The low complexity jump-counting pattern - an O(1) time complexity replacement for boolean skipfields

Matthew Bentley

May 25, 2023

Abstract

The low complexity jump-counting pattern is a replacement for boolean skipfields which enables O(1) time complexity for iteration from one unskipped object to the next. It achieves this without branching statements, which results in better performance on many processors[1][2]. It is similar to the high complexity jump-counting pattern[3] (formerly known as the 'advanced jump-counting skipfield'). Its main advantage over that pattern is that it provides O(1) amortized time complexity when changing single or multiple consecutive skipfield nodes from skipped to unskipped and vice-versa, as opposed to the high complexity jump-counting pattern's undefined time complexity in this area.

1 Introduction

A common pattern in data processing is the utilisation of a boolean skipfield to denote skipped objects in a sequence of objects. For an array of 10 integers, an accompanying boolean skipfield of the following form:

0 1 0 0 1 0 0 0 0 1

would skip the 2nd, 5th and 10th integers respectively during processing of the array.

The pseudocode for a single iteration between two unskipped objects in a sequence of objects using a boolean skipfield, reads as follows (where S denotes the skipfield and i the current index into that skipfield):

```
DO { $i := i + 1$ } WHILE ($S_i == 1$)
```

One use-case for this technique is to temporarily stop processing some objects, with an intention to reinstate processing on some or all of these objects at a later point in time. Another is to indicate erased objects ie. objects which will never be processed again. In the latter case, the advantage of using a skipfield, versus removing these objects from the array then consolidating the subsequent objects, is that no relocation of objects is necessary. This can result in performance gains for large objects, where large amounts of data would otherwise need to be relocated [4]. It also means that pointers to an array's non-erased objects remain valid regardless of whether erasures have occurred, again due to the lack of relocation as stated.

However boolean skipfields come with two downsides, the first being that branching code is necessary for every skipfield read, in order to determine whether to process or skip an object. This large amount of branching can cause performance issues on processors with deep pipelines and poor branch prediction performance, for example Intel Core2 class CPUs[1][2], and will have some effect even on processors with adequate performance in this area[5].

The second downside is that during iteration an unpredictable number of reads from the skipfield are necessary to ascertain the next unskipped object. For example, consider the following boolean skipfield:

0 1 1 1 1 0 0 0 1 0

When iterating from left to right, ascertaining the location of the second unskipped object requires five reads of the skipfield ('1 1 1 1 0') whereas finding the third requires one ('0') and the fourth requires 2 ('1 0'). In other words the time complexity for iterating from one unskipped node to the next is undefined. For large numbers of consecutive skipped nodes this can introduce considerable and unpredictable latency during iteration, making this approach inappropriate for any field of work involving high performance and timing sensitivity. Such fields include game development[6][7][8], high frequency trading[9][10][11], embedded development[12][13][14], data center software[15][16][17] and high performance computing[18][19].

The low complexity jump-counting pattern, by contrast provides O(1) time complexity during iteration; only a single read from the skipfield is ever necessary to ascertain the location of the next unskipped object. As a practical example, consider a data center housing 1,000,000 hard drives, where a process is attempting to find a hard drive which is not currently busy with another task. If an unmodified boolean pattern is used to denote the busy/available status of drives, the maximum possible number of skipfield reads necessary to find an available drive is 1,000,000. If a low complexity jump-counting pattern is used instead the maximum possible number of reads is 2.

It achieves this without introducing significant calculation into the process of changing of skipfield nodes from unskipped to skipped and vice-versa. In addition it entirely removes branching instructions from the iteration process. The pattern is currently utilised by a C++ data container, 'Colony' [20], as a method of denoting erased objects within the container. This makes it possible for a Colony to iterate from one non-erased object to the next with O(1) time complexity, which is a requirement for C++ container iterators [21]. Taking the boolean skipfield example above, an equivalent low complexity jump-counting pattern is:

0 4 0 0 4 0 0 0 1 0

This field is interpreted as follows: when iterating from left to right, from skipfield index 2 we skip forward 4 places and from skipfield index 9 we skip forward 1 place. When reverse-iterating from right to left, from index 9 we skip backward 1 place, and from index 5 we skip backward 4 places. A simple solution, but efficient modification of this type of skipfield becomes somewhat more complex.

A review of the scholarly literature has shown no similar pre-existing works on replacements for boolean skipfields other than the high complexity jump-counting pattern[3]. The differences between this and the low complexity jump-counting pattern are explored in Section 6, but can be summarised as: a middle node's value is irrelevant and unmaintained in the low complexity jump-counting pattern but relevant and maintained as a particular numeric sequence in the high complexity jump-counting pattern.

We will begin by providing definitions to aid explanation of the rest of the pattern, followed by basic procedures for iteration and skipfield modification. Then procedures for modification of multiple consecutive skipfield numbers ('nodes') will be discussed. Lastly, additional areas of interest such as parallel processing will be covered.

