An Investigation of Dead-Zone Pattern Matching Algorithms

by
Melanie Barbara Mauch

Thesis presented in fulfilment of the requirements for the degree of Master of Arts in the Faculty of Arts and Social Sciences at Stellenbosch University

Supervisor:	Prof.Dr.Dr. Bruce W. Watson
Co-supervisor:	Dr.Ir. Loek Cleophas

March 2016

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: .March 2016.

Abstract

Pattern matching allows us to search some text for a word or for a sequence of characters—a popular feature of computer programs such as text editors. Traditionally, three distinct families of pattern matching algorithms exist: the *Boyer-Moore (BM)* algorithm, the *Knuth-Morris-Pratt (KMP)* algorithm, and the *Rabin-Karp (RK)* algorithm. The basic algorithm in all these algorithmic families was developed in the 1970s and 1980s. However a new family of pattern matching algorithms, known as the *Dead-Zone (DZ)* family of algorithms, has recently been developed. In a previous study, it was theoretically proven that *DZ* is able to pattern match a text with fewer match attempts than the well-known *Horspool* algorithm, a derivative of the *BM* algorithm.

The main aim of this study was to provide empirical evidence to determine whether *DZ* is faster in practice. A benchmark platform was developed to compare variants of the *DZ* algorithm to existing pattern matching algorithms. Initial experiments were performed with four C implementations of the *DZ* algorithm (two recursive and two iterative implementations). Subsequent to this, *DZ* variants that make use of different shift functions as well as two parallel variants of *DZ* (implemented with Pthreads and CUDA) were developed. Additionally, the underlying skeleton of the *DZ* algorithm was tweaked to determine whether the *DZ* code was optimal.

The benchmark results showed that the C implementation of the iterative *DZ* variants performed favourably. Both iterative algorithms beat traditional pattern matching algorithms when searching natural language and genome texts, particularly for short patterns. When different shift functions were used, the only time a *DZ* implementation performed better than an implementation of the traditional algorithm was for a pattern length of 65536 characters. Contrary to our expectations, the parallel implementation of *DZ* did not always provide a speedup. In fact, the Pthreaded variants of *DZ* were slower than the non-threaded *DZ* implementations, although the CUDA *DZ* variants were consistently five times faster than a CPU implementation of *Horspool*. By using a *cache-friendly DZ* algorithm, which reduces cache misses by about 20%, the the original *DZ* can be improved by approximately 5% for relatively short patterns (up to 128 characters with a natural language text). Moreover, a cost of recursion and the impact of information sharing were observed for all *DZ* variants and have thus been identified as intrinsic *DZ* characteristics.

Further research is recommended to determine whether the *cache-friendly DZ* algorithm should become the standard implementation of the *DZ* algorithm. In addition, we hope that the development of our benchmark platform has produced a technique that can be used by researchers in future studies to conduct benchmark tests.

Abstrak

Patroonpassing word gebruik om vir 'n reeks opeenvolgende karakters in 'n blok van teks te soek. Dit word breedvoerig programmaties in rekenaarenaarprogramme gebruik, byvoorbeeld in teksredigeerders. Tradisioneel is daar drie afsonderlike patroonpassingalgoritme families: die *Boyer-Moore (BM)* familie, *Knuth-Morris-Pratt (KMP)* familie en *Rabin-Karp (RK)* familie. Die basisalgoritmes in hierdie algoritmefamilies was reeds in die 1970s en 1980s ontwikkel. Maar, 'n nuwe patroonpassingsalgoritme familie is egter onlangs ontwikkel. Dit staan as die *Dooie Gebied (DG)* algoritme familie bekend. 'n Vorige studie het bewys dat *DG* algoritmes in staat is om patroonpassing uit te voer met minder passingpogings as die welbekende *Hoorspool* algoritme, wat 'n afgeleide algortime van die BM algoritme is.

Die hoofdoel met hierdie studie was om die *DG* familie van algoritmes empiries te ondersoek. 'n Normtoets platform is ontwikkel om veranderlikes van die DG algoritme met bestaande patroonpassingsalgoritmes te vergelyk. Aanvanklike eksperimente is met vier C implementasies van die *DG* algoritme uitgevoer. Twee van die implementasies is rekursief en die ander twee is iteratief. Daarna was *DG* variante ontwikkel wat van verskillende skuif-funksies gebruik maak het. Twee parallelle variante van *DG* was ook ontwikkel. Een maak gebruik van "Pthreads' en die ander is in CUDA geimplementeer. Verder was die C kode weergawe van die basiese DG algoritme fyn aangepas om vas te stel of die kode optimaal was.

Die normtoetsresultate dui aan dat die C-implementasie van die iteratiewe *DG* variante gunstig presteer bo-oor die tradisionele patroonpassingsalgoritmes. Beide van die iteratiewe algoritmes klop die tradisionele patroonpassingsalgoritmes wanneer daar met relatiewe kort patrone getoets word. Die verrigting van verskeie skuif-funksies was ook geondersoek. Die enigste keer wanneer die *DG* algoritmes beter presteer het as die tradisionele algoritme, was vir patroonlengtes van 65536 karakters. Teen ons verwagtinge, het die parallelle implementasie nie altyd spoedtoename voorsien nie. Tewens, die "Pthread" variante van *DG* was stadiger as die nie-gerygde *DG* implementasies. Die CUDA *DG* variante was egter telkens vyf keer vinniger as die konvensionele SVE implementasie van *Horspool*. Die normtoetse het ook aangedui dat die oorspronklike *DG* kode naby aan optimaal was. Egter, deur 'n kas-vriendelike weergawe te gebruik wat kas oorslane met omtrent 20% verminder, kon die prestasie met naastenby 5% verbeter word vir relatiewe kort patrone (tot by

128 karakters met natuurlike taal teks). Verder was daar vir al die DG variante ‘n rekursiekoste en ‘n impak op inligtingdeling waargeneem wat as interne DG kenmerke geidentifiseer is.

Verdere navorsing word aanbeveel om vas te stel of die kas-vriendelike DG algoritme die standaard implementasie van die DG algoritme behoort te word. Bykomstiglik, hoop ons dat die ontwikkeling van ons normtoets platform ’n tegniek geproduseer het wat deur navorsers in toekomstige studies gebruik kan word om normtoetse uit te voer.

Preface

Before I acknowledge the people who have contributed to the production of this thesis, some background information needs to be provided. This is my master's thesis, submitted in fulfilment of the MA Socio-Informatics degree at Stellenbosch University. It is the result of my research for the FASTAR (Finite Automata Systems — Theoretical and Applied Research) group, from which I had three advisors: Prof.Dr.Dr. B.W. Watson, Dr.Ir. L. Cleophas and Prof.Dr. D. Kourie.

One of the core research interests of the FASTAR group is pattern matching. I joined the FASTAR research group in 2012 during my BSc (Honours) Computer Science degree at the University of Pretoria. My Honours project focused on benchmarking a new pattern matching algorithm (known as *Dead-Zone*) that was developed by members of the FASTAR group. I created a benchmarking framework for the *Dead-Zone* code that was written by Bruce Watson. The framework enabled us to compare *Dead-Zone* to already existing algorithms based on how long they took to find all occurrences of a pattern in a string. This study serves as a continuation of my Honours research.

I want to thank Bruce Watson and Derrick Kourie for allowing me to continue my *Dead-Zone* research and for guiding me through this research project (as well as previous projects). I also want to thank Loek Cleophas for his exceptionally helpful comments during the review process. I must also acknowledge Tinus Strauss for the parallelism advice at times of critical need.

I would also like to thank David Gregg from Trinity College Dublin and Jorma Tarhio from Aalto University for their contributions to the *Dead-Zone* codebase and for their comprehensive advice on how the *Dead-Zone* algorithm could be improved.

My sincere thanks also goes to my employer, Mike Love, not only for supporting me throughout the project, but also for letting me work part-time so that I could devote my time to my studies.

I must express my gratitude to my close friends and colleagues who provided a much needed escape from my studies and helped me stay sane through these difficult years. I am also grateful to my friends and family in Stellenbosch and Cape Town that helped me adjust to life in a new city.

A very special thanks goes out to the National Research Foundation (NRF) as

well as Stellenbosch University. I recognise that this research would not have been possible without the financial assistance and bursaries provided to me by these institutions.

Finally, I would like to thank my parents and Johann Koekemoer for their persistent encouragement, love and assistance. Completing this work would have been all the more difficult were it not for their precious support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ability to search some text for a word or for a sequence of characters is a common function that is incorporated into a wide array of computer programs. This search functionality is made possible through the use of pattern matching algorithms. Pattern matching also has a number of practical application areas, ranging from computer security (virus scanning) to bioinformatics (DNA sequencing).

Until now, there existed three main families of pattern matching algorithms, all three of which were developed during the 1970s and 1980s. These are the *Knuth-Morris-Pratt (KMP)* algorithm [21], the *Boyer-Moore (BM)* algorithm [9] and the *Rabin-Karp (RK)* algorithm [19]. Newer pattern matching algorithms are based on techniques that were first introduced in these three algorithms.

The recently developed *Dead-Zone (DZ)* algorithm performs pattern matching in a unique way that differs from these algorithms and can be viewed as a unique family of pattern matching algorithms. In a previous study [37], it was theoretically proven that, in the best case, the *DZ* algorithm requires fewer match attempts than an existing pattern matching algorithm (*Horspool*) to perform pattern matching, however empirical evidence produced by benchmark experiments is needed to determine whether *DZ* can be faster in practice. This is the area addressed by the research described in this thesis.

1.1 Related Work

It is evident that there is a lack of rigour involved in scientific benchmark experiments. Kalibera and Jones [18] examined 122 papers published at leading conferences in 2011 and found that the majority of those papers reported their experiments in a way that make them seemingly impossible to repeat. Mytkowicz et al. [25] also examined more than one hundred research papers and determined that measurement bias is significant and commonplace in papers with experimental results. Similarly, Vitek and Kalibera [36] found that it is common for computer science publications to have unclear benchmarking goals, measurement bias, inappropriate benchmarks and no comparisons with the state of the art.

To improve the quality of software benchmark experiments, Vitek and Kalibera [36] recommend that empirical evaluations should be properly documented, thus making the research study repeatable and reproducible. This is supported by Pieterse and Flater [32], who also suggest that the measurement of software performance should be conducted in such a way that others will be able to corroborate the validity of the findings.

