Parallel Pairwise Operations on Data Stored in DNA: Sorting, Shifting, and Searching

Tonglin Chen
Department of Electrical and Computer Engineering, University of Minnesota, USA

Arnav Solanki
Department of Electrical and Computer Engineering, University of Minnesota, USA

Marc Riedel[1]
Department of Electrical and Computer Engineering, University of Minnesota, USA

Abstract

Prior research has introduced the Single-Instruction-Multiple-Data paradigm for DNA computing (SIMD DNA). It offers the potential for storing information and performing in-memory computations on DNA, with massive parallelism. This paper introduces three new SIMD DNA operations: sorting, shifting, and searching. Each is a fundamental operation in computer science. Our implementations demonstrate the effectiveness of parallel pairwise operations with this new paradigm.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies; Applied computing → Computational biology

Keywords and phrases Molecular Computing, DNA Computing, DNA Storage, Parallel Computing, Strand Displacement

Digital Object Identifier 10.4230/LIPIcs.DNA.27.11

Funding The authors are funded by DARPA grant #W911NF-18-2-0032

Acknowledgements We thank David Soloveichik, Olgica Milenkovic, Boya Wang, and Cameron Chalk.

1 Introduction

Beginning with the seminal work of Adelman a quarter-century ago [1], DNA computing has promised the benefits of massive parallelism in operations. More recently, there has been considerable interest in DNA storage [3, 4]. A particularly promising approach is to encode data by "nicking" DNA with editing enzymes such as PfAgo and CRISPR-Cas9 [9, 12]. A novel paradigm that combines this form of data storage with computation, dubbed "SIMD DNA", was introduced in 2019 [13]. Data is stored on potentially long DNA strands, divided into "cells", each storing a single bit. Nicks and denaturing create open toeholds in each cell. Toehold-mediated strand displacement [10, 14] is used to implement computation on the stored values.

This paper first proposes a new encoding system for SIMD DNA computation, suitable for general pairwise operations. Then it presents three novel applications using the new encoding system. The first is a binary bubble sorting algorithm (equivalent to rule 184 with elementary cellular automata [7, 8]). We show that sorting can be performed in only N parallel steps, where N is the number of bits to be sorted. The second application is a left-shifting operation (equivalent to rule 170 with elementary cellular automata), performed in a single parallel step. The third application is a parallel search algorithm that returns an answer as to whether a query substring is present in a target string. In principle, the algorithm can return an answer

[1] corresponding author

in $\log(n)$ steps, but our implementation requires between $\log(n)$ and n steps to complete, depending on the problem size and implementation constraints, where n is the length of the query string. Note that the parallelism is still impressive, assuming that the query string length n is much smaller than the target string length m. All three applications are of immediate practical interest, as many forms of computation on stored data entail some form of sorting, shifting, and searching.

2 Background

2.1 Parallel computation using SIMD

SIMD is a computer engineering acronym for Single Instruction, Multiple Data [6], a form of computation in which multiple processing elements perform the same operation on multiple data points simultaneously. It contrasts with the more general class of parallel computation called MIMD (Multiple Instructions, Multiple Data), where multiple processing elements can perform completely different operations on multiple data points simultaneously. While general MIMD parallelism might be desirable, it is often not practical. Much of the modern progress in electronic computing power has come by scaling up SIMD computation with platforms such as graphical processing units (GPUs).

2.2 SIMD DNA structure

SIMD implemented on DNA is intriguing. It provides a means to transform stored data, perhaps large amounts of it, with a single parallel instruction. We will review the paradigm as we introduce our new encoding scheme and our new applications; of course, we do not claim credit for the original concepts. The reader is referred to [13].

SIMD DNA computation is predicated on the encoding scheme for data. Conceptually, we divide stretches of double-stranded DNA into "domains", where each domain is a contiguous sequence of nucleotides of some small specified length (typically 5 to 20). A sequence of several (typically 5 to 7) domains maps to a "cell" storing one binary bit. Whether a cell stores a 0 or a 1 depends upon topological variations, specifically the location of nicks, i.e., breaks in the DNA backbone. The nicks always occur on one strand of a double-stranded complex (generally the top strand in our examples); the other remains untouched.

The computation is carried out by a sequence of "instructions", where each instruction implements DNA strand displacement reactions on cells. Instructions are initiated by single-stranded "instruction strands" added to the solution. After the strand displacement cascades complete, any single-strand fragments in the solution are washed away; the original strand is kept and separated via a magnetic bead. After a sequence of instructions, the data is transformed to its final state. The readout can be performed via fluorescence or with Oxford nanopore devices [2], [9].

The general flow of SIMD DNA computation is summarized as follows and illustrated in Figure 1.

1. Design an encoding structure that best suits the algorithm.
2. Encode the data at specific locations, using enzymes to nick corresponding targets.
3. Gently denature the DNA, allowing segments between adjacent nicks to detach, exposing toeholds.
4. Execute instructions, implemented as strand-displacement operations.
5. Finally, read out data using fluorescence or with nanopores.

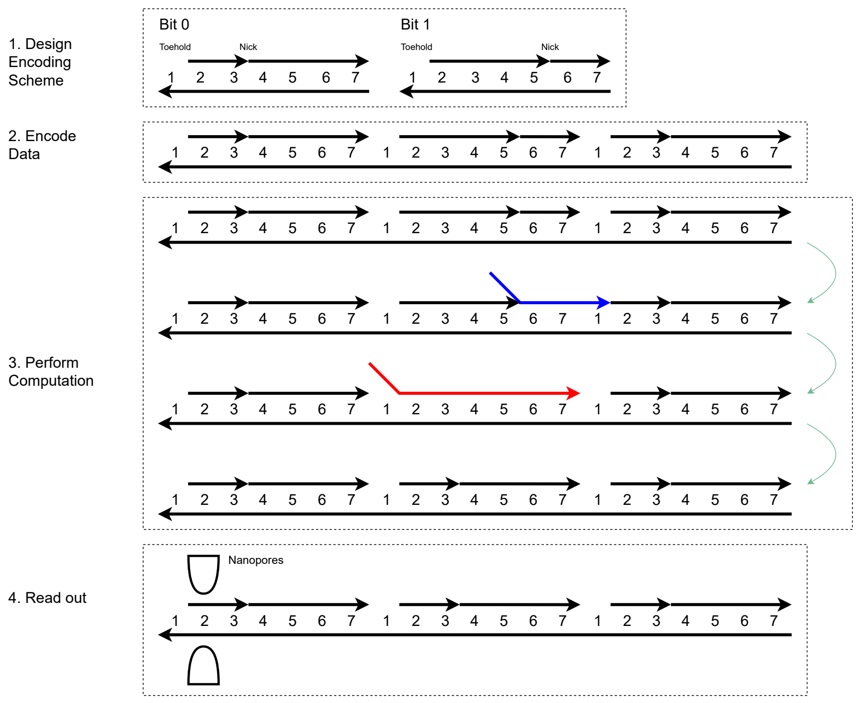


Figure 1 General Outline of SIMD DNA Computations. Arrowheads represent "nicks": breaks in the DNA backbone, performed with gene editing techniques. Integers represent "domains": contiguous sequences of nucleotides of some small, specified length. For convenience, we use the numbers 1 through 7 repeatedly; however, each copy of a number represents a distinct domain, consisting of a unique nucleotide sequence. Stage 1 shows the encoding of binary bits 0 and 1, based of different locations of toeholds and nicks. Note that domain 1 is always "exposed": the DNA backbone of the top strand is nicked, and the DNA is gently denatured until this segment falls off, exposing a toehold at this domain. Stage 2 shows an example of encoding the bits 010. Stage 3 illustrates the step in which computation is performed with strand displacement, in a general sense. Details of this step will be provided for specific algorithms in later sections. Note that, in this generic example, the location of nick in the second cell has changed at the end of stage 3. Stage 4 illustrates how nanopore sequencing could be used to perform readout.

3 Design of Encoding System

Several schemes for encoding binary data were proposed in prior work [13], each chosen to minimize the number of operations for a specific algorithm. Here we propose a new encoding scheme that works well for the broad class of algorithms that consist of parallel, pairwise operations. A requirement for running these algorithms is that the encoding scheme should allow the algorithm to recognize any combination of adjacent bits. This specification comes at the expense of more complexity for some algorithms, i.e., more operations per step than possible with a customized encoding.

