Algorithms in the Ultra-Wide Word Model

University of Waterloo Technical Report CS-2012-21

Arash Farzan[1], Alejandro López-Ortiz[2], Patrick K. Nicholson[2], and Alejandro Salinger[2]

[1] Max-Planck-Institut für Informatik
afarzan@mpi-inf.mpg.de
[2] David R. Cheriton School of Computer Science, University of Waterloo
{alopez-o,p3nichol,ajsalinger}@uwaterloo.ca

Abstract. The effective use of parallel computing resources to speed up algorithms in current multi-core and other parallel architectures remains a difficult challenge, with ease of programming playing a key role in the eventual success of these architectures. In this paper we consider an alternative view of parallelism in the form of an ultra-wide word processor. We introduce the Ultra-Wide Word architecture and model, an extension of the word-RAM model, that allows for constant time operations on thousands of bits in parallel. Word parallelism as exploited by the word-RAM model does not suffer from the more difficult aspects of parallel programming, namely synchronization and concurrency. In practice, the speedups obtained by word-RAM algorithms are moderate, as they are limited by the word size. We argue that a large class of word-RAM algorithms can be implemented in the Ultra-Wide Word model, obtaining speedups comparable to multi-threaded computations while keeping the simplicity of programming of the sequential RAM model. We show that this is the case by describing implementations of Ultra-Wide Word algorithms for dynamic programming and string searching. In addition, we show that the Ultra-Wide Word model can be used to implement a non-standard memory architecture, which enables the sidestepping of lower bounds of important data structure problems such as priority queues and dynamic prefix sums.

1 Introduction

In the last few years, multi-core architectures have become the dominant hardware platform. The potential of these architectures to improve performance through parallelism remains to be tamed, as effectively using all cores on a single application has proven to be a difficult challenge. We consider an alternate view of parallelism for a modern architecture in the form of an ultra-wide word processor. This can be implemented by replacing one or more cores with a very wide word Arithmetic Logic Unit (ALU) that can perform operations on a very large number of bits in parallel.

The idea of executing operations on a large number of bits simultaneously has been successfully exploited in different forms. In Very Long Instruction Word (VLIW) architectures [12] several instructions can be encoded in one wide word and executed in one single parallel instruction. Vector processors allow execution of one instruction on multiple elements simultaneously, implementing Single-Instruction-Multiple-Data (SIMD) parallelism. This form of parallelism led to the design of supercomputers such as the Cray architecture family [30], and is now present in Graphics Processing Units (GPUs) as well as in Streaming SIMD Extensions (SSE) extensions to scalar processors.

As CPU hardware advances so does the model used in theory to analyze it. The increase in word size was reflected in the word-RAM model in which algorithm performance is given as a function of the input size n and the word size w, with the common assumption that $w = \Theta(\log n)$. In its simplest version, the word-RAM model allows the same operations of the traditional RAM model. Algorithms in this model take advantage of bit-level parallelism through packing various elements in one word and operating on them simultaneously. Although similar to vector processing, the

word-RAM provides more flexibility in that the layout of data in a word depends on the algorithm, and data elements can be packed in an arbitrary way. Unlike VLIW architectures, the Ultra-Wide Word model we propose is not concerned with the compiler identifying operations which can be done in parallel but rather with achieving large speedups in implementations of word-RAM algorithms through operations on thousands of bits in parallel.

As multi-core chip designs evolve, chip vendors try to determine the best way to use the available area on the chip, and the options traditionally are more cores or more cache. We believe that the current stage in processor design allows for the inclusion of an architecture such as the one we present in this work. In addition, ease of programming is a major hurdle to the eventual success of parallel and multi-core architectures. In contrast, bit parallelism as exploited by the word-RAM model does not suffer from this drawback: there is a large selection of word-RAM algorithms (see, e.g. [1, 20, 18, 8]) that readily benefit from bit parallelism without having to deal with the more difficult aspects of concurrency such as mutual exclusion, synchronization and resource contention. In this sense, the advantage of an on-chip ultra-wide word architecture is that it would enable word-RAM algorithms to achieve speedups comparable to multi-threaded computations, while at the same time keeping the simplicity of sequential programming that is inherent to the RAM model. We argue that this is the case by showing several examples of implementations of word-RAM algorithms using the wide word, usually with simple modifications to existing algorithms, and extending the ideas and techniques from the word-RAM model. We also show how the Ultra-Wide architecture can be used to simulate a non-standard memory layout, which has been used to sidestep known lower bounds in important data structure problems [6, 7].

In terms of the actual architecture, we envision the Ultra-Wide ALU together with multi-cores on the same chip. Thus, the Ultra-Wide architecture adds to the computing power of current architectures. The results in this paper, however, do not use multi-core parallelism.

1.1 Summary of Results

In this paper, we introduce the Ultra-Wide Word architecture and model, which extends the w-bit word-RAM model by adding an ALU that operates on w^2-bit words. We show that several broad classes of algorithms can be implemented in this model. In particular:

- We describe Ultra-Wide Word implementations of dynamic programming algorithms for the subset sum problem, the knapsack problem, the longest common subsequence problem, as well as many generalizations of these problems. Each of these algorithms illustrates a different technique (or combination of techniques) for translating an implementation of an algorithm in the word-RAM model to the Ultra-Wide Word model. In all these cases we obtain a w-fold speedup over word-RAM algorithms.
- We also describe Ultra-Wide Word implementations of popular string searching algorithms: the Shift-And/Shift-Or algorithms [2, 34], and the Boyer-Moore-Horspool algorithm [22]. As with the dynamic programming algorithms, we obtain a w-fold speedup over the original algorithms.
- Finally, we show that the Ultra-Wide Word model is powerful enough to simulate a non-standard memory architecture called Random Access Memory with Byte Overlap (RAMBO). This allows us to implement data structures and algorithms that circumvent known lower bounds for the word-RAM model.

The rest of this paper is organized as follows. In Section 2 we describe the Ultra-Wide architecture and model of computation. We show in Sections 3 how to simulate a Random Access Machine

with Byte Overlap (RAMBO) memory architecture. In Sections 4 and 5 we show examples of UW-RAM implementations of algorithms for Dynamic Programming and String Searching. We present concluding remarks in Section 6.

2 The Ultra-Wide word-RAM model

The Ultra-Wide word-RAM model we propose is an extension of the word-RAM model. We briefly review the key features of the word-RAM model together with some of its most representative algorithms.

2.1 Algorithms in the word-RAM model

The word-RAM is a variation of the RAM model in which a word has length w bits, and the contents of the memory are assumed to be integers in the range $\{0, \ldots, 2^{w-1}\}$ [18]. This implies that $w \geq \log n$, where n is the size of an input problem, and that the size of the memory is at most $2^{O(w)}$; otherwise a memory cell cannot be addressed using a constant number of words. The word-RAM includes the usual load, store and jump instructions of the RAM model, allowing for immediate operands and for direct and indirect addressing. In this model, arithmetic operations on two words are modulo 2^w and the instruction set includes left and right shift operations (equal to multiplication and division by powers of two) and boolean operations. All instructions take constant time to execute. There are different versions of the word-RAM model depending on the instruction set assumed to be available. The *restricted model* is limited to addition, subtraction, left and right shifts and boolean operations AND, OR, and NOT. These instructions augmented with multiplication constitute the *multiplication model*. Finally the AC^0 model assumes that all functions computable by an unbounded fan-in circuit of polynomial size (in w) and constant depth are available in the instruction set and execute in constant time. This definition includes all instructions from the restricted model and excludes multiplication. We refer to the reader to the survey by Hagerup [18] for a more extended description of the model and a discussion of its practicality.

Algorithms in the word-RAM model take advantage of the intrinsic parallelism in instructions that operate on words. The simplest examples are boolean operations: in one instruction we can compute the AND or OR of w sets of 1 bit each. In general, word-RAM algorithms exploit this parallelism by operating on various elements in parallel using operations on w-bits words. Word-level parallelism is the only source of increased efficiency with respect to traditional RAM algorithms, and hence w is the maximum speedup that can be obtained [18].

There are various algorithms for fundamental problems that take advantage of word-level parallelism or a bounded universe, some of which fit into the word-RAM model, although are not explicitly designed for it [33]. Much attention has been given to sorting and searching, for which known lower bounds in the comparison model do not carry to the word-RAM model [15]. For example, in a word-RAM model with multiplication, sorting n words can be done in $O(n \log \log n)$ time and $O(n)$ space deterministically [20], and in expected $O(n\sqrt{\log \log n})$ time and $O(n)$ space using randomization [21]. Word-RAM techniques have also been applied in many different areas, such as succinct data structures [23, 26], computational geometry [8, 9], and text indexing [16].

2.2 Ultra-Wide RAM

The Ultra-Wide word-RAM model (UW-RAM) extends the word-RAM model by introducing an Ultra-Wide ALU with w^2-bit *wide words*, where w is number of bits in a word-RAM model. The

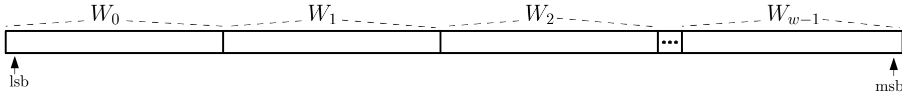

Fig. 1. A wide word in the Ultra-Wide Word architecture. The wide word is divided in w blocks of w bits each, shown here in increasing number of block from left to right.