2 Definitions

For the purposes of this document, the following terms are defined:

LCJC: Abbreviation of 'low complexity jump-counting'.

HCJC: Abbreviation of 'high complexity jump-counting'.

Skipfield: An array of integers used to skip over certain objects in an accompanying data structure during iteration or processing. Denoted by S in equations.

Node: A single integer within a skipfield, corresponding to some external object which may or may not be skipped, depending on the state of the skipfield. For a boolean skipfield this would be 0 or 1, while for a LCJC skipfield this can be any non-negative integer.

Skipblock: A contiguous sequence of skipped nodes within a skipfield. In the following LCJC skipfield:

0 4 0 0 4 0 0 0 1 0

all of the nodes from the first '4' to the second '4' are part of a skipblock. In addition the ‘1’ is also a skipblock (of length = 1). In a boolean skipfield, any node which is ‘1’ instead of ‘0’ is part of a skipblock. For example:

1 1 1 0 0 0 1 1 0 0

Here the first 3 nodes are part of a skipblock of length 3. The 7th and 8th nodes are part of a skipblock of length 2.

Start node: The first node in any skipblock. In the LCJC skipfield above the first ‘4’ is a start node, as is the ‘1’. In the Boolean skipfield above both the first and seventh nodes are start nodes. The index of this node is denoted by s in equations.

End node: The last node in any skipblock. In the LCJC skipfield above the second ‘4’ is a start node, as is the ‘1’. In the Boolean skipfield above the third node is an end node, as is the 8th. The index of this node is denoted by e in equations.

Middle node: Any node in a skipblock which is not a start or end node. In the LCJC skipfield example above the third and fourth nodes are middle nodes. In the Boolean skipfield example above only the second node is a middle node.

Current node: The node currently being modified during processing. The index of this node is denoted by i in equations.

3 Iteration

The LCJC skipfield is similar to a boolean skipfield in that a non-zero skipfield node indicates a skip. The difference being, for a boolean skipfield the skip is only ever of one node, whereas for a LCJC skipfield the skip can be many nodes. For the following skipfield:

0 3 0 3 0 0 0 0 0 0

the 1st node is unskipped. The second node is non-zero and indicates a skipblock of 3 nodes, meaning that the 2nd, 3rd and 4th nodes are all skipped. All subsequent nodes after the 4th are unskipped.

Iterating forwards from one node to the next follows this process:

1. Add one to the current skipfield index.

2. Add the value of the skipfield node at the current skipfield index, to the current skipfield index.

In equation form this can be expressed as follows:

$$i := i + 1 \tag{1}$$

$$i := i + S_i \tag{2}$$

Similarly, iterating in reverse follows this process:

1. Subtract one from the current skipfield index.

2. Subtract the value of the skipfield node at the current skipfield index, from current skipfield index.

$$i := i - 1 \tag{3}$$

$$i := i - S_i \tag{4}$$

So, when iterating over the following skipfield in reverse:

0 5 0 4 2 5 0 0 0 0

the last four nodes are zero, and therefore unskipped, so the second part of the procedure above subtracts nothing from the index. But when iterating from the 7th to the 6th node, whose value is 5, the following occurs:

1. i := i - 1 (i := 7 - 1)

2. i := i - S_i (i := 6 - 5)

which results in the index being 1.

4 Modification of singular skipfield nodes

4.1 Changing a skipfield node from unskipped to skipped

We start by reading the nodes to the left and right of the node we wish to modify. If the current node is at the beginning or the end of the skipfield, the respective non-existent node to the left or right counts as being zero ('unskipped') for the purposes of the pattern. There are four potential outcomes in this situation:

4.1.1 If both left and right nodes are non-zero

This means the current node is between two skipblocks and we must join those skip-blocks together. The procedure for doing so is:

1. Subtract the left node's value from the current node's index to get the index of the left skipblock's start node.

2. Add the value of the right node to the current node's index to get the index of the right skipblock's end node.

3. Set the left skipblock's start node and the right skipblock's end node to the value of the left node + the value of the right node + 1.

$$j := S_{i-1} \tag{5}$$

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

$$S_{i-j} := S_{i+k} := j + k + 1 \tag{7}$$

4.1.2 If only the left node is non-zero:

This indicates that the current node has an adjacent skipblock on the left, in which case we simply add the current node to that skipblock. The procedure for doing so follows:

1. Subtract the left node's value from the current node's index to get the index of the left skipblock's start node.

2. Set the value of the left skipblock's start node and the current node, to the left node's value + 1.

$$j := S_{i-1} \tag{8}$$

$$S_{i-j} := S_i := j + 1 \tag{9}$$

4.1.3 If only the right node is non-zero:

This means the node in question has an adjacent skipblock on the right, in which case we add the current node to that skipblock:

1. Add the right node's value to the current node's index to get the index of the right skipblock's end node.
2. Set the value of the right skipblock's end node and the current node, to the value of the right node + 1.