The encoding scheme is shown in Figure 2. Each cell stores a single binary value (a "bit"). Each cell consists of 7 domains. We do not specify the actual nucleotide sequence of the domains here for simplicity. While preparing this cell, the top DNA strand must be nicked before and after domain 1. This strand can then be displaced by denaturing, creating an exposed toehold. Domain 1 is always exposed as a toehold in this representation. Domains 2 through 7 are covered. When storing a bit 0, we will nick the top strand between domains 3 and 4; when storing a bit 1, we will nick between domains 5 and 6. There are four possible

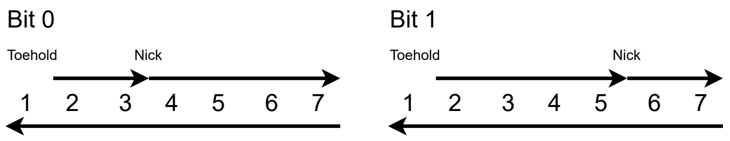


Figure 2 Bit representation in the encoding scheme. Horizontal lines represents DNA strands. Integers represent "domains": specific sequences of nucleotides. Arrow heads represent nicked positions: places where the phosphodiester bond in the backbone of the DNA strand has been broken, via gene-editing techniques. Cells store binary values. Each cell consist of 7 domains. Domain 1 is always exposed, forming a toehold.

pairings for two adjacent cells. Each will be detected using different domain combinations: for $(0, 0)$, domains 1, 2 and 3; for $(0, 1)$, domain 1 only; for $(1, 0)$, domains 6 through 3 with wrapping at domain 7 and 1; and for $(1, 1)$, domains 6, 7 and 1.

Before describing the implementation of specific algorithms for sorting, shifting, and searching, we will present some general algorithmic steps useful in implementing all of these.

3.1 Identifying Bit Pairs

A common task in our algorithms is "identifying" pairs of adjacent bits, i.e., recognizing the specific pair of cells at a location of interest. We will exploit the fact that domain 1 is always exposed to identify these specific pairs. Figure 3 illustrates our approach on the string 11001, which contains all 4 possible adjacent pairs: $00, 01, 10$ and 11.

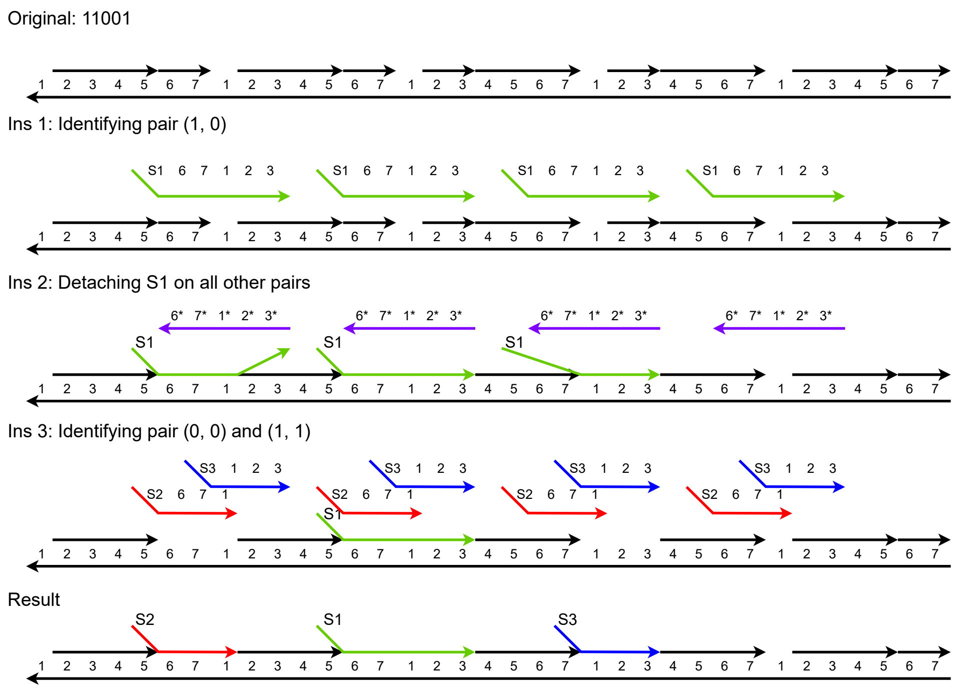

Original: 11001
Ins 1: Identifying pair (1, 0)
S1 6 7 1 2 3
Ins 2: Detaching S1 on all other pairs
6* 7* 1* 2* 3*
S1
Ins 3: Identifying pair (0, 0) and (1, 1)
S3 1 2 3
S2 6 7 1
Result
S2
S1
S3

Figure 3 Example of Identifying Different Pairs of Adjacent Bits.

Identification is performed with three instructions. In instruction 1, the strands (S_1 6 7 1 2 3) are issued to all pairs of bits. Through the toehold at domain 1 between each pair, the strand S_1 binds to domains 6, 7, 1 in the pair $(1, 1)$, leaving domains S_1, 2, 3 open. In the pair $(0,0)$, the strand S_1 binds to domains 1, 2, 3, leaving domains S_1, 6, 7 open. The strand S_1 binds to domains 6, 7, 1, 2, 3, in the pair $(1,0)$. The strand S_1 does not bind to the pair $(0,1)$ since the only exposed toehold is domain 1. We can then distinguish the pair $(1,0)$ from the open domains on strand S_1.

In instruction 2, using the complementary strands (6* 7* 1* 2* 3*), the strand S_1 that attaches to the pairs $(0,0)$ and $(1,1)$ is pulled out. This is done through the open domains 2, 3 in the pair $(0,0)$ and the open domains 6, 7 in the pair $(1,1)$ on strand S_1. After this instruction, strand S_1 remains only in the pair $(1,0)$.

In instruction 3, two instruction strands are issued at the same time: (S_2 6 7 1) and (S_3 1 2 3). Here (S_2 6 7 1) will bind to the pair $(1,1)$ and (S_3 1 2 3) will bind to the pair $(0,0)$. They will not bind with any other pairs since the only exposed toehold for binding would be domain 1; they will prefer the locations with more exposed domains.

The result is that the adjacent bit pairs $(1,1)$, $(1,0)$ and $(0,0)$ are each *labeled* with strands S_2, S_1 and S_3 respectively. Pairs $(0,1)$ are labelled with an exposed toehold at domain 1. This toehold could be replaced by a strand (S_x 4 5 6 7 1) or a strand (S_x 1 2 3 4 5); the choice would be made depending on the use case.

3.2 Rewriting a cell

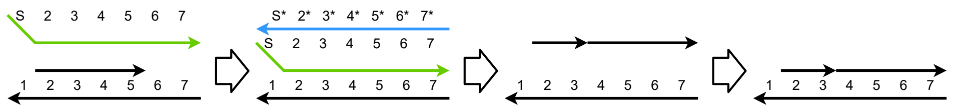


Figure 4 Example of Rewriting in Three Steps

By exposing toeholds across domains 2 through 7 in a cell, we can rewrite the content of that cell – so change a 1 to 0 or a 0 to 1 – with three instructions. The idea is that, since there are exposed domains, we can displace the content of the cell with a single strand covering all these domains. Then we can remove the covering strand through the exposed "tag" domain (S in Figure 4) using a complementary strand. The cell is now completely exposed. We can write a new bit to it by hybridizing the strands according to our encoding scheme, leaving domain 1 as a toehold and placing the nick at the desired location.

4 Parallel Binary Bubble Sorting

Sorting is a simple yet fundamental operation in computer science. Here we consider sorting binary values.[2] Sorting can be used to determine the "weight" of a vector of 0's and 1's: the count of the number of 1's relative to the length of the vector. It can also be used to compute the majority function: whether there are more 1's than 0's or not in the input set. Majority is a fundamental operation for many machine-learning algorithms.

Our SIMD DNA implementation performs parallel bubble sorting on binary bits [5]. It can be expressed as a pairwise operation in the form of $f(a,b) = (c,d)$, where (a,b) is the value of the input bit pair, and (c,d), the outputs, represent the action we take, whether to rewrite or to leave it as it is. The outputs can be 0 or 1, which means that we can arbitrarily

[2] Perhaps counter-intuitively, sorting binary values in hardware is as difficult algorithmically as sorting arbitrary values such as integers or real numbers [5]

change the value of the cell. They can also be X, meaning they remain unchanged. We discuss what kind of pairwise operations can be performed on our encoding in Section 7.1.

The sorting operation can be expressed in the following pairwise operation,

$$f(0,0) = (X,0) \qquad f(0,1) = (X,X) \qquad f(1,0) = (0,1) \qquad f(1,1) = (1,X).$$

Algorithmically, the following "bit swapping" is performed:

- If the current bit is 1, it changes it to 0 if and only if its right neighbor is 0.
- If the current bit is 0, it changes it to 1 if and only if its left neighbor is 1.

We argue that repeatedly performing such bit swapping will sort the entire sequence of binary values.

▷ Claim 1. Bit swapping will never happen more than once for any consecutive sequence of three bits. Such a sequence consists of two consecutive pairs, sharing the middle bit.

