SQream - Home Assignment – Compressing and Decompressing arrays of integers

Michael Tatevosov

Compression Algorithms:

- **Run-Length Encoding (RLE):**

  Replaces repeated consecutive elements with a count and single instance of the element.

  The array we get consists of random values, and random values spread equally so there will be no benefit.

- **Delta Encoding:**

  Delta encoding stores the difference between consecutive elements instead of the absolute values.

  As the values are 0 to 100 it won’t make them small enough so this method is also not fitting here.

- **Huffman Coding:**

  A variable-length prefix coding algorithm that assigns shorter codes to more frequently occurring elements.

  Also irrelevant here as we don’t have data about frequencies, and random values spread equally so there will be no benefit.

- **Bitpacking:**

  A compression technique that packs multiple integers into a single machine word, efficiently utilizing the available bits - **BEST FITTING SOLUTION.**

Solution

1. As the values are 0 -100 ,they fit in a char - or uint8_t. So I used **BitPacking** algorithm to make a char from every int. I then improved the solution by utilising the numbers in 7 bits - as 100, the max value, can be represented by 7 bits , meaning the compressed values can take even less than one byte:

   ```
   vector<uint8_t> BpackCompresser::Compress(const vector<int>& data);
   vector<int> BpackCompresser::Decompress(const vector<uint8_t>& compData);
   ```

2. I also implemented **BitPacking** combined with **Run-Length** algorithm, for cases with lots of repeated consecutive elements.

   ```
   vector<uint8_t> BpackRLCompresser::Compress(const vector<int>& dataToCompress);
   vector<int> BpackRLCompresser::Decompress(const vector<uint8_t>& compData);
   ```

3. Finally, I used **multi threading** for faster implementation of the BitPacking. I kept the order by using a Waitable Priority Queue (implemented during the course).

   ```
   vector<uint8_t> MultiThreadCompresser::Compress(int numOfThreads, const vector<int>& data);
   vector<int> MultiThreadCompresser::Decompress(int numOfThreads, const vector<uint8_t>& compData);
   ```

SIZE COMPARISON

Original data
10 MIL ints

40 Mb

Compressed data

Bitpacking

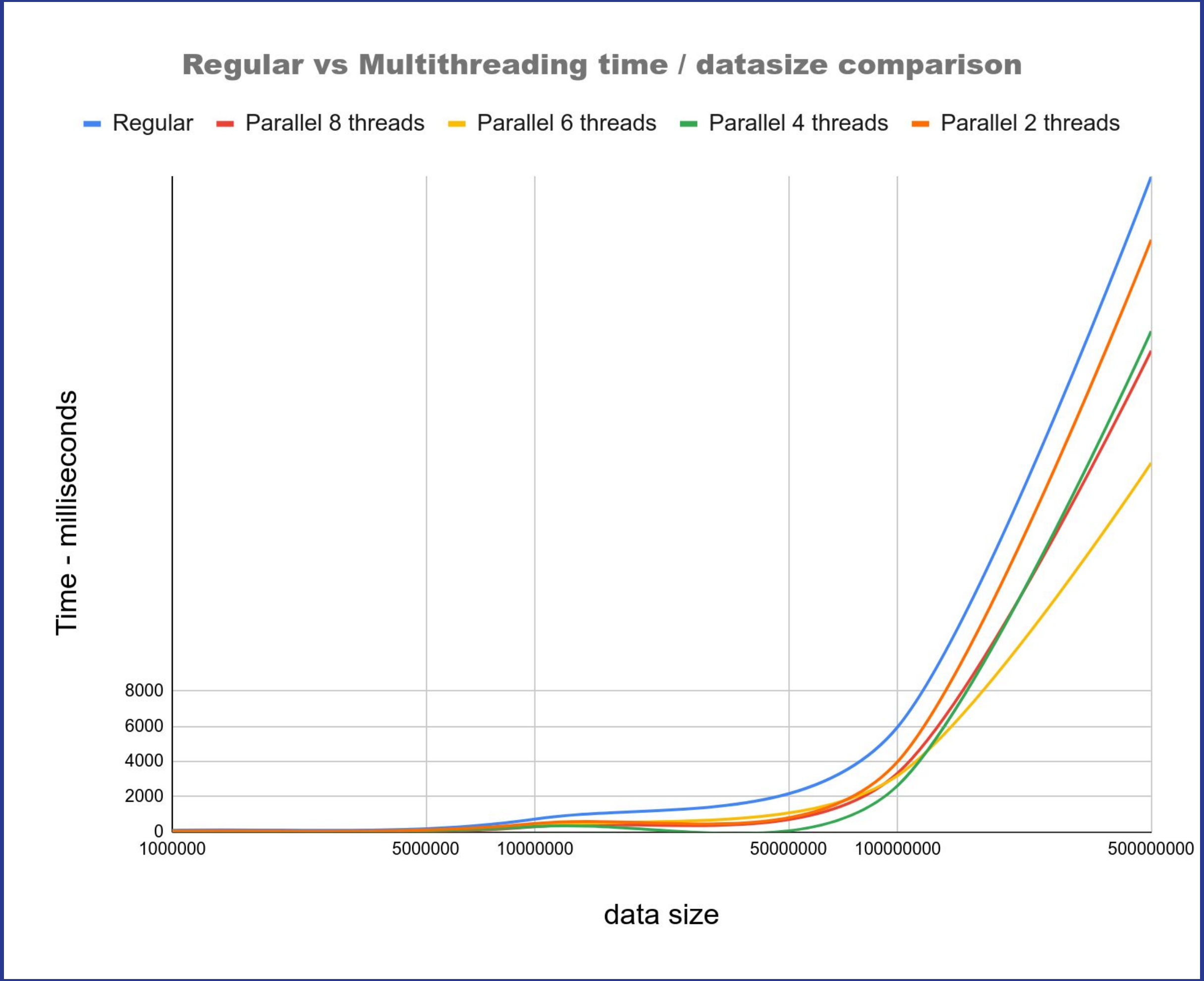
8,750,000 bytes

Bitpacking
And RLE
Almost No repetitions
8,834,555 bytes

Bitpacking
Multithreaded

8,750,000 bytes

Bitpacking
And RLE
With data repetitions
1,327,123 bytes



As can be seen, up to 5Mb of data almost no difference in performance.

But for large amounts of data, optimal value i=of threads seems to around half of available threads on the PC -
for me it was 6 threads.

Improvement suggestions

1. To use thread pool instead of opening each thread individually.

   It provides a pre-defined number of worker threads that can be reused. This approach can improve performance by minimizing the time spent on thread creation and management.
2. To add more compression algorithms that will inherit from Compress class like Huffman coding, Delta encoding , etc.
3. If the use of those algorithms is large we can create a factory that will create instances of each algorithm on demand for faster and more convenient use.