$$k := S_{i+1} \tag{10}$$

$$S_{i+k} := S_i := k + 1 \tag{11}$$

4.1.4 If both left and right nodes are zero:

This means the node in question has no skipblocks on either side, in which case we create a new skipblock of 1 nodes:

1. Set the value of the current node to 1.

$$S_i := 1 \tag{12}$$

4.1.5 Example

We will start utilising a LCJC skipfield with no skipped nodes:

0 0 0 0 0 0 0 0 0 0

If we want to change the state of the 5th node from unskipped to skipped, we read the values of the skipfield nodes to the left and right of the current node, and find both are zero. We therefore follow procedure 5.1.4 above, setting the 5th node to 1:

0 0 0 0 1 0 0 0 0 0

Now consider if we want to change the state of the 1st node from unskipped to skipped. Since this node is at the beginning of the skipfield, the node to the left is non-existent and is interpreted as being zero for the purposes of the pattern. So both left and right nodes are 'zero' and we again follow procedure 5.1.4, setting the 1st node to 1:

1 0 0 0 1 0 0 0 0 0

Next we wish to change the 4th node from unskipped to skipped. The node to the right is non-zero and the node to the left is zero, so we follow procedure 5.1.3. We add the value of the right node ('1') to the index of the current node (4) to find the index of the right skipblock's end node. We set this end node and the current node, to the right node's value + 1:

1 0 0 2 2 0 0 0 0 0

Now we wish to change the 2nd node from unskipped to skipped. The node to the left is non-zero while the node to the right is zero, so we follow procedure 5.1.2. We subtract the value of the left node ('1') from the current node's index (2) to find the index of the left skipblock's start node (1). Once again the start and end nodes of this skipblock are the same node. We set the start node and the current node, to the left node's value + 1:

2 2 0 2 2 0 0 0 0 0

Finally we wish to change the 3rd node from unskipped to skipped. Both nodes to the left and right are non-zero, so we must join the two skipblocks. First we subtract the value of the left node ('2') from the current node's index (3) to find the index of the left skipblock's start node (1). We then add the value of the right node ('2') to the current node's index to find the index of the right skipblock's end node (5). Lastly we set the value of the start node on the left and the end node on the right to the left node's value + the right node's value + 1:

5 2 0 2 5 0 0 0 0 0

It should be obvious at this point that the value of any nodes between the start and end node of a skipblock are irrelevant during iteration as they are skipped over, and are also not read during the modification of nodes from unskipped to skipped. They also have no effect on the modification of nodes from skipped to unskipped, as we will see in the following subsection.

4.2 Changing a skipfield node from skipped to unskipped

To accomplish this it is necessary to know whether the node to be modified is a start node, end node or middle node. If it is a middle node, we must also know the index of either the start node or the end node of that skipblock. From the start node we can find the end node's index by adding the start node's value to it's index then subtracting 1. And vice versa, from the end node we can find the index of the start node by subtracting the end node's value from it's index then adding 1.

In the LCJC pattern it is not possible to locate a start or end node from the value of a middle node, unlike the HCJC pattern. Storing or otherwise recording skipblock start and end node locations is left to the developer and the best approach will vary based on the situation. This is explored more fully in Section 6. There are four possible scenarios when changing a node from skipped to unskipped:

4.2.1 The node to be modified is the end node of a skipblock

In this case, we truncate the skipblock to the left. The process is as follows:

1. Subtract 1 from the current node's value and store the result as j.
2. Subtract j from the current node's index to find the index of the skipblock's start node.
3. Set the node to the left of the current node, and the end node, to j.

4. Set the current node to 0.

$$j := S_i - 1 \tag{13}$$

$$S_{i-j} := S_{i-1} := j \tag{14}$$

$$S_i := 0 \tag{15}$$

4.2.2 The node to be modified is the start node of a skipblock

Here we truncate the skipblock to the right:

1. Subtract 1 from the current node's value and store the result as j.
2. Add j to the current node's index to find the index of the skipblock's end node.
3. Set the node to the right of the current node, and the end node, to j.
4. Set the current node to 0.

$$j := S_i - 1 \tag{16}$$

$$S_{i+j} := S_{i+1} := j \tag{17}$$

$$S_i := 0 \tag{18}$$

4.2.3 The node to be modified is both the start and end node of a skipblock

In this scenario the current node is a 1-node skipblock and can simply be set to zero.

$$S_i := 0 \tag{19}$$

4.2.4 The node to be modified is neither the start nor the end node of a skipblock

In this scenario the node in question is a middle node and the skipblock must be split into two skipblocks, as follows:

1. Subtract the current node's index from the end node's index and set the value of the node to the right of the current node, and the end node, to the result.
2. Subtract the start node's index from the current node's index and set the value of the node to the left of the current node, and the start node, to the result.
3. Set the value of the current node to 0.