Proof. The only pair of consecutive bits that ever gets rewritten is the pair $(1,0)$ to $(0,1)$. It is impossible to have two consecutive, overlapping pairs $(1,0)$ sharing a common middle bit. ◁

Accordingly, bubble sorting binary values in parallel does not require an odd and even index addressing scheme, as does bubble sorting arbitrary values.

▷ Claim 2. Sorting completes in at most $(N-1)$ parallel steps where N is the total number of bits.

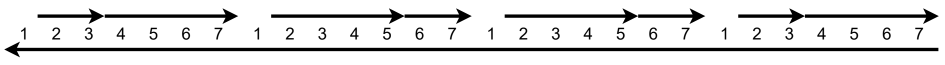
Proof. Suppose we have a sequence of binary bits of length N, in which all bits except the first are 0. When applying the algorithm, the 1 located at the start will be pushed back one position at a time with the $f(1,0) = (0,1)$ bit swap operation. Fully sorting the sequence, i.e., moving the 1 to the last position, requires $N-1$ total swaps. Now suppose we are sorting an arbitrary bit sequence. We argue that, after $N-1$ swaps, all the 1's will be at the end of the sequence. To see why, note that an $f(1,0) = (0,1)$ operations moves a 1 forward, while an $f(1,1) = (1,1)$ operation does not affect adjacent 1's. Thus, in $N-1$ steps, all 1's will have moved to end of the sequence. ◁

4.1 Implementation

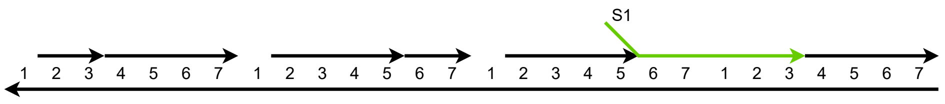
Here we give an instruction set for performing parallel binary bubble sort with SIMD DNA, using the encoding in Figure 2. It consists of 12 individual instructions. These are summarized as follows.

1. Label pairs $(1,0)$.
2. Uncover these, leaving domains 6 and 7 for the bits 1 and domains 2 and 3 for the bits 0 open in these pairs.
3. Protect the bits 0 of these pairs by covering the corresponding toehold at domains 2 and 3.
4. Flip the bits 1 to 0 in these pairs.
5. Release the protective covers; flip the bits 0 to 1 in these pairs.

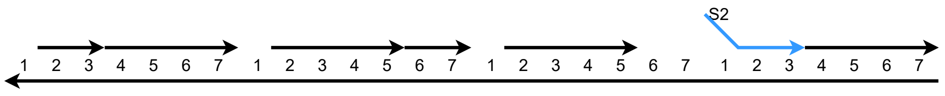
For the initialization, we can use the first two instructions described in Section 3.1, with an additional instruction to fix open domains for bits that do not change. We can use the rewriting method described in Section 3.2 to flip the bits. A full description of the implementation of sorting is provided in Appendix B.



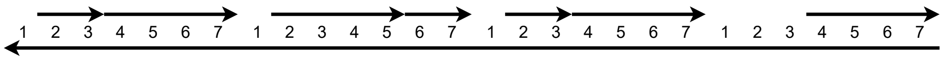
(a) Initial Sequence 0110



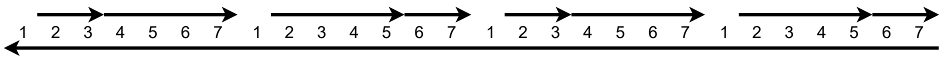
(b) After Recognizing $(1, 0)$



(c) Protection on Bit 0



(d) Flipped third bit to 0, Protection Removed



(e) Flipped fourth bit to 1, Result 0101

Figure 5 Outline of the SIMD DNA parallel binary sorting algorithm.

5 Parallel Left Shifting

We propose a SIMD DNA implementation of shifting, another fundamental operation in computer science. Shifting left corresponds to multiplying a binary number by 2; shifting right corresponds to dividing it by 2. It is a useful operation in general for aligning data in a variety of algorithms [5]. We present a left shift algorithm, one that shifts all N binary bits one position to the left, with the Least Significant Bit (LSB) remaining unchanged. This operation is, of course, a parallel left shift, moving all bits simultaneously in lockstep. Our implementation requires 11 instructions per shift. Note that unlike usual arithmetic or logical left shift that inserts a bit 0 to the LSB, the left shift operation described here keeps the original LSB, thereby duplicating the LSB. The usual left shift could be implemented by adding instructions rewriting the LSB to 0 after the instructions we provide here.

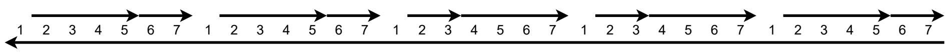
We describe the shift operation using the following pairwise operation as:

$$f(0, 0) = (0, X) \qquad f(0, 1) = (1, X) \qquad f(1, 0) = (0, X) \qquad f(1, 1) = (1, X)$$

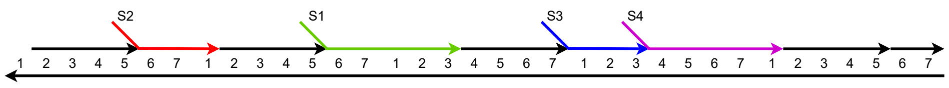
Here X means a value that does not change. For each bit pair, the operation writes the value of the right bit to the left bit. Since only the value of the left bit is changed in each bit pair, the operation is non-overlapping and can be implemented using the encoding scheme we propose. We illustrate with the example of shifting 11001 to 10011, shown in Figure 6.

1. Label all the bit pairs. Cover the toeholds for the pairs $(0, 0)$ and $(1, 1)$.
2. For the pairs $(1, 0)$, flip the bits 1 to 0.
3. For the pairs $(0, 1)$, flip the bits 0 to 1.
4. Finally, uncover all the toeholds for the pairs $(0, 0)$ and $(1, 1)$.

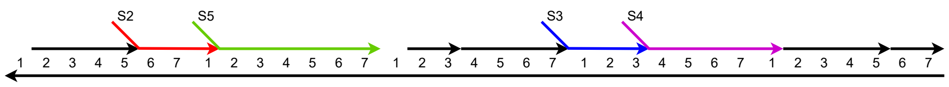
A full description of the implementation of shifting is given in Appendix C.



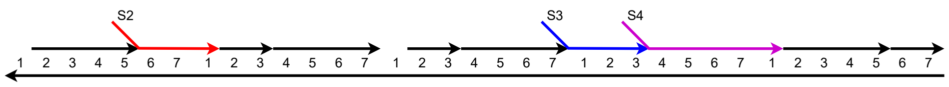
(a) Initial Sequence 11001



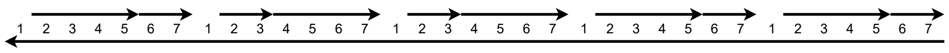
(b) After identifying all pairs



(c) Release S_1 from Pair $(1, 0)$



(d) Rewrite bit 1 in the previous pair with 0



(e) Release S_2, S_3 and S_4 then write 1, Result 10011

Figure 6 Outline of the SIMD DNA parallel left shift operations. The initial sequence S is 11001 and the result sequence T is 10011. The operation shift each bit to left one position (T[5:1]=S[4:0]), while keeping the Least Significant Bit unchanged.

6 Parallel Search Algorithm

Searching is fundamental to all branches of computer science that involve data storage and retrieval. We consider the problem of deciding whether a given substring exists in a stored string of bits. We first discuss a general algorithm that returns an answer to such a question in $\log(n)$ parallel steps, where n is the substring length. We then propose an implementation in SIMD DNA. Due to practical constraints, the time complexity of the implementation is not $O(\log(n))$; it is closer to $O(n)$, depending on the problem size and implementation details. We note that a requirement of our algorithm is that the length of the query string is a power of 2. We discuss the time complexity and constraints in detail in Section 7.3.

6.1 Algorithm

Suppose we have a *query* substring Q of a length n and we would like to search whether it appears in a much longer *target* string A. Pseudo-code for our approach is given as Listing 1. We will elucidate the pseudo-code by stepping through examples.

6.1.1 Parallel search procedure

We illustrate searching for a query string $Q = 1101$ in the following target string A:

$$\begin{aligned} A_0 &= 10101010110110100011110101000100 \\ A_1 &= a_2a_2a_2a_2a_3a_1a_2a_2a_0a_3a_3a_1a_1a_0a_1a_0 \\ A_2 &= b_0b_0b_1b_0b_2b_1b_3b_3 \end{aligned} \tag{1}$$

Listing 1 Pseudo-code for Parallel Search Algorithm. Note that the operations inside the two **foreach** loops can be performed in parallel since they are independent. The `pair` operation here is to find a corresponding symbol that replaces the two symbols in the lookup table, and the `identity` operation is to look up the symbol that represents the query string.

```
S = Query String
T = Target String
n = length of S
for i in range(0,n-1):
    T_i = T
    truncate first i characters of T_i
    p = 1
    while p <= n:
        j = 0
        while j < (length(T_i)-1):
            a = T_i[j]
            b = T_i[j+1]
            c = pair(a,b) # Pair 2 consecutive cells
            if c.identity(S): # Check if new pair is the query
                return True
            replace a,b in T_i with c
            j += 1
        p = 2*p
return False
```

The original string is A_0. In each step, two consecutive symbols are read and replaced with a single symbol. Here $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_0a_3, b_3 = a_1a_0$. Note that $Q = 1101 = a_3a_1 = b_1$. After three steps, we conclude that the query string exists in the target string, since there are two matches in the string A_2.