1.2 Thesis Aims

The general aim of this research is to determine the empirical performance of new variants of the *Dead-Zone* algorithm and how they compare to existing pattern matching algorithms. In order to do this, original variants of the *Dead-Zone* family of algorithms must be developed and implemented such that their performance can be assessed.

The following questions needed to be answered about the *Dead-Zone* implementations:

- Will the performance of the *Dead-Zone* algorithm improve if different shift functions (such as those used by the *Boyer-Moore* algorithm and variants) are used?
- Can parallel implementations of the *Dead-Zone* algorithm perform efficiently and achieve a speedup?
- Is the *Dead-Zone* code optimal—i.e. will the performance improve if the code skeletons are tweaked?

The main aims of the benchmark experiments were:

- To identify the time taken for an algorithm to find all occurrences of a pattern in a text.
- To analyse the captured data in order to establish the properties of the algorithms.
- To produce a technique that could be used by researchers to conduct benchmark tests.

1.3 Thesis Structure

This text consists of seven chapters and four appendices. The following paragraphs give an overview of each chapter.

Chapter 2 gives an introduction to pattern matching. It discusses a few of the traditional pattern matching algorithms and introduces the *Dead-Zone* family of algorithms. It concludes by stating the four basic variants of *Dead-Zone*.

Chapter 3 contains a summary of existing *Dead-Zone* research that was conducted during my Honours year. It describes the implementation of the four basic variants of *Dead-Zone*. It also describes the implementation of a benchmark platform and how the benchmark results influenced the iterative and incremental design of the benchmark platform. The subsequent chapters extend the work given in this chapter.

Chapter 4 investigates the performance of the *Dead-Zone* algorithm when different shift functions are used. The choice of shifters is explained and the implementation of the algorithms is given.

Chapter 5 describes two parallel implementations of *Dead-Zone*, one with POSIX Threads and the other using CUDA. The differences between the two implementations are highlighted. Performance results of the benchmark experiments are also discussed.

Chapter 6 provides nine new *Dead-Zone* skeletons that attempt to improve the performance of the original variants of the Dead-Zone algorithm. Also, for each skeleton, the modifications that were made to the code are discussed. It also examines the results of the benchmark experiments and makes a recommendation on which Dead-Zone skeleton to use.

Chapter 7 summarises the contributions of the thesis and provides possible directions for future work.

Appendix B provides the code for the four basic variants of the *Dead-Zone* algorithm.

Appendix C displays graphs pertaining to the multiple shifter implementations of *Dead-Zone*.

Appendix D displays graphs for both parallel *Dead-Zone* implementations.

Appendix E contains graphs relating to the performance of the different *Dead-Zone* skeleton implementations.

Chapter 2

Pattern Matching

The exact string matching problem is defined as finding all occurrences of a given pattern $p = p_0p_1...p_{m-1}$ in a text $t = t_0t_1...t_{n-1}$ where t and p are finite sequences from some finite character set Σ.

The exact string matching problem was first solved in 1975 by Aho and Corasick [1] and later by Boyer and Moore [9] and Knuth, Morris and Pratt [21] in 1977. Although more pattern matching algorithms have since appeared [12], many of which derived from the *Aho-Corasick*, *Boyer-Moore (BM)* and *Knuth-Morris-Pratt (KMP)* algorithms, the originals are still the most well-known.

Figure 2.1 displays the naive way to match a pattern in a string. The pattern is aligned at the beginning of the text and each character is compared from left to right. In the case of a mismatch, the pattern is always shifted one character to the right and characters are again compared starting with the first letter of the pattern. These redundant character comparisons are the reason for the *brute force* algorithm's time complexity of $\mathcal{O}(nm)$ [10].

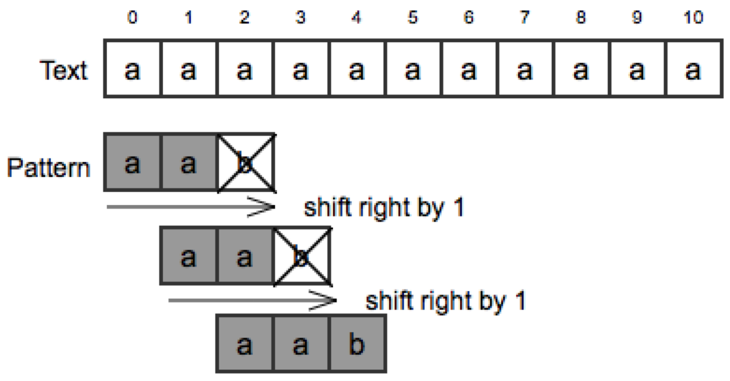


Figure 2.1: *Brute force* pattern matching

2.1 Traditional Algorithms

In order to perform clever shifts that skip character comparisons in t or p, one or more shift tables are used to determine the number of positions that P will be shifted. For each pattern, the shift tables are precomputed prior to performing the pattern matching. The details of computing shift tables can be found in a number of articles and books such as [9, 21, 26] and will not be discussed in this thesis.

KMP improves on the brute force approach and achieves a time complexity of $\mathcal{O}(n)$ [26]. It uses the concept of a string prefix and suffix: given that a and b are strings, a is a prefix of ab and b is a suffix of ab. When a mismatch occurs, the characters in p that were successfully matched with characters in t make up the characters of the prefix. p is shifted a precomputed number of positions to the right to align the prefix in p with the longest suffix of the current alignment window in t that is also a prefix in p. The next comparison will start at the character of p immediately following the prefix and go again from left to right.

Figure 2.2 illustrates how the *KMP* algorithm uses this prefix information to avoid redundant character comparisons. p is aligned at the beginning of t and matching occurs from left to right. A mismatch occurs for a at p_1 and for b at t_1. The prefix a in p cannot be aligned with b in t, thus p is shifted two positions to the right. Note that we already know the prefix aa in p matches the prefix aa in t; matching begins from p_2 and a mismatch is detected. The prefix aa in p matches the suffix aa in the current alignment window in t and p is shifted one position to the right.

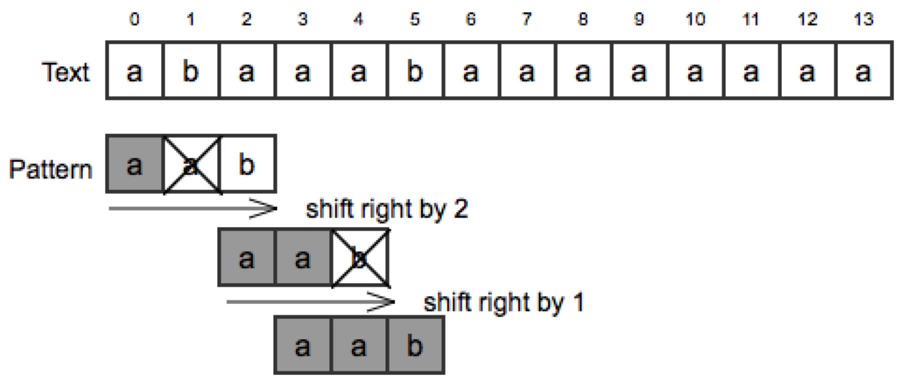


Figure 2.2: *Knuth-Morris-Pratt* pattern matching

Although the *KMP* algorithm attempts to minimise the number of characters in p involved in pattern matching, it is not possible to skip any characters in t. All characters in t are read from left to right when *KMP* pattern matching is performed. The *BM* algorithm, however, is able to skip over characters in t by searching for suffixes and matching p with t from right to left, not from left to right. Rules are used to precompute tables that determine the number of positions that p will shift.

Given that a mismatch occurs at t_i and p_j, the following bad-character heuristics apply:

1. If the mismatched character in t_i does not appear in p, align p_0 with t_{i+1}.
2. If the mismatched character in t_i occurs to the right of p_j, shift p to the right by one position.
3. If the mismatched character in t_i occurs only to the left of p_j, align t_i with the closest character to p_j that matches t_i.

Given that a mismatch occurs at t_i and p_j, the following good-suffix heuristics apply:

1. If p contains a suffix $p_{j+1}...p_{m-1}$ that is equal to a substring beginning to the left of p_j and not preceded by the mismatched character p_j, align the suffix in t with the right-most substring in p to the left of p_j that matches the suffix.
2. If the above rule does not apply, align the longest suffix after after t_i in the current alignment window of t with the closest prefix to p_j that matches the suffix.

When a mismatch is detected, the number of positions that p will shift is determined by the maximum between the shifts given by the bad-character and good-suffix rules.

The rules are highlighted in Figure 2.3. p is aligned with the beginning of t and matching occurs from right to left. A mismatch occurs for b at p_3 and a at t_3. Both the bad-character shift and good-suffix shift are equal to one. The good-suffix rule aligns b at p_3 with the suffix b at t_4 and p shifts one position to the right. Characters are compared from right to left and a mismatch is found for b at p_4 and c at t_5. c does not occur in p, thus, according to the bad-character rule, p shifts $|p| = 5$ positions to the right. Characters are again compared from right to left until a mismatch is detected for b at p_3 and a at t_9. Again, both the bad-character shift and the good-suffix shift are one. We align the prefix b at p_1 with the suffix b at t_{10}. Characters are compared from right to left and a mismatch immediately occurs for b at p_4 and a at t_{11}. The bad-character rule aligns a at position t_{11} with the closest a in p and p shifts two positions to the right.