$$S_e := S_{i+1} := e - i \tag{20}$$

$$S_s := S_{i-1} := i - s \tag{21}$$

$$S_i := 0 \tag{22}$$

4.2.5 Example

Let's take the final result of our example in section 4.1.5:

5 2 0 2 5 0 0 0 0 0

We'll start by modifying index 5, which is the end node, so that it is no longer skipped. Following the procedure in section 4.2.1, we take the value of that node ('5'), subtract 1 from it ('4'), and subtract that value from the end node's index (5) to find the start node's index (1). We then set both the start node and the node to the left of the current node, to the value of the current node - 1. Lastly we set the current (end) node to 0, which gives us the following result:

4 2 0 4 0 0 0 0 0 0

Now, suppose we change the start node at index 1 to be unskipped. Following the procedure in section 4.2.2, we take the start node's value ('4'), subtract 1 from it ('3'), and add that value to the start node's index to find the end node's index (4). Lastly we set the node to the right of the current node and the end node, to the current node's value - 1 ('3'), and set the current node to 0:

0 3 0 3 0 0 0 0 0 0

Lastly we shall modify the 3rd node in the skipfield, setting it to be unskipped. As discussed earlier, since this is a middle node we must already have knowledge of the start and end node's indexes prior to modification. Following the procedure in section 4.2.4, we first subtract the current node's index (3) from the end node's index (4) and store the result as j ('1'). We then set the value of the node to the right of the current node, and the end node, to j (in this case these two nodes are the same node).

Next we subtract the start node's index (2) from the current node's index (3) and store the result as k ('1'). We then set the value of the node to the left of the current node, and the start node, to k (in this case these two nodes are actually the same node). Finally we set the value of the current node to 0. The result is as follows:

```
0 1 0 1 0 0 0 0 0 0
```

As you can see the value of the 3rd node is unchanged, but it is no longer skipped during iteration.

5 Modification of multiple consecutive skipfield nodes at once

In some situations it is advantageous to consider how the modification of multiple nodes might be optimised; for example, in the context of a data container where several consecutive objects might be erased in a single function, and the skipfield updated accordingly. Transformations of multiple consecutive skipfield nodes from unskipped to skipped or vice-versa are O(1) amortized operations with the LCJC pattern. When compared to a boolean skipfield's O(N) time complexity for the same operation, a potential performance advantage for large N can be seen.

5.1 Changing multiple consecutive skipfield nodes from unskipped to skipped

To start with we specify the range of skipfield nodes we want to modify by defining a 'left node' which is the first node we wish to modify, and a 'right node' which is the last node we wish to modify. Both nodes must be unskipped nodes, but the nodes between them may be either unskipped or skipped. We then follow this process:

1. First we check the value of the node to the left of the left node. If it is zero (or the left node is at the start of the skipfield) we do nothing. But if it is non-zero, this indicates a pre-existing skipblock to the left. In the latter case we subtract the non-zero value from the left node's index to find the index of the left skipblock's start node. We then change the left node's index to the start node's index.

2. We check the value of the node to the right of the right node. If it is zero (or the right node is at the end of the skipfield), again we do nothing. If it is non-zero, this indicates a pre-existing skipblock to the right. In the latter case we add the non-zero value to the right node's index to find the index of the skipblock's end node. We then change the right node's index to this end node's index.

3. Finally we subtract the index number of the left node from the index number of the right node, add 1, and set the value of both the left and right nodes to this value.

Or in pseudocode (where l and r are the left and right node indexes respectively):

```
1. IF $l$ != 1 AND $S_{l-1}$ != 0 THEN
       $l := l - S_{l-1}$
2. IF $r$ != size($S$) AND $S_{r+1}$ != 0 THEN
       $r := r + S_{r+1}$
3. $S_l := S_r := (r - l) + 1$
```

5.1.1 Example

Taking the skipfield below:

0 0 0 0 0 0 0 0 0 0

If we wish to change nodes 3 through to 5 from unskipped to skipped, we take node 3 as the 'left node' and node 5 as the 'right node'. Following the pseudocode in section 5.1, we end up with:

1. l != 1 BUT $S_{l-1} = 0$, so no action taken

2. r != size(S) BUT $S_{r+1} = 0$, so no action taken

3. S_l := S_r := (r - l) + 1 (S_3 := S_5 := 3)

Which results in the following skipfield:

0 0 3 0 3 0 0 0 0 0

Immediately we can see that nodes 3 through to 5 will be skipped during iteration. Now, if we make a second modification of nodes, this time setting nodes 6 through to 7 to be skipped, we follow the process again:

1. l != 1 AND S_{l-1} != 0, so:
   l := l - S_{l-1} (l := 6 - 3)

2. r != size(S) BUT $S_{r+1} = 0$, so no action taken

3. S_l := S_r := (r - l) + 1 (S_3 := S_7 := 5)

Which results in:

0 5 0 3 0 5 0 0 0 0

In this instance we triggered the first line of the pseudocode's condition and extended the existing skipblock. We can see from the result that node values *between* the left and right nodes are not changed; node 5 is still at value 3. There is no need to change those values as they are skipped over during iteration.

