Random access of elements in data containers using multiple memory blocks of increasing capacity

Matthew Bentley

Abstract

In C++ terminology a 'random access' data container is a sequential container where any element within the container can be accessed with O(1) time complexity, given the index of that element in the iterative sequence. To date, no methods have been suggested for creating random access containers utilizing multiple memory blocks of increasing capacity. This paper details one method of doing so. Increasingly-large memory blocks have advantages both in terms of data locality[1][2] and subsequent cache performance, reducing memory waste for small numbers of elements, and in reducing the overall number of memory allocations, compared to multiple blocks of the same capacity.

1. Introduction

However much one would like it to be otherwise, there exist no truly universal data containers. Each computing scenario is different, and the most suitable container depends largely on context. In the modern C++ world the most commonly-used container is the "vector", often known as a "dynamic array" or "array list" in other languages. The primary reason for it's common usage is because, like an array, a vector's storage mechanism is a singular contiguous block of memory. This can have a positive performance effect on most modern processors due to the fact that data is read from main memory into the CPU's cache in large chunks. Hence reading one item from main memory into cache will typically read many subsequent items as well, if they are contiguous in memory.

And if after processing the first item, the second item is already in the CPU's cache, retrieving it is approximately 100 to 200 times faster than retrieving it from main memory on most modern computers. Storing data in singular contiguous memory blocks is one way of ensuring that subsequent items will be read into cache. If items are instead allocated individually in memory by a container, there is generally no guarantee that items subsequent to the first item retrieved will be in the same memory area, and hence they may not be in the CPU's cache after the first item is processed [3].

An alternative strategy which works particularly well on large items and for large numbers of items, is storing data in multiple memory blocks. This has an advantage in terms of expanding the container when it reaches full capacity.

When a single memory block container such as a vector expands, it must provide a new, larger memory block and then copy the existing items to that memory block. There is a performance cost to this [4] [5], but also it can create usage issues because it invalidates pointers to stored elements. Using multiple memory blocks means one does not have to reallocate elements when expanding the container's capacity, and so pointers to elements will not be invalidated, and the reallocation performance cost is avoided.

To date, most multiple memory block containers, such as double-ended queue (eg. C++'s std::deque [6]) implementations, have used memory blocks of the same capacity, which makes the math for accessing elements via their iterative index straightforward. However this comes at a cost of needing to allocate more memory blocks than would be necessary if increasingly-large memory blocks were used. So when memory allocation is an expensive operation, using many memory blocks of the same capacity is a sub-optimal strategy. In addition, creating larger memory blocks increases the overall contiguity of items in memory, which may increase performance as described above.

A better strategy is to use memory blocks of increasing capacity, and in this way reduce the number of overall allocations required. Most vector implementations use a doubling rule for capacity, that is to say when the vector expands, it allocates a memory block twice the capacity of the existing one, copies the existing elements to it, and then removes the original memory block. While there is some variation in terms of expansion ratios, this is generally a good "common sense" rule in terms of anticipating how many more items the program is likely to need in the complete absence of any guiding input from the programmer themselves.

Similarly in a multiple memory block container we can double the capacity with each expansion, by creating a new memory block with the same capacity as all of the previous memory blocks combined. As an example, a container might initially create a memory block of capacity 7, upon first expansion create another memory block of capacity 7, then upon second expansion create a memory block of capacity 14. At this point however a mechanism to allow for random access of individual items within the container based on their index, in O(1) time complexity, is not obvious.

The method to enable this which I illustrate below relies on each memory block's capacity being a power of two, and each new memory block doubling the overall capacity of the container. Hence if the first block's capacity were 8, the second would also need to be 8 and the third would have to be 16. This strategy enables the use of binary math to calculate, from any element's index, both the memory block that item is contained within, and the sub-block offset required to access the element. Full illustration of this principle follows.

2. Definitions

For the purposes of this paper the following terms are defined:

Container index: The index of an element within a container ie. it's number in relation to sequential access of elements.

Block index: The sequential number of any given memory block used by a container which utilizes multiple memory blocks.

Sub-block index: The sequential number of any element within a given memory block. Not related to the container index, although for the first memory block the container index and sub-block indexes will be the same.

Active bit: A bit set to 1.

