The high complexity jump-counting pattern

Matthew Bentley

December 10, 2019

Abstract

This document describes a numeric pattern for indicating skippable items in data sequences and methods to adjust this pattern when the skippable status for an individual item changes. The pattern provides up to 525% faster iteration speeds (dependent on the ratio of skipped to unskipped items) than an equivalent boolean skipfield on a modern pipelined CPU. This is due to a lack of branching code, reduced number of skipfield reads and a reduced instruction count during iteration, when compared with a boolean pattern. This efficiency is accomplished without significantly impacting the duration taken to change an item's skippable status. The pattern has application in generalized computer science but also specifically in the implementation of data containers. The 'high complexity' in the title refers to time complexity, to differentiate between this pattern and the 'low complexity' jump-counting pattern, which has lower time complexity in it's operations.

1 Introduction

In many areas of software engineering including video game development and high performance domains, there are scenarios where boolean fields are utilized to indicate whether elements within data containers are still active. This is sometimes preferred to actual erasure or removal of objects from the data container, as erasure can come with side-effects depending on the container in question. During iteration and processing

the boolean field nodes are checked so as to skip over objects marked as inactive. As an example, an array of objects in a physics simulation might contain a boolean field named "active" within each object. The flag is set to false when that object is destroyed and subsequently skipped during processing.

There are many advantages to this technique. Firstly, with a static container like an array, removal of an object from active memory is not possible. Secondly, if an extensible container designed for iterative speed is used (such as a "vector" or queue) then erasure of objects may have performance penalties [2, Standard Template Library, p. 164] due to reallocation of subsequent objects. Thirdly, the latter process of reallocating elements invalidates indexes and pointers to objects within the container [3, Sequential Containers, p. 485], which can be of concern in modular or object-oriented code.

(Note: highly-contiguous storage containers such as C++'s vector, deque and regular arrays are often preferred in high-performance programming over non-contiguous containers such as linked-lists [5, p.15] [6, p.44], despite the latter not exhibiting these side-effects. This is due to the poor processor cache performance associated with non-contiguous memory storage during iteration [7].)

The technique is simple and often effective, but inefficient for iteration of any container where large numbers of consecutive "inactive" objects are present, as iteration involves processing the boolean field of each inactive object. What this means is, as soon as one or more objects in the container are made inactive, the number of boolean fields which must be checked to iterate from from one active object to the next becomes unpredictable. There could be only one skipfield node check before the next active object is reached, or as many skipfield node checks as there are erased objects in the container. This creates jitter during iteration. An ideal solution would simply skip from one "active" object to the next in a singular step.

The "jump-counting" skipfield approach negates this inefficiency and provides O(1) time complexity for linear iterative traversal from one active object to the next, without incurring additional computational overhead. It is a numeric sequence which indicates

both when to skip over objects and how many objects to skip over, in any sequence of elements, and can also be used in any context where a boolean skipfield would typically be employed.

A review of scholarly journals has revealed no similar pre-existing works on replacements for boolean skipfields, the use of boolean skipfields to indicate object erasure in data sets, or numeric patterns similar to the jump-counting skipfield pattern other than the precursor for this paper (previous title: "The advanced jump-counting skipfield"[1]). A similar field at least conceptually is Carry-Skip Adders [4] however these differ in form, function and intended application. In form a Carry-Skip Adder consists specifically of hardware blocks or circuits/circuit simulations, and functionally it uses a boolean skipfield to allow carry bits to skip data transformation blocks, instead of skipping actual data as the jump-counting skipfield does.

There is also a low-complexity variant of the jump-counting pattern, which is explained in it's own paper[11]. This has advantages in some scenarios, since it has a reduced time complexity compared to it's high complexity counterpart, hence the name. However for the purposes of brevity, when we refer to a 'jump-counting' skipfield or pattern for the rest of this paper we are explicitly referring to the high-complexity version of that pattern. To begin, let us examine the pattern's methodology, then benchmarks will be presented to show it's performance compared to boolean skipfields.

2 Basic notation

A jump-counting skipfield is always an array (or extensible container) of an unsigned integer type. This type must be of sufficient bit-depth to describe the number of objects that could potentially be stored within the set of objects it is associated with. So a memory block which can store up to 65536 objects would require a skipfield made up of 16-bit unsigned integers, while a memory block which could store up to 1,000,000,000 objects would require a skipfield made up of 32-bit unsigned integers.

All unskipped objects are notated with zero, so if you have a set of ten unskipped objects in a memory block, the block's corresponding skipfield (S) would be:

$$S = (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0)$$

Any non-zero value indicates a skipped object. If a singular object is skipped and has no consecutive erased objects it is notated with "1":

$$S = (0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0)$$

From this point onward we will refer to numbers within the skipfield as "nodes" and sequences of consecutive erased objects (as notated in the skipfield) as "skipblocks". Below we show a skipfield where the 3rd, 4th, 5th and 6th objects are being skipped. The first node indicating a skipblock (eg. the first "4" in the example below) is called the "start node". The last node in that skipblock is called the "end node" (eg. the last "4" in the example below). The values of both start and end nodes are always set to the number of objects which need to be skipped. In the example above, the "1" is simultaneously the start and end node of a skipblock.

$$S = (0 \quad 0 \quad 4 \quad 2 \quad 3 \quad 4 \quad 0 \quad 0 \quad 0 \quad 0)$$

Nodes after the start node in a skipblock start from a value of 2 and then increment once per node. The purpose of this becomes apparent when changing the status of nodes from skipped to unskipped and will be explored in section 5.

3 Iteration

To iterate across a sequence of objects utilizing a jump-counting skipfield, an iterator must keep track of both the object being pointing to and that object's correspond-

ing skipfield node. The skipfield node's index will always correspond to the object's index ie. object[5] is always associated with skipfield[5]. Because of this index association an iterator only technically needs to keep track of the index number (i), but for performance reasons we might choose an implementation utilizing pointers to both the object array (e) and the skipfield array (s). To simplify discussion we will focus on an index implementation of the jump-counting pattern for the remainder of this document. To increment in this context we would:

1. increment the index number (i) by one, then
2. increment the index number by the value of skipfield node at index i (S_i).

$$i := i + 1 \quad (1)$$

$$i := i + S_i \quad (2)$$

Below is a demonstration of iteration over the skipfield example shown earlier:

(Notes: in all examples, ↓ indicates the skipfield node corresponding to the current index number i. Also we will skip the S = () array notation for all subsequent examples in the interest of visual clarity.)

