenti in questo spazio di ricerca -- di cui uno di questi è proprio SPDP. Tuttavia, Crusher Simulator permette di scegliere arbitrariamente il numero di componenti da concatenare.

2.2 STRUTTURA DEL PROGETTO

La cartella principale del progetto è **Crusher_Simulator**, in cui all'interno troviamo:

- **CrusherSimulator.py**: script Python che effettua i processi di compressione, verifica e decompressione;
- **tmp (tmp/tmp_compression, tmp/tmp_decompression)**: cartella che contiene tutti i file "temporanei" che vengono creati quando si passa da una componente ad un'altra nella fase di combinazioni. Include anche dei file di log;
- **file_test**: contiene tutti i dataset dei file di test utilizzati;
- **components**: repository degli eseguibili compilati delle componenti;
- **components/source**: cartella che contiene i file sorgenti delle componenti;
- **report.log**: file generato ad ogni fine processo di compressione/decompressione, contenente informazioni come tempo, ratio compression, space savings ecc.. di tutte le possibili combinazioni provate e della miglior combinazione trovata.

test_file	11/06/2019 15:05	Cartella di file	
components	11/06/2019 16:47	Cartella di file	
tmp	11/06/2019 17:39	Cartella di file	
CrusherSimulator.py	11/06/2019 16:56	Python File	25 KB
file_compresso.crusher	11/06/2019 17:40	File CRUSHER	1 KB
report.log	11/06/2019 17:40	Documento di testo	17 KB

Contenuto della cartella principale Crusher_Simulator

N.B. Il contenuto di *tmp* e *report.log* viene sovrascritto ad ogni esecuzione del tool.

2.3 SETUP ENVIRONMENT

Il tool non è altro che uno script in **Python** che può essere lanciato utilizzando l'interprete Python (versione usata 2.7.15rc1) . Lo scopo dell'utilizzo è del tutto didattico e dimostrativo, per cui è possibile che possono essere presenti debolezze software a livello di sicurezza. Inoltre, è pensato per essere eseguito in sistemi **Unix-based**.

Il tool necessita del modulo **tqdm** per gestire la progress bar durante il processo di compressione e decompressione. Per la sua installazione, si usa il seguente comando:

```
~$ pip install tqdm
```

Va verificato inoltre che ogni cartella e il file **CrusherSimulator.py** di **Crusher_Simulator** (inclusa quest'ultima) abbiano i permessi di scrittura, lettura e accesso/esecuzione (basta usare i comandi **chmod** e **chown** per assegnare i privilegi e proprietari dei file).

Gli eseguibili delle componenti sono stati ottenuti compilando i relativi sorgenti contenuti in **Crusher_Simulator/components** utilizzando i seguenti comandi:

Per gli eseguibili: **crotate**, **drotate**, **cdim**, **ddim**, **clnvs**, **dlnvs**, **clz**, **dlz**

```
~$: gcc -O3 -o [component_exe] [component_encode/decode].c
```

Per gli eseguibili invece: **crle**, **drle**

```
~$: gcc -o [component_exe] [component_encode/decode].c
```

Per **bwt**^[3] (**cbwt** e **dbwt**) e **huffman**^[4] (**chuffman**, **dhuffman**), invece bisogna usare il comando **make** (progetti presi da **github**)

N.B. La fase di compressione è estremamente onerosa in termini di tempo di esecuzione e risorse di sistema: la cartella **/tmp/tmp_decompression** a fine fase può arrivare a pesare circa decine di GB per file di input di dimensioni circa 150 MB

2.3 LE COMPONENTI

Il repository di componenti disponibili è diviso in due categorie:

- 1) **Shifters** (*mescolatori*): non modificano la dimensione dei blocchi di dati
 - **Rotate**: è un'operazione simile allo shift, che si distingue però dal fatto che i bit posizionati su un'estremità vengono rimessi all'altra estremità.
 - **Trasformata di Barrow-Wheelers**: accetta una sequenza di carattere, dove nessuno di questi cambia di valore dal momento che la trasformazione permuta soltanto l'ordine dei caratteri. Se la stringa originale contiene molte ripetizioni di certe sottostringhe, allora nella stringa trasformata troveremo diversi punti in cui lo stesso carattere si ripete tante volte.
 - **Dim** [*componente estrapolata da SPDP*]: accetta un parametro n che specifica la dimensionalità e raggruppa i valori di conseguenza. Per esempio, una dimensione di tre della sequenza lineare $x_1, y_1, z_1, x_2, y_2, z_2, x_3, y_3, z_3$ viene modificata in $x_1, x_2, x_3, y_1, y_2, y_3, z_1, z_2, z_3$ usando $n = 2, 4, 8$ e 12 . In questo caso è stato usato il valore 8 .
 - **Lnvs** [*componente estrapolata da SPDP*]: accetta due parametri k ed n . Sottrae l'ultimo n -esimo valore da quello corrente ed emette il residuo; se $k = 's'$, viene utilizzata la sottrazione aritmetica. Se invece $k = 'x'$, viene utilizzata la sottrazione bitwise (xor). In entrambi i casi,
 - Sottrae l'ultimo ennesimo valore dal valore corrente ed emette il residuo. Se $k = 's'$, viene utilizzata la sottrazione aritmetica. Se $k = 'x'$, viene utilizzata la sottrazione bit a bit (xor). In entrambi i casi, n varia $n = 1, 2, 3, 4, 8, 12, 16, 32$ e 64 . In questo caso noi abbiamo usato il valore $n=2$ e $k='s'$.
- 2) **Compressors** (*compressori*): comprimono la lunghezza di un blocco di dati.
 - **Run-Length**: cerca nei dati da comprimere una serie di elementi uguali e la sostituisce con un solo elemento, quindi un carattere speciale e infine il numero di volte che esso va ripetuto.
 - **LZ77** (una sua variante, estrapolata da SPDP): Usa una tabella hash 32768-entry per identificare le ultime occorrenze precedenti del valore corrente, quindi controlla se i valori n immediatamente precedenti a tali posizioni corrispondono ai valori n appena prima della posizione corrente. In caso contrario, viene emesso solo il valore corrente e il componente avanza al valore successivo. Se i valori n corrispondono, il componente conta quanti valori successivi al valore corrente corrispondono ai valori dopo quella posizione. La lunghezza della sottostringa corrispondente viene emessa e il componente avanza di molti valori. Consideriamo $n = 6$ (per evitare corrispondenze errate che comportano l'emissione di conteggi zero, che espandono invece di comprimere i dati)
 - **Huffman**: si basa sul principio di trovare il sistema ottimale per codificare stringhe, basato sulla frequenza relativa di ciascun carattere. La codifica di Huffman usa un metodo specifico per scegliere la rappresentazione di ciascun simbolo, risultando in un codice senza prefissi che esprime il carattere più frequente nella maniera più breve possibile.

2.4 LE MODALITÀ DI UTILIZZO

In questa sezione sono mostrate le possibili interazioni con il tool.

2.4.1 Encode mode

```
~$ python CruscherSimulator.py encode  
[file_input_to_compress] [file_output_compressed] [#_of_components_for_chain]
```

Effettua la compressione lossless di un file dato in input `[file_input_to_compress]`, utilizzando le componenti disponibili. Il tool effettua una ricerca esaustiva andando a concatenare un certo numero di componenti `[#_of_components_for_chain]`, determina il miglior algoritmo tra quelli candidati cioè che ha il miglior compression ratio per il dato file in input, dà in output il file compresso `[file_output_compressed]`.

Il file di output compresso avrà l'estensione **.crusher** che verrà aggiunta automaticamente dal tool; inoltre nel <<footer>> verrà scritta una stringa che identifica la catena di componenti utilizzata nel processo di compressione (e.g. l'algoritmo di compressione).

Alla fine del processo viene creato anche il file **report.log**, che contiene varie informazioni sulla compressione appena conclusa.

N.B. Ogni componente è implementata in modo che possa ricevere un blocco di dati come input, che trasforma in un blocco di output di dati per un'altra componente. Durante il processo di compressione, tutti i file intermedi (algoritmi candidati, e.g. catena di componenti) saranno contenuti nella cartella `/tmp/tmp_compression`.

In dettaglio

Supponendo di aver scelto di concatenare **#_of_components_for_chain** componenti, il processo di compressione andrà a concatenare le componenti in maniera tale che le prime **#_of_components_for_chain - 1** sono shifters, mentre l'n-esima componente è sempre un compressore.