The time complexity of the *BM* algorithm is $\mathcal{O}(mn)$, although in the average case it performs sub-linearly [26]. When using a large alphabet such as a natural language text, the bad-character heuristics produce the longest shifts. *Horspool* [17] used this notion to develop the first simplified version of the *BM* algorithm. To determine the number of positions that p will shift, the *Horspool* algorithm uses only the bad-character shift, and uses it on the last character in the current window in t instead of on the mismatched character. While

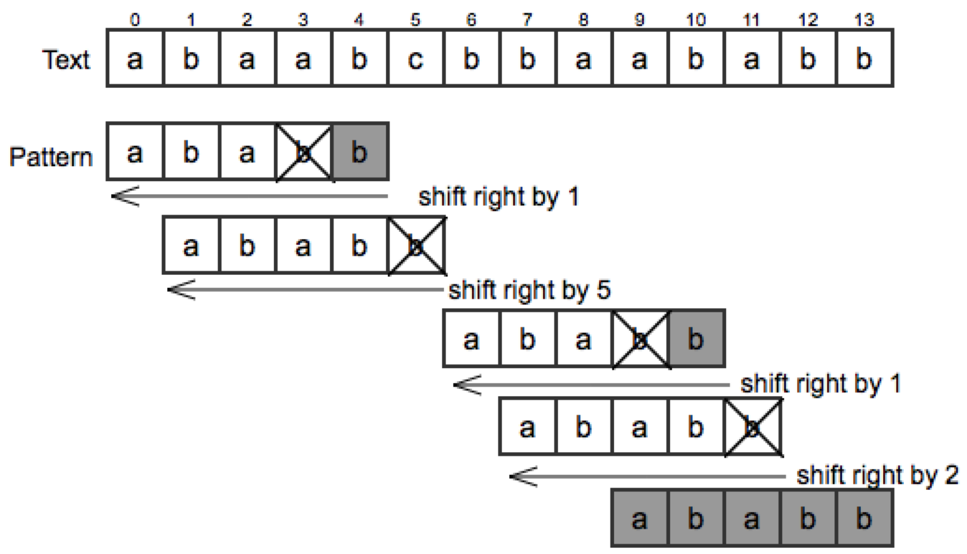


Figure 2.3: *Boyer-Moore* pattern matching

this modification yields on average longer shifts than the bad character shift of *BM* and has one less shift table, it has the same search time complexity as the original *BM* algorithm [6].

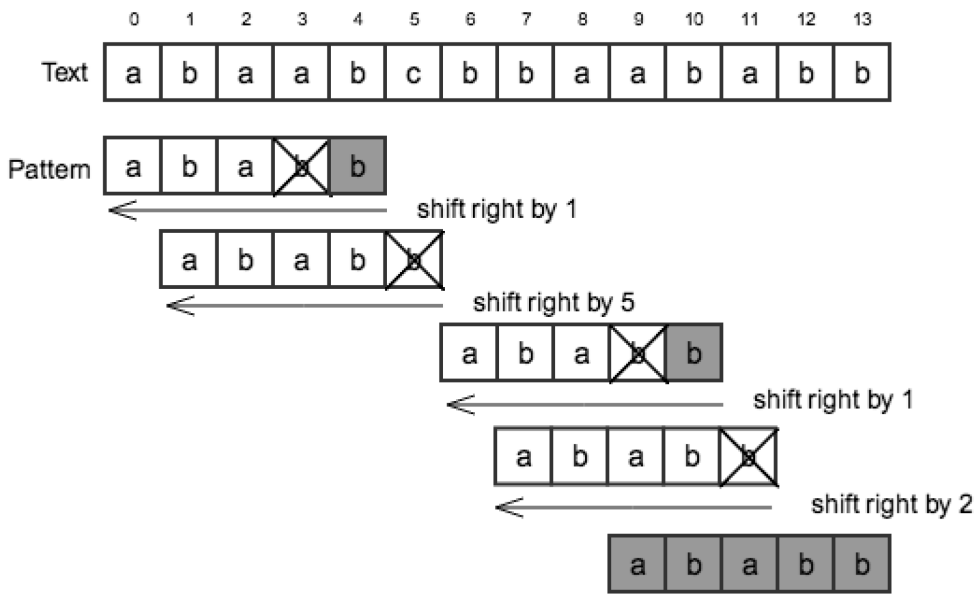


Figure 2.4: *Horspool* pattern matching

Horspool's bad-character shifts are shown in Figure 2.4. As in the *BM* algorithm, P is aligned with the beginning of t and matching occurs from right to left. A mismatch occurs for b at p_3 and for a at t_3. The last character in the text window, b at position t_4, is aligned with the closest b in p which shifts p one position to the right. A mismatch is found for b at p_4 and c at t_5. c does not occur in p, therefore p shifts $|p| = 5$ positions to the right. Characters are compared from right to left until a mismatch is found for b at p_3 and for a at t_9. The closest a in p is aligned with a at t_9, shifting p one position to the right. Matching begins on the right and a mismatch is immediately found for b at p_4 and a at t_{11}. The closest a in p is aligned with a at t_{11} and p shifts two positions to the right.

2.2 Dead-Zone Algorithms

The *Dead-Zone (DZ)* algorithms are a new family of single keyword pattern matching algorithms by Watson and Watson [38] that require less match probes than Horspool to determine whether p occurs in t [37].

The main *DZ* idea comprises of a growing number of dead zones: live zones in the text are searched and dead zones are generated as searching progresses. A dead zone is an area in the text where matching does not need to happen. Conversely, a live zone is an area in the text that has not been inspected—where matching still needs to occur. Prior to matching, t can be seen as one live zone. As matching occurs, p is placed somewhere in the middle of the live zone and then shifts both to the left and to the right in t, creating a dead zone.

It should be noted that because *DZ* is a family of algorithms, there are many versions of the abstract algorithm where the following parameters differ: match orders, shift functions and the match attempt point [37].

The *DZ* implementation in this study matches from left to right and uses the mid-point as a match attempt point. A *Horspool*-like shift function is used, unless otherwise stated. Chapter 4 discusses the performance of *DZ* when other shift functions are used.

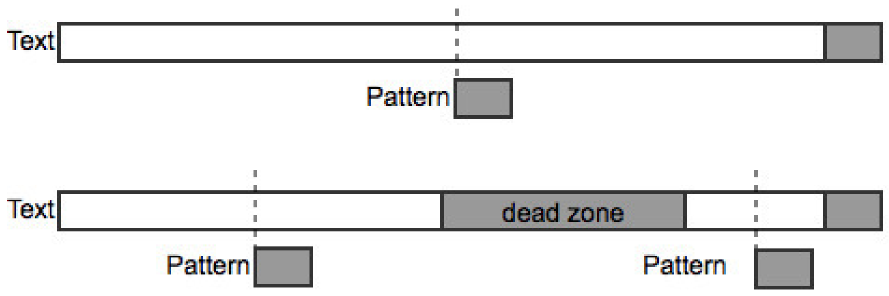


Figure 2.5: Dead zones created in live zones

Algorithm 1 (Abstract DZ Matcher)

```
proc $dzmat(live\_low, live\_high) \rightarrow$
        if $(live\_low \geq live\_high) \rightarrow$ skip
        ▯ $(live\_low < live\_high) \rightarrow$
            $j := \lfloor (live\_low + live\_high)/2 \rfloor$;
            $i := 0$;
            { invariant:
                    $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$ }
            do $((i < |p|)$ cand $(p_{mo(i)} = t_{j+mo(i)})) \rightarrow$
                $i := i + 1$
            od;
            { post: $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$
                    $\wedge\ ((i < |p|) \Rightarrow (p_{mo(i)} \neq t_{j+mo(i)}))$ }
            if $i = |p| \rightarrow print($'Match at ', $j)$
            ▯ $i < |p| \rightarrow$ skip
            fi;
            $new\_dead\_left := j - shift\_left(i, j) + 1$;
            $new\_dead\_right := j + shift\_right(i, j)$;
            $dzmat(live\_low, new\_dead\_left)$;
            $dzmat(new\_dead\_right + 1, live\_high)$
        fi
corp
```

The abstract recursive *DZ* algorithm from [24] is duplicated in Algorithm 1 such that an explanation of the algorithm can be given here. The recursive function is called dzmat. It searches the text t between the indices $[live\_low, live\_high)$ for all occurrences of the pattern p.

There cannot be a match in an area of text smaller than $|p|$, therefore the last $|p| - 1$ characters immediately become part of the dead zone. This means that the first invocation of dzmat uses a live zone with the dimensions $[0, |t| - |p| + 1)$.

The recursion terminates if the index of beginning of the live zone passes (or is equal to) the index of the end of the live zone i.e. $(live\_low \geq live\_high)$. This is the recursive base case of Algorithm 1. Otherwise, if $(live\_low < live\_high)$, the index of the live zone's mid-point is computed and stored as variable j. The first match attempt will occur at this index.

A loop matches characters in p with characters in t using variable i to reference an index in p, and i and j to reference an index in t. Matching is performed in the order specified by the match order function mo. As an aside, note that the code used in this study uses a left-to-right match order. This is in contrast to the *BM* and *Horspool* algorithms given in Section 2.1.

If a complete match is found, the loop terminates and j, the starting index for this iteration of matching, is printed out. Likewise, the loop terminates when the first mismatch occurs.

The new dead zone needs to be computed based on the characters that were successfully matched and, if a mismatch occurred, the position of the mismatched character. Two shift tables, *shift_left* and *shift_right*, determine how many characters to the left and to the right of j will become part of the dead zone. The variable *new_dead_left* is the lower bound of the dead zone, while variable *new_dead_right* is considered the upper bound of the dead zone.

To search the live zone areas that occur on either side of the newly computed dead zone, *dzmat* gets invoked twice. The first invocation attempts to match in the interval $[live\_low, new\_dead\_left\ )$, and the second invocation attempts to match in the remaining live zone in the interval $[new\_dead\_right + 1, live\_high)$.

Algorithm 2 (DZ Matcher with sharing)

```
proc dzmat_sh(live_low, live_high) →
    if (live_low ≥ live_high) → $\underline{d := live\_low}$
    ⫿ (live_low < live_high) →
      $j := \lfloor (live\_low + live\_high)/2 \rfloor$;
      $i := 0$;
      { invariant:
            $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$ }
      do $((i < |p|)$ cand $(p_{mo(i)} = t_{j+mo(i)}))$ →
          $i := i + 1$
      od;
      { post: $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$
                $\wedge\ ((i < |p|) \Rightarrow (p_{mo(i)} \neq t_{j+mo(i)}))$ }
      if $i = |p|$ → print('Match at ', $j$)
      ⫿ $i < |p|$ → skip
      fi;
      new_dead_left := $j - shift\_left(i, j) + 1$;
      new_dead_right := $j + shift\_right(i, j)$;
      dzmat_sh(live_low, new_dead_left );
      dzmat_sh($\underline{\max(d}, (new\_dead\_right + 1))$, live_high)
    fi
corp
```

While attempting to match in the left live zone, a dead zone may develop that is so large it overlaps with the right live zone. In Algorithm 1, information is not shared between the live zones, yet monitoring the growth of the left live zone and sharing this information with the right live zone limits the size of the right live zone when matching occurs there.