Before incrementing:

```
   ↓
0  0  4  2  3  4  0  0  0  0
```

After incrementing:

```
                  ↓
0  0  4  2  3  4  0  0  0  0
```

A similar pattern applies when iterating backwards across the skipfield, where we:

1. decrement the index (i) by one, then
2. subtract the value of the skipfield node at index (S_i) from the index.

$$i := i - 1 \quad (3)$$

$$i := i - S_i \quad (4)$$

Here is a demonstration of reverse-iteration using the example skipfield from above:

Before decrementing:

```
                  ↓
0  0  4  2  3  4  0  0  0  0
```

After decrementing:

```
   ↓
0  0  4  2  3  4  0  0  0  0
```

4 Changing a skipfield node from unskipped to skipped

To make this change we check the nodes to the left and to the right of the skipfield node:

(note: ∗ indicates nodes whose values we are assessing)

```
         ∗  ↓  ∗
0  0  0  0  0  0  0  0  0  0
```

If there is a non-zero node to the left, that node is the end node of a skipblock on the left. If there's a non-zero node to the right, that node is the start node of a skipblock on the right. It is useful to note at this point that if we know where a skipblock's start node is, we also know where the end node is and vice-versa, since both count the number of objects to be skipped. Therefore we can update both if we know the location and value of either one.

Once we have checked the value of the nodes to the left and right, there are four scenarios:

1. left and right are zero
2. left is non-zero and right is zero
3. left is zero and right is non-zero
4. left and right are non-zero

We will now examine how each of those scenarios is to be handled.

Scenario 1: Left and right are zero

We set the value of the current skipfield node to one. This indicates a single skipped object with no consecutive skipped objects. No further action is required.

$$S_i := 1 \tag{5}$$

(note: in this context ↓ indicates the skipfield node corresponding to the object we are wishing to skip)

Before:

```
                  ↓
0 0 0 0 0 0 0 0 0 0
```

After:

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

Scenario 2: Only left is non-zero

In this scenario the left-hand node is the end node of a preceding skipblock. We set the value of the current node to the value of the left-hand node, plus one. The current node is now the new end node. We then subtract the value of the left-hand node from the current node's index to find the start node's index, which we store as y, then set the start node's value to the current node's value. We do not change the value of the left-hand node.

$$S_i := 1 + S_{i-1} \tag{6}$$

$$y := i - S_{i-1} \tag{7}$$

$$S_y := S_i \tag{8}$$

Before:

```
                  ↓
0 0 0 3 2 3 0 0 0 0
```

After:

```
                  ↓
0 0 0 4 2 3 4 0 0 0
```

Scenario 3: Only right is non-zero

Here we take the value of the right-hand node and store it as x, then set the current node's value to $x + 1$. This is now the start node of the skipblock to the right. We update the value of each node to the right of the current node, starting from a value of two and incrementing by one per node, decrementing x by one per node and stopping when x is zero.

$$x := S_{i+1} \tag{9}$$

$$S_i := x + 1 \tag{10}$$

$$j := 1 \tag{11}$$

$$\textit{Repeat until } x = 0 \; \{S_{i+j} := j + 1 \tag{12}$$

$$j := j + 1 \tag{13}$$

$$x := x - 1\} \tag{14}$$

Before:

↓

0 0 0 3 2 3 0 0 0 0

After:

↓

0 0 4 2 3 4 0 0 0 0

Scenario 4: Both left and right are non-zero

We first take the value of the right-hand node, store it as x and increment x by one, and take the value of the left-hand node and store it as y. We subtract y from the current node's index to find the index of the left skipblock's start node, increment this start node by x, then add one to y. Then, starting from the current node and continuing to the right, we set every node to y, incrementing y by one for each node,

decrementing x by one per node, stopping when x is zero.

$$x := 1 + S_{i+1} \tag{15}$$

$$y := S_{i-1} \tag{16}$$

$$S_{i-y} := S_{i-y} + x \tag{17}$$

$$y := y + 1 \tag{18}$$

$$j := i \tag{19}$$

$$\textit{Repeat until } x \ = \ 0 \ \{S_j := y \tag{20}$$

$$j := j + 1 \tag{21}$$

$$y := y + 1 \tag{22}$$

$$x := x - 1\} \tag{23}$$

Before:

↓

2 2 0 3 2 3 0 0 0 0

After:

↓

6 2 3 4 5 6 0 0 0 0

5 Changing a skipfield node from skipped to unskipped

If we want to revert a skipfield node back to an "unskipped" status, we need to update the skipfield in such a way that subsequent iteration is still valid. To do so we must check the value of the skipfield nodes to the left and right of the skipfield node corresponding to the object whose status we wish to change. This is similar to the process we used during erasure. Once again we have four scenarios: either both left and right node values are zero, left is non-zero and right is zero, left is zero and right is non-zero, or both left and right are non-zero. Here's how we handle those scenarios in this case.

Scenario 1: Both left and right are zero

We set the value of the current node to zero. No further updates are necessary.

$$S_i := 0 \tag{24}$$

Before:											*After:*									
	↓											↓								
0	1	0	4	2	3	4	0	0	0		0	0	0	4	2	3	4	0	0	0

Scenario 2: Only left is non-zero

This indicates that the current node is the end node of a skipblock on the left. In this case we take the current node's value and store it (x), subtract one from x, then subtract x from the current node's index to find the index of the skipblock's start node. We then set the start node's value to x and the current node's value to zero.

$$x := S_i - 1 \tag{25}$$

$$S_{i-x} := x \tag{26}$$

$$S_i := 0 \tag{27}$$

Before:											*After:*									
						↓											↓			
0	0	0	4	2	3	4	0	0	0		0	0	0	3	2	3	0	0	0	0

Scenario 3: Only right is non-zero

This indicates that the current node is the start node of a skipblock continuing to the right. First we store the value of the current node as x and decrement x by one. We then set the value of the current node to zero and the value of the right-hand node to x, and subsequently decrement x by one again. Then, starting with the node after the right-hand node, we update the value of each subsequent node, starting with a value

of two and incrementing by one per node, decrementing x by one per node until x is zero.

$$x := S_i - 1 \tag{28}$$

$$S_i := 0 \tag{29}$$

$$S_{i+1} := x \tag{30}$$

$$x := x - 1 \tag{31}$$

$$j := 2 \tag{32}$$

$$\textit{Repeat until } x \ = \ 0 \ \{S_{i+j} := j \tag{33}$$

$$j := j + 1 \tag{34}$$

$$x := x - 1\} \tag{35}$$

Before:

$\downarrow$

0 0 0 4 2 3 4 0 0 0

After:

$\downarrow$

0 0 0 0 3 2 3 0 0 0

Scenario 4: Both left and right are non-zero

In this scenario the current node is within a skipblock, but neither at the beginning nor the end of it. The end result of the operation must be to split the singular skipblock into two separate skipblocks. For clarity we shall split this process into three phases. In the first phase of the transformation we store the current node's value as y and subtract $y - 1$ from the current node's index to find the index of the start node and store this as z. We store the value of the start node as x, decrement x by y and set

the value of the right-hand node to x.

$$y := S_i \tag{36}$$

$$z := i - (y - 1) \tag{37}$$

$$x := S_z - y \tag{38}$$

$$S_{i+1} := x \tag{39}$$

In the second phase we set the start node's value to $y - 1$, decrement x by one and set the value of the current node to zero.

$$S_z := y - 1 \tag{40}$$

$$x := x - 1 \tag{41}$$

$$S_i := 0 \tag{42}$$

In the third phase we update the values of each node to the right of the right-hand node, beginning with a value of two and incrementing by one per node, decrementing x by one per node until x is zero. If the value of x is already zero at this stage no update occurs.

$$j := 2 \tag{43}$$

$$\textit{Repeat until } x \;=\; 0\ \{S_{i+j} := j \tag{44}$$

$$j := j + 1 \tag{45}$$

$$x := x - 1\} \tag{46}$$

Before:

$$\begin{matrix} & & & \downarrow & & & & & & \\ 6 & 2 & 3 & 4 & 5 & 6 & 0 & 0 & 0 & 0 \end{matrix}$$

After first phase:

$$\begin{matrix} & & & \downarrow & & & & & & \\ 6 & 2 & 3 & 4 & 2 & 6 & 0 & 0 & 0 & 0 \end{matrix}$$

After second phase:

$$\begin{matrix} & & & \downarrow & & & & & & \\ 3 & 2 & 3 & 0 & 2 & 6 & 0 & 0 & 0 & 0 \end{matrix}$$

After third phase:

$$\begin{array}{cccccccccc} & & & \downarrow & & & & & & \\ 3 & 2 & 3 & 0 & 2 & 2 & 0 & 0 & 0 & 0 \end{array}$$

Reuse of the erased node is complete at this point and the skipfield can be iterated over correctly in both directions.

6 Performance results

In this section we benchmark in C++ the performance of a Colony[10] data container using a boolean skipfield to indicate erased objects to be skipped during iteration ("boolean colony"), versus a Colony container using a high-complexity jump-counting skipfield to skip erased objects ("jump-counting colony"), and as a reference point, a std::vector container. The std::vector should exhibit slower erasure and insertion but faster iteration. The test system is an Intel E8500 CPU running GCC 6.3 x64 as the compiler. Compiler settings are "-O2;-march=native". Tests are based on a sliding scale of number of runs vs number of objects, so a test using only 10 objects in a container will use 100,000 runs and average the results, whereas a test with 100,000 objects will use 10 runs and average the results. This gives reliable test averages without overly lengthening test times.

The insertion test measures inserting singular objects sequentially. The objects in question are small structs with an integer member containing a random number. In the erasure tests we iterate through the container objects and erase single objects at random. A C++ "remove_if" pattern might be relevant in this test if we were solely processing the std::vector, but such a pattern makes no difference to colony erasure performance, and thus is omitted. In the iteration tests we iterate forward through each container, adding the value of each struct's stored integer value to a total.

The full source code for these tests, including the colony containers, is available for download here: `http://www.plflib.org/jump_counting_vs_boolean_benchmarks.zip`

In Fig. 1 we can see that both colonies perform significantly better than std::vector