Now we make a third modification, changing nodes 2 through to 8 from unskipped to skipped:

1. l != 1 BUT $S_{l-1} = 0$, so no action taken

2. r != size(S) BUT $S_{r+1} = 0$, so no action taken

3. S_l := S_r := (r - l) + 1 (S_2 := S_8 := 7)

Which results in:

0 7 5 0 3 0 5 7 0 0

Here we see that although the range already contained a skipblock, but we essentially ignore this and execute step 3 of the pseudocode. The skipfield will still function as intended during iteration.

5.2 Changing multiple consecutive skipfield nodes from skipped to unskipped

Once again we specify the range of skipfield nodes we want to modify by defining a 'left node' and a 'right node'. But in this case, in order for the process to work both the left and right nodes must be skipped nodes, and the start and end nodes of the skipblock(s) involved must be known. This technique can span multiple skipblocks (essentially ignoring the unskipped nodes between those blocks), in which case, the start node is the start node of the skipblock which the left node is within, and the end node is the end node of the skipblock which the right node is within.

To start, we check to see if the left node is the start node. If it isn't, we subtract the start node's index from the left node's index, and set both the start node and the node to the left of the left node, to the resultant value. We then check to see if the right node is the end node. If it isn't, we subtract the right node's index from the end node's index, and set both the end node and the node to the right of the right node, to the resultant value. Lastly we set the value of all nodes from the left node through to the right node, to 0.

In pseudocode this reads as follows (where l and r are the indexes of the left and right nodes respectively):

```
1. IF $l$ != $s$ THEN
      $S_s := S_{l-1} := l - s$

2. IF $r$ != $e$ THEN
      $S_e := S_{r+1} := e - r$
```

3. $S_{l \to r} := 0$

5.2.1 Example

Consider the following skipfield:

0 7 5 0 3 0 5 7 0 0

If we want to change nodes 3 through to 6 to be unskipped (currently skipped), we follow the pseudocode from Section 5.2, with $l = 3$, $r = 7$, $s = 2$ and $e = 8$:

1. l != s, so:

   $S_s := S_{l-1} := l$ - s $\qquad (S_2 := S_2 := 1)$

2. r != e, so:

   $S_e := S_{r+1} := r$ - e $\qquad (S_8 := S_7 := 2)$

3. $S_{l \to r} := 0$

Which results in the following:

0 1 0 0 0 0 2 2 0 0

Since the node to the left of the left node was the start node, the first skipblock has a length of 1 (and is effectively written to twice in the above pseudocode).

6 Differences between the LCJC pattern and the HCJC pattern

In order to change a node's status from 'skipped' to 'unskipped' in an HCJC skipfield [3], you need only know the location of the node you wish to modify. From the value of this node you can work out the location of the skipblock's start and end nodes, and update the skipblock appropriately. However, in the LCJC pattern one must know both the location of the node one wishes to modify, and, if it is a middle node, also the location of either the start node or the end node of the skipblock it is contained within.

In the LCJC pattern, like the HCJC pattern, one can determine the location of the end node of a skipblock from the value of the start node, and vice-versa. Unlike the HCJC pattern, in the LCJC pattern only values for the start and end nodes are specified and maintained. The values of middle nodes in the HCJC pattern are used and maintained in a specific numeric sequence, whereas in the LCJC pattern those values are unused and unmaintained.

Hence while an HCJC skipfield of length 10 which skips items 1-3 and 6-9 will always look like this:

3 2 3 0 0 4 2 3 4 0

An equivalent LCJC skipfield could look like this:

3 0 3 0 0 4 0 0 4 0

or this:

3 2 3 0 0 4 3 0 4 0

or this:

3 5 3 0 0 4 2 6 4 0

The value of middle nodes in a LCJC skipfield depend on what prior modifications have occurred, but in themselves have no meaning or usage. Each LCJC skipfield above is equally valid, as all middle nodes are skipped over during iteration, and their values are not read when modifying skipblock nodes. The lack of maintenance of middle nodes enables O(1) time complexity when modifying nodes, unlike the HCJC pattern's undefined time complexity for the same process.

But it comes at the cost, when changing a node from skipped to unskipped, of needing prior knowledge of (a) whether the node being modified is a middle node, and if it is, then also (b) the location of a start or end node for the skipblock which that middle node is in. This is because changing a middle node to an unskipped status necessitates splitting a single skipblock into two skipblocks, so the start and end node values of the original skipblock must be modified in order for the skipfield to be read correctly during iteration.

Even if one were to iteratively read consecutive nodes to the left or right of a given middle node, it would not be possible to determine where it's skipblock began or ended based on the node values, as middle nodes for a LCJC pattern can have any value, including zero (see Section 7.2 for an exception to this).