Ultra-Wide ALU supports the basic operations available in a word-RAM model with multiplication on the entire word at once. Thus the supported operations are: addition, subtraction, left and right shift, bitwise boolean operations, and multiplication. In principle we allow multiplication, although the results of this paper require only two multiplications by constants, which can be replaced by straightforward AC^0 operations. The model maintains the standard w-bit ALU, as well as w-bit memory addressing. In terms of real world parameters, the wide word in the Ultra-Wide ALU would presently have between 1,000 and 10,000 bits, and could increase even further in the future.

Provided that the UW-RAM supports the same operations as the word-RAM, the techniques to achieve bit-level parallelism in the word-RAM extend directly to the UW-RAM. However, since the word-RAM assumes that a word can be read from memory in constant time, many operations in word-RAM algorithms can be implemented through table lookups. For example, counting the number of one bits in a word of $\log n$ bits can be implemented through two table lookups to a precomputed table that stores the number of set bits for each number of $\log n/2$ bits. The space used by the table is $\sqrt{n}$ words. We cannot expect to achieve the same constant time lookup operation with words of w^2 bits since the size of the lookup tables would be prohibitive. However, we allow for parallel table lookup operations within a wide word, implemented through parallel memory accesses in blocks within a wide word.

Before describing the memory access operations supported by the model we introduce some notation. Let W denote a w^2-bit word. Let $W[i]$ denote the i-th bit of W, and let $W[i..j]$ denote the contiguous sub-block of W from bit i to bit $j > i$, inclusive. The least significant bit of W is at $W[0]$, and thus $W = \sum_{i=0}^{w^2-1} W[i] \times 2^i$. For the sake of memory access operations we divide W into w-bit blocks that can access different w-bit words in memory. Let W_j denote the j-th contiguous block of w bits in W, for $0 \leq j \leq w-1$, and let $W_j[i]$ denote the i-th bit within W_j. Thus $W_j = W[jw..(j+1)w-1]$ and $W = \sum_{j=0}^{w-1} 2^j \times (\sum_{i=0}^{w-1} W_j[i] \times 2^i)$. The division of a wide word in blocks is solely intended for certain memory access operations, and other operations have no notion of block boundaries. Figure 1 shows a representation of a wide word, which depicts bits with increasing significance from left to right. Thus, shifts to the left (right) by i are equivalent to division (multiplication) by 2^i. In the description of operations with wide words we generally refer to variables with uppercase letters, and to regular variables that use one w-bit word with lower case. In addition, we use **0** to denote a wide word with value 0. We use standard C-like notation for operations AND ('&'), OR ('|'), NOT ('∼') and shifts ('<<','>>').

Memory access operations. In this architecture, w (not necessarily contiguous) words from memory can be transferred into the w blocks of a wide word W in constant parallel time. These blocks can be written to memory in parallel as well. Let MEM denote regular RAM memory, indexed by addresses to words[3]. The memory access operations provided by the model that involve

[3] A more sophisticated version of the model could consider accessing half-words and individual bytes as well, which would contribute to space savings for some algorithms.

Name	Input	Semantics
read_block	W, j, address	W_j ←MEM[address+j]
read_word	W, address	for all j in parallel: W_j ←MEM[address+j]
read_content	W, address	for all j in parallel: W_j ←MEM[address+W_j]
write_block	W, j, address	MEM[address+j]← W_j
write_word	W, address	for all j in parallel: MEM[address+j]← W_j
write_content	W, V, address	for all j in parallel: MEM[address+V_j]← W_j

Table 1. Wide word memory access operations supported by the UW-RAM.

wide words are specified in Table 1. Note that reading several (possibly non-contiguous) words from memory simultaneously is an assumption that is already made by any shared memory multiprocessing model. While, in reality, simultaneous access to all addresses in actual physical memory (e.g., DRAM) might not be possible, in shared memory systems, such as multi-core processors, the slowdown is mitigated by truly parallel access to private and shared caches, and thus the assumption is reasonable. We therefore follow this assumption in the same spirit.

2.3 UW-RAM Subroutines

We now describe some operations that will be used throughout the UW-RAM implementations we describe in later sections. A procedure we call *transpose* serves to bring together bits from all blocks into one block in constant time, while a procedure called *reverse transpose* is the inverse function. We also describe a parallel comparator, which is a standard technique used in word-RAM algorithms.

Transpose Let W be a word in which all bits are zero except possibly for the first bits of each block. The transpose operation moves the first bit of each block of a word W to the first block of the word, i.e., if $X = \text{transpose}(W)$, then $X[j] = W_j[0]$ for $0 \leq j < w$, and the rest of the bits of X are zero (See Figure 2). The transpose operation can be implemented by using the compression operation described by Brodnik [5, Ch 4.]. This operations takes a (regular size) word x whose bits all zeros but possibly for the bits in t blocks of k bits each that can have set bits (not necessarily equal across blocks). These blocks start at regular intervals of s bits[4]. The operation returns a word y with the contents of all blocks of x concatenated at the beginning of the word, and zeros in the rest of the word. More specifically, $y[i + j \cdot k] = x[i + j \cdot s]$ for $0 \leq i < k$ and $0 \leq j < t$ [5]. This operation can be implemented in constant time by first multiplying x by the constant $c = \frac{2^{ts} - 2^{tk}}{2^{s-k} - 1}$, then shifting the result to the left by $(t-1)s+k$, and finally doing a bitwise AND with $2^{tk} - 1$ [5]. In the case of the transpose operation we have $t = s = w$ and $k = 1$. Note that the three constants can be computed in constant time in the UW-RAM model, and since it t, s, k are fixed for any call to transpose, they can be *hardwired* in the transpose procedure. Note as well that the multiplication in the compression operation leaves relevant bits of the result in up to bit $ts - s + tk + k - 1$, which is w^2 in our model. In order to accommodate this operation, we assume that the wide word has a constant number of extra bits after the $(w^2 - 1)$-th bit in a word. The pseudocode for the transpose procedure is shown in Algorithm 1.

[4] This is called an (s, k)-sparse register in [5].

Algorithm 1 transpose(W)

$X \leftarrow W \times \frac{2^{w^2}-2^w}{2^{w-1}-1}$ {brings bits to contiguous positions in $X[(w-1)w+1..w^2]$}
$X \leftarrow (X << (w-1)w+1)$ & (2^w-1) {shifts bits to X_0 and cleans the rest of X}
return X

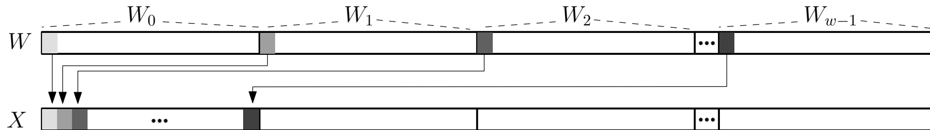

Fig. 2. The *transpose* operation takes a wide word W whose set bits are restricted to the first bits of each block and compresses them to the first block of a wide word.

Reverse Transpose This operation is the inverse of the transpose operation. It takes a word W whose set bits are all in the first block and spreads them across blocks of a word X so that $X_j[0] = W[j]$ for $0 \leq j < w$. We implement this operation by modifying the spreading operation in [5, Algorithm 4.3] to avoid reversing the order of the bits, and to end with all bits at the beginning of each block. We first replicate the contents of the first block in all blocks, and then extract the j-th bit of each block j. We then move this bit to the first bit of the block via four constant time operations. Algorithm 2 describes the details of this procedure.

Note that the *transpose* and *reverse transpose* operations require one multiplication by a constant each, and these are all the multiplications that we use in our results. Both operations are clearly in AC^0, and thus the model can also be regarded as a restricted model with two non-standard AC^0 operations.

Comparators Many word-RAM algorithms perform operations on pairs of elements in parallel by packing these elements in *fields* within one word. It is useful to be able to do fieldwise comparisons between two words. Suppose a word (either regular or wide) is divided in f-bit fields, with each field representing an $(f-1)$-bit number. Let G and F be two such words and let F_i and G_i denote the contents of the i-th field in F and G, respectively. Suppose that we want to identify all F_i such that $F_i \geq G_i$. Fieldwise comparisons can be done by setting the most significant bit of each field in F as a test bit and computing $H = F - G$. The most significant bit of the i-th field in H will be 1 if and only if $F_i \geq G_i$ [18]. Suppose we now want to operate only on the values of F that are greater than or equal to their corresponding value in G. We first mask away all but the test bits in H. A mask M with ones in all bits of the relevant fields and zeros everywhere else (including test bits) can be obtained by computing $M = H-(H << (f-1))$. The result of $(A$ & $F)$ contains then only the values of fields that pass the test [18]. Clearly this operation takes constant time, and it can be easily adapted to other standard comparisons. We shall assume that direct comparisons as well as operations that build on these (such as taking the fieldwise maximum between two words) are available and take constant time [18].