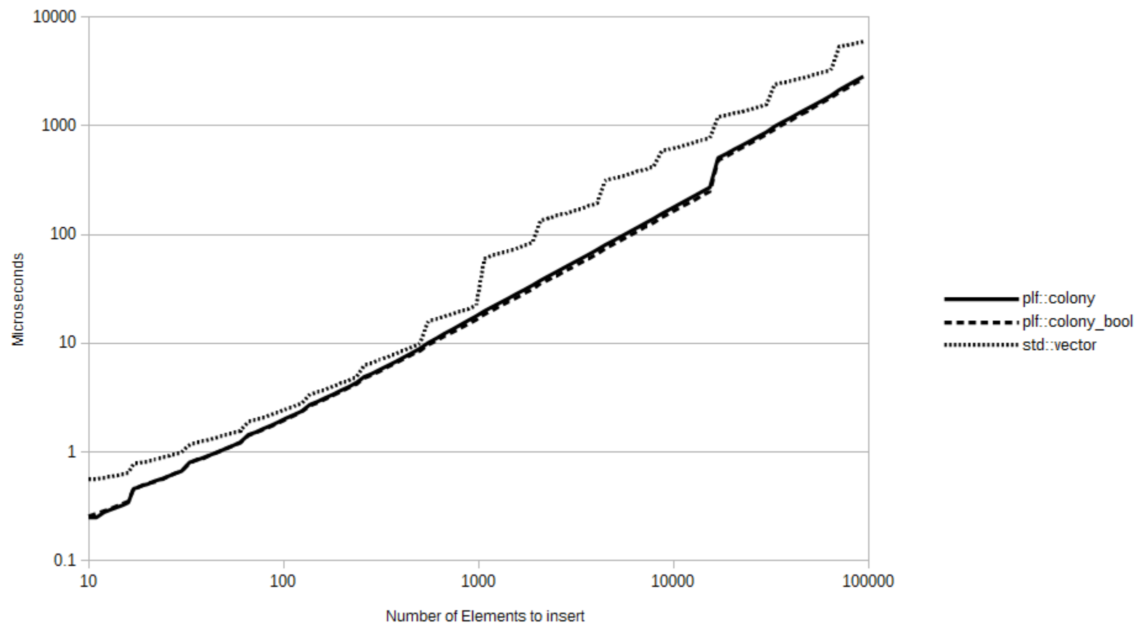

Figure 1: Duration to insert all objects - logarithmic scale

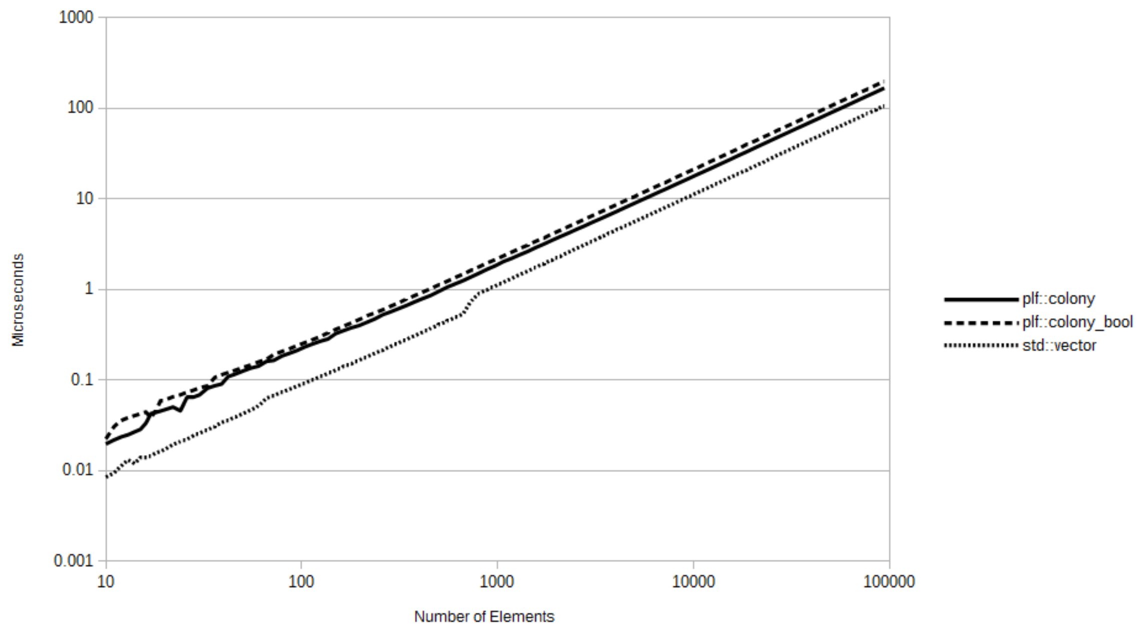

Figure 2: Duration to iterate over objects, prior to erasures - logarithmic scale

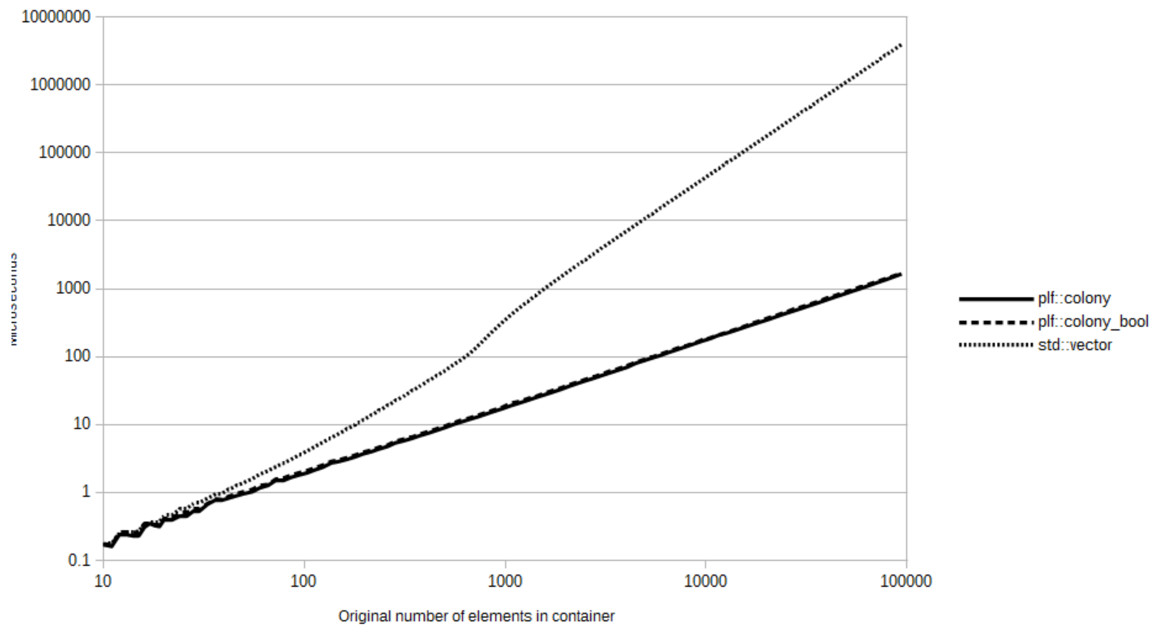

Figure 3: Duration to erase 25% of all objects - logarithmic scale

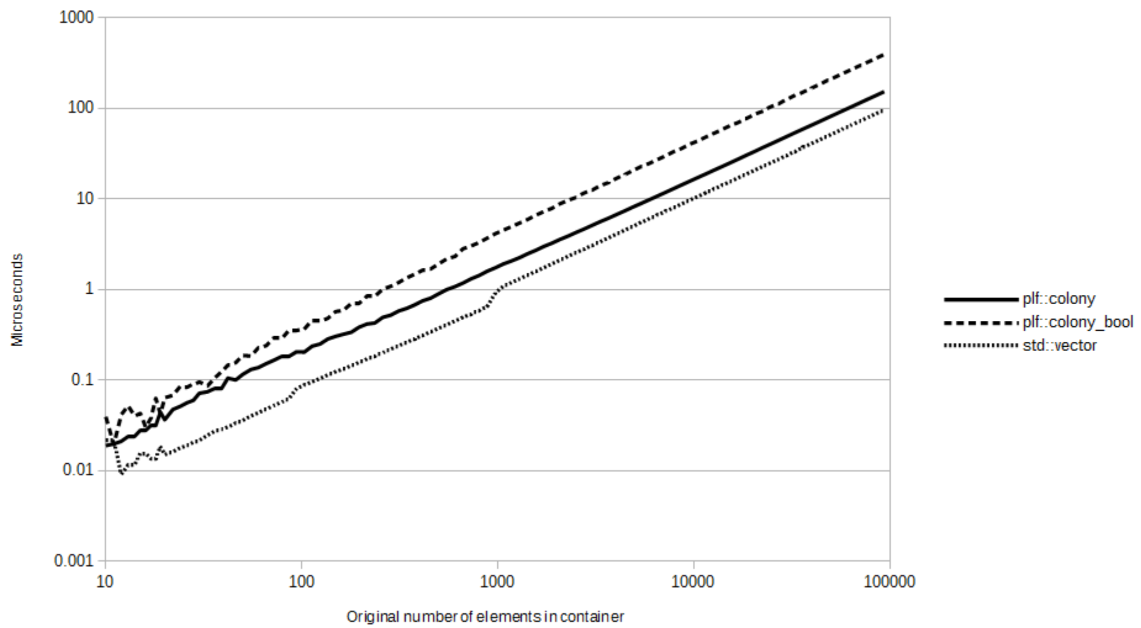

Figure 4: Duration to iterate after erasing 25% of all objects - logarithmic scale