Supponendo di avere a disposizione **#_shifters** componenti shifters e **#_compressors** componenti compressors, il numero totale di possibili combinazioni in cui possiamo legare tra loro le componenti risulta:

$$\begin{array}{lcl} \text{Disposizioni_Semplici}(\text{\#_shifters}, \text{\#_of_components_for_chain} - 1) & * & \\ \text{\#_compressors} & & = \\ \hline \text{\#_combinations} \end{array}$$

Nel nostro repository abbiamo che `#_shifters = 4` e `#_compressors = 3`, per cui `#_combinations = 72` se scegliamo `#_of_components_for_chain = 4`.

Il tool dunque effettuerà, in maniera sequenziale, la compressione del file dato in input utilizzando tutte le combinazioni ottenute, generando 72 algoritmi candidati (con 4 componenti per volta); di conseguenza abbiamo 72 file di output compressi. Ciascuno di questi, conterrà al suo interno come <<footer>> una stringa che identifica la catena di componenti utilizzata (e.g. l'algoritmo di compressione).

Tra tutti i file di output si verifica quale tra questi ha taglia minore e di conseguenza si determina il miglior algoritmo possibile tra quelli candidati per quel dato file di input.

2.4.2 Decode mode

```
~$ python CruscherSimulator.py decode  
[file_input_to_decompress][file_output_decompressed]
```

Effettua la decompressione di un file dato in input `[file_input_to_decompress]`, andando ad utilizzare la catena delle componenti in ordine inverso con la quale è avvenuto il processo di compressione. In output verrà generato il file decompresso `[file_output_decompressed]`.

Alla fine del processo viene creato anche il file **report.log**, che contiene varie informazioni sulla decompressione appena conclusa.

N.B. Il processo di decompressione si avvia solo se il file in input ha come estensione **.crusher**.

N.B. Ogni componente è implementata in modo che possa ricevere un blocco di dati come input, che trasforma in un blocco di output di dati per un'altra componente. Durante il processo di compressione, tutti i file intermedi (algoritmi candidati, e.g. catena di componenti) saranno contenuti nella cartella `/tmp/tmp_decompression`.

In dettaglio

Avviato il processo di decompressione, il tool andrà a leggere dal <<footer>> del file in input una stringa che identifica la catena di componenti utilizzata nel processo di compressione (e.g. l'algoritmo di compressione), dopodichè invertirà l'ordine delle componenti ottenendo l'algoritmo di decompressione, e darà in output il file decompresso.

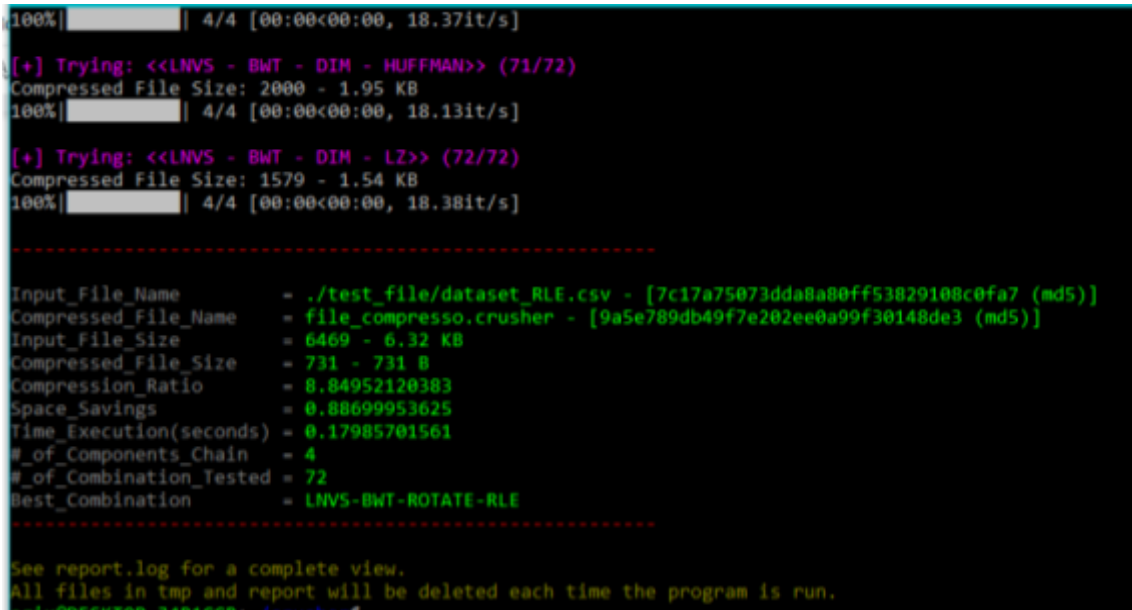
2.4.3 Verify mode

```
~$ python CruscherSimulator.py verify [file_1] [file_2]
```

Verifica se due file `[file_1]` e `[file_2]` hanno la stessa firma hash; la funzione di hashing utilizzata è **MD5**.