3 Simulation of RAMBO

The Random Access Machine with Byte Overlap (RAMBO) is a model of computation first introduced by Fredman and Saks [13] and further described by Brodnik [5]. In the standard RAM

Algorithm 2 reverse_transpose(W)

$C \leftarrow \frac{2^{w^2}-1}{2^w-1}$ $\{C_j = 1$ for all $j\}$
$X \leftarrow W \times C$ $\{X_j = W_0$ for all $j\}$
$D \leftarrow \frac{2^{(w+1)w}-1}{2^{w+1}-1}$ $\{D_j = 2^j$ for all $j\}$
$X \leftarrow X$ & D $\{X_j[j] = W[j]\}$
$M \leftarrow C >> (w-1)$ $\{M_j = 2^{w-1}$ for all $j\}$
$X \leftarrow M - X$ $\{X_j[w-1] = 0 \Leftrightarrow X_j[j] = 1$ for all $j\}$
$X \leftarrow (\sim X$ & $M) << (w-1)$ $\{X_j = W[j]$ for all $j\}$
return X

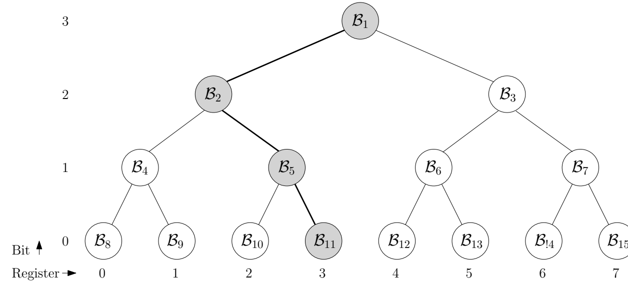

Fig. 3. Yggdrasil memory layout: each node in a complete binary tree is a RAMBO bit and registers are defined as paths from a leaf to the root. For example, register 3 contains bits $\mathcal{B}_{11}, \mathcal{B}_5, \mathcal{B}_2$, and $\mathcal{B}_1$ (shaded nodes).

model of computation, memory is organized in registers or words, each word containing a set of bits. Any bit in a word belongs to only that word. In contrast, in the RAMBO model words can overlap, that is, a single bit of memory can belong to several words. The topology of the memory, i.e., a specification of which bits are contained in which words, defines a particular variant of the RAMBO model. Variants of this model have been used to sidestep lower bounds for important data structure problems. For example, Brodnik *et al.* [6] use a variant of RAMBO called *Yggdrasil* in a data structure that achieves constant time for the operations insert, delete, membership, min, max, deletemin, deletemax, predecessor, and successor over a bounded universe of integers (known as the *discrete extended priority queue problem* [31]). In the Yggdrasil variant of RAMBO, words in memory are organized as paths from leaves to the root in a complete binary tree. Thus, bits might belong to several paths. Figure 3 shows an example of this this RAMBO layout.

We show how the UW-RAM can be used to implement memory access operations for any given RAMBO of word size at most w bits in constant time. Thus, the time bounds of any algorithm in the RAMBO model carry directly to the UW-RAM.

3.1 Implementing RAMBO operations in the UW-RAM

Let $\mathcal{B}_1, \ldots, \mathcal{B}_B$ denote the bits of RAMBO memory. A particular RAMBO memory layout can be defined by its *appearance sets*, this is, the locations of each bit $\mathcal{B}_i$ in the RAMBO memory [5]. For example, in the Yggdrasil model depicted in Figure 3, the appearance set of $\mathcal{B}_1$ is $\{\text{reg}[i].\text{bit}[3] | i = 0, \ldots, 7\}$, the one of $\mathcal{B}_4$ is $\{\text{reg}[0].\text{bit}[1], \text{reg}[1].\text{bit}[1]\}$, and the one of $\mathcal{B}_{12}$ is $\{\text{reg}[4].\text{bit}[0]\}$. Equivalently, the layout can be specified by the registers and the bits contained in them. In the example above, reg[0]=$\mathcal{B}_8\mathcal{B}_4\mathcal{B}_2\mathcal{B}_1$, and in general reg[$i$].bit[$j$]= $\mathcal{B}_k$, where $k = \lfloor i/2^j \rfloor + 2^{m-j-1}$ ($m = 4$ in the example) [6].

Algorithm 3 rambo_read(t)

1: read_block($W, \mathcal{R}[t]$) $\{w_j \leftarrow \mathcal{R}[t, j]\}$
2: read_content($W, \&A$) $\{w_j \leftarrow A[\mathcal{R}[t, j]]\}$
3: transpose(W)
4: write_block($W, 0, \&ret$) $\{ret \leftarrow W_0\}$
5: **return** ret

Algorithm 4 rambo_write($t, \mathcal{B} = \mathcal{B}_{i_0} \dots \mathcal{B}_{i_{b-1}}$)

1: read_block($W, 0, \mathcal{B}$) $\{w_0 \leftarrow \mathcal{B}\}$
2: reverse_transpose(W)
3: read_block($V, \mathcal{R}[t]$) $\{V_j \leftarrow \mathcal{R}[t, j]\}$
4: write_content($W, V, \&A$) $\{A[\mathcal{R}[t, j]] \leftarrow W_j\}$

In order to implement memory access operations on a given RAMBO using the UW-RAM, we need to represent the memory layout of RAMBO in standard RAM. We assume that the RAMBO memory layout is given as a table $\mathcal{R}$ that stores, for each register and bit within the register, the number of the corresponding RAMBO bit. Thus, if reg[i].bit[j]= $\mathcal{B}_k$, for some k, then $\mathcal{R}[i, j] = k$. This is without loss of generality as this representation can be easily precomputed from the appearance sets. We assume that $\mathcal{R}$ is stored in row major order.

Given a RAMBO memory of r registers of $b \leq w$ bits each, and $B \leq br$ distinct appearance sets, we want to store its contents in RAM. For this, we simply store each bit $\mathcal{B}_i$ in a different word. Thus, $A[i]$ stores the value of $\mathcal{B}_i$, where A is an array of integers in RAM. The total space used by this representation is then Bw bits, where w is the number of bits in a RAM word. Naturally we could store more than one bit in each word of A, however, this representation allows us to avoid concurrent writes to a same word.

Given an index t of a register of a RAMBO represented by $\mathcal{R}$, we can read the values of each bit of reg[t] from RAM and return the b bits in a word. Doing this sequentially for each bit might take $O(b)$ time. Using the wide word we can take advantage of parallel reading and the transpose operation to retrieve the contents of reg[t] in constant time. Let reg[t]= $\mathcal{B}_{i_0} \dots \mathcal{B}_{i_{b-1}}$. The read operation first reads the value of a bit $\mathcal{B}_{i_j}$ into block W_j of W by assigning $W_j = A[\mathcal{R}[t, j]]$. The second step consists of one transpose operation, after which the b bits are stored in W_0. Algorithm 3 shows the read operation, which takes constant time.

In order to implement the write operation reg[t]= $\mathcal{B}_{i_0} \dots \mathcal{B}_{i_{b-1}}$ of RAMBO, we first set $W_0 = \mathcal{B}_{i_0} \dots \mathcal{B}_{i_{b-1}}$ and perform a reverse transpose operation to place each bit $\mathcal{B}_j$ in block W_j. We then write the contents of each W_j in $A[\mathcal{R}[t, j]]$. Algorithm 4 shows this operation, which takes constant time as well.

Since the read and write operations described above are sufficient to implement any operation that uses RAMBO memory (any other operation is implemented in RAM), we have the following result.

Theorem 1. *Let $\mathcal{R}$ be any RAMBO memory layout of r registers of at most b bits each, and B distinct appearance sets, with $b \leq w$ and $\log B \leq w$. Let A be any RAMBO algorithm that uses $\mathcal{R}$, and runs in time T. Algorithm A can be implemented in the UW-RAM, to run in time $O(T)$, using $rb + B$ additional words of RAM.*

Proof. Table $\mathcal{R}$ indicating the RAMBO bit identifier for each register and bit within register can be stored in rb words of RAM, while the values of each bit can be stored in B words of RAM.

Since both rambo_read and rambo_write are constant time operations, any t-time operation that uses RAMBO memory can be implemented in UW-RAM in the same time t. In the case that $\mathcal{R}$ is not given, it can be computed from the appearance sets in $O(rb)$ time. This translation from appearance sets to $\mathcal{R}$ is fixed for the same RAMBO layout and needs only to be precomputed once. □

By Theorem 1, we can implement any arbitrary RAMBO memory layout and word-RAM algorithm with a moderate space overhead. Note that since any RAMBO implementation requires at least B bits of RAMBO memory, the relative overhead in space is reduced. The space overheads above are stated for a generic implementation of RAMBO. However, for particular RAMBO memory layouts one can save space by storing more than one RAMBO bit per RAM register, or by replacing table $\mathcal{R}$ with a constant time calculation of bit numbers from RAMBO registers and bits within registers (and adjusting rambo_read and rambo_write appropriately).

3.2 Constant time priority queue

Brodnik *et al.* [6] use the Yggdrasil RAMBO memory layout to implement priority queue operations in constant time using $3M$ bits of space ($2M$ of ordinary memory and M of RAMBO memory), where M is the size of the universe. This problem has non-constant lower bound for several models, including an $\Omega\left(\min\left(\frac{\lg\lg M}{\lg\lg\lg M}, \sqrt{\frac{\lg N}{\lg\lg N}}\right)\right)$ lower bound in the RAM model when the memory is restricted to $N^{O(1)}$, where N is the number of elements in the set to be maintained [32].