6.1.2 Search procedure with offset

It is possible that the query string does not align with divisions of length n in the target string. Thus we need to repeat the operation with offsets. The following example illustrates the operation with an offset of 2 bits.

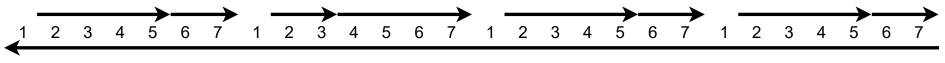
$$\begin{aligned}
A_0 &= \cancel{10}10101101011000000111100010001 00\\
A_1 &= \cancel{10}a_2a_2a_3a_1a_1a_2a_0a_0a_3a_3a_0a_1a_0a_1a_0\\
A_2 &= \cancel{10}b_0b_1b_2b_3b_4b_5b_5\cancel{a_0}
\end{aligned} \tag{2}$$

Here, the replacement is given by the aggregated pairs $a_0 = 00, a_1 = 01, a_2 = 10, a_3 = 11, b_0 = a_2a_2, b_1 = a_3a_1, b_2 = a_1a_2, b_3 = a_0a_0, b_4 = a_3a_3, b_5 = a_0a_1$. Again, an instance of the query string is found in the target string.

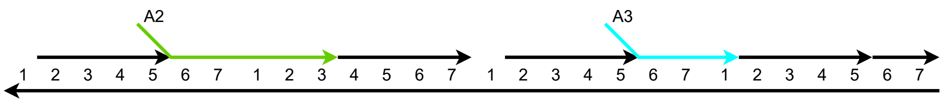
Searching for a query string with a given offset requires at most $\log(n)$ steps. In general, for an arbitrary query string of a length n (a power of 2), the search must be performed n times with offsets ranging from 0 to $n - 1$. In principle, all of these searches could be performed in parallel, as none would interfere with any other. Accordingly, our parallel implementation of searching completes in $\log(n)$ steps.

Note that the number of aggregated pair identifiers needed – the a's and b's in the example above – grows exponentially with the length of the target string. However, these can be synthesized once and reused for every query. If we consider the restricted problem of searching for a *specific* query string, meaning that we only use pair identifiers for matching pairs, then the number of identifiers needed is $\sum_{i=1}^{\log(n)} 2^i = n - 1$.

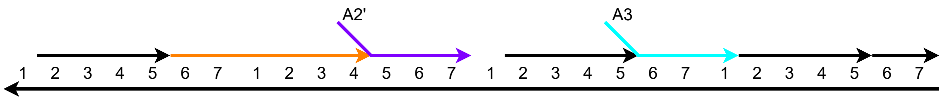
6.2 Implementation



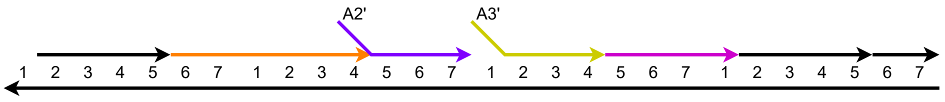
(a) Initial Sequence 1011



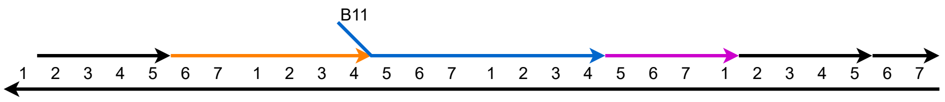
(b) Identifier A_2 captures first pair 10, A_3 captures second pair 11



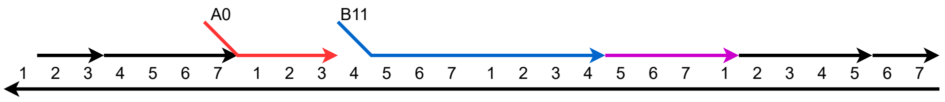
(c) covering the domain 1 between the two bit pairs



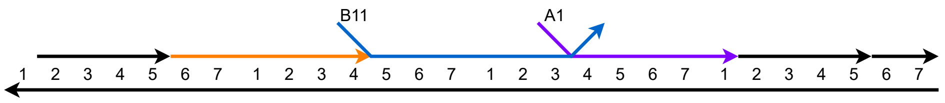
(d) Rewrite the content in the pair so that new identifiers are close to the middle



(e) Two identifier strands replaced by a single identifier if there is a perfect match



(f) Initial sequence is 0011. It will result in an open domain 4 in the cell left to the identifier



(g) Initial sequence is 1010. It will result in an open domain 4 on the identifier itself

Figure 7 Example implementation of search algorithm on target sequence 1011

To implement the algorithm in SIMD DNA, we do not issue instruction strands to each pair of overlapping bits. Instead, we consider the non-overlapping bit pairs. In the example shown in Figure 7, for the bit sequence 1011, we would consider operations on bit pair 10 and 11, but not on bit pair 01.

Figure 7 shows the critical steps on searching a target sequence 1011. It provides an example of a successful search and also the potential outcome of two failed searches. To implement the search operation with an offset, we can simply skip the number of bits according to the offset. We use the word *symbol* to represent the consecutive cells that we search for on a certain level. For example, in the first level, the symbols are 10 and 11. We can use the bit identifying steps described in Section 3.1 to recognize these symbols. We use identifiers $A_0 = 00, A_1 = 01, A_2 = 10, A_3 = 11$ to represent symbols in this level. We then move on to the next level, searching for consecutive symbols A_2A_3, which corresponds to the target string 1011.

In the first step of the second level, we first rewrite the topological structure at symbols that appear to be a query result. In this example, A_2 should be found as the left symbol, and A_3 should be found as the second symbol. We pull identifier A_2 out from every *odd* symbol (we only look at the first, third, fifth, etc.) and rewrite the entire symbol with the technique described in Section 3.2. After rewriting, we have the identifier A_2' that covers domains (5 6 7) in the *right most* cell, as seen in Figure 7c. For the second symbol A_3, we repeat the step described, except we pull the identifier out from every *even* symbol and the new identifier A_3' covers domains (2 3 4) in the *left most* cell. Through these steps, we have essentially "moved" the identifier of the matching symbols to the middle. In the final step, we issue the new identifier strand (B_{11} 5 6 7 1 2 3 4) to the location between every two symbols. It will result in a perfect binding only if there is a match at the current symbol level. Figure 7e shows the example of a matching result. Figure 7f and 7g show two potential examples of imperfect binding, indicating a non-matching result. We can pull them out through the open domains either on the identifier itself or a nearby open domain on the base strand. Therefore, the presence of the identifier B_{11} indicates a successful match.

We can repeat the process to recognize multiple symbols at the same level. When we move to the next level $l+1$, we can use the identifiers from this level l as a starting point for rewriting. To identify a symbol $S_{l+1,c} = S_{l,a}S_{l,b}$ at level $l+1$, we simply pull out identifiers for $S_{l,a}$ at odd symbols and $S_{l,b}$ at even symbols at level l. Then we "move" the identifier to the middle. Finally, we give identifiers for $S_{l+1,c}$ to the middle of each pair and identify the symbol.

A possible weakness of our implementation is that the strand used for rewriting could potentially be very long. This could cause problems when performing these operations *in vitro* due to branch migration complications. Lastly, this search operation can handle multiple overlapping queries within the reference string, but this requires careful consideration of the base-pair sequence of the cells in designing identifier strands.

7 Discussion

We discuss the features and implementation constraints of the proposed algorithms.

7.1 Ability to compute any non-conflicting pairwise operation

In Section 4 and Section 5, we presented examples of algorithms that perform pairwise operations, namely sorting and shifting, respectively. Given the ability to identify pairs of bits and a universal way to rewrite a cell, we can readily implement any algorithm that performs non-conflicting pairwise operations. Such operations only entail rewriting pairs of adjacent bits. The result of the operation on a specific sequence should always be the same, irrespective of the execution order. To illustrate, consider the following operation:

$$f(0,0) = (X,X) \qquad f(0,1) = (X,1) \qquad f(1,0) = (X,X) \qquad f(1,1) = (0,X)$$

Here X indicates a value that does not change. This operation *is* conflicting. To see why, consider its effect on the sequence 011. The second bit should change to 1 when the operation is applied to the first pair. However, this bit should change to 0 when the operation is applied to the second pair. Depending on the order of execution, the final result will be different. To ensure an operation is non-conflicting, for every three adjacent bits that two operations are performed on, the middle bit should be set to the same value.