Algorithm 2 shows that information sharing is easily implemented with an integer variable d to keep track of the upper bound of the left live zone. However, sharing information incurs a running-time penalty [24] because variable d is updated in once in the code and also read once in the code.

Additionally, there exists an iterative version of the *DZ* algorithm, as shown in Algorithm 3. The first recursive call (into the left live zone) is eliminated and a stack is manually implemented for the second recursive call (into the right live zone).

Algorithm 3 (DZ Matcher with iteration)

```
proc $dzmat\_iter(\langle live\_low, live\_high\rangle) \rightarrow$
     var $Todo$ : $stack\ of\ low/high\ index\ pairs$;
     $push\ \langle live\_low, live\_high\rangle\ onto\ Todo$;
     do $Todo \neq \emptyset \rightarrow$
        $pop\ \langle live\_low, live\_high\rangle\ onto\ Todo$;
        if $(live\_low \geq live\_high) \rightarrow$ skip
        $[\!]$ $(live\_low < live\_high) \rightarrow$
          $j := \lfloor (live\_low + live\_high)/2 \rfloor$;
          $i := 0$;
          { invariant:
              $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$ }
          do $((i < |p|)$ cand $(p_{mo(i)} = t_{j+mo(i)})) \rightarrow$
             $i := i + 1$
          od;
          { post: $(\forall\ k : k \in [0, i) : p_{mo(k)} = t_{j+mo(k)})$
                  $\wedge\ ((i < |p|) \Rightarrow (p_{mo(i)} \neq t_{j+mo(i)}))$ }
          if $i = |p| \rightarrow print($'Match at ', $j)$
          $[\!]$ $i < |p| \rightarrow$ skip
          fi;
          $new\_dead\_left := j - shift\_left(i, j) + 1$;
          $new\_dead\_right := j + shift\_right(i, j)$;
          $push\ \langle (new\_dead\_right + 1), live\_high\rangle\ onto\ Todo$;
          $push\ \langle live\_low,\ new\_dead\_left\ \rangle\ onto\ Todo$
        fi
     od
corp
```

A version of Algorithm 3 with sharing also exists.

This dissertation predominantly focuses on four basic variants of the DZ family of algorithms:

DZ(rec,nsh) This is a recursive non-sharing implementation, as shown in Algorithm 1.

DZ(rec,sh) This is a recursive sharing implementation, as shown in Algorithm 2.

DZ(iter,nsh) This is an iterative non-sharing implementation, as shown in Algorithm 3.

DZ(iter,sh) This is an iterative sharing implementation that combines the sharing of *DZ(rec,sh)* with the iterative loop of *DZ(iter,nsh)*.

Chapter 3

Dead-Zone Performance

3.1 Introduction

Theoretically, it has been proven that (in the best case) the *DZ* algorithm requires fewer match attempts than the *Horspool* algorithm to find all occurrences of a pattern in a text [37] because of *DZ*'s ability to doubly claim real estate to both the left and the right of the pattern. Empirical evidence is, however, required to see whether it outperforms the *Horspool* algorithm in practice.

The aim of this chapter is to explore the actual empirical processing speed of *DZ* algorithms compared to the *KMP*, *BM* and *Horspool* algorithms.

This chapter begins by examining the experimental design of the study and arguing the choice of data used. Then, the test procedure and implementations used in the study are explained in detail. Subsequently, the results of the study are analysed and discussed.

It should be noted that this chapter is a summary of joint work with Bruce Watson, Derrick Kourie and Tinus Strauss that has been previously published as [24].

3.2 Experimental Design

The experiment was carried out on a 2011 model MacBook Pro with the following specifications:

- Operating System: Mac OS X version 10.7.4
- Processor: Intel Core i7
- Processor speed: 2.8 GHz
- Number of cores: 2

- L2 Cache (per core) : 256 KB
- L3 Cache (per core): 4 MB
- Memory: 4 GB, 1333 MHz, DDR3

The executables for the benchmark experiment were compiled with Xcode 4.2 into Release builds. This corresponds to -O3 optimisation on GCC and most other compilers. Input symbols (chars) are used to index the shift tables, thus a compiler option for `unsigned char` was also used. All benchmark tests were performed using only one core with hyper-threading disabled. Furthermore, all unnecessary processes were terminated such that the process performing the benchmarking was the only user process utilising the core.

The experiment was conducted in two phases. In the first phase, a C++ version of the recursive sharing *DZ* algorithm given in Algorithm 2 was implemented. It made heavy use of object-oriented and template features of a C++ framework that was set up. This *DZ* implementation will be referred to as *DZ(rec,sh,OO)* because it relies on recursion, information sharing (as explained in Section 2.2) and object-orientation. Additionally, C++ versions of *BM*, *Horspool* and *KMP* were implemented and compared to *DZ(rec,sh,OO)*. In this phase, apart from using the optimising compiler, no further attempts were made to optimise the processing time of the *DZ* algorithm. Thus, *DZ(rec,sh,OO)* would serve as the upper bound on *DZ*'s empirical performance.

In the second phase, *DZ(rec,sh,OO)* was optimised by removing the object-oriented and template features. This resulted in four different *DZ* variants:

- *DZ(rec,sh)*,
- *DZ(rec,nsh)*,
- *DZ(iter,sh)* and
- *DZ(iter,nsh)*.

The code for these four different *DZ* variants can be found in the appendix.

Because the implementations of the *BM*, *Horspool* and *KMP* algorithms are uncomplicated they could be regarded as C implementations within a C++ benchmarking environment. Sanity checks were performed against the SMART platform [23] and it was established that the *BM*, *Horspool* and *KMP* implementations did not require optimising.

3.3 The Data

Pattern matching was performed using selected texts from the SMART corpus [23]. To determine the effects of alphabet size on the performance of the algorithms, we wanted to use a small alphabet as well as a large alphabet. Therefore, we chose a genome text, with an alphabet size of four, and a natural language text (the Bible in English), with a theoretical alphabet size of 256. However, a sed script that was run on the Bible text established that exactly 63 different characters appeared in it. Both the genome text file and the natural language text file have a size of approximately 4 MB.

Although the alphabet size of the two selected texts differed, patterns of the same length from both alphabets occupied the same amount of storage. This is because characters from both texts were stored as C++ `int` values, even though genetic data only requires 2 bits per symbol.

Patterns were chosen in two ways:

1. Patterns were randomly generated from the alphabet using the built-in C++ pseudo-random number generator.

2. Patterns were randomly chosen from the input text.

The first approach has a high chance of generating a pattern that does not appear in the text, especially if the alphabet size is large or the pattern is long. The latter case guarantees that at least one instance of the pattern will be found in the text and is the generally preferred method.

Initially, pattern lengths of 2^n were used where $n = 2, \ldots, 12$. However, in later benchmarks this was increased to $n = 2, \ldots, 14$ to see what effect larger patterns would have on the algorithms' performance.

3.4 Test Procedure

The SMART framework [23] was investigated as a possible platform for running the benchmark tests. However, for several reasons we decided to create our own benchmarking platform that would allow us to achieve more precise results with more control and repeatability.

Firstly, SMART requires C code. The starting point for implementing the DZ algorithms made use of object-orientation in C++ where different variations of the base abstract *DZ* algorithm could be implemented as subclasses using inheritance.

Secondly, we required a high resolution timing mechanism. SMART captures time in milliseconds, yet we did not know a priori whether measuring time with millisecond resolution would sufficiently discern the differences between the performance of the algorithms. The overhead of setting up shift tables is also in the SMART timing data. Furthermore, SMART runs a number of tests for each algorithm and returns the mean value for the runs. No other timing data is available.

Moreover, while experimenting with the SMART framework we experienced a number of difficulties. These are discussed in Section 3.9.

We developed a standard test procedure (described in Figure 3.1) that gets applied to each algorithm (*BM*, *Horspool*, *DZ(rec,sh,OO)* *DZ(rec,sh)* *DZ(rec,nsh)* *DZ(iter,sh)* and *DZ(iter,nsh)*) using each of the texts (genome and natural language) and each of the two approaches to choosing patterns (randomly generated from the alphabet or randomly chosen from the text). Pattern matching is performed with the algorithms and the time taken for each algorithm to find all occurrences of a pattern p is recorded in nanoseconds.

Be advised that the setting up of shift tables is precomputed prior to pattern matching and is not included as part of the timing data.

```
for $n = 2$ to $psize$
    for $i = 1$ to $pnum$
        Generate $p_i$ such that $|p_i| = 2^n$
        Set up algorithm tables
        for $j = 1$ to $pmin$
            Start timer
            Search $s$ for $p_i$
            Accumulate number of hits
            Stop timer and record time as $t(s, p_i, j)$
            Record total hits
        rof
        $t_{min}(s, p_i) := MIN : j \in [1, pmin] : t(s, p_i, j)$
    rof
    $t_{avg}(s, n) := (\sum_{i=1}^{pmin} t_{min}(s, p_i))/pnum$
rof
```

Figure 3.1: Test procedure in pseudo-code

The loop maxima have been parameterised in the pseudo-code because the technicalities changed slightly as testing proceeded. As explained in Section 3.3, during the initial tests *psize* was chosen as 12, but was subsequently increased to 14. Likewise, *pnum* was chosen as 500 and *pmin* as 1 — i.e. 500 different patterns were tested for each pattern length. The reason for this was because we did not want to deviate too much from the SMART framework [23] which, by default, generates sets of 500 patterns. These 500 results are analysed for average behaviour over the 500 runs as well as for minimum behaviour with respect to the 500 runs. However, this is not shown in the pseudo-code.

During the initial tests, it was found that it took a long time to complete the 500 runs. Moreover, 500 runs seemed needlessly large for the experiment when 30 observations are regarded as large enough to draw statistically valid conclusions. Thus, *pnum* was changed to 100 and *pmin* to 30 — i.e. 30 runs of the same pattern repeated 100 times for each pattern length. The minimum of the thirty runs, captured as $t_{min}(s, p_i)$, is used as the result for a given algorithm and pattern combination. This is repeated one hundred times and the average of the hundred minimum values is then computed as $t_{avg}(s, n)$. The decision to repeat each algorithm thirty times on the same data and record the minimum time was a precaution, expected to minimise the effect of outliers that could occur from unpredictable operating system behaviour.

A subsequent study by Kourie et al. [22] found that "[computing the average of minimum times taken over several iterations on the same data] appears to be a fairly robust and accurate performance metric for comparing minimum time behaviour of algorithms". This corroborates our experimental design. Consequently, a similar experimental design was used for all of the experiments discussed in this dissertation.

A sanity check was done to ensure that all of the algorithms find the same number of occurrences of p for all of the runs.

Note that, in the case of our experiments, it was identified that the optimising compiler optimised out all code that does not produce a side effect. Therefore, we needed to include a counter for the number of times a pattern is found and also record this count. This is shown in the pseudo-code in Figure 3.1 with "Accumulate number of hits" and "Record total hits". Without them the search code is removed by the optimiser.

3.5 Implementation

It has already been mentioned that the *BM*, *Horspool* and *KMP* algorithms were implemented in C code within a C++ environment. As an aside, note that, in regard to *Horspool*, the size of the shift table depends on the size of the alphabet being used, while in the case of *KMP*, the size of the shift table depends on the length of the pattern being tested. *BM* has one shift table that depends on the alphabet size and another shift table that depends on the length of the pattern.

In addition to the *BM*, *Horspool* and *KMP* algorithms, five variants of the *DZ* algorithm were benchmarked:

DZ(rec,sh,OO) This is a C++ implementation of Algorithm 2 using the architecture described in [37]. It makes use of object-oriented programming and C++ best practices from [14], including:

- The `string` class from the C++ standard library.
- The `vector` class from the C++ Standard Template Library (STL), used for shift tables.
- Emphasising code readability and relying on the optimising compiler.
- Hardly any virtual functions, for performance reasons.

- Preferring template parameterisation over inheritance, for performance reasons.
- Separate classes for different match orders, used as template parameters to the main pattern-matcher class.
- Separate classes for different probe choosers (where to make match attempts), used as template parameters to the main pattern-matcher class.
- Separate classes for different shifters, representing various shift functions.

DZ(rec,nsh) This is a C implementation of Algorithm 1. As opposed to *DZ(rec,sh,OO)*, almost all aspects were coded manually and without the use of libraries such as STL or `string`. For example, instead of division by two, the probe chooser that computed the average between low and high used a binary right shift.

DZ(rec,sh) This is a C implementation of Algorithm 2 in the same style as *DZ(rec,nsh)*.

DZ(iter,nsh) This is a C implementation of Algorithm 3 in the same style as *DZ(rec,nsh)*.

DZ(iter,sh) This is a C implementation that combines the iterative loop of *DZ(iter,nsh)* with the sharing of *DZ(rec,sh)*.

All five variants relied on *Horspool*'s right shift table (shift-right in Algorithm 1) and a left shift table (shift-left in Algorithm 1) that looks at the current text character aligned with the first pattern character and specifies how many characters can be safely shifted to the left — i.e. the left shift table is the mirror of *Horspool*'s right shift table.

3.6 High Resolution Timer

In order to accurately measure the performance of the algorithms, our experiments required a high precision timer with nanosecond resolution. The Mach 3.0 kernel of Mac OS X provides an efficient way to do time management on Apple computers [4]. It provides a monotonic clock that uses the Mach absolute time unit. This unit is CPU dependent and is converted to other units of time (such as nanoseconds) by using the `mach_timebase_info` API. However, since the CPU increments the absolute time unit, the monotonic clock stops when the CPU is powered down—which includes when the system goes to sleep. Consequently, the computer performing the benchmark experiment was prevented from going into sleep mode by using the `caffeinate` [3] terminal command.

Timers belong to the real time scheduling thread in the Mach kernel [5]. If there are a large number of real time threads trying to be executed then there will be contention over which thread executes first, and the timers will lose precision. To avoid this situation, we only create one timer object in the test harness and reuse it to capture the time taken for each of the algorithms to find all the patterns in the text.

3.7 Output Data

An overview of the benchmarking data that was captured for this study is presented in Table 3.1. Fourteen different benchmark experiments were conducted using the testing harness developed in this study. Each benchmark test generated a separate set of data that was captured in its own file. In total, 430 MB of raw data was stored and analysed. The resulting data was used to change the DZ implementations over the course of the study, as mentioned in the preceding sections. Moreover, based on the results of the benchmarking, the benchmarking platform was also improved and developed further.

Benchmark Number	**Text**	**Patterns**	**Description of Data**
1	Ecoli	Up to a length of 64 characters, randomly generated with pseudorandom number generator	Initial tests.
2	Ecoli	Up to a length of 64 characters, randomly generated with pseudorandom number generator	100 runs per pattern length. Discovered that the code in the loops where matches were found was being optimised away.
3	Ecoli	Up to a length of 256 characters, randomly generated with pseudorandom number generator and randomly chosen from text	100 runs per pattern length. First tests with *DZ(rec,sh,OO)*.
4	Ecoli	Up to a length of 4096, randomly chosen from text	100 runs per pattern length.
5	Ecoli	Up to a length of 4096, randomly chosen from text	100 runs per pattern length. Optimised version of DZ.
6	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length. Optimised version of DZ.
7	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length as well as the average of 30 minimums over 100 runs. Improved DZ.
8	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length. Improved DZ.

9	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length. 3 versions of DZ.
10	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length. 6 versions of DZ.
11	Ecoli	Up to a length of 4096, randomly chosen from text	500 runs per pattern length. 7 versions of DZ.
12	Ecoli	Up to a length of 16384, randomly chosen from text	Average of 30 minimums over 100 runs. 4 refined versions of DZ mentioned in the preceding sections.
13	Bible	Up to a length of 4096, randomly chosen from text	Average of 30 minimums over 100 runs. 4 refined versions of DZ mentioned in the preceding sections.
14	Bible	Up to a length of 16384, randomly chosen from text	Average of 30 minimums over 100 runs. 4 refined versions of DZ mentioned in the preceding sections.

Table 3.1: Overview of captured data

3.8 Overview of Results

The graph in Figure 3.2 shows a broad overview of the timing data. It is not intended to give a detailed evaluation of the data at this point, as this will be discussed later. The graph represents the time taken for six of the eight algorithms (*DZ(rec,sh,OO)* and *KMP* are not shown) to search the natural language text for all occurrences of a given pattern. The minimum time of thirty runs with the same data averaged over one hundred different patterns of the same length is depicted. Patterns were chosen randomly from the text.

The general trend had only a slight variation when the smaller alphabet was used. It also remained similar when patterns were randomly generated instead of chosen from the text. Therefore, the subsequent discussions will assume that the data set refers to the natural language text where patterns have been randomly chosen from the text, and where the average minimum time over one hundred observations for the same pattern length, with the minimum taken from thirty runs with the same data, has been used.

3.9 SMART Results

Section 3.4 already highlighted some of the reasons why the SMART framework [23] was not suitable for our tests. Furthermore, while experimenting with

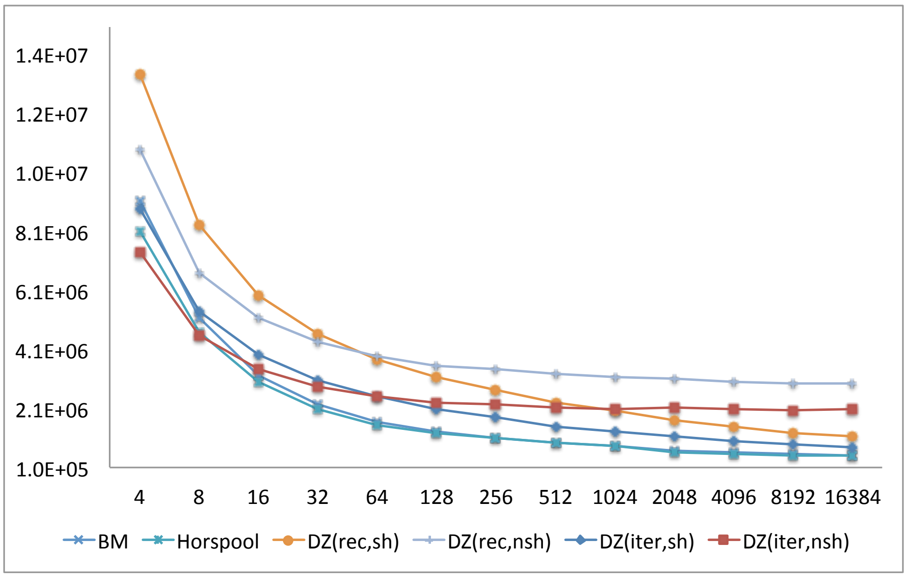


Figure 3.2: Illustrative raw averaged minimum time data
Source: [24]

SMART, we found our implementations of *BM* and *Horspool* to be faster than the SMART implementations.

Table 3.2 shows the difference in timing between the *BM* and *Horspool* implementations in the SMART platform (t_{SMART}) and the *BM* and *Horspool* implementations in our platform (t_{us}). The difference is expressed as a percentage of our times — i.e. $100 \times (t_{SMART} - t_{us})/t_{us}$. The comparisons are drawn from the mean time (in milliseconds) to find all occurrences of a pattern in 1 MB of genome data. As pattern length increases, the percentage difference in time between the two *BM* implementations increases, and the percentage difference in time between the two *Horspool* implementations stays fairly constant, except for one outlier. This outlier suggests that the SMART timings have a somewhat erratic quality — something that was also detected in subsequent tests.

3.10 Cost of Object-Orientation

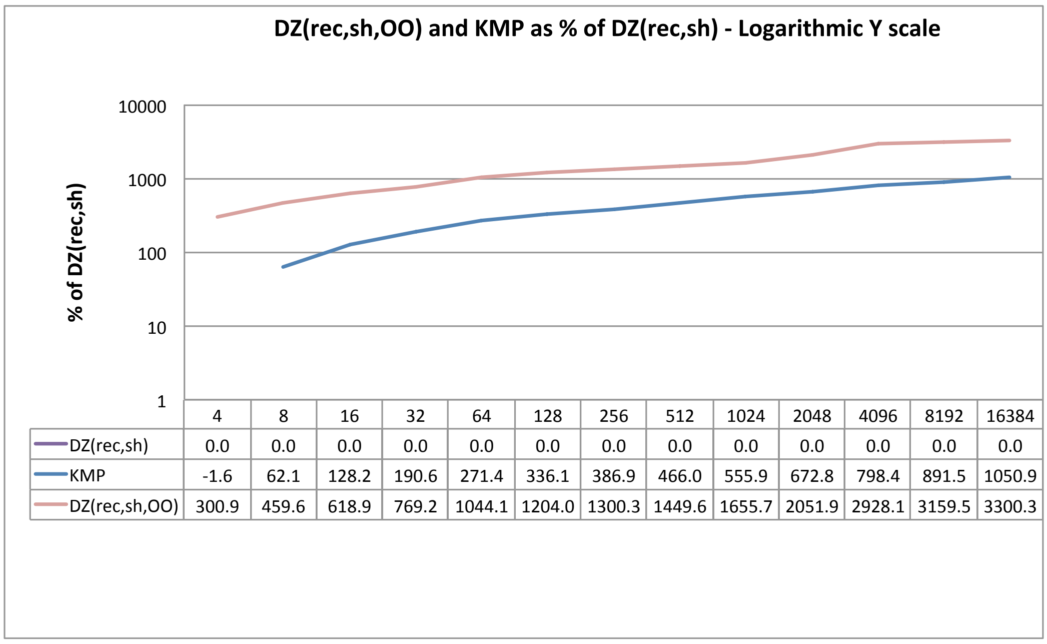
Figure 3.3 shows the performance of *DZ(rec,sh,OO)* and *KMP* using the performance of *DZ(rec,sh)* as a base line. The only difference between the *DZ(rec,sh,OO)* and *DZ(rec,sh)* implementations is that the former uses C++ templates and object-oriented features discussed in Section 3.5.

Note that the vertical axis uses a logarithmic scale. It is evident that the performance difference between *DZ(rec,sh,OO)* and *DZ(rec,sh)* gets larger as pattern length increases. In the best case (pattern length of 4), *DZ(rec,sh,OO)* is three times slower than *DZ(rec,sh)*, and with a pattern length of 16384 (the longest pattern we tested), *DZ(rec,sh,OO)* is thirty-three times slower

Table 3.2: Differences between SMART's timings and our timings (expressed as a percentage of our implementations).

Pattern Length	*BM*	*Horspool*
4	44	60
8	79	66
16	100	69
32	114	101
64	128	68
128	148	62
256	161	50
512	178	61
1024	205	56
2048	229	58
4096	276	63

than *DZ(rec,sh)*. Every time the pattern length is doubled, the performance difference between *DZ(rec,sh,OO)* and *DZ(rec,sh)* can be expected to increase by approximately 250% [24].



	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
DZ(rec,sh)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
KMP	-1.6	62.1	128.2	190.6	271.4	336.1	386.9	466.0	555.9	672.8	798.4	891.5	1050.9
DZ(rec,sh,OO)	300.9	459.6	618.9	769.2	1044.1	1204.0	1300.3	1449.6	1655.7	2051.9	2928.1	3159.5	3300.3

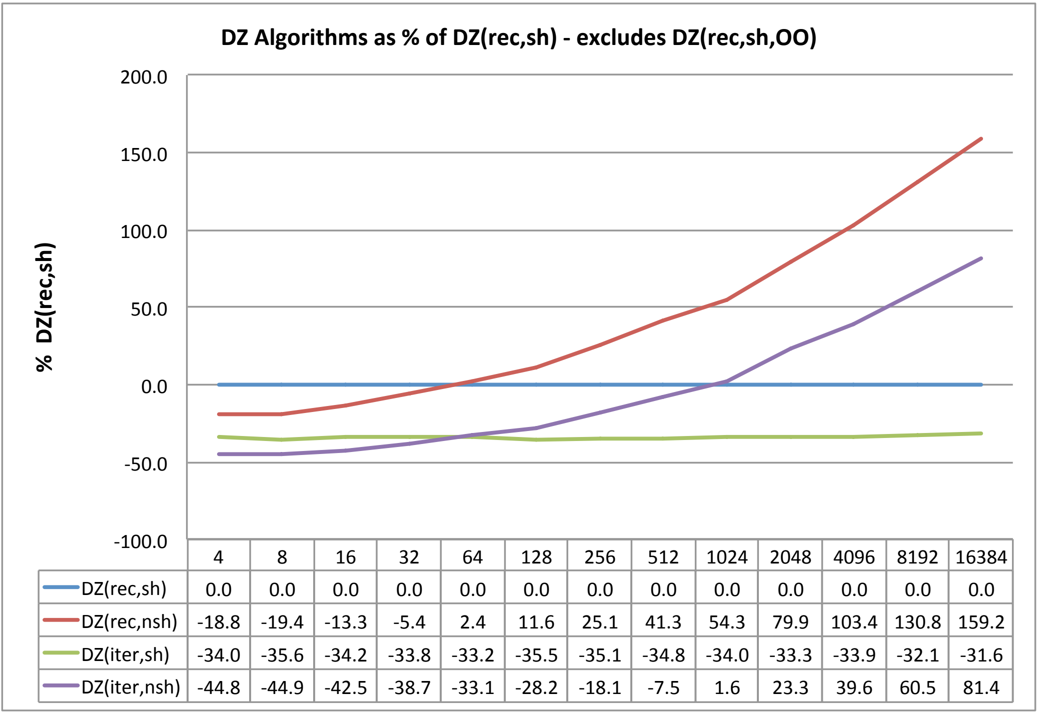
Figure 3.3: Cost of Object-Orientation
Source: [24]

Although *KMP* outperformed *DZ(rec,sh,OO)*, it performed poorly compared to *DZ(rec,sh)*, particularly with longer patterns. Moreover, it was found that *KMP* had a fairly constant performance as pattern length increased, unlike most of the other algorithms that had performance improvements as patterns got longer. In hindsight, *KMP* was not a notably interesting algorithm for our tests, and has been accordingly excluded from most of the results.

3.11 Cost of Recursion

Figure 3.4 shows the performance of *DZ(rec,nsh)*, *DZ(iter,sh)* and *DZ(iter,nsh)* relative to *DZ(rec,sh)*. It is evident that the iterative sharing version (*DZ(iter,sh)*) is consistently between 31% and 36% faster than its recursive counterpart (*DZ(rec,sh)*). Similarly, the iterative non-sharing version (*DZ(iter,nsh)*) is also quicker than the recursive non-sharing version (*DZ(rec,nsh)*).

We expected that the optimiser would recognise the relatively simple recursive calls and produce machine code similar to that of *DZ(iter,sh)* and *DZ(iter,nsh)* using the tail-recursion elimination transform. However, it is apparent that requiring the compiler to maintain a stack of live zone boundaries for the recursive calls instead of doing it oneself is time intensive and costs approximately one third of the total *DZ* pattern matching time.



	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
DZ(rec,sh)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DZ(rec,nsh)	-18.8	-19.4	-13.3	-5.4	2.4	11.6	25.1	41.3	54.3	79.9	103.4	130.8	159.2
DZ(iter,sh)	-34.0	-35.6	-34.2	-33.8	-33.2	-35.5	-35.1	-34.8	-34.0	-33.3	-33.9	-32.1	-31.6
DZ(iter,nsh)	-44.8	-44.9	-42.5	-38.7	-33.1	-28.2	-18.1	-7.5	1.6	23.3	39.6	60.5	81.4

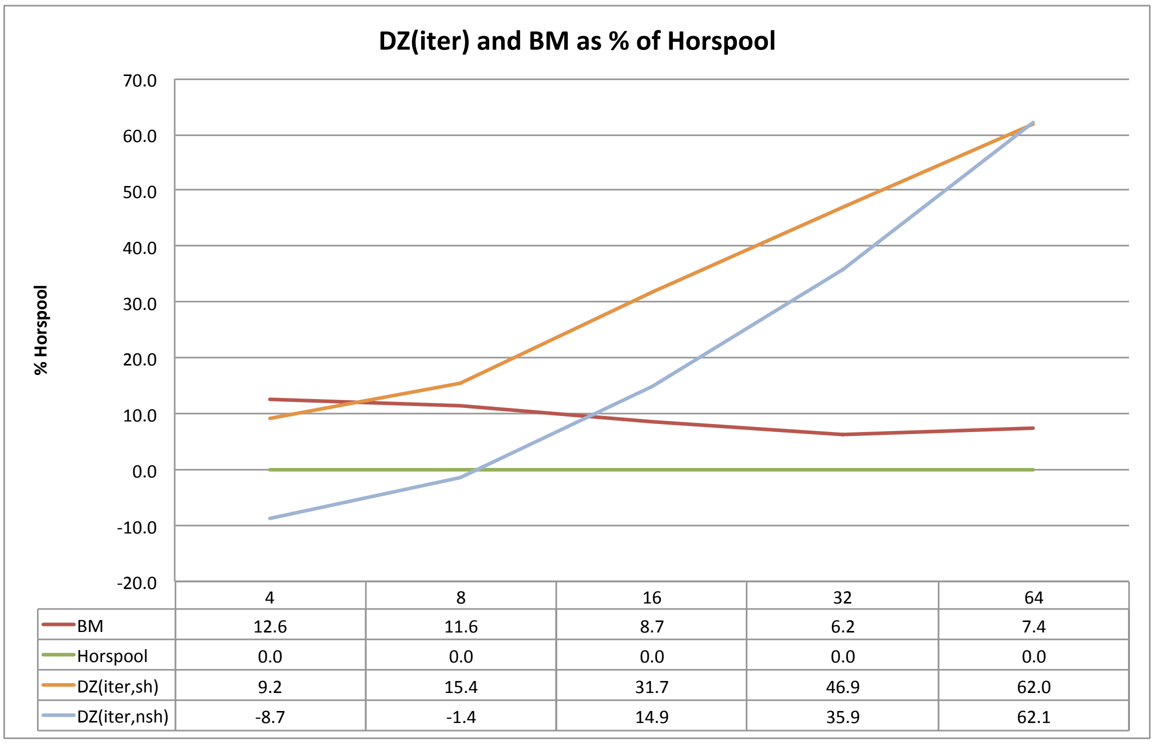
Figure 3.4: Cost of Recursion
Source: [24]

3.12 Impact of Information Sharing

In Figure 3.4, the impact of information sharing is also shown. Information sharing makes use of a variable d that gets updated in two places (discussed in Section 2.2), thus incurring a running-time penalty. Because of this penalty, when patterns are small, the non-sharing versions of DZ outperform their sharing counterparts. With a pattern length of about 64, *DZ(rec,sh)* and *DZ(rec,nsh)* have an almost identical performance. The same applies to *DZ(iter,sh)* and *DZ(iter,nsh)* with a pattern length of 64. When pattern lengths get larger than 64, the non-sharing versions perform progressively worse than the sharing versions. The reason is that longer patterns generally produce longer shifts, which are statistically more likely to grow the dead zone discovered during match attempts in the left live zone into the right live zone. It is evident that a pattern length of about 64 is the break-even point between sharing live zone information and suffering running-time penalties or not sharing live zone information.

3.13 Best Performing Algorithms

The results discussed thus far clearly show that the two iterative variants of *DZ*, *DZ(iter,sh)* and *DZ(iter,nsh)*, perform the best. Figure 3.5 shows *DZ(iter,sh)* and *DZ(iter,nsh)* compared to *BM* and *Horspool* for pattern lengths up to 64. It illustrates that *DZ(iter,sh)* only manages to outperform *BM* with a pattern length of 4, but after that it starts to perform steadily worse while *BM* performs better, until *BM* surpasses the performance of *Horspool* at a pattern length of 1024 (not shown in Figure 3.5).



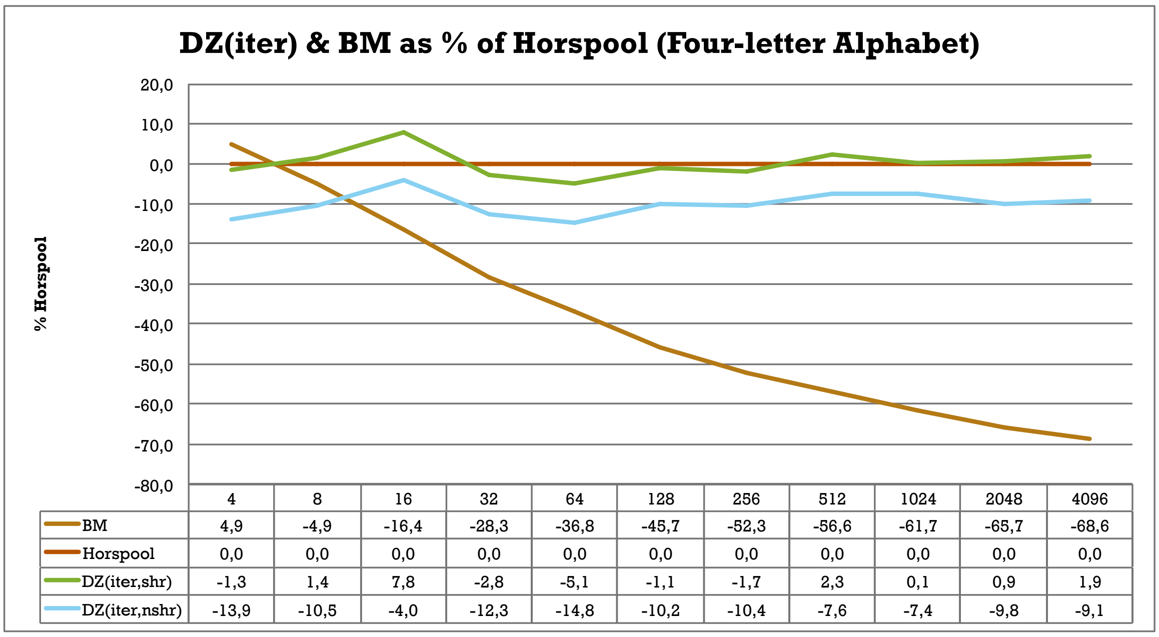
	4	8	16	32	64
BM	12.6	11.6	8.7	6.2	7.4
Horspool	0.0	0.0	0.0	0.0	0.0
DZ(iter,sh)	9.2	15.4	31.7	46.9	62.0
DZ(iter,nsh)	-8.7	-1.4	14.9	35.9	62.1

Figure 3.5: Best Performing Algorithms
Source: [24]

Under the best of circumstances (with a pattern length of 4), *DZ(iter,nsh)* outperforms *Horspool* by 8%, and continues to outperform *Horspool* for all pattern lengths up to about 9. Moreover, *DZ(iter,nsh)* performs better than *BM* for pattern lengths less than approximately 14. This indicates that *DZ(iter,nsh)* might be useful for natural language processing when relatively short patterns will be searched for.

3.14 Effect of Smaller Alphabets

Benchmark experiments were also performed using the genome text to explore the effects of smaller alphabets on the performance of the *DZ* algorithms. In these tests, the parameters in Figure 3.1 differ to what has been previously described—i.e. $psize = 12$, $pnum = 500$, $pmin = 1$. Thus 500 different patterns were selected for each pattern length from $2^2 to 2^{12}$, and the tests were not repeated 30 times using the same data. This decision was supported by noting that the timing data obtained from rerunning the tests using the same data usually only had slight variations. Two data sets were computed from the test results: the mean time per pattern length over the 500 observations, and the minimum time per pattern length over the 500 observations.

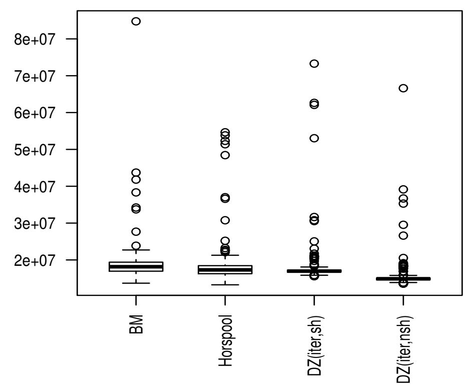


	4	8	16	32	64	128	256	512	1024	2048	4096
BM	4,9	-4,9	-16,4	-28,3	-36,8	-45,7	-52,3	-56,6	-61,7	-65,7	-68,6
Horspool	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
DZ(iter,shr)	-1,3	1,4	7,8	-2,8	-5,1	-1,1	-1,7	2,3	0,1	0,9	1,9
DZ(iter,nshr)	-13,9	-10,5	-4,0	-12,3	-14,8	-10,2	-10,4	-7,6	-7,4	-9,8	-9,1

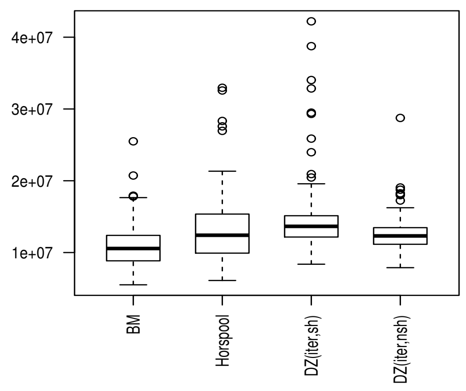
Figure 3.6: Best Performing Algorithms for Genome Text
Source: [24]

Figure 3.6 shows that *DZ(iter,nsh)* consistently outperformed *Horspool* by between 4% and 14%, and it outperformed *BM* up to a pattern length of about 9. Although *BM* started off slower than the other algorithms, it increasingly outperformed the rest as pattern length increased. The performance of *Horspool* and the performance of *DZ(iter,sh)* were very much alike.

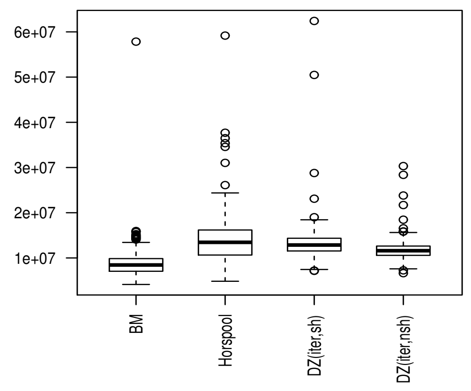
Figure 3.6 also illustrates that the sharing of information does not have as much of an impact when using a small alphabet as it does with a large alphabet. The performance of *DZ(iter,sh)* does not improve and eventually surpass that of



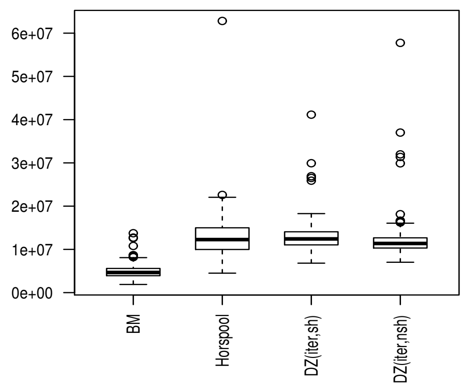
(a) Pattern length 4



(b) Pattern length 16



(c) Pattern length 64



(d) Pattern length 1024

Figure 3.7: Box plots of minimum results from pattern matching with the smaller alphabet

Source: [24]

DZ(iter,nsh) as pattern length increases. This differs from the results discussed in Section 3.12 that were collected from the 256 letter alphabet.

The best-case performance of each of the four algorithms at pattern lengths 4, 16, 64 and 1024 can be seen in Figure 3.7. It shows that, when looking at the minimum time per pattern length over the 500 observations, *Horspool* consistently outperformed *DZ(iter,nsh)* and *DZ(iter,sh)*. In fact, only the behaviour of the *BM* algorithm corresponded to what is shown for the average case in Figure 3.6. Note that the circles in Figure 3.7 represent outliers. Also, each subfigure is drawn to a different scale and should not be compared against one other. It is, however, interesting to note that only the bottom whisker in the *Horspool* plots is lower than that of *DZ(iter,nsh)* as well as *DZ(iter,sh)*. These box plots highlight that statistical claims about performance (whether best-, worst- or average-case) should not be allowed to obscure the possibility of significant deviations in terms of outliers [24].