For a universe of size $M = 2^m$, the Yggdrasil RAMBO layout consists of $r = (M+1)/2$ registers of $b = \log M$ bits each, and $B = M - 1$ distinct appearance sets (See Figure 3 for an example with $M = 16$). Thus, applying Theorem 1 we obtain the following:

Corollary 1. *The discrete extended priority queue problem can be solved in $O(1)$ worst case time per operation using $2M + ((M+1)/2)\log Mw + (M-1)w$ bits, and thus in $O(M \log M)$ words of RAM.*

3.3 Constant time dynamic prefix sums

Brodnik *et al.* [7] use a modified version of the Yggdrasil RAMBO to solve the dynamic prefix sums problem in constant time. The dynamic prefix sums problem consists of maintaining an array A of size N, supporting the operations $update(j, d)$ which sets $A[j]$ to $A[j] + d$, and $retrieve(j)$, which returns $\sum_{i=0}^{j} A[i]$ [7], where the *sum* can be any binary operation. This RAMBO implementation sidesteps various lower bounds for dynamic prefix sums on different models: there is an $\Omega(\log N)$ algebraic complexity lower bound [14] as well as and under the semi-group model of computation [19], and a $\Omega(\log N/\log\log N)$ information-theoretic lower bound [14].

The result of Brodnik et al. [7] uses a complete binary tree on top of array A as leaves. The tree is similar to the one used in the priority queue problem, but it differs in that only internal nodes store any information, and that there are $m = \lceil \log M \rceil$ bits stored in each node, where M is the size of the universe. This tree is stored in a variant of the Yggdrasil memory called m-Yggdrasil, in which each register correspond again to a path from a leaf to the root, but this time each node stores not only one bit but the m bits containing the sum of all leaves in the left subtree of that node [7]. It is assumed that $nm \leq w$, where $n = \lceil \log N \rceil$ and w is the size of the word in bits. Thus an entire path from leaf to root fits in a word and can be accessed in constant time. An update

or retrieve operation consists of retrieving the values along a path in the tree and processing them in constant time using bit-parallelism and table lookup operations. The space used by the lookup table can be reduced at the expense of an increased time for the retrieve operation. In general, both operations can be supported in time $O(\iota+1)$ with $(N-1)m$ bits of m-Yggdrasil memory and $O(M^{n/2^{\iota}} \cdot m + m)$ bits of RAM [7].

In order to represent the m-Yggdrasil memory in our model, we treat each bit of a node in the tree as a separate RAMBO bit. Thus the RAMBO memory has $r = N$ registers of $b = nm$ bits each, and there are $B = (N-1)m$ distinct bits to be stored. Thus by Theorem 1 we have:

Corollary 2. *The operations update and retrieve of the dynamic prefix sums problem can be supported in the UW-RAM model in $O(\iota+1)$ time with $O(M^{n/2^{\iota}} \cdot m + Nmnw)$ bits of RAM. For constant time operations ($\iota = 1$) the space is dominated by the first term, i.e. the space is $O(M^{\sqrt{\log N}})$ bits. For $\iota = \log\log N$, the time is $O(\log\log N)$ and the space is $O(Nmnw)$ bits.*

4 Dynamic programming

In this section we show how to speed up various dynamic programming algorithms in the UW-RAM. We show that an existing word-RAM algorithm for the subset sum problem can be directly translated to the UW-RAM, and show how to adapt an existing algorithm for the knapsack problem. We note that these problems have many generalizations that can be solved using the same techniques. Based on similar techniques, we describe a word-RAM algorithm (and UW-RAM implementation) for the longest common subsequence (LCS) problem. The implementation for subset sum as well as the first solution to LCS are examples of pure bit parallelism, while the knapsack implementation and the second algorithm for LCS use the parallel lookup power of the UW-RAM.

4.1 Subset Sum

Given a set $S = \{x_1, x_2, \ldots, x_n\}$ of nonnegative integers (weights), and an integer t (capacity), the subset sum problem is to find $S' \subseteq S$ such that $\sum_{a_i \in S'} a_i = t$. The optimization version asks for the solution of maximum weight which does not exceed t [10]. This problem is NP-hard but it can solved in pseudopolynomial time via dynamic programming in $O(nt)$ time, using the following recursion by Bellman [4]: for each $0 \leq i \leq n$ and $0 \leq j \leq t$, $C_{i,j}$ is true if there is a subset of elements $\{a_1, \ldots, a_i\}$ that adds up to j. Thus $C_{0,0}$ is true, $C_{0,j}$ is false for all $j > 0$, and value $C_{i,j}$ is true if $C_{i-1,j}$ is true or $C_{i-1,j-a_i}$ is true ($C_{i,j}$ is false for any $j < 0$). The problem admits a solution if $C_{n,t}$ is true.

Pisinger [29] gives an algorithm that implements this recursion in the word-RAM model with word size w by representing up to w values of a row of C. Using bit parallelism, w bits of a row can be updated simultaneously in constant time from the values of the previous row: row C_i is updated by computing $C_i = (C_{i-1} \mid (C_{i-1} >> a_i))$ (which might require shifting words containing C_{i-1} first by $\lfloor a_i/w \rfloor$ words and then by $a_i - \lfloor a_i/w \rfloor$) [29]. Assuming $w = \Theta(\log t)$, this approach leads to a $O(nt/\log t)$ solution in $O(t \log t)$ space. The actual values in S' that compose the solution can be then recovered with the same space and time bounds with a recursive technique by Pferschy [28].

Pisinger's algorithm can be implemented directly in the UW-RAM: entries of a row C_i are stored contiguously in memory, thus we can load and operate on w^2 bits simultaneously when updating each row. Hence, UW-RAM implementation runs in $O(nt/\log^2 t)$ time using the same $O(t \log t)$ space.

4.2 Knapsack

Given a set S of n elements with weights and values, the knapsack problem asks for a subset of S of maximum value such that the total weight is below a given capacity bound b. Let $S = \{(w_i, v_i)\}_{i=1}^n$ where w_i and v_i are the weight and value of the i-th element. Just like the subset sum problem, this problem is NP-hard but can be solved in pseudopolynomial time using the following recurrence by Bellman [4]. Let $C_{i,j}$ be the maximum value of a solution containing elements in the subset $S_i = \{(w_k, v_k)\}_{k=1}^i$ with maximum capacity j. Then $C_{0,j} = 0$ for all $0 \leq j \leq b$, and $C_{i,j} = \max\{C_{i-1,j}, C_{i-1,j-w_i} + v_i\}$. The value of the optimal solution is $C_{n,b}$. This leads to a dynamic program that runs in $O(nb)$ time.

The word-RAM algorithm by Pisinger represents partial solutions of the dynamic programming table with two binary tables g and h and operates on the $O(w)$ entries at a time [29]. More specifically, the entry $g_{i,w} = 1$ and $h_{i,v} = 1$ if and only if there is a solution with weight w and value v that is not dominated by an other solution in $C_{i,*}$ (i.e. there is no entry $C_{i,w'}$ such that $w' < w$ and $C_{i,w'} \geq v$). Pisinger shows how to update each entry of g and h with a constant time procedure, which can be encoded as constant size lookup table. By composing this table $\alpha = w/10$ times, α entries of the tables can be computed in constant time, so an entire row can be computed in $O(m/w)$ time and $O(m/\log m)$ space, where m is the maximum of the capacity b and the value of the optimal solution[5]. The optimal solution can then be computed in $O(nm/w)$ time [29].

Compared to the subset sum algorithm, which relies mainly on bit-parallel operations, this word-RAM algorithm for knapsack relies on table precomputation and lookup to achieve a w speedup. In this sense, the UW-RAM implementation of the knapsack algorithm is a good example of the parallel lookup power of the architecture. While we cannot precompute a composition of $\Theta(w^2)$ lookup tables to compute $\Theta(w^2)$ entries of g and h at a time, we can use the same tables with $\alpha = w/10$ as in Pisinger's algorithm and use the block of the wide word to make w simultaneous lookups to the table. Since the values a row i of h and g depend only on row $i-1$ of these tables, then there are no dependencies between values in the same row.

One difficulty, however, is that in order to compute the values in row i in parallel we must first preprocess row $i-1$ in both h and g, such that we can return the number of one bits in both $g_{i-1,0}, ..., g_{i-1,j}$ and $h_{i-1,0}, ..., h_{i-1,j}$ in $O(1)$ time for any column $j \in \{0, m-1\}$. That is, the prefix sums of the one bits in the row $i-1$ up to column j. Note that since we are only required to compute the prefix sums of row $i-1$ one time, this is *not* the same as the dynamic problem described in Section 3.3, i.e., it is a static problem. Furthermore, since the algorithm is the same for both g and h, we describe the computation for g alone.[6]

Static Prefix Sums: We begin by computing the number of ones in $g_{i-1,k}, ..., g_{i-1,k+w-1}$ for each column $k \in \{0, w, 2w, ..., \lfloor m/w \rfloor w\}$ using a lookup table, and store the results in an array $\mathcal{A}$ of length $\lfloor m/w \rfloor$. Next, we compute the prefix sums of $\mathcal{A}$ in two steps. The first step is to set

$$\mathcal{A}'[k] = \mathcal{A}[\lfloor k/w \rfloor w] + \mathcal{A}[\lfloor k/w \rfloor w + 1] + ... + \mathcal{A}[k] \ ,$$

for each $k \in \{0, ..., \lfloor m/w \rfloor\}$. This can be done for w array indices at a time, $k_0, ..., k_{w-1}$, where $k_\ell = k' + \ell w$, and k' iterates over the sequence

$$0, 1, ..., w-1, w^2, w^2+1, ..., w^2+w-1, 2w^2, 2w^2+2, ... \ .$$

[5] This value is not known in advanced but an upper bound of at most twice the optimal value can be used [29, 11].