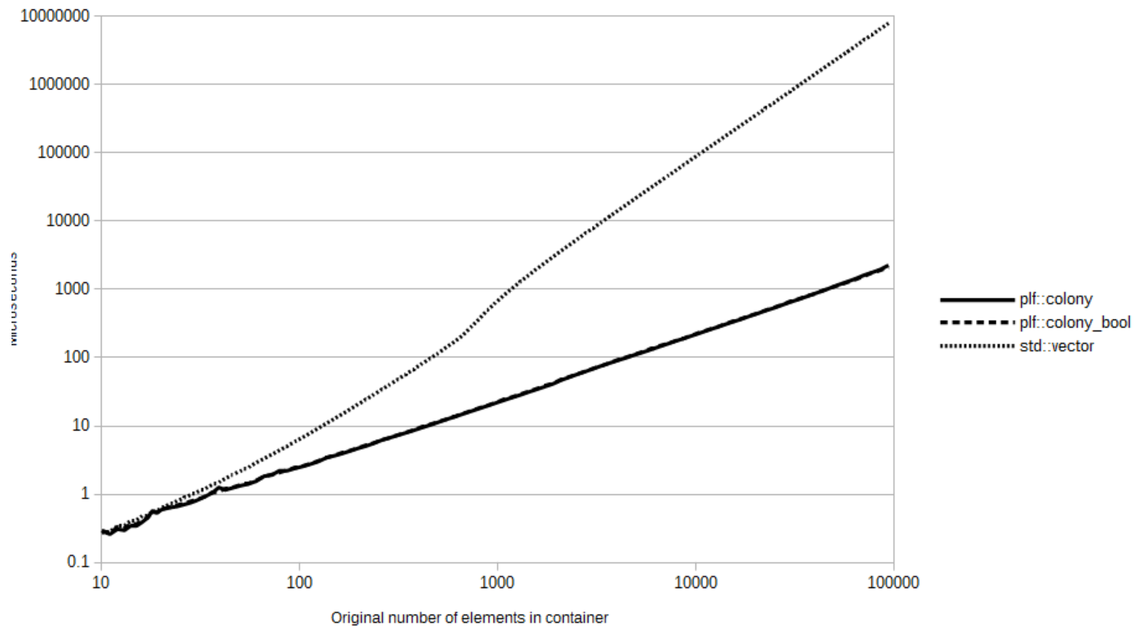

Figure 5: Duration to erase 50% of all objects - logarithmic scale

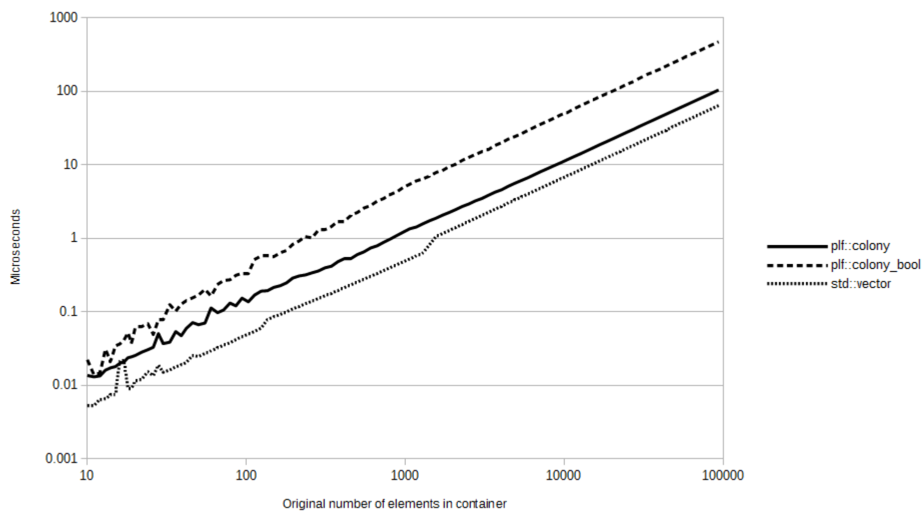

Figure 6: Duration to iterate after erasing 50% of all objects - logarithmic scale

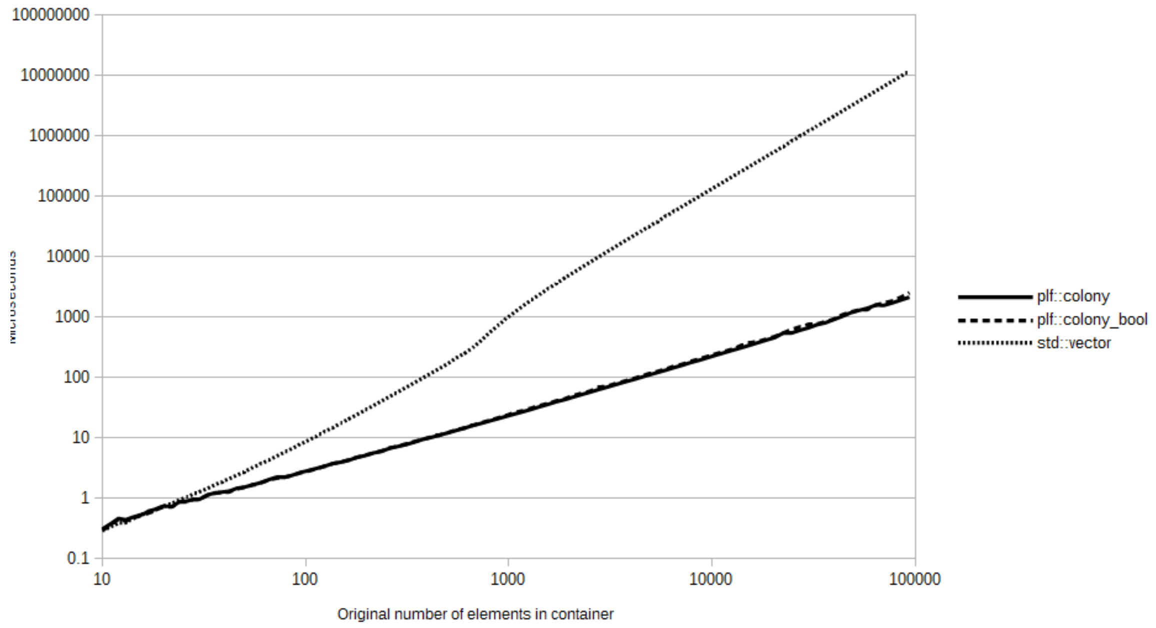

Figure 7: Duration to erase 75% of all objects - logarithmic scale

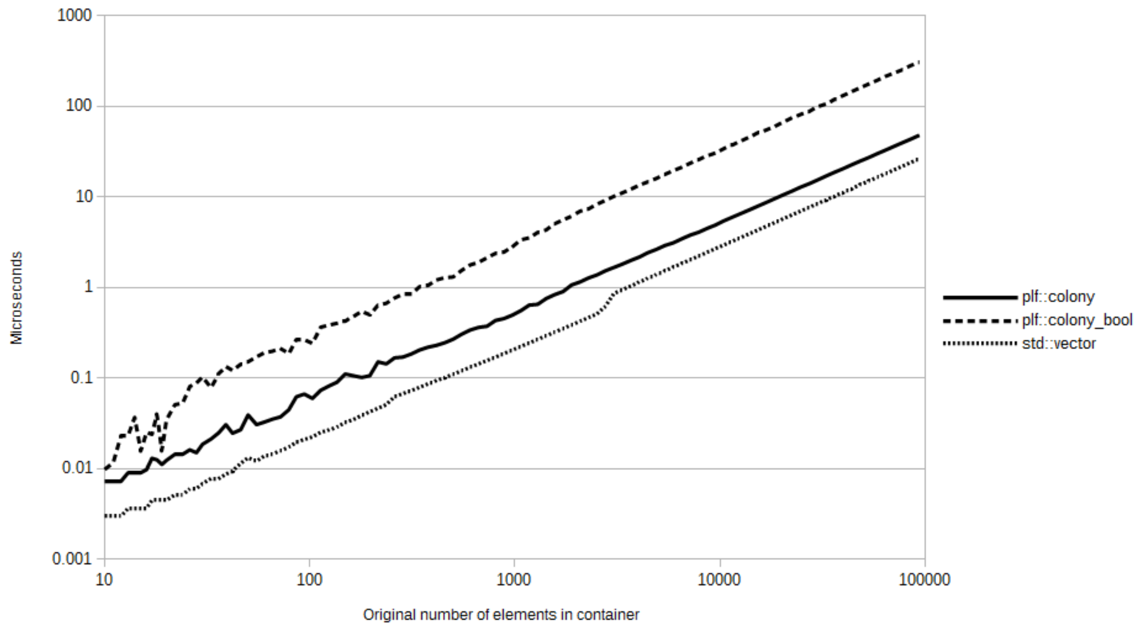

Figure 8: Duration to iterate after erasing 75% of all objects - logarithmic scale

during insertion, and that the colony with a boolean skipfield slightly outperforms the colony with a jump-counting skipfield due to its simpler procedures. At this point no erasures have occurred to either container, hence no changes to the colony skipfields have taken place.

In Fig. 2 std::vector comes in first place, iterating approximately twice as fast as both colonies on average, with the jump-counting colony second and the boolean colony third, 15% slower than the jump-counting colony on average. At this stage the difference in speed between the two colonies is primarily due to the additional branch test necessary to iterate over the boolean skipfield compared to the simple addition needed for iteration over the jump-counting skipfield. Although this test (testing whether or not to skip this element) will always equate to false in this scenario, and thus generates no cache misses, it still consumes more CPU time than the addition necessitated in the jump-counting skipfield version. Now we will measure erasure performance when iterating over the containers and erasing 25% of all objects at random.

As shown in Fig. 3, std::vector has predictably poor erasure performance due to its need to reallocate all the subsequent objects after each erased object (in order to preserve object contiguity), and also due to the random erasure pattern. The colony containers perform far better due to a lack of reallocation, requiring only a fraction of std::vector's duration to complete. The boolean colony has a small advantage over the jump-counting colony due to the lower amount of potential work needed to change a node's state from unskipped to skipped. This trend continues in Figs. 5 and 7. Now let's see how iteration performance has altered post-erasure.

In Fig. 4 we immediately see a performance advantage for the jump-counting colony over the boolean colony. The jump-counting colony's performance increases due to the reduced number of objects and skipfield nodes it is required to read, but the boolean colony's performance almost halves when compared to the iteration results prior to erasures. There are two reasons for this. Firstly, the boolean colony must check the value of each skipfield node to determine the erased status of the corresponding object,

and hence has no ability in the case of multiple consecutive erased objects to directly skip from one non-erased object to the next. The jump-counting colony has this ability, and so its iteration speed is greatly increased by comparison.

The second reason is that in the first iteration test there were no erased objects. And so, while the boolean colony had branching code (to enable skipping any individual object when its corresponding skipfield node indicated erasure), the CPU's branch prediction could lower the performance cost of the branching code significantly [8] [9]. This is because there was only one path for the branch to take at this point due to a lack of erasures. But with 25% of all objects erased, the iterator will encounter a skipped object 25% of the time, also rendering the branch prediction which the CPU makes wrong 25% of the time, creating cache misses. To further explore the impact of these factors on performance, we will now increase the erasure percentage to 50% of all objects in the original containers.

In Fig. 6 the boolean colony's performance is worse than in the 25% erasure iteration benchmark, because with 50% of all objects erased at random there is no opportunity for the CPU's branch prediction to be correct any more than 50% of the time. By contrast the jump-counting colony's iteration performance continues to scale proportionally to the number of erasures, as does std::vector's. Now we will erase 75% of all objects in the original containers.

Both the jump-counting colony and std::vector's iterative performance in Fig. 8 continue to scale proportionately according to the percentage of erasures. The boolean colony in this case performs slightly better than in the 25% erasure iteration test, the reason being that with 75% erased objects there are 25% non-erased objects, so the CPU's branch prediction can work just as effectively, but in this case only 25% of the original objects are having their values read, as opposed to the 75% being read in the 25% erasure iteration test.

Overall we can see that both the ability to skip directly from each active object to the next, and the lack of reliance on CPU branch prediction, play roles in the performance of a jump-counting skipfield compared to a boolean skipfield. On a processor

with no branch prediction, we can expect the boolean pattern to perform worse in the majority of cases, with average iterative performance being closer to that of the 50% erasure iteration test.

7 Additional areas of application and manipulation

7.1 Skipfield subdivision

Take the example of a large data center with 1,048,576 hard drives in various configurations, accessed asynchronously from various locations. To achieve maximum throughput and lowest-possible latency we would ideally assign each task to a hard drive which is not currently in use, as opposed to adding requests to a queue for a hard drive; though the latter may become necessary if the data center became saturated with requests. In addition the timing for when two similar requests complete on two different hard drives will be asynchronous, and for this reason a first-in-first-out strategy for determining available hard drives is not technically feasible. Hence we might implement a randomly-updatable skipfield to assess the available/busy status for each hard drive.

If we use a boolean field in this case we are presented with a problem: for any request we may be required to make anywhere between 1 and 1,048,576 boolean field checks in order to find an available hard drive. This creates variable latency as the time complexity for finding a hard drive will be O(random). If we instead use an advanced jump-counting skipfield pattern we will only require a singular field check to find the next available hard drive, ie. a O(1) time-complexity operation. While the operations to switch any given skipfield node between "busy" and "available" will take longer when compared to a boolean state change, the overall time-saving will be in the factors of 10 as the benchmarks in section 4 show.

In extreme situations where almost all available hard drives are busy, using a jump-counting skipfield in this way could however result in slow state change operations for individual skipfield nodes, as depending on the location of the changed node and

number of consecutive "busy" (skipped) nodes, anywhere between 1 and 1,048,576 skipfield nodes might need to be updated for a single node change. Because these updates don't involve significant branching, the performance cost of this operation would still be reasonably low, but it can be improved upon.

To ameliorate the performance detriment we can split the group of 1,048,576 drives into 4096 smaller groups of 256 drives, creating 8-bit individual skipfields for each group. We can then create a secondary top-level 16-bit jump-counting skipfield with 4096 nodes, in order to identify which hard-drive groups have currently-available hard drives. This increases the maximum number of field checks necessary to find an available hard drive to 2, but decreases the maximum potential number of writes necessary to make a drive's "available" status change, down to 256 8-bit writes plus 4096 16-bit writes (and the latter only if the drive's entire group has become unavailable), from a previous potential maximum of 1,048,576 32-bit writes.

A downside of using the jump-counting skipfield in this context is that concurrent writes to different nodes within one skipfield cannot occur at the same time, as is possible with a boolean skipfield. However if we subdivide, we create multiple independent skipfields, which means that more concurrent writes can occur within the system as a whole.

7.2 Compression

Any skipfield, boolean or otherwise, can be considered as a sequence of alternating runs of skipped and unskipped elements. Therefore all that is necessary to record when saving a skipfield for future purposes is the length of each run, and the value of the first run (skipped or unskipped). To properly compress this sequence, it would be beneficial to waste as few bits as possible. Therefore we need to know the maximum and median length of runs, to find the optimal bitdepth to store run lengths in. Take the following jump-counting skipfield:

0 0 0 0 0 3 2 3 0 0 1 0 9 2 3 4 5 6 7 8 9 0 0

This can be expressed as:

0 5 3 2 1 1 9 2

where the first number indicates the value of the first run (0 = unskipped) and each subsequent number details the length of each unskipped/skipped run. This is similar to run-length encoding[15] but without the necessity to record the value of each run.

The maximum number is 9, requiring a bit-depth of 4 to be represented. Second-highest is 5, required 3 bits to be represented. However the runs of 3, 2, 1, 1, and 2 (the majority) only require 2 bits to be stored, making 2 the median bitdepth, which we choose as the default bitdepth for our encoded sequence. By using 0 as a flag (a run of 0 is not possible, hence this is a 'special' value) we can indicate that a given run requires double the bitdepth of the default bitdepth, which tells the decoder to read the next 4 bits as one value.

Of course, the maximum bitdepth may be more than double the median bitdepth, and we need a way of communicating that, and there may be some balancing required at an initial scanning stage to ascertain ratios between median and larger required bitdepths, in order to choose the best default bitdepth for encoding with maximum efficiency of bits.

We use the first bit in our encoded sequence to indicate the state of the first run (0 or 1, unskipped or skipped). Assuming a maximum platform bitdepth of 64 bits, the next 7 bits would be used to record the default bitdepth (representing a default bitdepth of between 1 and 64). A further 6 bits after this records the maximum bitdepth. Since the maximum number 6 bits can represent is 63, and a maximum bitdepth must be more than 1 (otherwise it could not be larger than the lowest default), we subtract 1 from the maximum bitdepth before storing it, resulting in a possible bitdepth between 2 and 64.

Returning to our example above, with it's initial unskipped run (0), default bit-depth of 2 (0 0 0 0 0 1 0) and maximum bitdepth of 4 (with 1 subtracted: 0 0 0 0 1 1), the initial 14 bits look like this:

0 0 0 0 0 0 1 0 0 0 0 0 1 1

The runs can then be encoded as follows, where the two leading zeroes for 5 and 9

indicate a subsequent multiplied value using twice the default number of bits:

5 : 0 0 0 1 0 1

3 : 1 1

2 : 1 0

1 : 0 1

1 : 0 1

9 : 0 0 1 0 0 1

2 : 1 0

Combining both the header and run bit sequences above results in a total sequence of 36 bits:

0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 1

0 0 1 0 1 0 0 1 0 0 1 1 0

Optionally, if we are transmitting or recording multiple packets of data, we could use the 'special value' of 0, followed by a maximum bitdepth value of 0, as a 'special sequence' to indicate the end of the compressed skipfield. For the above sequence this is encoded as 0 0 0 0 0 0 ('special value' of '0 0' followed by a 'maximum bitdepth' sequence of 0 0 0 0).

This would result in an overall bit sequence of 42 bits total:

0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 0 1 1 1 1

0 0 1 0 1 0 0 1 0 0 1 1 0 0 0 0 0 0 0

A summary of the compression process follows:

1. Convert skipfield to a sequence of run-lengths, preceded by an indication of the first run's state (unskipped: 0, skipped: 1) while creating a histogram of the bitdepths necessary to encode each individual run.

2. Calculate the optimal default bitdepth and multiplier using the histogram, taking into account the overhead added by using the 'special value' of 0 to indicate a subsequent multiplied bitdepth and possibly doing multiple calculations to find the optimal default bitdepth and multiplier.

3. Record as the first 14 bits, the 1 bit state of the first run, the 7 bit value of the default bitdepth, and the 6-bit value of the maximum bitdepth (subtracting 1 before storing).

4. Record each run using the default bitdepth, unless the run length is number greater than what the default bitdepth can represent. In this case use a value of zero for the run to indicate a subsequent maximum (multiplied) bitdepth.

5. At the end of the sequence, if necessary in a given scenario, include a sequence of 6 zeroes to indicate end-of-sequence.

7.3 Intersection (overlaying multiple skipfields)

Multiple high complexity jump-counting skipfields may be intersected using a "tail-chasing" technique. The time complexity of this technique is $O(n + j)$ where n is the number of non-skipped nodes in the intersected output skipfield, and j is the number of skipblocks in the input skipfields which do not entirely overlap. All skipfields being intersected are processed synchronously, starting from the first node in each skipfield and proceeding sequentially through each subsequent node. The current skipfield index for all skipfields being processed is stored as i and the process continues until the end of all skipfields is reached. The steps for this process follow:

1. If the node value at index i in every skipfield is zero, a zero is written to the output skipfield at index i and i is incremented by one. We repeat step 1 until a non-zero value is encountered in any input skipfield or until the end of all skipfields have been reached.

2. When a non-zero value is encountered in an input skipfield, we take that value, add it to i and store the result of the addition as j. This is the index number of the node after the skipblock which has just been identified in that particular

input skipfield. We copy all skipfield node values from that skipfield to the output skipfield, starting from index i and continuing until index j is reached.

3. If j is not beyond the end of all input skipfields, we now check the values of the nodes in the other input skipfields at index j to see if any are non-zero. If none are, we set the output skipfield node at index j to zero, set i to $j + 1$ and go back to step 1.

4. Otherwise we take the first skipfield which contained a non-zero value, subtract the value of the node at index $j - 1$ in this skipfield from j, and store this value as k. k is the index of the start node of the skipblock identified in that skipfield.

5. We increment k by the value of the node at index k in this skipfield. k is now the index of the node after the skipblock in that skipfield. We also take the value of the node in the output skipfield at index i and store it as t.

6. We increment t by 1 and copy it to the output skipfield at index j, then add one to j. We repeat step 6 until j is equal to k.

7. Lastly we copy the value of t to the output skipfield at index i, and go back to step 3.

7.4 Parallel Computing

The jump-counting algorithms are principally serial in design and as such are unlikely to be of use in parallel architectures such as CUDA[12]. Data processing in such environments is typically performed on many items at once rather than iterating over

single items sequentially, and sequential iteration is where the jump-counting skipfield's performance advantages are found. But neither does the pattern prevent parallel usage. For example, if we use an advanced jump-counting skipfield instead of a boolean skipfield to indicate inactive data in a parallel-processing environment, this would not prevent a "compact" operation utilizing an exclusive[13] or inclusive[14] scan. In this situation one can predicate any non-zero value in the skipfield as a zero and any zero value as a one and then subsequently scan. The simple jump-counting pattern is not usable in this context as nodes within skipblocks may contain zero values while still being skipped (see section 3, scenario 4).

However as was mentioned in section 5.1, skipfield subdivision can lead to significant increases in simultaneous skipfield writes, as well as significant decreases in the number of necessary skipfield writes. Further research is needed.

8 Summary

Given an appropriate use case, implementing a high complexity jump-counting skipfield instead of a boolean skipfield will improve iteration performance substantially without significantly impacting on skipfield node state-change performance. This is due to the lack of branching code in the jump-counting skipfield's iteration and the reduced number of required skipfield reads. Improvements in performance and concurrent write-access can be realized for very large skipfields by subdividing the skipfield, thereby lowering the bit-depth of skipfield nodes (eg. a single skipfield of 255 nodes only requires an unsigned 8-bit node type) and the maximum number of necessary skipfield updates upon a change to a skipblock.

This pattern can be used for any situation where a single binary aspect of each object in a set decides whether or not to skip that object (for example erased/non-erased, alive/dead, active/inactive, blue/green). Although a boolean skipfield takes far less time to implement and can be simpler to process, the overall performance benefits

of a jump-counting skipfield outweigh this consideration for any domain requiring high performance, such as high-frequency game loops, container implementation and object processing.

9 Acknowledgements

I would like to thank both Dr Gisela Klette of The Auckland University of Technology and Dr Robert Hurley for their invaluable advice and support in both the critiquing and editing of this document. Also thanks for Steven Cantwell for his advice and Baptiste Wicht for his assistance in building useful benchmarks.

References

[1] Matthew Bentley (2017), Computer Game Journal, Vol. 6, Issue 3, pp 153169. Springer.

[2] Bulka, Dov, and David Mayhew., *Efficient C++: performance programming techniques.*, Addison-Wesley Professional, 2000.

[3] Marc Gregoire., *Professional C++*, John Wiley & Sons, 2014.

[4] Alain Guyot, Bertrand Hochet, Jean-Michel Muller, *A Way to Build Efficient Carry-Skip Adders.*, IEEE Transactions on Computers, Vol. C-36, No. 10., October 1987.

[5] J. Daniel Garcia and Bjarne Stroustrup, *Improving performance and maintainability through refactoring in C++11*, `http://www.stroustrup.com/improving_garcia_stroustrup_2015.pdf`, 2015.

[6] Goldthwaite, Lois., *Technical report on C++ performance.*, ISO/IEC PDTR 18015, 2006.

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

[8] Babka, Vlastimil and Marek, Lukáš and Tuma, Petr, *"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, 2009.

[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, *Comparative architectural characterization of SPEC CPU2000 and CPU2006 benchmarks on the intel Core 2 Duo processor*, Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS, 2008.

[10] Bentley, Matthew R., *Introduction of std::colony to the standard library*, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0447r9.html`, 2019.

[11] Bentley, Matthew R., *The low complexity jump-counting pattern - An O(1) time complexity replacement for boolean skipfields*, `https://plflib.org/matt_bentley_-_the_low_complexity_jump-counting_pattern.pdf`, 2019.

[12] Nvidia (2017), C.U.D.A. Programming guide., `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[13] Blelloch, Guy E. (1990), Prefix sums and their applications.

[14] Hillis, W. Daniel, and Guy L. Steele Jr. (1986), Data parallel algorithms, Communications of the ACM 29.12, 1170-1183.

[15] Hauck, E.L. (1986), Data compression using run length encoding and statistical encoding, US Patent 4,626,829.