In order to find the start and end nodes of a skipblock from a middle node without prior knowledge, one would have to iterate from the start of the skipfield until the skipblock containing the middle node was found, by comparing node indexes before and after skipblocks with the index of the middle node in question. As a skipfield can be large, this precludes efficient modification of middle nodes from skipped to unskipped without prior knowledge of a start or end node index.

This could mean that in some scenarios it is more practical to avoid modifying middle nodes entirely and to only modify start or end nodes in a LCJC pattern. This may result in more efficiency as modification of start and end nodes does not involve splitting a skipblock into two, and therefore invokes fewer operations than modifying a middle node. Furthermore, splitting a skipblock into two skipblocks results in more skip operations during iteration over the skipfield, which decreases data locality and in turn may inhibit performance [22] [23]. So avoiding modification of middle nodes may also has the benefit of enabling more efficient iteration.

7 Additional areas of application and manipulation

7.1 A method for reducing skipfield memory usage

Whatever bit-depth is used to construct an LCJC skipfield (for example, 8-bit, 16-bit or 32-bit unsigned integers), that bit-depth determines the maximum jump possible for a given skipfield node. For example the maximum number expressible by an 8-bit unsigned integer is 255, so an LCJC skipfield constructed from 8-bit unsigned integers is limited to a maximum jump range of 255.

In order to express larger runs of skipped nodes, one must use a larger bit-depth for the skipfield, which increases memory use. An unsigned 16-bit integer skipfield allows a maximum jump-range of 65535 nodes, 32-bit integers allow a jump-range of 4294967295 nodes, and so on. Larger bit-depths however constitute a waste of memory, particularly if large jump ranges are rare. On embedded platforms where memory is often limited and carefully controlled, this additional memory use could be problematic [24].

However there is a method of circumventing this memory loss, if one is willing to sacrifice some performance. If we take the following LCJC unsigned 8-bit integer skipfield:

254   0   0   (insert 248 zero nodes here)   0   0   254   0   0   0

If we change the second-from-last and third-from-last nodes from unskipped to skipped, now we have a jump-length of 256 for the skipblock, which is unexpressible in 8-bit integers. But what we can do, is the following:

255   1   0   (insert 248 zero nodes here)   0   0   1   0   255   0

What we are doing above is (a) making 255 a special number which is not a jump-range and (b) treating the two nodes subsequent to the start node (and prior to the end node) as a singular number which is the actual jump-range, but only when the start/end nodes are 255. We will call this singular number the "extended jump-range". The extended jump-range in the example above is a pair of 8-bit unsigned integers and therefore the number is a 16-bit unsigned integer constituted of a 1 and a 0, which results in the following bits when concatenated:

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

ie. 256 in binary. If we want an extended jump-range larger than 16-bits we can concatenate more nodes; for example, 4 consecutive nodes results in a 32-bit extended jump-range, 8 in a 64-bit extended jump-range. This means that using only 8-bit unsigned integers we can construct a skipfield such that jump-ranges can be up to the maximum number expressible in 64-bits.

This technique comes with a cost, that of increased instructions during iteration and manipulation. During iteration we add an IF statement to check whether the skipfield node is 255 in order to trigger the above "extended jump-range" behaviour, and a concatenation statement to read the two-or-more subsequent/prior (depending on whether we are iterating forwards or in reverse over the skipfield) nodes as an integer with a greater bit-depth. Forward iteration becomes (where X is the number of nodes we want to concatenate for our extended jump-range):

1. Add one to the current skipfield index.
2. Check to see if the value of the skipfield node at the current skipfield index is 255.
3. If it is, concatenate X subsequent nodes as a singular integer and add this integer to the current skipfield index.
4. If it is not, add the value of the skipfield node at the current skipfield index, to the current skipfield index.

Reverse iteration becomes:

1. Subtract one from the current skipfield index.
2. Check to see if the value of the skipfield node at the current skipfield index is 255.
3. If it is, concatenate X prior nodes as a singular integer and subtract this integer from the current skipfield index.
4. If it is not, subtract the value of the skipfield node at the current skipfield index, from the current skipfield index.

This introduces a branch into jump-counting iteration, compared to the standard iteration method, which is branch-free.

Manipulation of the skipfield ie. changing nodes from skipped to unskipped and vice-versa, also becomes slightly more complicated. One has to also check here if the start/end node one is 255, and if so, make appropriate adjustments to manipulate and move the extended jump-range values. In addition, if changing from skipped to unskipped, once the extended jump-range goes below the maximum expressible value of the skipfield's bit-depth (eg. 255 in the example above), the start/end nodes go back to the normal way of expressing a jump, containing a value lower than the maximum expressible value (eg. 254 in the example above).

If changing from unskipped to skipped and the node being changed is consecutive to an existing skipblock, one must check if the existing skipblock is already at the skipfield bit-depth's maximum value minus 1, and if so change the skipblock so it is using an extended jump-range sequence instead. In the example above, this would be when a consecutive skipblock had a start/end node value of 254 prior to adjustment.

7.2 Intersection of multiple skipfields

Multiple LCJC skipfields may be intersected using a 'tail-chasing' technique. All input skipfields being intersected are processed simultaneously, starting from the first node in each skipfield and proceeding sequentially through each subsequent node until the end of all skipfields is reached. This process does not assume or require that all skipfields be of the same length. The steps for this process follow below. Here, i is the index of the nodes simultaneously being processed in each skipfield, C is the set of all input skipfields currently being processed and O is the output skipfield. All nodes in the output skipfield are assumed to be zero before processing begins.

1. (a) Here we check all node values at index i in C. If a non-zero value is found, we go to step 2. Otherwise, i is incremented by one.
   (b) If at this point the end of any of the individual skipfields is reached, that skipfield is removed from the set of C.
   (c) If the end of all skipfields in C is reached, we end the process. Otherwise we repeat step 1.

2. (a) We take the largest non-zero value encountered and store this as k. This is the size of the largest skipblock we have encountered.

   (b) We add k to i and store the result as j. This is the index of the node immediately after the skipblock.

   (c) We also store the current value of i as s (the index of the start node of the skipblock encountered).

   (d) Lastly we store a reference to the skipfield within which we encountered the skipblock as X, remove X from C and proceed to step 3.

3. (a) We continue to iterate over C, checking for non-zero values and incrementing i by 1 until $i = j$.

   (b) If a non-zero value is encountered, we add the value to i and store it as l. If l is smaller or equal to j we ignore the result and repeat step 3. If l is larger than j we go to step 4.

   (c) Once $i = j$, we write k to O at indexes s and $j - 1$, then go back to step 1.

4. (a) We've found a skipblock which extends further than the previously-encountered skipblock, so we now add $(l - j)$ to k, then set j to the value of l.

   (b) We then add the skipfield currently stored as X back into the set of C.

   (c) Lastly we set X to the skipfield where the new non-zero value was encountered, remove X from C, then go back to step 3.

7.3 Parallel processing

The LCJC skipfield is specifically serial in nature and as such is unlikely to be of great use in parallel architectures such as CUDA[25]. In such environments processing is typically performed on many items at once rather than iterating over singular items sequentially. This negates the LCJC pattern's advantages which are primarily related to serial iteration time complexity. Further, the default numeric layout of the LCJC skipfield is not ideal for parallel processing. Operations such as 'compact' utilising an exclusive[26] or inclusive[27] scan would require a LCJC skipfield to be converted to the equivalent boolean skipfield prior to processing via a serial iteration over the skipfield.

This serial iteration is necessary simply because middle nodes can have a value of zero, and therefore cannot be recognised as being skipped nodes from their value alone - their skipped status comes from the skipfield as a whole. However with a minor modification the LCJC pattern can be made more usable in parallel computing environments. Specifically this involves setting middle nodes to the value of 1 (or any other fixed non-zero value), instead of leaving them unmodified.

The effect of this modification is that the skipfield can be used as a mask (using zero values to indicate items to be processed and non-zero values to indicate items not to be processed), or converted to a mask appropriate to the architecture via a parallel operation. This mask could then be used by gather and scatter [28][29] operations for SIMD [30] processing, for example using AVX2 or AVX512F instructions on an x86 processor [31].

To re-implement the LCJC pattern in this way, the following changes are necessary:

1. When changing a singular skipfield node from unskipped to skipped status, if the node is between two skipblocks, (ie. the value of both left and right nodes are non-zero) it's value is set to 1 (or any other non-zero value), instead of being left as 0.

2. When changing multiple consecutive nodes from unskipped to skipped statuses, those nodes which do not become start or end nodes of a skipblock are set to a value of 1 (or any other non-zero value), instead of being left as 0.

Since the above represents additional operations per modification, I have kept this separate from the default LCJC pattern, which is intended for serial usage and may benefit from the reduced instruction count. Note that adding the above instructions means that it also becomes possible to determine the location of the start and end nodes of a skipblock from any given middle node, simply by iterating to the left and right of the node until a node with a value of 0 is reached. Without the above instructions this is not possible because a middle node could be 0, as mentioned.

7.4 Multi-threaded/Multi-CPU access

Concurrent modification of a LCJC skipfield is more difficult than with a boolean skipfield, as the node states cannot be changed independently of the state of adjacent nodes. Hence, concurrent modification would likely necessitate the use of mutexes [32]. With a boolean skipfield each node is entirely independent of all other nodes, making concurrent modification possible via per-node systems such as Atomics [33].

8 Conclusion

There are many use cases which may benefit from switching from a boolean pattern to a LCJC pattern for skipfields, including: object containers which employ skipping mechanisms to avoid reallocation[4] and its associated computational costs[34], signal processing and resource grids. These benefits take the form of time complexity reduction and branch-free computation during iteration. While it is hard to predict all potential areas of application, we can assume that any use case involving large numbers of skipped objects and sequential access of a skipfield may be appropriate for substitution of a LCJC skipfield. In situations where parallel access is required, however, a boolean skipfield will likely still be preferable.