[6] We note that it is possible to simulate the standard parallel prefix sums algorithm (for w processors in this case) [24] using the UW-RAM, though we believe the algorithm described in the sequel to be more straightforward.

Note that we can compute all w array entries in constant time, for each k' in the previous sequence, since

$$\mathcal{A}'[k' + \ell w] = \begin{cases} \mathcal{A}'[k' + \ell w - 1] + \mathcal{A}[k' + \ell w] & \text{if } k \neq 0 \mod w, \\ \mathcal{A}[k' + \ell w] & \text{otherwise.} \end{cases}$$

The second step is to set $\mathcal{A}'[k] = \mathcal{A}'[k] + \mathcal{A}'[\lfloor k/w \rfloor w - 1]$, for $k \in \{w, w+1, ..., \lfloor m/w \rfloor\}$. This can also be done for w values at once, $k_0, ..., k_{w-1}$, where $k_\ell = k' + \ell$, and $k' \in \{w, 2w, ..., \lfloor m/w \rfloor\}$. At this point $\mathcal{A}'$ contains the prefix sums of $\mathcal{A}$, and took $O(|\mathcal{A}|/w) = O(m/w^2)$ time to compute, by exploiting the block read and write operations of the UW-RAM.

Let f be the number of ones in $g_{i-1,\lfloor j/w \rfloor}, ..., g_{i-1,j}$, which can be computed using the lookup table. To compute $g_{i-1,0}, ..., g_{i-1,j}$ we return $f + \mathcal{A}'[\lfloor j/w \rfloor]$. Since each row of g and h requires $O(m/w^2)$ to compute, and there are n rows, the total time to compute g and h (and thus to compute the optimal solution) on the UW-RAM is $O(nm/w^2)$. This achieves a w-speedup over Pisinger's word-RAM solution.

4.3 Generalizations of Subset Sum and Knapsack Problems

Pisinger [29] uses the techniques of the word-RAM algorithm for subset sum and knapsack to obtain a word-RAM algorithms for obtaining a path in a layered network: given a graph $G = (V, E)$, a source $s \in V$ and a terminal $t \in V$, and a weight for each edge, is there a path of weight b from s to t? Again, this algorithm translates directly to a UW-RAM algorithm, thus yielding a w speedup over the word-RAM algorithm. Pisinger uses these algorithms to implement word-RAM solutions for other generalizations of subset sum and knapsack problems, such as: the bounded subset sum and knapsack problems (each element can be chosen a bounded number of times), the multiple choice subset sum and knapsack problems (the set of numbers is divided in classes and the target sum must be matched with one number of each class), the unbounded subset sum and knapsack problems (each element can be chosen an arbitrary number of times), the change-making problem, and, finally, the two-partition problem. UW-RAM implementations for all these generalizations are direct and yield a w speedup over the word-RAM algorithms.

4.4 Longest Common Subsequence

The final dynamic programming problem we examine is that of computing the Longest Common Subsequence (LCS) of two string sequences. Given a sequence of symbols $X = x_1 x_2 \dots x_m$, a sequence $Z = z_1 z_2 \dots z_k$ is a subsequence of X if there exists an increasing sequence of indices $i_1, i_2, \dots, i_k$ such that for all $1 \leq j \leq m$, $x_{i_j} = z_j$ [10]. Let Σ be a finite alphabet of symbols, where $\sigma = |\Sigma|$. Given two sequences $X = x_1 x_2 \dots x_m$ and $Y = y_1 y_2 \dots y_n$, where $x_i, y_j \in \Sigma$, the LCS problem asks for a sequence $Z = z_1 z_2 \dots z_k$ of maximum length such that Z is a subsequence of both X and Y. This problem can be solved via a classic dynamic programming algorithm in $O(nm)$ time. In what follows, we show how to combine techniques used for subset sum and knapsack, as well as the four Russians technique, in order to achieve further speedups in the UW-RAM model. The first algorithm presented runs in $O(\frac{nm}{w^2} \log \sigma + m + n)$ time, while the second is more involved and runs in time $O(n^2 \log^2 \sigma / w^3 + n \log \sigma / w)$, assuming $m = n$ for simplicity.

Let $c_{i,j}$ denote the length of the LCS of $X[1..i] = x_1 x_2 \dots x_i$ and $Y[1..j] = y_1 y_2 \dots y_j$, then the following recurrence allows us to compute the length of the LCS of X and Y [10]:

$$c_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ c_{i-1,j-1} + 1, & \text{if } x_i = y_j \\ \max\{c_{i,j-1}, c_{i-1,j}\}, & \text{otherwise} \end{cases} \quad (1)$$

The length of the LCS is $c_{m,n}$, which can be computed in $O(mn)$ time. Consider an $(m+1) \times (n+1)$ table C storing the values $c_{i,j}$. The idea of the UW-RAM algorithm is to compute various entries of this table in parallel. We assume $w = \Theta(\max\{\log n, \log m\})$.

Let d_k denote the values in the k-th a diagonal of table C, this is $d_k = \{c_{i,j} | i + j = k\}$. Since a value in a cell $i, j > 0$ depends only on the values of cells $(i-1, j)$, $(i-1, j-1)$ and $(i, j-1)$, all values in the same diagonal d_k are independent of each other and can be computed in parallel. Thus, we use the wide word to compute various entries of a diagonal in constant time. Since each value in the cell might use up to $\min\{\log n, \log m\}$ bits, each value might use up to an entire block of the wide word (if $m = \Theta(n)$), thus w cells can be computed in parallel. Since the total number of cells is $O(mn)$ and the critical path of the table is $m+n+2$ cells, this approach requires $O(mn/w+m+n)$ parallel time, resulting in a speedup of w. However, we can obtain better speedups by using fewer bits per entry of the table, which enables us to operate on more values in parallel. For this sake, instead of storing the actual values of the partial longest common subsequences we store differences between consecutive values as described in [25] for the related string edit distance problem.

Let $V_{i,j} = c_{i,j} - c_{i-1,j}$ and $H_{i,j} = c_{i,j} - c_{i,j-1}$ for $1 \le i \le m$ and $1 \le j \le n$ denote the tables of vertical and horizontal differences of values in C. We adapt Corollary 1 in [25] for the computation of V and H:

Proposition 1. *Let* $[x_i = y_j] = 1$ *if* $x_i = y_j$ *and 0 otherwise. Then* $V_{i,j} = \max\{[x_i = y_j] - H_{i-1,j}, 0, V_{i,j-1} - H_{i-1,j}\}$ *and* $H_{i,j} = \max\{[x_i = y_j] - V_{i,j-1}, 0, H_{i-1,j} - V_{i,j-1}\}$.

Proof. Directly from Recurrence 1 we obtain $V_{i,j} = 1 - H_{i-1,j}$ if $x_i = y_j$ and $V_{i,j} = \max\{0, V_{i,j-1} - H_{i-1,j}\}$ otherwise. Similarly $H_{i,j} = 1 - V_{i,j-1}$ if $x_i = y_j$ and $H_{i,j} = \max\{0, H_{i-1,j} - V_{i,j-1}\}$ otherwise. It is easy to verify from the definition of longest common subsequence and Recurrence 1 that $0 \le H_{i,j} \le 1$ and $0 \le V_{i,j} \le 1$ for all i, j, which implies that the maximum in $\max\{[x_i = y_j] - H_{i-1,j}, 0, V_{i,j-1} - H_{i-1,j}\}$ and $\max\{[x_i = y_j] - V_{i,j-1}, 0, H_{i-1,j} - V_{i,j-1}\}$ is equal to the first term if $x_i = y_j$ and to the second or third terms otherwise. □

We compute tables H and V according to Proposition 1 diagonal by diagonal using bit parallelism in the wide word. Assume an alphabet $\Sigma = \{0, 1, 2, \ldots, \sigma-1\}$ with $\lceil \log \sigma \rceil \le w-1$. Although all entries in tables H and V are either 0 or 1 we will use fields of $O(\log \sigma)$ bits to store these values, since we can only compare at most $w/\log \sigma$ symbols simultaneously in the wide word. We divide the wide word W in f-bit fields with $f = \max(\lceil \log \sigma \rceil, 2) + 1$. Each field will be used to store both symbols and intermediate results for the computation of diagonals of H and V, plus an additional bit to serve as a test bit in order to implement fieldwise comparisons as described in Section 2.3. We require at least 3 bits because although all entries in tables H and V use one bit, intermediate results in calculations can result in values of -1. Thus we require 2 bits to represent values -1, 0, and 1, and a test or sentinel bit to prevent carry bits resulting from subtractions to interfere with neighbouring fields. We represent -1 in two's complement. It is not hard to extend the techniques for comparisons and maxima to the case of positive and negative numbers [18].