Inactive bit: A bit set to 0.

LSB: Least significant bit.

MSB: Most significant bit.

Bitwise index: The index of a bit within a number, when measuring from the LSB to the MSB. In the unsigned integer 33 the index of the first active bit in the number is 0 ("1", representing a value of 1), while the index of the last active bit is 5 ("1", representing a value of 32). For unsigned integers the bitwise index is also equivalent to log base 2 of the number which the bit represents.

MSAB: The most significant *active* bit of any given number ie. the highest bit which is set to 1. In the unsigned integer 33 the MSAB is at bitwise index 5.

Note: in this paper all index numbers begin at zero. This means the index of the LSB in any number is 0, the index of the first memory block in a container is 0, and the sub-block index of the first element within any given memory block is also 0.

3. Method

As mentioned, in order for this method to work it is required that the capacity of the first memory block must be a power of two, and each subsequent memory block must double the overall capacity of the container. The first block's capacity is stored as f. The bitwise index of the MSAB of the first block's capacity is also stored as m. So, for example if the first block's capacity was 8, it's MSAB's bitwise index would be 3 as counted from the LSB (whose index is 0):

```
          ↓
0  0  0  0  1  0  0  0
```

Now, say we have a container following this principle of multiple memory blocks with power-of-2 sizes, which has expanded multiple times and now contains 254 items, and it's initial overall capacity was 8. This would mean it's overall capacity would now be 256. The capacity of each individual memory block it contains at this point would be 8, 8, 16, 32, 64, and 128 respectively. To calculate which memory block any given item is contained within, we first require the item's index in the container ie. what it's number is in terms of sequential access. We store this index as i then process as follows.

1. First we subtract 1 from m and store the result as n. If n is negative, we round upward to zero. Alternatively we can use multiplication with the result of a comparison with zero to achieve the same result (see pseudocode in next section for example).
2. We then bit-shift i to the right (in the direction of the LSB) by n and store the result as j. If i was 9 and n was 2, for example, j would be 2:
   Before:
   ```
             ↓
   0  0  0  0  1  0  0  1
   ```
   After:
   ```
                ↓
   0  0  0  0  0  0  1  0
   ```
3. Taking j, we calculate it's MSAB index and store this as b, which is the index of the memory block which our item at container index i is stored within. Continuing our earlier example of a container with a first memory block capacity of 8, if i is 9 then b would be 1 (the second memory block).
4. If b is zero, our item is within the first memory block, and we use i as our index into that block to find the item. Otherwise we bitshift 1 to the left by $(b + n)$ and store this as k. This has the effect of storing only the MSAB of i in k. We then subtract k from i and store the result as s, which is our index into the memory block at index b. Continuing the example above, i is 9, j is 1 and n is 2 so k would be 8 and s would be 1.

4. Pseudocode

In pseudocode the above process looks like this:

1. m := MsabIndex(f)
2. n := (m != 0) * (m - 1)
3. j := i >> n
4. b := MsabIndex(j)
5. k := (j != 0) << (b + n)
6. s := i - k

(where the result of a '=' or '!=' comparison is 1 for true and 0 for false).

The $MsabIndex$ function will differ depending on platform, but most modern processors contain an instruction designed specifically to calculate the MSAB's index for any given integer (for example, the BSRL/BSRW instructions on Intel x86 CPUs), which makes the first and fourth steps of the pseudocode above computationally inexpensive. In situations where such processor instructions are not available, there are commonly-understood bit-twiddling processes to achieve the same result, for example the following C++ code:

```
unsigned int number; // 32-bit integer we want to find the log base 2 of
unsigned int result = 0; // result will become log base 2 of (number)

while (number >>= 1) { ++result; }
```

(https://graphics.stanford.edu/ seander/bithacks.html#IntegerLogObvious)

Returning to our example from the section above (where i is 9 and f is 8), this process becomes:

1. $m :=$ MsabIndex(f)	($m := 3$)
2. $n :=$ (m != 0) * (m - 1)	($n := 1$ * (3 - 1) $:= 2$)
3. $j := i >> n$	($j := 9 >> 2 := 2$)
4. $b :=$ MsabIndex(j)	($b := 1$)
5. $k :=$ (j != 0) << ($b + n$)	($k := 1 << (1 + 2) := 8$)
6. $s := i$ - k	($s := 9$ - 8 $:= 1$)

This results in a block index (b) of 1, and a sub-block index (s) of 1. Given that the first block (with index 0) has a capacity of 8, this makes sense.