Non-conflicting operations can be performed in parallel on all bit pairs. In the first step, we identify the four bit pairs described in 3.1. After this step, we supply strands with four labels covering the four bit pairs. Then, we release strands with specific labels one at a time to obtain write access to specific bit pairs. (Write access refers to a domain being exposed.) We rewrite these cells with the operation described in Section 3.2. The full operation requires rewriting all four bit pairs.

We conclude that our encoding scheme and design method are generally applicable to parallel bitwise algorithms, provided that they can be expressed in terms of such non-conflicted pairwise operations.

7.2 Converting to Different Encoding Schemes

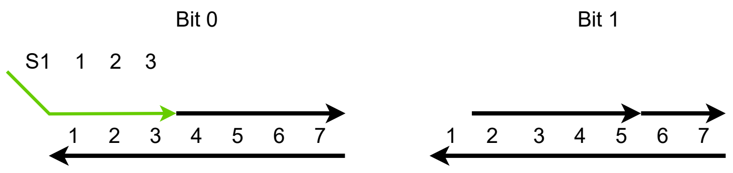


Figure 8 One strand could be used to differentiate two bits

A benefit of the encoding scheme that we are proposing is that it can easily be converted to any other similar scheme since each cell always has an exposed domain 1. In the original SIMD DNA scheme proposed in [13], the authors designed two specific encoding schemes for the two applications proposed (rule 110 and a binary counter). We suggest that our encoding scheme could be used as an intermediate form when converting to other encoding schemes, designed for particular algorithms. Figure 8 illustrates how we can use a single strand (S_1 1 2 3) to differentiate bit values of 0 from bit values of 1. We can use the technique discussed in 3.2 to re-write the data with a different encoding scheme, so long as the scheme also encodes each bit with 7 domains. Complete instructions for performing such encoding changes are given in Appendix A.

7.3 Time Complexity of Parallel Search

While the time complexity of the proposed parallel search is $O(\log(n))$ in principle, where n is the query substring length, the time complexity of our SIMD DNA implementation is somewhat worse. While the abstract search algorithm finds the query in the reference string by pairing individual characters in parallel, and thus completes in $O(\log(n))$ steps, our implementation searches for and identifies distinct symbols sequentially, that is to say, it first searches for a specific symbol across all possible locations at once, then it searches for the next symbol across all locations at once, and so on.

The abstract algorithm assumes all symbols are identified in one pass to allow for further pairing. If we consider all the different symbols in a query string, counting repeated symbols, $\frac{n}{2^i}$ symbols must be searched sequentially at level i in our implementation. Accordingly, the total number of sequential search steps could be as high as $O(n)$. However, at each level, all the occurrences of a specific symbol are identified simultaneously. At level i, each symbol represents a binary string with a length of 2^i, so there are at most 2^{2^i} distinct symbols at level i. For example, in the first level, instead of searching for $\frac{n}{2}$ symbols, we only search for four distinct symbols. In the second level, there are only 16 distinct symbols. Since we only search for distinct symbols, the number of steps in the first few levels will be greatly reduced.

Our parallel search algorithm currently only works on query strings having a length that is a power of two. However, we believe that our implementation could be modified to allow for arbitrary-length query strings. We do not provide details here, as they are cumbersome, but we outline the method as follows.

Note that, in parallel search, the query string is searched reductively: at each level, two symbols are reduced to one symbol. When working with query strings having any arbitrary length, there might be an odd number of symbols in the current level, meaning that the last symbol cannot be reduced for the next level. In this case, we can add a method to identify the trailing odd symbol at the current level and replace it in the next level. The reduction can still be completed in a logarithmic number of levels.

8 Conclusion

We have presented algorithms for basic parallel operations within the SIMD DNA framework. We note that there are, in fact, two layers of parallelism possible:

1. Bit-level Parallelism: instructions applied to all bits in an array at once.
2. Data-level Parallelism: the same instructions applied to *multiple* arrays at once.

While operations on DNA are slow and error-prone, with these levels of parallelism, perhaps DNA computation could scale to a truly impressive regime. Consider the following back-of-an-envelop estimates. Suppose:

- we have 10^{12} independent cells in parallel in a single test tube;
- a single operation takes approximately 10 minutes to complete.
- different cells use the same DNA sequence. Using distinct sequences for different cells, as in our search operation, can result in a solution with multiple competing DNA molecules. At larger scales, this would result in an increase in reagent volume and could diminish reaction rates.

This means that we can perform approximately 10^9 operations per second in a single test tube, already impressive. Now suppose that:

- we have 100 test tubes.

This means we can compute at 100,000 MIPS (million instructions per second). This is comparable to what very respectable existing silicon systems can achieve. The key conceptual difference between the SIMD DNA approach and other forms of DNA computing is that it exploits a substrate on which data is stored. This enables the SIMD parallelism.

Many experimental hurdles remain in demonstrating and deploying this paradigm. DNA synthesis remains prohibitively expensive. A possible alternative is to use gene-editing techniques to encode data on naturally occurring DNA [11].

References

1 Leonard M Adleman. Molecular computation of solutions to combinatorial problems. *Science*, pages 1021–1024, 1994.

2 Nagendra Athreya, Olgica Milenkovic, and Jean-Pierre Leburton. Detection and mapping of dsDNA breaks using graphene nanopore transistor. *Biophysical Journal*, 116(3):292a, 2019.

3 Luis Ceze, Jeff Nivala, and Karin Strauss. Molecular digital data storage using DNA. *Nature Reviews Genetics*, 20(8):456–466, Aug 2019. doi:10.1038/s41576-019-0125-3.

4 George Church, Yuan Gao, and Sriram Kosuri. Next-generation digital information storage in DNA. *Science (New York, N.Y.)*, 337:1628, 08 2012. doi:10.1126/science.1226355.

5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

6 M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

7 Joachim Krug and Herbert Spohn. Universality classes for deterministic surface growth. *Physical Review A*, 38(8):4271, 1988.

8 Wentian Li. Power spectra of regular languages and cellular automata. *Complex Systems*, 1(1):107–130, 1987.

9 Ke Liu, Chao Pan, Alexandre Kuhn, Adrian Pascal Nievergelt, Georg E Fantner, Olgica Milenkovic, and Aleksandra Radenovic. Detecting topological variations of DNA at single-molecule level. *Nature communications*, 10(1):1–9, 2019.

10 David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010. URL: https://www.pnas.org/content/107/12/5393, arXiv:https://www.pnas.org/content/107/12/5393.full.pdf, doi:10.1073/pnas.0909380107.

11 S. Tabatabaei, Boya Wang, Nagendra Athreya, Behnam Enghiad, Alvaro Hernandez, Christopher Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature Communications*, 11, 12 2020. doi:10.1038/s41467-020-15588-z.

12 S Kasra Tabatabaei, Boya Wang, Nagendra Bala Murali Athreya, Behnam Enghiad, Alvaro Gonzalo Hernandez, Christopher J Fields, Jean-Pierre Leburton, David Soloveichik, Huimin Zhao, and Olgica Milenkovic. DNA punch cards for storing data on native DNA sequences via enzymatic nicking. *Nature communications*, 11(1):1–10, 2020.

13 Boya Wang, Cameron Chalk, and David Soloveichik. SIMD||DNA: Single instruction, multiple data computation with DNA strand displacement cascades. In Chris Thachuk and Yan Liu, editors, *DNA Computing and Molecular Programming*, pages 219–235, Cham, 2019. Springer International Publishing.

14 Bernard Yurke. A DNA-fuelled molecular machine made of DNA. *Nature*, 406(6796: 605), 2000.

A Instructions for Converting to Another Scheme

Instruction 1 identifies and distinguishes the two different bits. In instruction 1, strand (S_1 1 2 3) is issued. In bit 0, the strand will displace the short strand over domains 2 and 3 but does not edit bit 1 since domain 1 is the only open domain for binding. In instruction 2, all domains in bit 1 are replaced by a single strand covering all domains with identifier S_a. Then in instruction 3, the strand S_1 is detached, so domains 1, 2, and 3 on bit 0 are exposed. In Instruction 4, all domains in bit 0 are replaced by a single strand covering all the domains with the identifier S_b. Then any encoding scheme with 7 domains in 1 cell could be written to the bits by first detaching strand S_a and writing the encoding for bit 1, then detaching strand S_b and writing the encoding for bit 0.