Let H_k and V_k denote the k-th diagonal of H and V, respectively, i.e., $H_k = \{H_{i,j} | i + j = k\}$ and $V_k = \{V_{i,j} | i + j = k\}$. Consider table H. We will operate with each diagonal H_k using $\lceil |H_k|/\ell \rceil$

words, where $\ell = w^2/f$. Let $f_0, \ldots, f_{\ell-1}$ denote the fields within W, in increasing order of bit significance within W. In each wide word, cells of H_k will be stored in increasing order of column, i.e., if $H_{i,j}$ is stored in field f_r, then f_{r+1} stores $H_{i-1,j+1}$. In order to compute each diagonal we must compare the relevant entries of strings X and Y. We assume that each symbol of X and Y is stored using $\lceil \log \sigma \rceil + 1$ bits (including the test bit), and that X is stored in reverse order. X and Y can be preprocessed in $O(m+n)$ to arrange this representation, which will allow us to do parallel comparisons of symbols for each diagonal loading contiguous words of memory in wide words.

Consider a diagonal H_k. Assume that the entire diagonal fits in a word W. This will not be the case for most diagonals, but we describe the former case for simplicity. The latter case is implemented as a sequence of steps updating portions of the diagonal. We update the entries of H_k as follows: (1) we load the symbols of the relevant substrings of X and Y into words W_X and W_Y, with the substring of X in reverse order. More specifically, for a diagonal k, $W_Y = y_{j_1} y_{j_1+1} \ldots y_{j_2}$ where $j_1 = k - \min(m, k-1)$ and $j_2 = \min(n, k)$, and $W_X = x_{i_2} x_{i_2-1} \ldots x_{i_1}$ with $i_2 = k - j_1$ and $i_1 = k - j_2$. We subtract W_Y from W_X, mask out all non-zero results and write a 1 in each field that resulted in 0. We store the resulting word in W_{eq}, a 1 in each field corresponding to a cell (i,j) with $x_i = y_j$ and a 0 otherwise (this can be implemented through comparisons as described in Section 2.3). (2) We load V_{k-1} into a word W_V and subtract it from W_{eq} to obtain $[a_i = b_j] - V_{i,j-1}$ for all i,j in H_k simultaneously and store the result in W_1. (3) We load H_{k-1} into a word W_H and subtract W_V to it to obtain $H_{i-1,j} - V_{i,j-1}$ for all i,j in H_k, storing the result in W_2. (4) Finally, using fieldwise comparisons we obtain the fieldwise maximum of W_1, W_2 and and the word $\mathbf{0}$. The resulting word is H_k. The procedure to compute V_k is analogous. Note that the entries corresponding to base cases in the first row and column in the LCS table correspond to the base cases of the horizontal and vertical vectors, respectively. When computing diagonals H_k with $k \leq n+1$ and V_k with $k \leq m+1$, the entries corresponding to base cases are not computed from previous diagonals but should be added appropriately at the end of H_k and beginning of V_k.

Example 1. Let $X = abbab$ and $Y = aabbba$ be two strings. Figure 4 shows the entries of the dynamic programming table for computing the LCS of X and Y, as well as the values of horizontal and vertical differences.

	j		1	2	3	4	5	6
LCS			a	a	b	b	b	a
i		0	0	0	0	0	0	0
1	a	0	1	1	1	1	1	1
2	b	0	1	1	2	2	2	2
3	b	0	1	1	2	3	3	3
4	a	0	1	2	2	3	3	4
5	b	0	1	2	3	3	4	4

	j	1	2	3	4	5	6
H		a	a	b	b	b	a
i		0	0	0	0	0	0
1	a	1	0	0	0	0	0
2	b	1	0	1	0	0	0
3	b	1	0	1	1	0	0
4	a	1	1	0	1	0	1
5	b	1	1	1	0	1	0

	j		1	2	3	4	5	6
V			a	a	b	b	b	a
i								
1	a	0	1	1	1	1	1	1
2	b	0	0	0	1	1	1	1
3	b	0	0	0	0	1	1	1
4	a	0	0	1	0	0	0	1
5	b	0	0	0	1	0	1	0

Fig. 4. Dynamic programming tables for the longest common subsequence and vector differences for $X = abbab$ and $Y = aabbba$.

In this example $\sigma = 2$, thus we use one bit for each symbol ('a'=0, 'b'=1), but we use $f = 3$ bits per field. Consider the diagonal H_6 in table H (in dark gray). We now illustrate how to obtain H_6 from H_5 and V_5 (in light gray). In what follows we represent the number in each field in decimal and do not include the details of fieldwise comparison and maxima.

$$
\begin{array}{rll}
W_X = & 1\ 0\ 1\ 1\ \ 0 & (=x_5x_4x_3x_2x_1) \\
W_Y = & 0\ 0\ 1\ 1\ \ 1 & (=y_1y_2y_3y_4y_5) \\
W_{eq} = & 0\ 1\ 1\ 1\ \ 0 & (W_{eq}[f \cdot (j-1)] = 1 \Leftrightarrow x_{|H_5|-j} = y_j) \\
V_5 = & 0\ 0\ 0\ 1\ \ 1 & \\
W_1 = W_{eq} - V_5 = & 0\ 0\ 1\ 0\ \text{-}1 & \\
\hline
H_5 = & 1\ 0\ 1\ 0\ \ 0 & \\
W_2 = H_5 - V_5 = & 1\ 0\ 1\ \text{-}1\ \text{-}1 & \\
\hline
\max\{W_1, W_2, \mathbf{0}\} = & 1\ 1\ 1\ 0\ \ 0 & \\
H_6 = & 1\ 1\ 1\ 0\ \ 0\ \ 0 & \text{(last 0 is the base case)}
\end{array}
$$

Once all diagonals are computed, the final length of the longest common subsequence of X and Y can be simply computed by (sequentially) adding the values of the last row of H or the values of last column of V (which can be done while computing H and V). The entire procedure is described in Algorithm 5 and leads to the following theorem.

Theorem 2. *Let Σ be an alphabet of size σ. Given two strings X and Y over Σ of lengths m and n, respectively, the length of the longest common subsequence of X and Y can be computed in $O(\frac{nm}{w^2}\log\sigma + m + n)$ and $O(\min(n,m)w/\log\sigma)$ memory words in addition to the input.*

Proof. A diagonal of H and V of length ℓ entries can be computed in time $O(\ell\log\sigma/w^2 + 1)$. Adding this time over all $m+n$ diagonals yields the total time. For the space, each diagonal is represented in $\lceil \ell f/w^2 \rceil$ wide words, where $f = O(\log\sigma)$ is the number of bits per field. Since we can compute each diagonal H_k and V_k using only H_{k-1} and V_{k-1}, we only need to store 4 diagonals at any given time. Since the maximum length of a diagonal is $\min(n,m)$ and each wide word can be stored in w regular words of memory, the result follows. ◻

Recovering a Longest Common Subsequence It is known that given a dynamic programming table storing the values of the LCS between strings X and Y one can recover the actual subsequence by, starting from $c_{m,n}$ following the path through the cells corresponding to the values used when computing each value $c_{i,j}$ according to Recurrence (1): if $x_i = y_j$ then we add x_i to the LCS and continue with cell $(i-1, j-1)$. Otherwise the path follows the cell corresponding to the maximum of $c_{i-1,j}$ or $c_{i,j-1}$. Although Algorithm 5 does not compute the actual LCS table, a path of an LCS can be easily computed using tables H and V. The path starts at cell (m,n) (of either table) and to continue from a cell (i,j), if $x_i = y_j$ then x_i is part of the LCS and we continue with cell $(i-1, j-1)$. Otherwise, if $H_{i,j} = 1$ and $V_{i,j} = 0$ then we continue with cell $(i-1, j)$, and if $H_{i,j} = 0$ and $V_{i,j} = 1$ we continue with cell $(i, j-1)$ (and with any of the two if $H_{i,j} = V_{i,j} = 0$). This can be easily done in $O(m+n)$ time if all diagonals of tables V and H are kept in memory while computing the LCS length in Algorithm 5. This would require Algorithm 5 to use $O(nmw/\log\sigma)$ words of memory to store all diagonals.

Four Russians Technique The computation of the longest common subsequence can be made even faster by combining the diagonal-by-diagonal order of computation described above with the Four Russians technique. The Four Russians technique [33] was applied to the string edit problem (and also the LCS) by Masek and Paterson [25], and it consists of dividing the dynamic programming table in blocks of size $t \times t$ cells. In a precomputation phase, all possible blocks are