2.5 STRUTTURA DEL REPORT

Al termine della modalità di compressione (encode mode) o decompressione (decode mode) viene generato un file di log **report.log** che contiene vari parametri sul relativo processo sia per le varie combinazioni testate, sia sulla miglior combinazione risultante.



```
100%|██████████| 4/4 [00:00<00:00, 18.37it/s]
[+] Trying: <<LNVS - BWT - DIM - HUFFMAN>> (71/72)
Compressed File Size: 2000 - 1.95 KB
100%|██████████| 4/4 [00:00<00:00, 18.13it/s]
[+] Trying: <<LNVS - BWT - DIM - LZ>> (72/72)
Compressed File Size: 1579 - 1.54 KB
100%|██████████| 4/4 [00:00<00:00, 18.38it/s]

-----

Input_File_Name      = ./test_file/dataset_RLE.csv - [7c17a75073dda8a80ff53829108c0fa7 (md5)]
Compressed_File_Name = file_compresso.crusher - [9a5e789db49f7e202ee0a99f30148de3 (md5)]
Input_File_Size      = 6469 - 6.32 KB
Compressed_File_Size  = 731 - 731 B
Compression_Ratio     = 8.84952120383
Space_Savings         = 0.88699953625
Time_Execution(seconds) = 0.17985701561
#_of_Components_Chain = 4
#_of_Combination_Test = 72
Best_Combination      = LNVS-BWT-ROTATE-RLE

-----

See report.log for a complete view.
All files in tmp and report will be deleted each time the program is run.
```

Screenshot della terminazione dell'encode mode

Le informazioni riportate dal miglior algoritmo sono:

- **Input_File_Name:** nome del file di input e relativo hash in md5;
- **Output_File_Name:** nome del file di output e relativo hash in md5;
- **Input_File_Size:** la dimensione del file di input;
- **Output_File_Size:** la dimensione del file di output;
- **Compression_Ratio:** Il rapporto di compressione ottenuto dall'algoritmo;
- **Space_Savings:** il risparmio di spazio ottenuto dall'algoritmo;
- **Time_Execution:** tempo di esecuzione espresso in secondi dell'algoritmo;
- **#_of_Components_Chain:** il numero di componenti concatenate che definiscono l'algoritmo;
- **#_of_Combination_Test:** il numero di combinazioni effettuate (algoritmi candidati) [solo nel report di compressione]
- **Best_Combination:** miglior combinazione scelta (algoritmo migliore) [solo nel report di compressione]

Le informazioni riportate da ciascun algoritmo candidato sono:

- La catena di componenti utilizzate che determinano l'algoritmo candidato;
- **Compressed_File_Size:** la dimensione del file di output compresso;
- **Compression_Ratio:** Il rapporto di compressione ottenuto dall'algoritmo candidato;
- **Space Savings:** Il rapporto di compressione ottenuto dall'algoritmo candidato;
- **Time_Execution:** tempo di esecuzione espresso in secondi dell'algoritmo candidato;

I parametri che vengono calcolati per ciascun algoritmo candidato sono:

- **Compression Ratio:** Il rapporto di compressione dei dati è definito come il rapporto tra la dimensione non compressa e la dimensione compressa.

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

- **Space Savings:** il risparmio di spazio è definito come la riduzione delle dimensioni rispetto alla dimensione non compressa.

$$\text{Space Savings} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}$$

- **Time Execution:** il tempo di esecuzione è espresso in secondi, ed è calcolato come la differenza tra il tempo di inizio della compressione/decompressione e il tempo di terminazione del processo.



```
21 #####
22 #####
23 #####
24 #####
25 #####
26 #####
27 #####
28
29 TIMESTAMP = 2019-06-11 17:03:58
30
31
32
33
34
35
36
37
38
39
40
41 [+] Trying: <<ROTATE - DIM - BWT - RLE>> (1/72)
42 Compressed File Size: 869184 - 848.81 KB
43 Compression Ratio: 2.81447886754
44 Space Savings: 0.644694436496
45 Time Execution (seconds): 0.22936201096
46
47 [+] Trying: <<ROTATE - DIM - BWT - HUFFMAN>> (2/72)
48 Compressed File Size: 1017461 - 993.6 KB
49 Compression Ratio: 2.40434183071
50 Space Savings: 0.584085762171
51 Time Execution (seconds): 0.21213607652
52
53 [+] Trying: <<ROTATE - DIM - BWT - LZ>> (3/72)
54 Compressed File Size: 1322514 - 1.17 MB
55 Compression Ratio: 2.00104047888
56 Space Savings: 0.500259984466
57 Time Execution (seconds): 0.15405988493
58
```