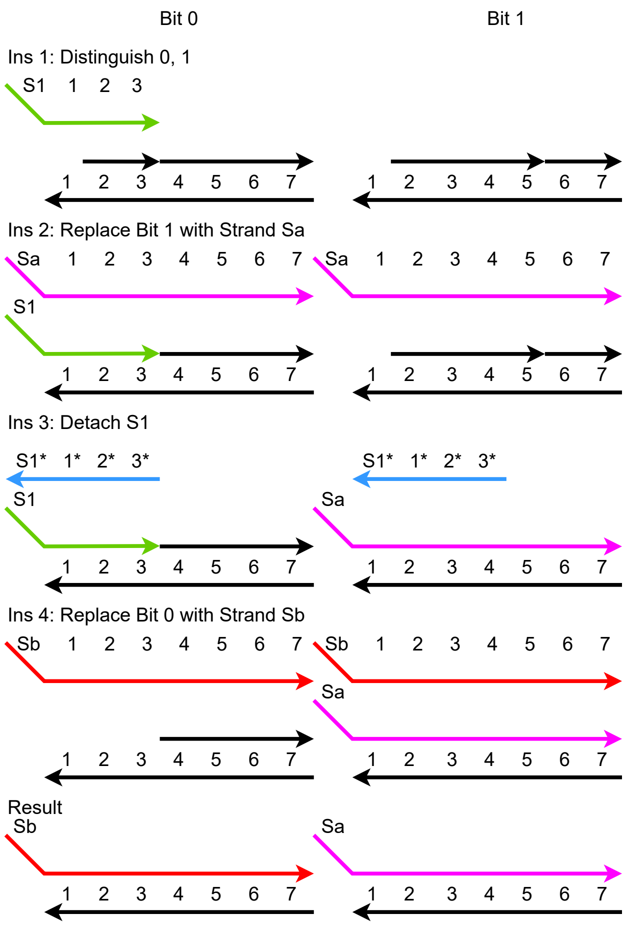


Figure 9 Current coding scheme could be converted to other coding scheme

B Detailed Implementation of Each Step for Parallel Sorting

Here we give an instruction set for parallel binary bubble sort with the previously defined encoding scheme. We can implement each step of the sorting algorithm in 12 individual operations. Details of the design are shown in Figure 10.

The 12 instruction falls to 2 stages. The first stage is "identifying." During instructions 1-4, all the pairs (0, 1) are identified, and in both bit 0 and 1, a toehold is exposed for writing new data. More specifically, Instructions 1 and 2 identify the combination of $(1, 0)$. In instruction 1, (S_1 6 7 1 2 3) is issued to each pair of bits. In pair $(0, 0)$, S_1 and domains 6, 7 are exposed. In pair $(0, 1)$, since the only open domain is 1, it will not form a strong enough bind. In pair $(1, 0)$, only S_1 is exposed. In pair $(1, 1)$, S_1 and domains 2, 3 are exposed. In instruction 2, strand (6* 7* 1* 2* 3*) is issued to each pair of bits. Since pair $(1, 0)$ is the only pair that does not have exposure 5 or 2, this strand will detach strand S_1 in each pair except pair $(1, 0)$. After Instruction 2, the toehold between a bit value of 1 and a bit value of 0 in the pair $(1, 0)$ is replaced by a strand with an identifier of S_1. Instruction 3 seals off the domain exposed in the other pairs during Instruction 1 and 2 so that it will not be edited later. In instruction 4, the strand with identifier S_1 is detached, exposing domains 6 and 7 in the left cell containing bit 1, or domains 2 and 3, in the right cell containing bit 0. After this instruction, toeholds are exposed only in the 1s and 0s in pair $(1, 0)$. Other bits are not affected.

The second stage is flipping the bits in the pair $(1, 0)$. In instruction 5, in the case of a bit value of 0, domains 2 and 3 are temporarily covered by a strand with identifier S_2 so that the writing process will not interfere with the identified 0s at this moment. In instruction 6, a bit value of 1 is replaced by a strand with identifier S_3 via the open toehold at domains 6 and 7. The strand is then detached in instruction 8, exposing all the domains of that bit. Then, the bit value of 0 is written to the location of a bit value of 1 in instruction 8. In instruction 9, the temporary cover for a bit 0 is lifted. Then, in instructions 10 through 12, a bit 1 is written to the location of a bit value of 0 using the same scheme as instructions 6 through 8. Throughout the process, only bits identified in the first stage with toeholds exposed are affected.

C Detailed Implementation of Each Step for Parallel Left Shift cell

The instructions are shown as followed, with an example of shifting 11001 to 10011.

The first three instructions are exactly the same as those for identifying bit pairs in Section 3.1. In instruction 1, the strand (S_1 6 7 1 2 3), which identifies the different patterns of two bits, is issued to each pair of bits. In instruction 2, strand (6* 7* 1* 2* 3*) is issued, detaching strands with open domains 6 and 7, or 2 and 3. After this instruction, strands with identifier S_1 only remain at pair (1, 0). In instruction 3, we issue two species of strands at the same time: (S_2 6 7 1) and (S_3 1 2 3). (S_2 6 7 1) will bind with pair $(1, 1)$ and (S_3 1 2 3) will bind with pair $(0, 0)$. S_2 will not form a stable binding with pair $(0, 0)$ or $(0, 1)$ because the binding area is only one domain. Same goes with S_3 and pair $(1, 1)$ or $(0, 1)$. After this instruction, only domain 1 between pair $(0, 1)$ is still exposed. In instruction 4, strand (S_4 4 5 6 7 1) is issued. Through the open domain 1 between pair $(0, 1)$, the strand in bit 0 is replaced by S_4. After this step, the first bit in pair $(1, 0)$ is identified with the strand S_1, and the first bit in pair (0, 1) is replaced with the strand S_4.

Instructions 5 to 9 rewrite the first bit in pair $(1, 0)$ to 0. In instruction 5, the strand S_1 is detached, exposing domains 6, 7, 1, 2 and 3. The exposed domains 2 and 3 are sealed off in instruction 6 to not interfere with subsequent instructions. In instruction 7, strand (S_5 2 3 4 5 6 7) is issued through the open toehold on domains 6 and 7 in the bit 1 in pair $(1, 0)$, and displaces the strand in that bit. Since domains 2 and 3 are sealed off, bit 0 will not be modified in this instruction. In instruction 8, strand S_5 is detached, leaving the domains in the bit open. In instruction 9, strands (2 3) and (4 5 6 7), which represent 0, are written to the bit containing open domains.

In the final two instructions, we write 1 to the first bit in pair $(0, 1)$. In instruction 10, 3 strands are issued to each pair of bits: (S_2* 6* 7* 1*), (S_3* 1* 2* 3*) and (S_4* 4* 5* 6* 7* 1*). S_2, S_3 and S_4 are detached through these strands. Since S_4 covers the bit 0 in pair (0, 1), after this step, domain 3 and 4 are exposed in these bits, ready to be written to 1. In the final step, strands (2 3), (2 3 4 5), and (6 7) are issued to each cell. Strand (2 3) and (6 7) will fix the exposed domains from strand S_2 or S_3, and strand (2 3 4 5) will write bit 1 to the bit with domain 3 and 4 exposed. Details of the design are shown in Figure 11.

For all the pairs of $(0, 0)$ and $(1, 1)$, the first bit in those pairs will not be modified since the toehold 1 will be covered with S_2 or S_3 in the process.

D Detailed Implementation of the Second Level in Parallel Search

Here we discuss the *second* level of the parallel search operation. The first level of search operation uses the instructions that were described in Section 3.1, except we now only issue strands to non-overlapping bit pairs. We use identifiers $A_0 = 00, A_1 = 01, A_2 = 10, A_3 = 11$ to represent symbols in this level. For instance, to search for the target string 1011, we search for the symbol A_2 in odd symbols and A_3 in even symbols. The cases of A_2 in even symbols and A_3 in odd symbols are covered by searching with offset.

In the first instruction of the second level, we uncover the A_2 in the odd symbols, creating an open region. In instruction 2, we use a long strand to cover the entire right half of the symbol, from the start of identifier A_2 to the rightmost cell. This strand is pulled out in instruction 3. In instruction 4, we use an identifier A_2' to cover domains 5, 6, 7 in the rightmost cell while covering all other domains.

Instructions 5 to 8 are essentially the same as instructions 1 to 4, but with two significant differences. Firstly, since A_3 is the second symbol in the current level of query, we only search for even-numbered symbols (2, 4, 6, etc.). Secondly, instead of rewriting the right half of the symbol, we write the left half. We make the new identifier A_3' to cover domains 2, 3, 4 in the left-most cell. In instruction 9, we use identifier ($B_1$1 5 6 7 1 2 3 4) to recognize the two consecutive symbols A_2 and A_3. Since, in the regular encoding, no strand starts from domain 5 or ends at domain 4, it will only form a perfect binding with a matched result.