Algorithm 5 LCS-length$(X, Y, m = |X|, n = |Y|, \sigma)$

1: $f \leftarrow \max(\lceil \log \sigma \rceil, 2) + 1$ {field length in bits}
2: $H_1^1 \leftarrow \mathbf{0}$ $\{H_{0,1} = 0\}$
3: $V_1^1 \leftarrow \mathbf{0}$ $\{V_{1,0} = 0\}$
4: $length \leftarrow 0$ {length of longest common subsequence}
5: **for** $k = 2$ to $m + n$ **do**
6: $\quad \ell \leftarrow \min(n, k-1) + \min(m, k-1) - k + 1$ {length of diagonal}
7: $\quad j_1 \leftarrow k - \min(m, k-1)$ {indices of relevant substrings of X and Y}
8: $\quad j_2 \leftarrow \min(n, k)$
9: $\quad i_2 \leftarrow k - j_1$
10: $\quad i_1 \leftarrow k - j_2$
11: $\quad j \leftarrow j_1$
12: $\quad i \leftarrow i_2$
13: $\quad s \leftarrow \lceil \ell f / w^2 \rceil$ {number of wide words per diagonal}
14: $\quad$ **for** $t = 1$ to s **do**
15: $\qquad j' \leftarrow \min(j + s - 1, j_2)$
16: $\qquad i' \leftarrow \max(i + s - 1, i_1)$
17: $\qquad W_Y \leftarrow Y[j..j']$
18: $\qquad W_X \leftarrow X[i..i']$ {substring of X is in reverse order}
19: $\qquad W_{eq} \leftarrow$equal(W_X, W_Y)
20: $\qquad W_1 \leftarrow W_{eq} - V_{k-1}^t$
21: $\qquad W_2 \leftarrow H_{k-1}^t - V_{k-1}^t$
22: $\qquad H_k^t \leftarrow \max(W_1, W_2, \mathbf{0})$ {base case is implicitly added at rightmost field}
23: $\qquad W_1 \leftarrow W_{eq} - H_{k-1}^t$
24: $\qquad W_2 \leftarrow V_{k-1}^t - H_{k-1}^t$
25: $\qquad V_k^t \leftarrow \max(W_1, W_2, \mathbf{0})$
26: $\qquad$ **if** $t = 1$ AND $k \leq m + 1$ **then**
27: $\qquad\quad V_k^t \leftarrow V_k^t >> f$ {add 0 in the first field for the base case}
28: $\qquad i \leftarrow i' + 1$
29: $\qquad j \leftarrow j' + 1$
30: $\qquad$ **if** $t = 1$ AND $k \geq m + 1$ **then**
31: $\qquad\quad length \leftarrow length + H_k^1[0..f-1]$ $\{length = length + H_{m,k-m}\}$
32: **return** $length$

computed and stored as a data structure indexed by the first row and column of each block. The LCS can be then computed by looking up relevant values of the table one block at a time using the data structure. In a RAM with indirect addressing and under a suitable value of t, the last row and column of a block can be obtained by looking up the entry corresponding to the first row and column of that block in constant time. This technique yields a speedup of $O(t^2)$ with respect to computing all cells in the table, for a total time of $O(n^2/t^2)$ (for two strings of length n) plus the time for the precomputation of all blocks. By setting $t = O(\log n)$ and encoding the table with difference vectors the precomputation time can be absorbed by the time to compute the main table (See [25, 17] for a more detailed description of the technique).

We can use the power of the parallel memory access of the UW-RAM to speedup the computation of the LCS even further by looking up blocks in parallel, in a similar fashion to the diagonal-by-diagonal approach described above. For simplicity assume $m = n$. Using the same encoding for H and V, we first precompute all possible blocks of H and V of size $t \times t$. Since a block is completely determined by the first column and rows, whose values are 0 and 1, and the two substrings of length t (over an alphabet of size σ), there are $O((2\sigma)^{2t})$ possible blocks. Note that we can encode each cell now with one bit, since we do not need to do symbol comparisons

in parallel. Each block can be computed in $O(t^2)$ time with the standard sequential algorithm, so the precomputation time is $O((2\sigma)^{2t}t^2)$. We set $t = \log_{2\sigma} n/2$, and thus the precomputation time is $O(n \log^2 n)$ [17]. Since $t \leq w/2$ we can use each block of the wide word to lookup the entry for each block by using a parallel lookup operation. Thus, as described previously, we can compute tables H and V in diagonals of blocks, computing $\min(\ell, w)$ blocks simultaneously in a diagonal of length ℓ blocks. There are $(n/t)^2$ blocks to compute and the critical path of the table has length n/t blocks. Therefore, the computation of H and V can be carried out in time $O(n^2/(t^2 w) + n/t) = O(n^2 \log^2 \sigma/w^3 + n \log \sigma/w)$, since $t = \Theta(w/\log \sigma)$. For a constant alphabet size and $w = \Theta(\log n)$ this time is $O(n^2/\log^3 n)$.

5 String Search

Another example of a problem where a large class of algorithms can be sped up in the UW-RAM is String Searching. Given a text T of length n and a pattern P of length m, both over an alphabet Σ, the string matching problem consists of reporting all the occurrences of P in T. We focus here on on-line searching, this is, with no preprocessing of the text (though preprocessing the pattern is allowed), and we assume in general that $n \gg m$. We use two classic algorithms for this problem to illustrate different ways of obtaining speedups via parallel operations in the wide word. More specifically, we obtain w-speedups for UW-RAM implementations of the Shift-And and Shift-Or algorithms [2, 34], and the Boyer-Moore-Horspool algorithm [22]. For a string S, let $S[i]$ denote the i-th character of S, and let $S[i..j]$ denote the substring of S starting at position i and ending at position j. Indices start at 1.

5.1 Shift-And and Shift-Or

The Shift-And and Shift-Or algorithms keep a sliding window of length m over the text T. On a window at positions at substring $T[i-m+1..i]$, the algorithms keep track of all prefixes of P that match a suffix of $T[i-m+1..i]$. Thus, if at any time there is one such prefix of length $|P|$ then an occurrence is reported at $T[i-m+1]$. This is equivalent to running the $(m+1)$-state non-deterministic automaton that recognizes P starting from every position of T. For a window $T[i-m+1..i]$ in T, the j-th state of the automaton is active if and only if $P[0..j] = T[i-j+1..i]$. These algorithms represent the automaton as a bit vector and update the active states using bit-parallelism. More specifically, the Shift-And algorithm keeps a bit vector $\boldsymbol{v} = b_0 b_1 \ldots b_{m-1}$, where $b_j = 1$ whenever the j-th state is active. If $\boldsymbol{v}_i$ represents the automaton for the window ending at $T[i]$, then $\boldsymbol{v}_{i+1} = ((\boldsymbol{v}_i >> 1) \mid 1) \,\&\, Y[T[i]]$, where $Y[\sigma]$ is a bit vector with set bits in the positions of the occurrences of σ in P. The OR with a 1 corresponds to the first state always being active to allow a match to start at any position. The Shift-Or algorithm is similar but it saves this operation by representing active states with zeros instead of ones. We describe two UW-RAM algorithms for Shift-And that illustrate different techniques, noting that the UW-RAM implementation of Shift-Or is analogous. The running times of the UW-RAM algorithms are $O(nm/w^2 + n)$ and $O(nm/w^2 + n/w)$, which are $O(n)$ and $O(n/w)$, resp., for $m = O(w^2)$.

w^2-bit Automaton The straightforward way of taking advantage of the wide word when implementing Shift-And is to use the entire wide word for bit vectors. We first compute the mask array $Y[\sigma]$ for each $\sigma \in \Sigma$ and store each w^2-bit vector in contiguous words of memory starting at address

Algorithm 6 Shift_And($T, P, n = |T|, m = |P|, \Sigma$)

```
1: {Preprocessing}
2: for each σ ∈ Σ do
3:     Y[σ] ← 0
4: for j = 1 to m do
5:     Y[P[j]] ← Y[P[j]] | (1 >> (j − 1))
6: {Search}
7: V ← 0
8: C ← 1 >> (m − 1)
9: for i = 1 to n do
10:    V = ((V >> 1) | 1) & Y[T[i]]
11:    if V & C ≠ 0 then
12:        report an occurrence at i − m + 1
```

$Y + \sigma$. Then the code of the UW-RAM is essentially the same as the original code, replacing all references to the array Y with memory access operations for the wide word: assuming $m \leq w^2$, reading and writing to $Y[\sigma]$ implemented with by read_word($W, Y + \sigma$) and write_word($W, Y + \sigma$), for some word W. Otherwise bit vectors are represented in $\lceil m/w^2 \rceil$ wide words (and stored in memory in $\lceil m/w^2 \rceil w$ words). The rest of the operations are done on registers and constants are part of the precomputation. The pseudocode for this algorithm is shown in Algorithm 6, which assumes $m \leq w^2$ and is based on the pseudocode for Shift-And given in [27, Ch 2.2.2]. Since we can now update $\boldsymbol{v}$ in $O(m/w^2 + 1)$ time, the running time of Algorithm 6 is $O(nm/w^2 + n)$. Thus, compared to the original algorithm, the UW-RAM algorithm achieves a speedup of w when $m \geq w^2$, and a speedup of $\lceil m/w \rceil$ otherwise (no speedup is achieved for $m \leq w$).

$\boldsymbol{w}$-bit Parallel Automata Another way of using the wide word to speedup the Shift-And algorithm is take advantage of the parallel memory access operations of the UW-RAM to perform w parallel searches on disjoint portions of the text. This is done by using each block of a wide word to represent the automaton in each search: block j is used to search P in $T[jn/w..(j+1)n/w - 1]$, for $0 \leq j \leq w - 1$ (we assume w divides n). Since the operations involved in updating the automata are the same across blocks, an update to all w automata can be done with a constant number of single wide word operations. All bit vectors of the precomputed table Y are now again w-bit long, as in the original algorithm. In each step of the search, w entries of Y are read in parallel to each block according to the current character in T in the search in each portion. The pseudocode for this procedure is shown in Algorithm 7. The code assumes $m \leq w$, though it is straightforward to modify it for the $m > w$ case. The running time of this algorithm is now $O(nm/w^2 + n/w + occ)$, where occ is the number of occurrences found. This is always faster than the first version above.