Likewise if we think of a container where the first block's capacity (f) is 4, and the container index of the item we wish to access (i) is 31, we get the following:

1. $m :=$ MsabIndex(f)	($m := 2$)
2. $n :=$ (m != 0) * (m - 1)	($n := 1$ * (2 - 1) $:= 1$)
3. $j := i >> n$	($j := 31 >> 1 := 15$)
4. $b :=$ MsabIndex(j)	($b := 3$)
5. $k :=$ (j != 0) << ($b + n$)	($k := 1 << (3 + 1) := 16$)
6. $s := i$ - k	($s := 31$ - 16 $:= 15$)

Resulting in a block index (b) of 3, and a sub-block index (s) of 15.

When the first block capacity is 4, subsequent block capacities will be 4, 8, 16, 32 and so on. Matching these up with the results above, we can see that the block index of 3 would match the block of capacity 16, and that the sub-block index of 15 matches where an element at container index 31 in the container would be.

5. Optimization strategies

The most immediately obvious optimization is to store the bitwise index of the MSAB of the first block, minus 1 (ie. n) for the lifetime of the container, thereby making the first two instructions of the pseudocode unnecessary. In addition, if one were to introduce a branching instruction before the first entry in the pseudocode above, to determine whether the index is smaller than the capacity of the first block (and if so simply returning $b = 0$ and $s = i$), many instructions could be avoided when one is indexing into the first block. If we do, we can also change line 5 of the pseudocode from:

$k := (j \mathrel{!}= 0) << (b + n)$

to:

$k := 1 << (b + n)$

This would remove a comparison operation at that stage. However, whether this would result in a performance increase largely depends on the processor in question and the compiler; in many modern processors the avoidance of branch instructions is associated with better performance due to effects on the pipeline of branch prediction failure [7] [9] [8]. This strategy might also not be appropriate for domains involving latency sensitivity, due to the variation in timing depending on which path the branch takes.

6. Conclusion

By utilizing memory blocks with capacities equal to powers of two and doubling the overall capacity of the container upon every memory block allocation, we can create a container with O(1) time complexity when randomly-accessing elements. The calculations necessary are trivial and involve no additional memory allocations or lookup tables. Iterating over such a structure is also trivial, and the capacity of each memory block is calculable from it's index. Overall this method shows substantial promise in terms of enabling fewer memory allocations for multiple-memory-block containers such as deques while making reallocation upon capacity expansion unnecessary.

References

[1] Kennedy, Ken, and Kathryn McKinley (1992), Optimizing for Parallelism and Data Locality, CRPC-TR92190 January 1992.

[2] Denning, P.J.. (2006), The locality principle. Communications of the ACM Volume 48 Issue 7: 19-24.

[3] Markus Kowarschik and Christian Weiß (2003), An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms, Algorithms for Memory Hierarchies: Advanced Lectures, Springer Berlin Heidelberg.

[4] Marc Gregoire. (2014), Professional C++, John Wiley & Sons.

[5] Bulka, Dov, and David Mayhew. (2000), Efficient C++: performance programming techniques., pp 76 Addison-Wesley Professional.

[6] ISO/IEC (2013), Programming Languages — C++, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf`.

[7] Agner Fog (2017), Branch prediction in Intel Haswell, Broadwell and Skylake, The microarchitecture of Intel and AMD CPUs, `http://www.agner.org/optimise/microarchitecture.pdf`, pp.28.

[8] Babka, Vlastimil and Marek, Lukáš and Tuma, Petr (2009), *When misses differ: Investigating impact of cache misses on observed performance.*, Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on, pp. 112-119. IEEE.

[9] Arun Kejariwal, Center for Embedded Computer Systems University of California (Irvine, USA), Alexander V. Veidenbaum, Alexandru Nicolau, Xinmin Tian, Milind Girkar, Hideki Saito, Utpal Banerjee (2008), Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel® Core™ 2 Duo processor, Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS.