Bit-(Parallelism)^2: Getting to the Next Level of Parallelism

Domenico Cantone, Simone Faro, and Emanuele Giaquinta

Dipartimento di Matematica e Informatica, Università di Catania, Italy
{cantone, faro, giaquinta}@dmi.unict.it

Abstract. We investigate the problem of getting to a higher instruction-level parallelism in string matching algorithms. In particular, starting from an algorithm based on bit-parallelism, we propose two flexible approaches for boosting it with a higher level of parallelism. These approaches are general enough to be applied to other bit-parallel algorithms. It turns out that higher levels of parallelism lead to more efficient solutions in practical cases, as demonstrated by an extensive experimentation.

Keywords: string matching, bit parallelism, text processing, design and analysis of algorithms, instruction-level parallelism.

1 Introduction

Given a text t of length n and a pattern p of length m over some alphabet Σ of size σ, the *string matching problem* consists in finding *all* occurrences of the pattern p in the text t. This problem has been extensively studied in computer science because of its direct applications to such diverse areas as text, image and signal processing, speech analysis and recognition, information retrieval, computational biology and chemistry. String matching algorithms are also basic components in many software applications. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science. Finally, they also play an important rôle in theoretical computer science by providing challenging problems.

In this paper we focus on one of such engaging problems, namely the problem of enhancing the instruction-level parallelism of string matching algorithms.

The *instruction-level parallelism* (ILP) is a measure of how many operations in an algorithm can be performed simultaneously. Ordinary programs are typically written under a sequential execution model, in which instructions are executed one after the other and in the order specified by the programmer. ILP allows one to overlap the execution of multiple instructions or even to change the order in which instructions are executed. The extent to which ILP is present in programs heavely depends on the application. In certain fields, such as graphics and scientific computing, ILP is largely used. However, workloads such as cryptography exhibit much less parallelism.

```
sequence a
a1.  $p[j] \leftarrow p[j] + 2$
a2.  $h \leftarrow p[i] \times 3$
a3.  $p[i] \leftarrow p[j] + h$
```

```
sequence b
b1.  $a \leftarrow b \leftarrow 0$
b2.  for $i = 1$ to $n$ do
b3.      $a \leftarrow a + (q[i] \times p[i])$
b4.      $b \leftarrow b + (3 \times p[i])$
```

```
sequence c
c1.  $j \leftarrow 0$
c2.  for $i = 1$ to $n$ do
c3.      $j \leftarrow j + (p[i] \times (3000 + q[i]))$
c4.  $a \leftarrow j \mod 1000$
c5.  $b \leftarrow \lfloor j/1000 \rfloor$
```

Fig. 1. Three sequences of instructions.

Consider, for instance, the sequences of instructions shown in Fig. 1. In sequence a, operation a_3 depends on the results of operations a_1 and a_2, and thus it cannot be calculated until both operations are completed. However, operations a_1 and a_2 can be calculated simultaneously as they are independent of each other. If we assume that each operation can be completed in one time unit, then these three instructions can be completed in two time units, yielding an ILP of 3/2.

Two main techniques can be adopted to increase the ILP of a sequence of instructions: micro-architectural and software techniques.

Micro-architectural techniques used to exploit ILP include, among others, *instruction pipelining*, where the execution of multiple instructions can be partially overlapped, and *superscalar execution*, in which multiple execution units are used to execute multiple instructions in parallel. For instance, in sequence b, operation b_3 and b_4 are independent, so (if two different processors are available) they can be calculated in parallel for each iteration of the cycle in b_2, thus halving the time needed for their execution.

Instead, software techniques are generally more challenging as they strongly depend on the processed data. Consider again sequence b shown in Fig. 1. Assuming that the sum $\sum_{i=1}^{n} p[i]q[i]$ is less than 1000, a smart programmer could modify the sequence in the form proposed in sequence c, achieving again an ILP of 2 while using a single processor.

Several string matching algorithms have been proposed to take advantage of micro-architectural techniques for increasing ILP (see for instance [6, 7, 10, 5]). However, most of the work has been devoted to develop software techniques for ILP to simulate efficiently the parallel computation of nondeterministic finite automata (NFAs) related to the search pattern, whose number of states is about the pattern size (see for instance [2, 8, 9, 3]). Such simulations can be done efficiently using the *bit-parallelism* technique, which consists in exploiting the intrinsic parallelism of the bit operations inside a computer word [2]. In some cases, bit-parallelism allows to reduce the overall number of operations up to a factor equal to the number of bits in a computer word. Thus, although string matching algorithms based on bit-parallelism are usually simple and have very low memory requirements, they generally work well with patterns of moderate length only.

When the pattern size is small enough, in favorable situations it becomes possible to carry on in parallel the simulation of multiple copies of a same NFA or of multiple distinct NFAs, thus getting to a second level of parallelism.

In this paper we illustrate this idea in the case of BNDM-like algorithms. More specifically, not satisfied with the degree of parallelism of the bit-parallel implementation of a variant of the Wide-Window algorithm, we present two different approaches which yield a better ILP if compared with the original algorithm. The methods we present turn out to be quite flexible and, as such, can be applied to other bit-parallel algorithms as well.[1]

The rest of the paper is organized as follows. In Section 2 we introduce the notations and terminology used in the paper. In Section 3, we introduce the bit-parallel technique and describe some string matching algorithms based on it. Next, in Section 4, we present two techniques to enhance ILP and illustrate two new algorithms resulting from their application. Experimental data obtained by running the algorithms under various conditions are presented and compared in Section 5. Finally, we draw our conclusions in Section 6.

2 Notations and Terminology

Throughout the paper we will make use of the following notations and terminology. A string p of length $m \geq 0$ is represented as a finite array $p[0\,..\,m-1]$ of characters from a finite alphabet Σ of size σ (in particular, for $m = 0$ we obtain the empty string, also denoted by ε). Thus $p[i]$ will denote the $(i+1)$-st character of p, for $0 \leq i < m$, and $p[i\,..\,j]$ will denote the *factor* or *substring* of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p, for $0 \leq i \leq j < m$. A factor of the form $p[0\,..\,i]$, also written p_i, is called a *prefix* of p and a factor of the form $p[i\,..\,m-1]$ is called a *suffix* of p for $0 \leq i \leq m-1$. We write $\mathit{Suff}(p)$ for the collection of all suffixes of p. In addition, we write $p.p'$ or, more simply, pp' to denote the concatenation of the strings p and p'. Finally, we denote the reverse of the string p by $\bar{p}$, i.e. $\bar{p} = p[m-1]p[m-2]\ldots p[0]$.

The *nondeterministic suffix automaton* with ε-transition $\mathit{NSA}(p) = (Q, \Sigma, \delta, I, F)$ for the language $\mathit{Suff}(P)$ of the suffixes of p is defined as follows:

- $Q = \{I, q_0, q_1, \ldots, q_m\}$ (I is the initial state)
- $F = \{q_m\}$ (F is the set of final states)
- the transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow \mathit{Pow}(Q)$ is such that $\delta(q, c) = \{q_{i+1}\}$, if $q = q_i$ and $c = p[i]$ $(0 \leq i < m)$; $\delta(q, c) = \{q_0, q_1, \ldots, q_m\}$, if $q = I$ and $c = \varepsilon$; and $\delta(q, c) = \emptyset$, otherwise.

We will also make use of the following C-like notations to represent some bitwise operations. In particular, "|" represents the bitwise Or; "&" denotes the bitwise And; "$\sim$" represents the one's complement; "$\gg$" and "$\ll$" denote respectively the bitwise right shift and the bitwise left shift.

[1] A similar technique has been exploited in [4] to boost the approximate search of a pattern, under the edit distance; the algorithm proposed in [4] performs a left to right scan of the text while processing $\lfloor w/m \rfloor$ text segments simultaneously (w is the word size in bits). We thank an anonymous referee for pointing this out to us.

3 Bit-Parallelism: Starting from the First Level

Bit-parallelism exploits the intrinsic parallelism of bit operations inside computer words, allowing in favorable cases to cut down the overall number of operations up to the number of bits in a computer word.

Among the several algorithms based on bit-parallelism which have been developed over the years, the Backward-Nondeterministic-DAWG-Matching algorithm (BNDM, for short) deserves particular attention as it has inspired various other algorithms and is still considered one of the fastest algorithms based on bit-parallelism [8].

The BNDM algorithm uses bit-parallelism to simulate the nondeterministic suffix automaton $NSA(\bar{p})$ for the reverse of the search pattern p.

To this purpose, the states of the automaton (except states I and q_0) are put in a one-one correspondence with the bits of a bit mask D, having size equal to the length L of the pattern p.[2] In this context, in any automaton configuration bits corresponding to active states are set to 1, while bits corresponding to inactive states are set to 0. Additionally, for each character c of the alphabet Σ, the algorithm maintains a bit mask $\mathsf{B}[c]$ whose i-th bit is set to 1 iff $p[i] = c$. Thus, the array B requires $|\Sigma| \cdot L$ bits.

The BNDM algorithm works by shifting a window of length m over the text. For each window alignment, it searches the pattern by scanning the current window backwards, updating the automaton configuration for each character read. Whenever bit $(m-1)$ of D is set, a suffix of $\bar{p}$ (i.e., a prefix of p) has been found and the current position is recorded. A search ends when either D becomes zero or the algorithm has performed m iterations. The window is then shifted to the start position of the longest recognized proper suffix of $\bar{p}$.

Because of the ϵ-closure of the initial state I, at the beginning of any search all states are active, i.e. $\mathsf{D} = 1^m$. A transition on character c is implemented (with the exception of the first one[3]) as follows:

$$\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \ \& \ \mathsf{B}[c] \, .$$

The BNDM algorithm scales in function of the number $\lceil m/\omega \rceil$ of words needed to represent each of D and $\mathsf{B}[c]$, for $c \in \Sigma$, where ω is the size of the computer word in bits. Its worst case time complexity is $\mathcal{O}(nm\lceil m/\omega \rceil)$, though in practice exhibits a sublinear behavior.

Several variants of the BNDM algorithm have been proposed over the years. In what follows we briefly describe an efficient variant, the Wide-Window algorithm (WW, for short) [3], which is particularly suitable for our purposes. However, we slightly depart from its original version so as to make the algorithm parallelizable in ways that will be explained in the next section.

[2] Note that if $L \leq \omega$ the entire bit mask D fits in a single computer word, otherwise $\lceil L/\omega \rceil$ computer words are needed to represent it.

[3] The first transition is simply encoded as $\mathsf{D} \leftarrow \mathsf{D} \ \& \ \mathsf{B}[c]$.

3.1 The Wide-Window Algorithm

Let p be a pattern of length m and let t be a text of length n. The WW algorithm locates $\lfloor n/m \rfloor$ *attempt positions* in t, namely positions $j = km - 1$, for $k = 1, \ldots, \lfloor n/m \rfloor$. For each such position j, the pattern p is searched for in the *attempt window* of size $2m - 1$ centered at j, i.e. in the substring $t[j - m + 1 \mathinner{..} j + m - 1]$. Each of such search phases is divided into two steps.

In the first step, the right side of the attempt window, consisting of the last m characters, is scanned from left to right with the automaton $\mathit{NSA}(p)$. In this step, the start positions (in p) of the suffixes of p aligned with position j in t are collected in a set

$$\mathcal{S}_j = \{0 \leq i < m \mid p[i \mathinner{..} m - 1] = t[j \mathinner{..} j + m - 1 - i]\}\,.$$

In the second step, the left half of the attempt window, consisting of the first m symbols, is scanned from right to left with the automaton $\mathit{NSA}(\bar{p})$. During this step, the end positions (in p) of the prefixes of p aligned with position j in t are collected in a set

$$\mathcal{P}_j = \{0 \leq i < m \mid p[0 \mathinner{..} i] = t[j - i \mathinner{..} j]\}\,.$$

Taking advantage of the fact that an occurrence of p is located at position $(j - k)$ of t if and only if $k \in \mathcal{S}_j \cap \mathcal{P}_j$, for $k = 0, \ldots, m - 1$, the number of all the occurrences of p in the attempt window centered at j is readily given by the cardinality $|\mathcal{S}_j \cap \mathcal{P}_j|$.

Fig. 2(A) shows a simple schematization of the structure of an iteration of the WW algorithm at a given position j in t. The two sequential phases are represented by the arrows labeled 1 and 2, respectively.

It is straightforward to devise a bit-parallel implementation of the WW algorithm. The sets $\mathcal{P}$ and $\mathcal{S}$ can be encoded by two bit masks P and S, respectively. The nondeterministic automata $\mathit{NSA}(p)$ and $\mathit{NSA}(\bar{p})$ are then used for searching the suffixes and prefixes of p on the right and on the left parts of the window, respectively. Both automata state configurations and final state configuration can be encoded by the bit masks D and $\mathsf{M} = (1 \ll (m-1))$, so that $(\mathsf{D} \mathbin{\&} \mathsf{M}) \neq 0$ will mean that a suffix or a prefix of the search pattern p has been found, depending on whether D is encoding a state configuration of the automaton $\mathit{NSA}(p)$ or of the automaton $\mathit{NSA}(\bar{p})$. Whenever a suffix (resp., a prefix) of length $(\ell + 1)$ is found (with $\ell = 0, 1, \ldots, m - 1$), the bit $\mathsf{S}[m - 1 - \ell]$ (resp., the bit $\mathsf{P}[\ell]$) is set by one of the following bitwise operations:

$$\begin{array}{ll} \mathsf{S} \leftarrow \mathsf{S} \mid ((\mathsf{D}\&\mathsf{M}) \gg \ell) & \text{(in the suffix case)} \\ \mathsf{P} \leftarrow \mathsf{P} \mid ((\mathsf{D}\&\mathsf{M}) \gg (m - 1 - \ell)) & \text{(in the prefix case)}\,. \end{array}$$

If we are only interested in counting the number of occurrences of p in t, we can just count the number of bits set in $(\mathsf{S} \mathbin{\&} \mathsf{P})$. This can be done in $\log_2(\omega)$ operations by using a *population count* function, where ω is the size of the computer word in bits (see [1]). Otherwise, if we want also to retrieve the matching positions of p in t, we can iterate over the bits set in $(\mathsf{S} \mathbin{\&} \mathsf{P})$ by repeatedly computing the index of the highest bit set and then masking it. The function that computes the highest bit set of a register x is $\lfloor \log_2(x) \rfloor$, and can be implemented efficiently in either a machine dependent or machine independent way (see again [1]).

The resulting algorithm based on bit parallelism is named Bit-Parallel Wide-Window algorithm ($\mathsf{B_pW_w}$, for short). It needs $\lceil m/\omega \rceil$ words to represent the bit masks D, S, P, and $\mathsf{B}[c]$, for $c \in \Sigma$. The worst case time complexity of the $\mathsf{B_pW_w}$ algorithm is $\mathcal{O}(n\lceil m/\omega \rceil + \lfloor n/m \rfloor \log_2(\omega))$.

Additionally, we observe that the $\mathsf{B_pW_w}$ algorithm can be easily modified so as to work on windows of size $2m$. For the sake of clarity, we have just discussed a simpler but slightly less efficient variant.

4 Bit-(Parallelism)^2: Getting to the Second Level

In this section we present two different approaches which lead to a higher level of parallelism. By way of demonstration we apply them to a bit-parallel version of the Wide-Window algorithm,[4] but our approaches can be applied as well to other (more efficient) solutions based on bit-parallelism.

The two approaches can be summarized as follows:

- **first approach:** if the algorithm searches for the pattern in fixed-size text windows then, at each attempt, process simultaneously two (adjacent or partially overlapping) text windows by using in parallel two copies of a same automaton;
- **second approach:** if each search attempt of the algorithm can be divided into two steps (which possibly make use of two different automata) then execute simultaneously the two steps, by running the two automata in parallel.

Both variants use the SIMD (Single Instruction Multiple Data) paradigm. This approach, on which vectorial instructions sets like MMX and SSE are based, consists in executing the same instructions on multiple data in a parallel way. Tipically, a register of size ω is logically divided into i blocks of k bits which are then updated simultaneously.

In both variants of the $\mathsf{B_pW_w}$ algorithm, we divide a word of ω bits into *two* blocks, each being used to encode a suffix automaton. Thus, the maximum length of the pattern gets restricted to $\lfloor \omega/2 \rfloor$. We denote with B the array of bit masks encoding the suffix automaton $\mathit{NSA}(p)$ and with C the array of bit masks encoding the suffix automaton $\mathit{NSA}(\bar{p})$.

4.1 The Bit-Parallel (Wide-Window)^2 Algorithm

In the first variant, named Bit-Parallel (Wide-Window)^2 ($\mathsf{B_pW_w^2}$, for short), two partially overlapping windows in t, each of size $2m-1$, centered at consecutive

[4] We chose the Wide-Window algorithm in our case study since its structure makes its parallelization simpler.

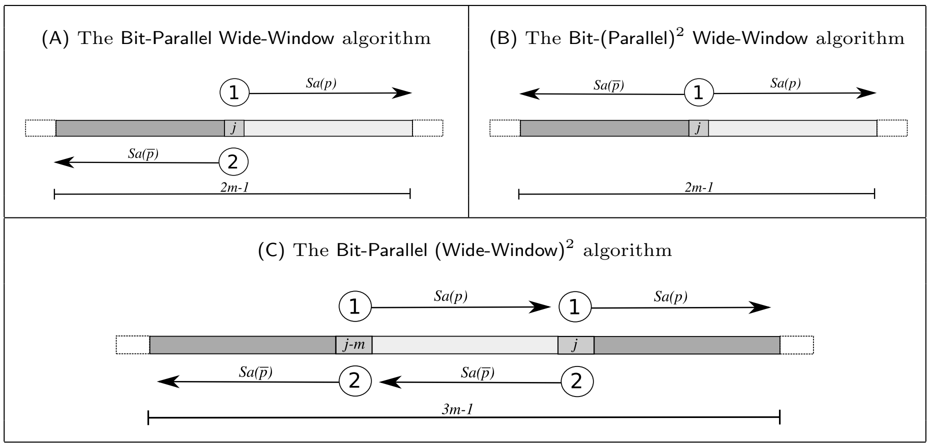
(A) The Bit-Parallel Wide-Window algorithm
1
Sa(p)
j
Sa($\bar{p}$)
2
2m-1
(B) The Bit-(Parallel)^2 Wide-Window algorithm
Sa($\bar{p}$)
1
Sa(p)
j
2m-1
(C) The Bit-Parallel (Wide-Window)^2 algorithm
1
Sa(p)
1
Sa(p)
j-m
j
Sa($\bar{p}$)
2
Sa($\bar{p}$)
2
3m-1

Fig. 2. Structure of a searching iteration at a given position j in the text t of (A) the $\mathsf{B_pW_w}$ algorithm, (B) the $\mathsf{B_p^2W_w}$ algorithm, and (C) the $\mathsf{B_pW_w^2}$ algorithm.

attempt positions $j - m$ and j, are processed simultaneously. For the parallel simulation two automata are represented in a single word and updated in parallel.

Specifically, each search phase is again divided into two steps. During the first step, two copies of $NSA(p)$ are operated in parallel to compute simultaneously the sets $\mathcal{S}_{j-m}$ and $\mathcal{S}_j$ (lines 13-18). Likewise, in the second step, two copies of $NSA(\bar{p})$ are operated in parallel to compute the sets $\mathcal{P}_{j-m}$ and $\mathcal{P}_j$ (lines 20-25). To represent the automata with a single word, the bit masks D, M, S, and P are logically divided into two blocks, each of $k = \omega/2$ bits.

During the first step, the most significant k bits of D encode the state of the suffix automaton $NSA(p)$ that scan the attempt window centered at $j - m$. Similarly, the least significant k bits of D encode the state of the suffix automaton $NSA(p)$ that scans the attempt window centered at j. An analogous encoding is used in the second step, but with the automaton $NSA(\bar{p})$ in place of $NSA(p)$. Fig. 2(C) schematizes the structure of a search iteration of the $\mathsf{B_pW_w^2}$ algorithm, at given attempt positions $j - m$ and j of t.

The most significant k bits of the bit mask S (resp., P) encode the set $\mathcal{S}_{j-m}$ (resp., $\mathcal{P}_{j-m}$), while the least significant k bits encode the set $\mathcal{S}_j$ (resp., $\mathcal{P}_j$). Thus, to properly detect suffixes in both windows, the bit mask M is initialized (lines 8-9) with the value

$$\mathsf{M} \leftarrow (1 \ll (m + k - 1)) \mid (1 \ll (m - 1))$$

and transitions of the automata are performed in parallel with the following bitwise operations (lines 14-15 and lines 21-22)

$$\begin{aligned} &\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \;\&\; ((\mathsf{B}[t[j - m + \ell]] \ll k) \mid \mathsf{B}[t[j + \ell]]) \qquad \text{(in the first phase)} \\ &\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \;\&\; ((\mathsf{C}[t[j - m - \ell]] \ll k) \mid \mathsf{C}[t[j - \ell]]) \qquad \text{(in the second phase)}, \end{aligned}$$

for $\ell = 1, \ldots, m - 1$ (when $\ell = 0$, the left shift of D does not take place).

The remaining bitwise operations are left unchanged, as the automata configurations are updated using the same instructions. Since two windows are simultaneously scanned at each search iteration, the shift becomes $2m$, therefore doubling the length of the shift with respect to the WW algorithm. The pseudocode of the algorithm $\mathsf{B_pW_w^2}$ is reported in Fig. 3 (on the left).

4.2 The Bit-(Parallel)^2 Wide-Window Algorithm

The second variant of the WW algorithm which we present next is called Bit-(Parallel)^2 Wide-Window algorithm ($\mathsf{B_p^2W_w}$, for short). The idea behind it consists in processing a single window at each attempt (as in the original WW algorithm) but this time by scanning its left and right sides simultaneously. Fig. 2(B) schematizes the structure of a searching iteration of the $\mathsf{B_p^2W_w}$ algorithm, while Fig. 3 (on the right) shows the pseudocode of the $\mathsf{B_p^2W_w}$ algorithm.

As above, let p be a pattern of length m, and t be a text of length n. The bit masks B and C which are used to perform the transitions on both automata $NSA(p)$ and $NSA(\bar{p})$ are computed as in the $\mathsf{B_pW_w}$ algorithm (lines 3-7).

Automata state configurations are again encoded simultaneously in a same bit mask D. Specifically, the most significant k bits of D encode the state of the suffix automaton $NSA(p)$, while the least significant k bits of D encode the state of the suffix automaton $NSA(\bar{p})$. The $\mathsf{B_p^2W_w}$ algorithm uses the following bitwise operations to perform transitions[5] of both automata in parallel (lines 14-15,17):

$$\mathsf{D} \leftarrow (\mathsf{D} \ll 1) \,\&\, ((\mathsf{B}[t[j+\ell]] \ll k) \mid \mathsf{C}[t[j-\ell]]) \,,$$

for $\ell = 1, \ldots, m-1$. Note that in this case the $\mathsf{left\ shift}$ of k positions can be precomputed in B by setting $\mathsf{B}[c] \leftarrow \mathsf{B}[c] \ll k$, for each $c \in \Sigma$.

Using the same representation, the final-states bit mask M is initialized as

$$\mathsf{M} \leftarrow (1 \ll (m+k-1)) \mid (1 \ll (m-1)) \qquad \text{(lines 8-9)}\,.$$

At each iteration around an attempt position j of t, the sets $\mathcal{S}_j$ and $\mathcal{P}_j^*$ are computed, where $\mathcal{S}_j$ is defined as in the case of the $\mathsf{B_pW_w}$ algorithm, and $\mathcal{P}_j^*$ is defined as $\mathcal{P}_j^* = \{0 \le i < m \mid p[0\,..\,m-1-i] = t[j-(m-1-i)\,..\,j]\}$, so that $\mathcal{P}_j = \{0 \le i < m \mid (m-1-i) \in \mathcal{P}_j^*\}$.

The sets $\mathcal{S}_j$ and $\mathcal{P}_j^*$ can be encoded with a single bit mask PS, in the rightmost and the leftmost k bits, respectively. Positions in $\mathcal{S}_j$ and $\mathcal{P}_j^*$ are then updated simultaneusly in PS by executing the following operation (line 16):

$$\mathsf{PS} \leftarrow \mathsf{PS} \mid ((\mathsf{D} \,\&\, \mathsf{M}) \gg l)\,.$$

At the end of each iteration, the bit masks S and P are retrieved from PS with the following bitwise operations (lines 19-20):

$$\mathsf{P} \leftarrow \mathsf{reverse}(\mathsf{PS}) \gg (\omega - m)\,, \qquad \mathsf{S} \leftarrow \mathsf{PS} \gg k\,,$$

where $\mathsf{reverse}$ denotes the bit-reversal function, which satisfies $\mathsf{reverse}(x)[i] = x[\omega-1-i]$, for $i = 0, \ldots, \omega-1$ and any bit mask x. In fact, to obtain the correct

[5] For $\ell = 0$, D is simply updated by $\mathsf{D} \leftarrow \mathsf{D} \,\&\, ((\mathsf{B}[t[j+l]] \ll k) \mid \mathsf{C}[t[j-l]])$.

```
Bit-Parallel (Wide-Window)$^2$ $(p, m, t, n)$
count $\leftarrow 0$
$k \leftarrow \omega/2$
for $c \in \Sigma$ do B$[c] \leftarrow 0$
for $c \in \Sigma$ do C$[c] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    B$[p[i]] \leftarrow$ B$[p[i]] | (1 \ll i)$
    C$[p[m-1-i]] \leftarrow$ C$[p[m-1-i]] | (1 \ll i)$
H $\leftarrow 1 \ll (m-1)$
M $\leftarrow$ (H $\ll k$) | H
$j \leftarrow 2m-1$
while $j < n-m$ do
    D $\leftarrow \sim 0, l \leftarrow 0,$ S $\leftarrow 0$
    while D $\neq 0$ do
        H $\leftarrow$ (B$[t[j-m+l]] \ll k$)|B$[t[j+l]]$
        D $\leftarrow$ D & H
        S $\leftarrow$ S | ((D & M) $\gg l$)
        D $\leftarrow$ D $\ll 1$
        $l \leftarrow l+1$
    D $\leftarrow \sim 0, l \leftarrow 0,$ P $\leftarrow 0$
    while $D \neq 0$ do
        H $\leftarrow$ (C$[t[j-m-l]] \ll k$) | C$[t[j-l]]$
        D $\leftarrow$ D & H
        P $\leftarrow$ P | ((D&M) $\gg (m-1-l)$)
        D $\leftarrow$ D $\ll 1$
        $l \leftarrow l+1$
    count $\leftarrow$ count + popcount(P&S)
    $j \leftarrow j+2m$
```

```
Bit-(Parallel)$^2$ Wide-Window $(p, m, t, n)$
count $\leftarrow 0$
$k \leftarrow \omega/2$
for $c \in \Sigma$ do B$[c] \leftarrow 0$
for $c \in \Sigma$ do C$[c] \leftarrow 0$
for $i \leftarrow 0$ to $m-1$ do
    B$[p[i]] \leftarrow$ B$[p[i]]$ | $(1 \ll (k+i))$
    C$[p[m-1-i]] \leftarrow$ C$[p[m-1-i]] | (1 \ll i)$
H $\leftarrow 1 \ll (m-1)$
M $\leftarrow (H \ll k)$ | H
$j \leftarrow m-1$
while $j < n-m$ do
    D $\leftarrow \sim 0, l \leftarrow 0,$ PS $\leftarrow 0$
    while D $\neq 0$ do
        H $\leftarrow$ C$[t[j-l]]$ | B$[t[j+l]]$
        D $\leftarrow$ D & H
        PS $\leftarrow$ PS | ((D & M) $\gg l$)
        D $\leftarrow$ D $\ll 1$
        $l \leftarrow l+1$
    P $\leftarrow$ reverse(PS) $\gg (\omega - m)$
    S $\leftarrow$ PS $\gg k$
    count $\leftarrow$ count + popcount(P&S)
    $j \leftarrow j+m$
```

Fig. 3. The Bit-Parallel (Wide-Window)^2 algorithm (on the left) and the Bit-(Parallel)^2 Wide-Window algorithm (on the right) for the exact string matching problem.

value of P we used bit-reversal modulo m, which has been easily achieved by right shifting reverse(PS) by $(\omega - m)$ positions. We recall that the reverse function can be implemented efficiently with $\mathcal{O}(\log_2(\omega))$ operations (see [1]).

5 Experimental Results

We present next the results of an extensive experimental comparison of our proposed variants $B_p^2 W_w$ and $B_p W_w^2$ with the $B_p W_w$ and BNDM algorithms. In particular, we have tested two different implementations of the $B_p^2 W_w$ and $B_p W_w^2$ algorithms, characterized by a different implementation of the *population-count* function. One implementation uses the builtin version of the GNU C compiler (algorithms $B_p^2 W_w$ and $B_p W_w^2$), while the second implementation uses the population-count function described in [1] (algorithms $B_p^2 W_w^{\underline{bc}}$ and $B_p W_w^{2\,\underline{bc}}$). Thus, we compared the following string matching algorithms, in terms of running time:

- the Bit-Parallel Wide-Window algorithm ($B_p W_w$)
- the Bit-(Parallel)^2 Wide-Window algorithm ($B_p^2 W_w$)
- the Bit-(Parallel)^2 Wide-Window algorithm with bit-count ($B_p^2 W_w^{\underline{bc}}$)
- the Bit-Parallel (Wide-Window)^2 algorithm ($B_p W_w^2$)
- the Bit-Parallel (Wide-Window)^2 algorithm with bit-count ($B_p W_w^{2\,\underline{bc}}$)
- the Backward-Nondeterministic-DAWG-Matching algorithm (BNDM).

All algorithms have been implemented in the C programming language and tested on a PC with Intel Core2 processor of 1.66GHz running Linux and with a 32 bit word. In particular, all algorithms have been tested on seven Randσ problems, for $\sigma = 2, 4, 8, 16, 32, 64, 128$, where a Rand$\sigma$ problem consists of searching a set of 400 random patterns of a given length in a 5Mb random text over a common alphabet of size σ, with a uniform character distribution.

Only short patterns of length $m = 2, 4, 6, 8, 10, 12, 14, 16$ have been considered in our tests, since the bit size of a word was 32 in our case. However, the same approach could be applied with 64-bit processors or using Intel Processors with SSE instructions on 128 bit registers, to process patterns up to a length of 32 and of 64, respectively. Moreover, we observe that $\lceil 2m/\omega \rceil$ different words could be used for representing our suffix automata in case of longer patterns, overcoming the bound on the value of m, though at the price of an increased running time.

In the following tables, running times are expressed in hundredths of seconds. The best results among all bit-parallel WW variants have been boldfaced and underlined. Additionally, running times relative to the BNDM algorithm have been boldfaced and underlined when the BNDM algorithm outperforms the other algorithms.

m	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	816.50	634.00	521.25	442.50	381.50	334.25	300.75	271.00
$\mathsf{B^2_pW_w}$	775.00	481.50	**332.25**	**255.50**	**211.75**	**183.00**	**162.25**	**146.00**
$\mathsf{B^2_pW^{\underline{bc}}_w}$	**642.25**	**455.50**	353.50	287.50	243.00	210.75	186.50	167.25
$\mathsf{B_pW^2_w}$	905.75	700.75	554.50	455.25	383.00	331.50	291.75	268.50
$\mathsf{B_pW^{2\,\underline{bc}}_w}$	659.75	566.75	478.50	404.50	349.25	306.75	274.50	256.50
BNDM	690.75	545.75	410.50	308.00	244.50	201.50	172.00	150.00

Results for a Rand2 problem

m	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	615.75	381.25	278.00	221.50	185.50	159.25	140.25	125.50
$\mathsf{B^2_pW_w}$	527.00	**311.75**	**232.75**	**186.75**	**155.75**	**134.00**	**118.25**	**106.25**
$\mathsf{B^2_pW^{\underline{bc}}_w}$	**503.50**	342.75	260.00	207.50	172.75	148.25	130.50	116.50
$\mathsf{B_pW^2_w}$	645.00	386.25	279.25	219.50	179.75	152.25	132.75	119.25
$\mathsf{B_pW^{2\,\underline{bc}}_w}$	557.75	375.50	276.25	217.25	180.00	154.00	135.50	122.25
BNDM	555.50	312.00	**215.75**	**166.25**	**137.00**	**117.00**	**102.00**	**90.50**

Results for a Rand4 problem

m	2	4	6	8	10	12	14	16
$\mathsf{B_pW_w}$	422.75	273.50	199.75	153.75	123.75	103.50	89.25	79.00
$\mathsf{B^2_pW_w}$	**375.75**	**234.50**	173.25	137.50	115.25	100.50	89.50	80.75
$\mathsf{B^2_pW^{\underline{bc}}_w}$	378.25	252.25	186.00	147.50	123.25	107.25	95.25	86.00
$\mathsf{B_pW^2_w}$	407.75	239.00	**164.75**	**126.25**	**103.25**	**89.00**	**78.25**	**70.75**
$\mathsf{B_pW^{2\,\underline{bc}}_w}$	398.50	249.25	172.25	131.00	107.50	92.00	81.50	73.25
BNDM	**369.25**	236.50	167.75	**124.50**	**99.25**	**81.25**	**69.75**	**61.00**

Results for a Rand8 problem

m	2	4	6	8	10	12	14	16
B_pW_w	353.75	204.00	154.75	127.50	108.25	92.75	80.75	69.50
$B_p^2W_w$	258.25	185.00	145.00	114.25	94.00	78.75	67.75	59.00
$B_p^2W_w^{bc}$	**265.00**	193.50	151.50	118.50	98.25	82.00	71.00	62.00
$B_pW_w^2$	278.50	**173.50**	**129.00**	**99.25**	**79.75**	**65.75**	**56.25**	**48.50**
$B_pW_w^{2\,bc}$	282.75	181.25	134.25	103.25	82.75	68.25	58.00	50.25
BNDM	**265.00**	**169.0**	132.00	108.50	91.00	77.00	66.00	57.25

Results for a Rand16 problem

m	2	4	6	8	10	12	14	16
B_pW_w	273.75	160.75	119.25	98.25	85.00	76.00	68.50	62.75
$B_p^2W_w$	**191.50**	136.50	114.75	95.75	81.50	71.50	63.00	56.00
$B_p^2W_w^{bc}$	197.00	140.50	119.00	98.00	84.50	73.50	64.25	56.75
$B_pW_w^2$	215.00	**129.00**	**98.75**	**82.00**	**70.50**	**61.00**	**53.50**	**46.25**
$B_pW_w^{2\,bc}$	219.50	132.50	101.25	83.25	72.00	62.50	54.75	47.50
BNDM	218.75	**128.25**	**98.00**	**82.00**	72.00	64.50	58.25	52.75

Results for a Rand32 problem

m	2	4	6	8	10	12	14	16
B_pW_w	250.00	136.75	98.50	79.50	67.50	59.50	53.75	49.25
$B_p^2W_w$	**160.00**	107.00	89.00	75.00	65.25	59.75	54.00	50.25
$B_p^2W_w^{bc}$	162.75	111.50	91.50	75.00	67.50	60.50	54.25	50.00
$B_pW_w^2$	184.25	**104.25**	**77.00**	**63.25**	**55.25**	**49.50**	**45.00**	**41.25**
$B_pW_w^{2\,bc}$	186.00	105.75	78.25	64.25	55.75	49.75	45.50	41.75
BNDM	197.50	109.00	79.25	64.25	**55.25**	**49.50**	**45.00**	41.75

Results for a Rand64 problem

m	2	4	6	8	10	12	14	16
B_pW_w	238.50	126.25	88.25	69.25	58.00	50.00	44.50	40.25
$B_p^2W_w$	**145.75**	92.75	73.50	61.50	53.00	49.00	43.50	41.50
$B_p^2W_w^{bc}$	148.00	96.00	76.00	61.50	54.75	49.00	43.00	41.00
$B_pW_w^2$	168.25	**91.25**	**65.25**	**52.50**	**44.25**	**39.25**	**36.00**	**33.00**
$B_pW_w^{2\,bc}$	169.00	92.00	65.75	**52.50**	45.00	39.75	**36.00**	**33.00**
BNDM	187.25	99.50	70.25	55.50	46.75	40.75	36.50	33.50

Results for a Rand128 problem

The above experimental results show that the algorithms obtained by applying a second level of parallelism perform always better then the original B_pW_w algorithm. The gap is more evident in the case of short patterns or small alphabets. In particular the $B_p^2W_w$ algorithm achieves its best performances with small alphabets, while the $B_pW_w^2$ algorithm turns out to be the best choice for patterns with a length greater than 4.

The BNDM algorithm obtains the best results in some cases and it performs always better than the B_pW_w algorithm. It is interesting to observe that the BNDM algorithm is outperformed by the $B_p^2W_w$ algorithm when the alphabet is small and by the $B_pW_w^2$ algorithm in the case of large alphabets.

6 Conclusions

We have presented two variants of the Bit-Parallel Wide-Window algorithm which use a second level of parallelization inspired by the *Single Instruction Multiple Data* paradigm. While the $\mathsf{B}_\mathsf{p}^2\mathsf{W}_\mathsf{w}$ variant is quite entangled to the original algorithm, as it uses two different automata in parallel, the other one ($\mathsf{B}_\mathsf{p}\mathsf{W}_\mathsf{w}^2$) is quite general and much the same approach can possibly be applied to other bit-parallel algorithms. As the experimental results show, this technique provides a non negligible speedup; this is particularly true for the second variant as it allows to double the size of window shifts.

References

1. J. Arndt. *Matters Computational*. 2009. `http://www.jjj.de/fxt/`.
2. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Commun. ACM*, 35(10):74–82, 1992.
3. Longtao He, Binxing Fang, and Jie Sui. The wide window string matching algorithm. *Theor. Comput. Sci.*, 332(1-3):391–404, 2005.
4. Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro. Increased bit-parallelism for approximate and multiple string matching. *J. Exp. Algorithmics*, 10:2.6, 2005.
5. K. Kaplan, L. L. Burge III, and M. Garuba. A parallel algorithm for approximate string matching. In H. R. Arabnia and Youngsong Mun, *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '03*, pages 1844–1848, 2003.
6. Wei Lin and Bin Liu. Pipelined parallel AC-based approach for multi-string matching. In *Proc. 14th International Conference on Parallel and Distributed Systems (ICPADS 2008)*, pages 665–672, 2008.
7. P. D. Michailidis and K. G. Margaritis. A programmable array processor architecture for flexible approximate string matching algorithms. *J. Parallel Distrib. Comput.*, 67(2):131–141, 2007.
8. G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In M. Farach-Colton, editor, *Proc. of the 9th Annual Symposium on Combinatorial Pattern Matching*, number 1448 in Lecture Notes in Computer Science, pages 14–33, Piscataway, NJ, 1998. Springer-Verlag, Berlin.
9. H. Peltola and J. Tarhio. Alternative algorithms for bit-parallel string matching. In M. A. Nascimento, E. Silva de Moura, and A. L. Oliveira, editors, *SPIRE 2003*, volume 2857 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2003.
10. D. Semé and S. Youlou. Parallel algorithms for string matching problems on a linear array with reconfigurable pipelined bus system. In M. J. Oudshoorn and S. Rajasekaran, editors, *Proc. of the ISCA 18th International Conference on Parallel and Distributed Computing Systems*, pages 55–60, 2005.