9 Acknowledgements

Many thanks to Dr Robert Hurley for proof-reading and correcting many aspects of this document, as well as illuminating new areas of exploration. You have a good eye, and an extensive knowledge of subjective bias.

10 References

References

[1] 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.

[2] 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.

[3] Bentley, Matthew R., *The high complexity jump-counting pattern*, `https://plflib.org/matt_bentley_-_the_high_complexity_jump-counting_pattern.pdf`, 2019.

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

[5] 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.

[6] Claypool, Mark, and Kajal Claypool (2010). "Latency can kill: precision and deadline in online games." Proceedings of the first annual ACM SIGMM conference on Multimedia systems. ACM.

[7] Claypool, Mark (2005), "The effect of latency on user performance in real-time strategy games.", Computer Networks 49.1 (2005): pp 52-70.

[8] Savery, C. and Graham, T.C. (2011). It's about time: confronting latency in the development of groupware systems. Proceedings of the ACM 2011 conference on Computer supported cooperative work (pp. 177-186). ACM.

[9] Lockwood, John W., et al (2012), "A low-latency library in FPGA hardware for high-frequency trading (HFT).", 2012 IEEE 20th annual symposium on high-performance interconnects. IEEE.

[10] Moallemi, Ciamac C., and Mehmet Salam. (2013), "OR ForumThe cost of latency in high-frequency trading.", Operations Research 61.5: 1070-1086.

[11] Goldstein, Michael A., Pavitra Kumar, and Frank C. Graves. (2014), "Computerized and highfrequency trading.", Financial Review 49.2: 177-202.

[12] Bamakhrama, Mohamed A., and Todor Stefanov. (2012), "Managing latency in embedded streaming applications under hard-real-time scheduling.", Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis. ACM.

[13] Ivutin, Alexey, and Eugene Larkin (2014), "Estimation of latency in embedded real-time systems.", 2014 3rd Mediterranean Conference on Embedded Computing (MECO). IEEE.

[14] Eisenring, Michael, Lothar Thiele, and Eckart Zitzler (2000), "Conflicting criteria in embedded system design.", IEEE Design and Test of Computers 17.2: 51-59.

[15] Alizadeh, Mohammad, et al. (2012), "Less is more: trading a little bandwidth for ultra-low latency in the data center.", Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12).

[16] Xu, Yunjing, et al. (2013), "Small is better: Avoiding latency traps in virtualized data centers.", Proceedings of the 4th annual Symposium on Cloud Computing. ACM.

[17] Munir, Ali, Ihsan Ayyub Qazi, and Saad Bin Qaisar. (2013), "On achieving low latency in data centers.", 2013 IEEE International Conference on Communications (ICC). IEEE.

[18] Younge, Andrew J., et al. (2011), "Analysis of virtualization technologies for high performance computing environments.", 2011 IEEE 4th International Conference on Cloud Computing. IEEE.

[19] McCalpin, John D. (1995), "Memory bandwidth and machine balance in current high performance computers.", IEEE computer society technical committee on computer architecture (TCCA) newsletter 1995: 19-25.

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

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

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

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

[24] Lekatsas, Haris and Wolf, Wayne. (1998), Code compression for embedded systems, Proceedings of the 35th Annual Design Automation Conference: 516-521.

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

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

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

[28] Lewis, John G., and Horst D. Simon. (1988), The impact of hardware gather/scatter on sparse Gaussian elimination., SIAM Journal on Scientific and Statistical Computing 9.2 : 304-311.

[29] Tan, Hongbing & Chen, Haiyan & Liu, Sheng & Wu, Jianguo. (2017), Modeling and evaluation for gather/scatter operations in Vector-SIMD architectures. 143-148. 10.1109/ASAP.2017.7995271.

[30] Weiss, Michael. (1989), Parallel languages, vectorization, and compilers., Proceedings of the Thirteenth Annual International Computer Software & Applications Conference. IEEE.

[31] Maly, Lukas, Jan Zapletal, and Michal Merta. (2019) Vectorized evaluation of boundary integral operators., AIP Conference Proceedings. Vol. 2116. No. 1. AIP Publishing.

[32] Courtois, Pierre-Jacques, Frans Heymans, and David Lorge Parnas (1971), Concurrent control with 'readers' and 'writers'., Communications of the ACM 14.10, pp 667-668.

[33] Oni, Joo-On, Fawnizu Azmadi Hussin, and Nordin Zakaria. (2018), Fine-Grained Overhead Analysis Utilizing Atomic Instructions for Cross-ISA Dynamic Binary Translation on Multicore Processor., International Conference on Intelligent and Advanced System (ICIAS). IEEE.

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