Report dell'encode mode - parte iniziale

```

450 Compressed File Size: 846630 ~ 826.75 KB
451 Compression Ratio: 2.88545584258
452 Space Savings: 0.653914074316
453 Time Execution (seconds): 26.5350730419
454
455 [+] Trying: <<LHVS - BWT - DIM - RLE>> (70/72)
456 Compressed File Size: 2471136 ~ 2.36 MB
457 Compression Ratio: 0.989949561659
458 Space Savings: -0.0101524751666
459 Time Execution (seconds): 26.5593209267
460
461 [+] Trying: <<LHVS - BWT - DIM - HUFFMAN>> (71/72)
462 Compressed File Size: 958521 ~ 936.06 KB
463 Compression Ratio: 2.55216108985
464 Space Savings: 0.608175203368
465 Time Execution (seconds): 26.6456861496
466
467 [+] Trying: <<LHVS - BWT - DIM - LZ>> (72/72)
468 Compressed File Size: 1791618 ~ 1.71 MB
469 Compression Ratio: 1.34541383264
470 Space Savings: 0.267621305645
471 Time Execution (seconds): 26.4851610661
472
473 -----
474 Input_File_Name      = ./test_file/num_plasma.trace.fpc - [e21c99705679dbff23fbd9f4af0e583 (md5)]
475 Compressed_File_Name = num_plasma_compr.crusher - [189ef36acdb7f5120cb15f5f7bd23c1f (md5)]
476 Input_File_Size     = 2446300 ~ 2.33 MB
477 Compressed_File_Size = 249304 ~ 243.46 KB
478 Compression_Ratio    = 9.81251805025
479 Space_Savings        = 0.898089359441
480 Time_Execution(seconds) = 0.0737779140472
481 #_of_Components_Chain = 4
482 #_of_Combination_Test = 72
483 Best_Combination      = DIM-ROTATE-LHVS-LZ
484 -----
485 See report.log for a complete view.
486 All files in tmp and report will be deleted each time the program is run

```

Report dell'encode mode - parte finale

3. Test e risultati

I test sono stati condotti su una macchina di Amazon Web Service, dunque con una CPU Intel Xeon E5-2676 v3, con 8GB di RAM dedicati, e il Sistema Operativo utilizzato è Ubuntu Server 18.04.

I file di test utilizzati sono stati presi dal sito dell'Università del Texas^[5] e sono gli stessi utilizzati nel paper di Steven Claggett, Sahar Azimi e Martin Burtscher; i dataset forniti sono compressi utilizzando il compressore FPC^{[6][7]}, tuttavia in bibliografia viene lasciato il riferimento su come estrarli.

La natura dei dataset (double precision) riguarda misurazioni da strumenti scientifici, risultato di simulazioni numeriche e messaggi numerici inviati da un nodo in un sistema parallelo che eseguono applicazioni NAS Parallel Benchmark (NPB) e ASCI Purple.

La misura che è stata presa in considerazione è il rapporto di compressione, i cui valori risultanti sono stati confrontati con quelli di SPDP; inoltre sono state riportate anche le combinazioni delle componenti che hanno permesso tale risultato di compression ratio.