After the identifier B$_1$ 1 binds, we also need to clean up the imperfect bindings in case of a mismatch. Figure 12 shows the instructions for the cleanup process. In instruction 10, we first use the complementary strand (5* 6* 7* 1* 2* 3* 4*) to pull out the imperfect bond identifier $B_1$1. Then we issue strands covering the exposed domain. We first issue strands covering fewer domains, then in following instructions, we issue strands covering more domains. As a result, we always obtain a perfect fit; the strands will not be pulled out in potential unrelated rewriting processes.

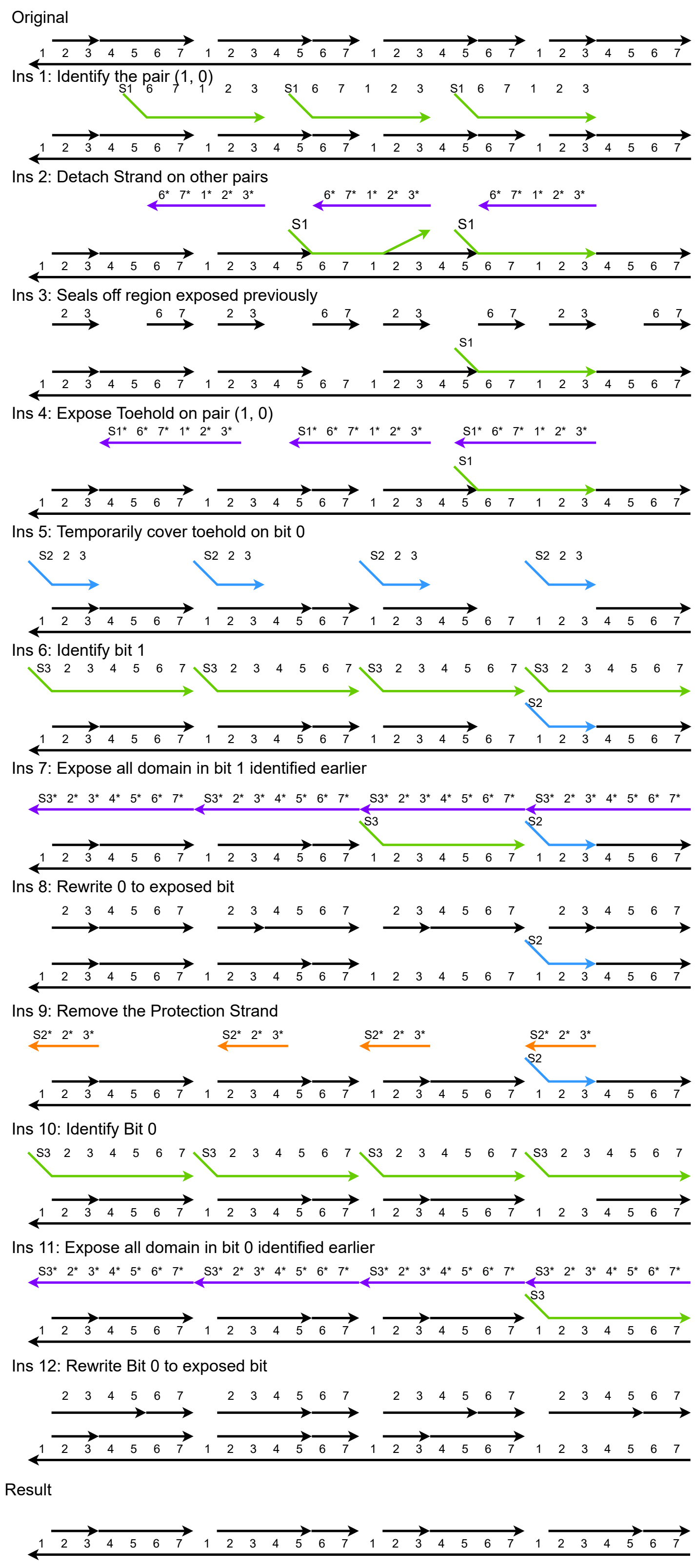

Original
Ins 1: Identify the pair (1, 0)
Ins 2: Detach Strand on other pairs
Ins 3: Seals off region exposed previously
Ins 4: Expose Toehold on pair (1, 0)
Ins 5: Temporarily cover toehold on bit 0
Ins 6: Identify bit 1
Ins 7: Expose all domain in bit 1 identified earlier
Ins 8: Rewrite 0 to exposed bit
Ins 9: Remove the Protection Strand
Ins 10: Identify Bit 0
Ins 11: Expose all domain in bit 0 identified earlier
Ins 12: Rewrite Bit 0 to exposed bit
Result
S1
S2
S3

Figure 10 Instructions for Parallel Sorting

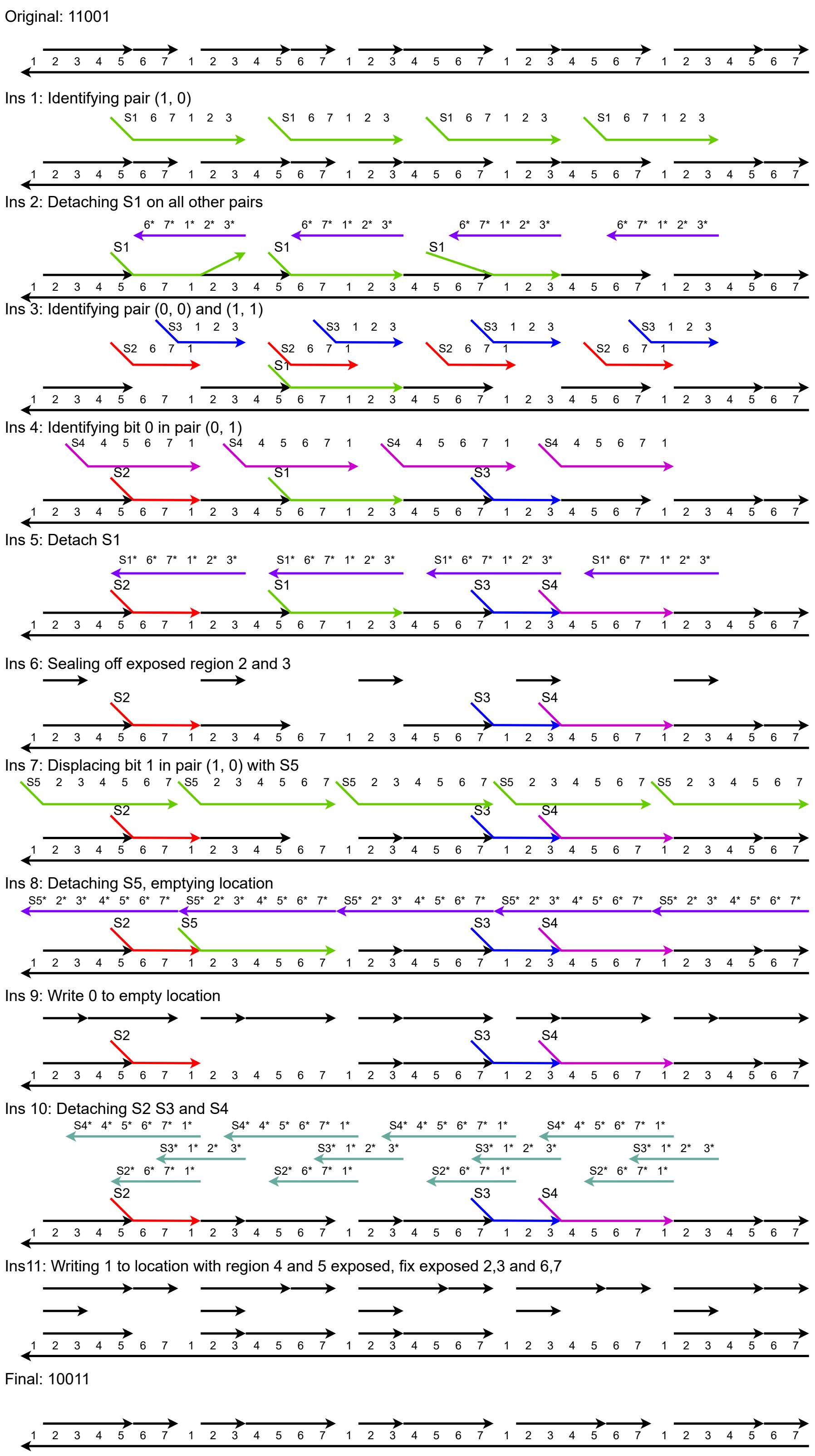

Original: 11001
Ins 1: Identifying pair (1, 0)
S1 6 7 1 2 3
Ins 2: Detaching S1 on all other pairs
6* 7* 1* 2* 3*
S1
Ins 3: Identifying pair (0, 0) and (1, 1)
S3 1 2 3
S2 6 7 1
Ins 4: Identifying bit 0 in pair (0, 1)
S4 4 5 6 7 1
Ins 5: Detach S1
S1* 6* 7* 1* 2* 3*
Ins 6: Sealing off exposed region 2 and 3
Ins 7: Displacing bit 1 in pair (1, 0) with S5
S5 2 3 4 5 6 7
Ins 8: Detaching S5, emptying location
S5* 2* 3* 4* 5* 6* 7*
Ins 9: Write 0 to empty location
Ins 10: Detaching S2 S3 and S4
S4* 4* 5* 6* 7* 1*
S3* 1* 2* 3*
S2* 6* 7* 1*
Ins11: Writing 1 to location with region 4 and 5 exposed, fix exposed 2,3 and 6,7
Final: 10011
1 2 3 4 5 6 7