5.2 Boyer-Moore-Horspool (BMH)

We give one more example of how to use the wide word to speed up string searching by using the BMH [22] algorithm as an example. This algorithm keeps a sliding window of length m over the text T and searches backwards in the window for matching suffixes of both the window and the pattern. More specifically, for a window $T[i..i + m - 1]$, the algorithm checks if $T[i + j - 1] = P[j]$ starting with $j = m$ and decrementing j until either $j = 0$ (there is a match) or a mismatch is found. Either way, the window is then shifted by so $T[i + m - 1]$ is aligned with the last occurrence of this character in P (not counting $P[m]$). The worst case running time of BMH is $O(nm)$ (when

Algorithm 7 Parallel_Shift_And($T, P, n = |T|, m = |P|, \Sigma$). For technical reasons assume $T[n+j] = \$$ for $j = 1, \ldots, m-1$, with $\$ \notin \Sigma$, and that $w \geq \log(n+m)$. In order to report matches at each step in time proportional to the number of matches (and not the number of blocks), we move directly to blocks with matching positions by using a function that for every word of length w returns an array A with the positions of set bits. For example, for $w = 5$ and $x = 01011$, $A = \{1, 3, 4\}$. We do this by table look up to a table with $w/2$-bit entries, whose space is $O(2^{w/2}w)$ words, which for $w = \log n$ is $O(\sqrt{n \log n})$.

```
1:  {Preprocessing}
2:  for each σ ∈ Σ do
3:      Y[σ] ← 0 {|Y[σ]| = w}
4:  for j = 1 to m do
5:      Y[P[j]] ← Y[P[j]] | (1 >> (j − 1))
6:      Y[$] ← 0
7:  V ← 0
8:  ONES ← (2^{w^2} − 1)/(2^w − 1) {ONES_j = 1 for all j}
9:  C ← ONES >> (w − 1) {C_j = 2^{w−1} for all j}
10: {Search}
11: n′ ← n/w
12: POSNS ← 0 {current positions in text}
13: for j = 0 to w do
14:     POSNS ← POSNS | ((jn′ + 1) >> wj)
15: for i = 1 to n′ + m − 1 do
16:     V1 ← (V >> 1) | ONES
17:     V2 ← POSNS
18:     read_content(V2, T) {load characters in each position (V2_j = T[POSNS_j])}
19:     read_content(V2, Y) {lookup masks in array Y (V2_j = Y[T[POSNS_j]])}
20:     V ← V1 & V2
21:     W ← V & C {check for matches at each block}
22:     transpose(W << w − 1)
23:     matches ← W_0 {matches[j] = 1 if there was a match at block j}
24:     write_word(POSNS, matching_positions) {write all current positions in an array matching_positions}
25:     A ← lookup(matches) {position in T of k-th matching block is at matching_positions[A[k]]}
26:     while A[k] ≠ −1 do
27:         report match at matching_positions[A[k]]
28:         k ← k + 1
29:     V ← V & ∼C {clear most significant bit in each block}
30:     POSNS ← POSNS + ONES {update positions in T (POSNS_j ≤ n + m − 1 for all j, thus there is no carry
        across blocks)}
```

the entire window is checked for all window positions) but on average the window can be shifted by more than one character, making the running time $O(n)$ [3]. We can take advantage of the wide word to make several character comparisons in parallel, achieving a w speedup over the worst case behaviour of the standard algorithm.

First, we divide each wide word in f-bit fields so that each field contains one, thus $f = \lceil \log \sigma \rceil$. At each position of the window, we do a field-wise comparison between a wide word containing the characters of the text and one containing the characters of the pattern. We do this simply by subtracting both words. Since we only care if all symbols in the words match, we only need to check if the result is zero, without having to worry about carries crossing fields (and hence we do not need a test bit). We shift the window to the next position if the result is not zero. Note that this check can be done in constant time and it is quite simple as we do not need to identify where there was a mismatch. Thus in each window we can compare up to w^2/f symbols in parallel, and

Algorithm 8 BMH($T, P, n = |T|, m = |P|, \Sigma$). For simplicity we assume that w divides $m \log \sigma$. We assume also that T and P are represented with $\log \sigma$ bits per symbol. We still use $T[i]$ to denote one character, which can be easily obtained from the packed representation in constant time (the same applies to the actual address of starting characters of substrings).

```
1: {Preprocessing}
2: for each σ ∈ Σ do
3:     jump[σ] ← m
4: for j = 1 to m − 1 do
5:     jump[P[j]] ← m − j
6: m′ ← w²/ log σ {characters per wide word}
7: {Search}
8: i = 0
9: while i ≤ n − m do
10:     k ← m′/m {number of window segment}
11:     while k > 0 do
12:         W ← T[i + (k − 1)m′ + 1..i + km′] {W contains the substring of T of k-th window segment}
13:         V ← P[(k − 1)m′ + 1..km′] {V contains the substring of P of k-th window segment}
14:         if W − V ≠ 0 then
15:             break
16:         else if k = 1 then
17:             report occurrence at i + 1
18:         k ← k − 1
19:     i ← i + jump[T[i + m]]
```

hence the running time in the worst case becomes $O(mn \log \sigma / w^2)$. We show the pseudocode in Algorithm 8 which, again, is based on the pseudocode of this algorithm presented in [27, Ch. 2.3.2]. Note that for a given input the distance of the shifts is exactly the same as in the original version of the algorithm, and therefore the expected running time remains the same. Note as well that the expected running time can be reduced by using each block to search in disjoint parts of the text at the expense of increasing the worst case time to $O(mn \log \sigma / w)$ due to the reduction in the number of characters that can be compared simultaneously.

6 Concluding Remarks

We have introduced the Ultra-Wide Word architecture and model, and showed that several classes of algorithms can be readily implemented in this model to achieve a speedup over traditional word-RAM algorithms. The examples described in this paper already show the potential of this model to enable parallel implementations of existing algorithms with speedups comparable to that of multi-core computations. We believe that this architecture could serve as well to simplify many existing word-RAM algorithms, that in practice do not perform well due to large constant factors. Finally, we conjecture that this model will lead to new efficient algorithms and data structures that can sidestep existing lower bounds.

References

1. Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.
2. Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, October 1992.

3. Ricardo A. Baeza-Yates and Mireille Régnier. Average running time of the boyer-moore-horspool algorithm. *Theoretical Computer Science*, 92(1):19 – 31, 1992.
4. Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.
5. Andrej Brodnik. *Searching in Constant Time and Minimum Space.* PhD thesis, Unviersity of Waterloo, 1995. Also available as Technical Report CS-95-41.
6. Andrej Brodnik, Svante Carlsson, Michael L. Fredman, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. *Journal of Systems and Software*, 78(3):249 – 256, 2005.
7. Andrej Brodnik, Johan Karlsson, J. Ian Munro, and Andreas Nilsson. An O(1) solution to the prefix sum problem on a specialized memory architecture. In Gonzalo Navarro, Leopoldo E. Bertossi, and Yoshiharu Kohayakawa, editors, *IFIP TCS*, volume 209 of *IFIP*, pages 103–114. Springer, 2006.
8. Timothy M. Chan. Point location in o(log n) time, voronoi diagrams in o(n log n) time, and other transdichotomous results in computational geometry. In *FOCS*, pages 333–344. IEEE Computer Society, 2006.
9. Timothy M. Chan and Mihai Patrascu. Transdichotomous results in computational geometry, i: Point location in sublogarithmic time. *SIAM J. Comput.*, 39(2):703–729, 2009.
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.
11. George B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):pp. 266–277, 1957.
12. Joseph A. Fisher. Very long instruction word architectures and the eli-512. *SIGARCH Comput. Archit. News*, 11:140–150, June 1983.
13. M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM.
14. Michael L. Fredman. The complexity of maintaining an array and computing its partial sums. *J. ACM*, 29(1):250–260, January 1982.
15. M.L. Fredman and D.E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.
16. R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.
17. Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, New York, NY, USA, 1997.
18. Torben Hagerup. Sorting and searching on the word ram. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *STACS 98*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0028575.
19. Haripriyan Hampapuram and Michael L. Fredman. Optimal biweighted binary trees and the complexity of maintaining partial sums. *SIAM J. Comput.*, 28(1):1–9, 1998.
20. Yijie Han. Deterministic sorting in o(nlog logn) time and linear space. *J. Algorithms*, 50:96–105, January 2004.
21. Yijie Han and Mikkel Thorup. Integer sorting in 0(n sqrt (log log n)) expected time and linear space. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 135–144, Washington, DC, USA, 2002. IEEE Computer Society.
22. R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.
23. G. Jacobson. Space-efficient static trees and graphs. *Foundations of Computer Science, IEEE Annual Symposium on*, pages 549–554, 1989.
24. Joseph JáJá. *An introduction to parallel algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.
25. William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.
26. J. Ian Munro. Tables. In Vijay Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 37–42. Springer, 1996.
27. G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.
28. Ulrich Pferschy. Dynamic programming revisited: Improving knapsack algorithms. *Computing*, 63(4):419–430, 1999.
29. David Pisinger. Dynamic programming on the word ram. *Algorithmica*, 35:128–145, 2003. 10.1007/s00453-002-0989-y.
30. Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1):63–72, 1978.

31. P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976. 10.1007/BF01683268.
32. Peter van Emde Boas. Machine models and simulation. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 1–66. 1990.
33. Arlazarov V.L., Dinic E.A., Kronrod M.A., and Faradzev I.A. On economic construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk SSSR*, 194:487–488, 1970. (In Russian). English translation in Soviet Math. Dokl., 11,1209-1210, 1975.
34. Sun Wu and Udi Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, October 1992.