La tabella seguente contiene i dati relativi ai test effettuati:

FILE	TAGLIA INIZIALE (MB)	TAGLIA COMPRESSA (MB)	COMPRESSION RATIO	SPDP (lv9) COMPRESSION RATIO	COMPONENTI
obs_error	56.28	36.99	1.60	1.61	DIM-LNVS-ROTATE-LZ
obs_info	18.05	9.29	1.94	1.95	ROTATE-DIM-LNVS-LZ
obs_spitzer	189	174.45	1.08	0.98	DIM-ROTATE-BWT-HUFFMAN
obs_temp	38.08	36.65	1.04	1.03	ROTATE-DIM-LNVS-LZ
num_plasma	33.46	982.48 (KB)	34.878	33.17	LNVS-ROTATE-DIM-LZ
num_comet	102.38	86.42	1.18	1.16	ROTATE-DIM-BWT-LZ
num_brain	135.27	114.21	1.18	1.20	DIM-ROTATE-LNVS-LZ
num_control	152.12	144.72	1.05	1.01	DIM-BWT-ROTATE-HUFFMAN
msg_bt	197.57	196.44	1.00	1.33	BWT-LNVS-ROTATE-HUFFMAN
msg_lu	185.13	149.32	1.23	1.26	DIM-ROTATE-LNVS-LZ
msg_sweep3d	119.91	42.22	2.84	3.01	DIM-ROTATE-LNVS-LZ

Dalla tabella si evince che, per alcuni dataset, il rapporto di compressione risulta essere migliore con il nostro tool, mentre in altri i valori non si discostano in modo significativo da SPDP. Questo perché, come suggerito negli sviluppi futuri del paper di SPDP, sono state aggiunte altre componenti algoritmiche.

Quindi, dati i risultati, si può affermare che l'obiettivo del nostro lavoro, cioè simulare il framework CRUSHER, è stato raggiunto. Inoltre, notiamo come il compressore RLE (RUN-LENGTH) non è stato mai selezionato; la motivazione sta nel fatto che questi dataset non presentano molte ripetizioni di numeri consecutivi simili.

Riportiamo anche i tempi di esecuzione, relativi alla compressione, per ciascun dataset

Dataset	Tempo di esecuzione (in secondi)
obs_error	0,62
obs_info	0,19
obs_spitzer	31,36
obs_temp	0,53
num_plasma	0,32
num_comet	131,85
num_brain	2,09
num_control	30,90
msg_bt	30,49
msg_lu	2,40
msg_sweep3d	1,20

Concludiamo il documento vedendo i possibili sviluppi futuri.

4. Conclusioni e Sviluppi futuri

In questo lavoro di gruppo è stato simulato il framework CRUSHER, arricchendolo con altre componenti, e sono stati confrontati i valori con quelli dell'algoritmo SPDP.

Sono state aggiunte altre componenti algoritmiche al framework CRUSHER in modo da ottenere ulteriori algoritmi, i quali risultano avere un miglior rapporto di compressione rispetto ai test di SPDP per i dataset dati.

Nel lavoro futuro si potrebbe testare il tool in altri domini (file di immagini per esempio) per verificare se è possibile avere una compressione più efficiente. Inoltre, si potrebbero aggiungere nuove combinazioni, ripetendo una componente più volte, per poter quindi verificare la possibilità di migliorare il rapporto di compressione. Per esempio, date le componenti ABCD, verranno valutate le combinazioni ABAC e ABAD.

Infine, si potrebbero studiare altri algoritmi di compressione, come Zstd, ed estrarre le parti fondamentali in modo da sintetizzare algoritmi ancora migliori.

5. Bibliografia

- [1] Real-Time Synthesis of Compression Algorithms for Scientific Data - Martin Burtscher:
<https://userweb.cs.txstate.edu/~burtscher/papers/sc16.pdf>
- [2] SPDP: An Automatically Synthesized Lossless Compression Algorithm for Floating-Point Data
Steven Claggett, Sahar Azimi, and Martin Burtscher:
<https://userweb.cs.txstate.edu/~mb92/papers/dcc18.pdf>
- [3] An ANSI C implementation of the Burrows-Wheeler transformation (BWT) -
MichaelDipperstein: <https://github.com/michaeldipperstein/bwt>
- [4] Huffman encoder/decoder - intended for educational purposes - drichardson:
<https://github.com/drichardson/huffman>
- [5] Scientific IEEE 754 64-Bit Double-Precision Floating-Point Datasets:
<https://userweb.cs.txstate.edu/~burtscher/research/datasets/FPdouble/>
- [6] High Throughput Compression of Double-Precision Floating-Point Data - Martin Burtscher and
Paruj Ratanaworabhan: <https://userweb.cs.txstate.edu/~burtscher/papers/dcc07a.pdf>
- [7] FPC lossless compressor/decompressor for IEEE 754 64-bit double-precision floating-point data
- Martin Burtscher: <https://userweb.cs.txstate.edu/~burtscher/research/FPC/>