Figure 11 Instructions for the Left Shift cell

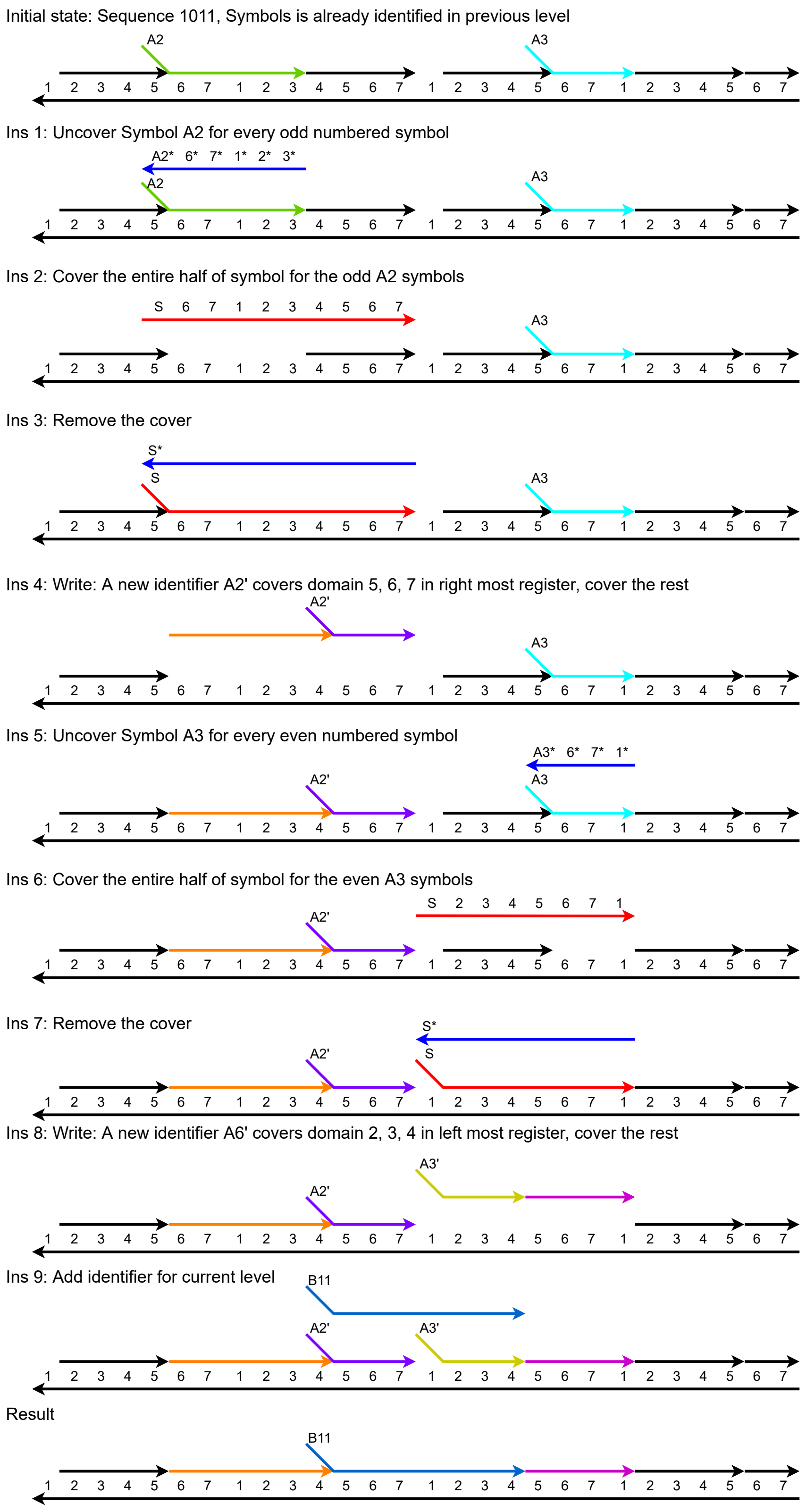

Initial state: Sequence 1011, Symbols is already identified in previous level
A2
A3
1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7 1 2 3 4 5 6 7
Ins 1: Uncover Symbol A2 for every odd numbered symbol
A2* 6* 7* 1* 2* 3*
Ins 2: Cover the entire half of symbol for the odd A2 symbols
S 6 7 1 2 3 4 5 6 7
Ins 3: Remove the cover
S*
S
Ins 4: Write: A new identifier A2' covers domain 5, 6, 7 in right most register, cover the rest
A2'
Ins 5: Uncover Symbol A3 for every even numbered symbol
A3* 6* 7* 1*
Ins 6: Cover the entire half of symbol for the even A3 symbols
S 2 3 4 5 6 7 1
Ins 7: Remove the cover
Ins 8: Write: A new identifier A6' covers domain 2, 3, 4 in left most register, cover the rest
A3'
Ins 9: Add identifier for current level
B11
Result

Figure 12 Instructions for a search operation of target sequence 1011

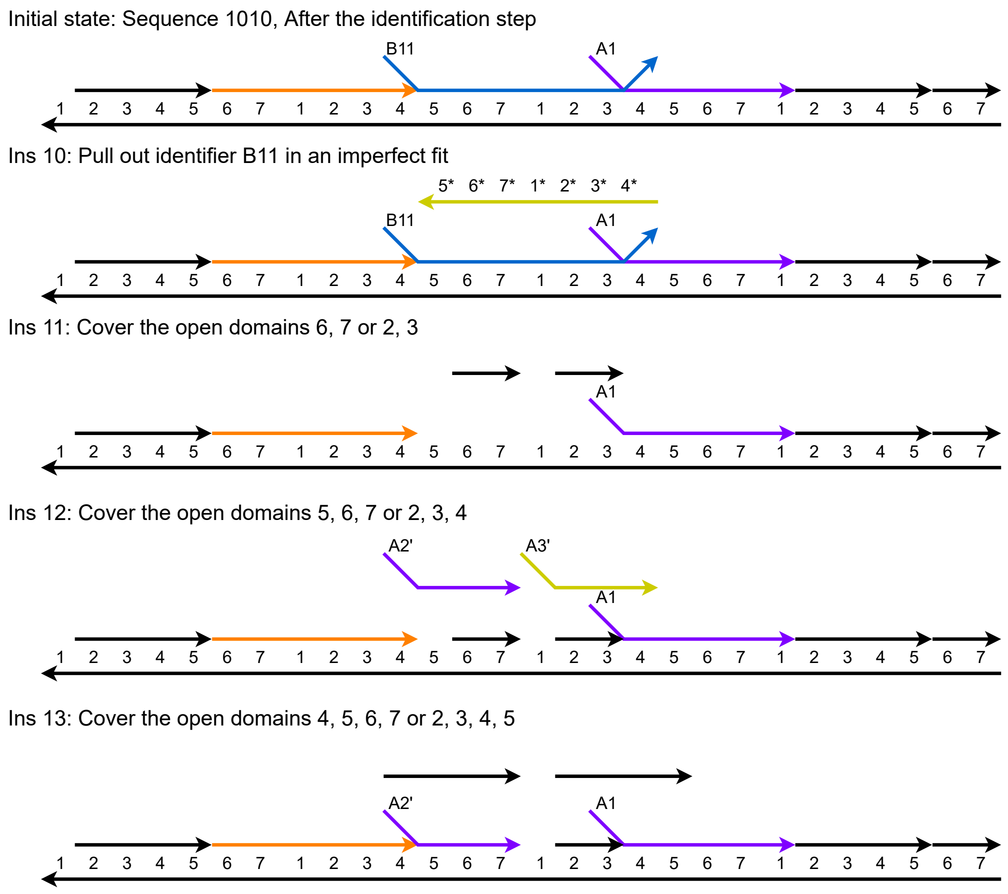


Figure 13 Instructions for the clean up process for a failed searching, these instructions won't affect the result of a successful search.