3.15 Conclusion

Many lessons were learned during this study, not all of which are related to pattern matching. It is informative to briefly mention the knowledge gained to highlight how it influenced the studies mentioned in Chapters 4, 5 and 6:

- Earlier research proved that the *DZ* algorithm requires fewer probes than the *Horspool* algorithm to find all occurrences of a pattern in a text [37]. For this reason, we expected that this experiment would simply be a matter of comparing the performance of the existing C++ implementation of *DZ(rec,sh,OO)* to the performance of the *BM* and *Horspool* algorithms, anticipating a good *DZ* performance. The "cost of object-orientation" was an unforeseen discovery. We suspect that researchers and the computer science community are not aware of how substantial this cost is for such fundamental algorithms.
- Making use of our own benchmark platform (written in C++) instead of the SMART platform [23] (written in C) was a good decision. The SMART platform was not flexible enough and the timings were too erratic for our requirements. In addition, algorithms coded in C and executed in our C++ benchmarking environment were as efficient as the standard C executions.
- We did not expect that the compiler would optimise out all code that did not produce any side effects. Because of this, we hoped that it would optimise out all the effects of the "two-tail recursive" calls, but we were, however, disappointed. This can be achieved manually with code, so there should be an optimising compiler that can handle a class of "doubly tail-recursive" problems, which suggests a possible research topic for compiler researchers.
- Because of the running-time penalties, we did not know whether the sharing of information would be at all beneficial in the *DZ* algorithm. Incidentally, the payoff for sharing dead-zone boundary information becomes increasingly noticeable as pattern length increases. The sharing variants perform better than the non-sharing variants at a pattern length of approximately 64 characters.
- When matching a genome text, the *DZ(iter,nsh)* algorithm performed consistently better the *Horspool* algorithm and outperformed the *BM* algorithm for short patterns. With a natural text, the *DZ(iter,nsh)* algorithm outperformed both the *Horspool* algorithm and the BM algorithm for shortish patterns.
- The performance of the *DZ* algorithms showed a tendency to be similar to the performance of the *Horspool* algorithm, rather than the performance

of the *BM* algorithm. This can be easily explained by the fact that *Horspool*-based shift tables were used. The impact of different shift tables is explored in Chapter 4

- The *DZ* algorithm can be simply converted to a threaded implementation by executing the two recursive calls in parallel. This is presented in the study in Chapter 5.

Chapter 4

Multiple Shifters

4.1 Introduction

Each iteration (or recursion) of a *DZ* algorithm entails a right shift and a left shift. The use of *Horspool*'s right shift table and a *Horspool*-based left shift table explains why the performance of the *DZ* algorithms tended to be similar to that of *Horspool* in Chapter 3. However, shift tables from algorithms other than *Horspool* could also be used in implementations of *DZ* because *DZ* algorithms are not restricted to using *Horspool*-based shift tables. Furthermore, shifters from various algorithms can be used in different combinations—i.e. determining the shift distance using a right shift table from some pattern matching algorithm and a left shift table based on another algorithm.

Accordingly, the aim of this chapter is to find out whether there would be any significant changes in the behaviour of the *DZ* algorithms when different left and right shifters are used.

First, this chapter explains the shifters that were used in the benchmark experiments. Then, the details of the experiment are discussed. Finally, this chapter reports on the impact of using different combinations of shift tables in *DZ* implementations.

4.2 Shifters Used

There are a wide variety of pattern matching algorithms in the literature [11], each having strengths and weaknesses and each associated with its own right shift table(s). However, symmetry arguments can convert right shift tables into left shift tables. Consequently, each shift table found in the literature can therefore be used as the basis for the left- and right shift tables required in the *DZ* algorithm.

We considered using shift tables from the best performing character comparison based algorithms identified in [12], however, most have shift table implementations which are incompatible with the test harness used in this study (for example, one algorithm makes use of hashing). As a result, we used shift tables, each in their respective left and right shifter versions, from three traditional algorithms that were compatible with the C implementation of *DZ* used in this study. Specifically, shift tables from Sunday's *Quick Search (QS)* algorithm [33] (denoted by q), the *Berry-Ravindran (BR)* algorithm [8] (denoted by b) and the *Horspool* algorithm [17] (denoted by h) were used. Nine left-right shifter combinations can be formed out of these shifters, namely $\{h\text{-}h, h\text{-}q, h\text{-}b, q\text{-}h, q\text{-}q, q\text{-}b, b\text{-}h, b\text{-}q, b\text{-}b\}$. The details of the various shift tables can be found in the respective literature and will not be discussed in this paper.

4.3 Experimental Design

The experiment was carried out on a 2012 model MacBook Pro with the following specifications:

- Operating System: Mac OS X version 10.8.5
- Processor: Intel Core i7
- Processor speed: 2.6 GHz
- Number of cores: 4
- L2 Cache (per core): 256 KB
- L3 Cache (per core): 6 MB
- Memory: 8 GB, 1600 MHz, DDR3

All executables were compiled with the GCC compiler, using the optimisation option -O3.

4.4 The Data

Two texts from the SMART corpus [23] were used to carry out the pattern matching experiments, namely a genome text and a natural language text (the Bible). These texts are described in Section 3.3.

Patterns of length 2^n were used, where $n = 1, \ldots, 16$ to determine the effect of very short ($2^1 = 2$) and very long ($2^{16} = 65536$) patterns. Patterns were selected randomly from the text by using a pseudorandom number generator to provide an index into the text as the start of a pattern of a given length. To ensure a cross-comparison of performance, different implementations used the same randomly generated patterns for matching by seeding the pseudorandom number generator with the same number.

4.5 Implementation

Each of the nine left-right shifter pairs $\{h\text{-}h, h\text{-}q, h\text{-}b, q\text{-}h, q\text{-}q, q\text{-}b, b\text{-}h, b\text{-}q, b\text{-}b\}$ was used to implement four instantiations of the abstract algorithm: *DZ(rec,sh, *-*)*, *DZ(rec,nsh,*-*)*, *DZ(iter,sh,*-*)* and *DZ(iter,nsh,*-*)*. Thus we had $4 \times 9 = 36$ different implementations of the *DZ* algorithm. As an aside, the experiments discussed in Chapter 3 were based on *DZ(*,*,h-h)*.

The *DZ* variants were implemented in C code (in a C++ environment) with each variant having three separate files for its skeleton, left shifter and right shifter. Instead of explicitly coding 36 different versions of *DZ*, we made us of the C preprocessor in the test harness to define 36 different combinations of skeletons, left shifters and right shifters.

Data structures were not space-optimised to account for different alphabet sizes. The patterns and texts to be searched were all treated as `const unsigned char` arrays, although the number of characters occurring in these arrays would be limited by the alphabet size of the data set. One-dimensional arrays of type `int` with a size of 256 were used to implement the h and q shift tables, and two-dimensional arrays of type `int` with a size of 256 in each dimension were used to implement the b shift tables. Again, this ignored the potential space gains that would have been possible by tuning these.

Additionally, C versions of the *Horspool*, *QS* and *BR* algorithms were implemented and compared to the 36 different *DZ* variants. In this chapter, we refer to these *Horspool*, *QS* and *BR* implementations as the standard versions of the algorithms.

4.6 Test Procedure

The same test harness described in Section 3.4 and shown in Figure 3.1 was applied to each of the 36 *DZ* variants (*DZ(iter,sh,h-q)*, *DZ(rec,nah,q-b)*, etc.), using each of the texts (bible and genome). Pattern matching is performed with the algorithms and the time taken for each algorithm to find all occurrences of a pattern p is recorded in nanoseconds. This happens for patterns of length 2^n where $n = 1, \ldots, 16$, $pnum = 100$ and $pmin = 30$.

The harness provides access to the shift tables appropriate for a given *DZ* variant by invoking the corresponding version of a `static inline int function` that returns the required integer shift value.

4.7 Output Data

A file that captures the data of the benchmark run gets written for each variant of *DZ*. Hence, each file contains $16 \times 100 \times 30 = 48000$ run times for each *DZ* variant.

Given the large amount of output data generated by the benchmarking experiments, it was important that we had sanity checks to ensure that no error had occurred. Firstly, we confirmed that all of the algorithms found the same number of occurrences of p for all of the runs.

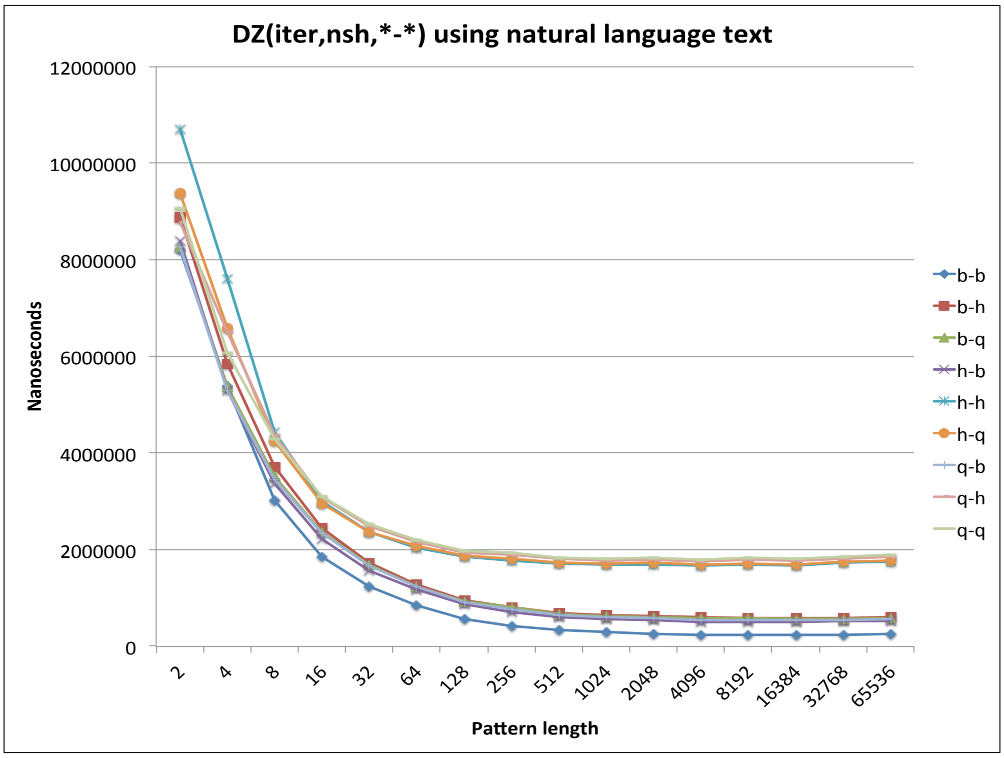
A second sanity check was based on the fact that theoretical time taken for a *DZ* algorithm to process a pattern p on a string S is given by $\mathcal{O}(\frac{|S|}{|p|})$ [37]. Therefore, one would anticipate a roughly positive linear relationship between processing time and the reciprocal of the pattern length when a fixed search text is used. Consequently, for each variant of *DZ* the (Pearson product-moment) correlation coefficient was computed between the two variables: the raw averaged minimum time and the inverse of the pattern length. The minimum coefficient value over all 36 values computed was 0.97, indicating a highly significant statistical correlation in all cases.

4.8 Overview of Results

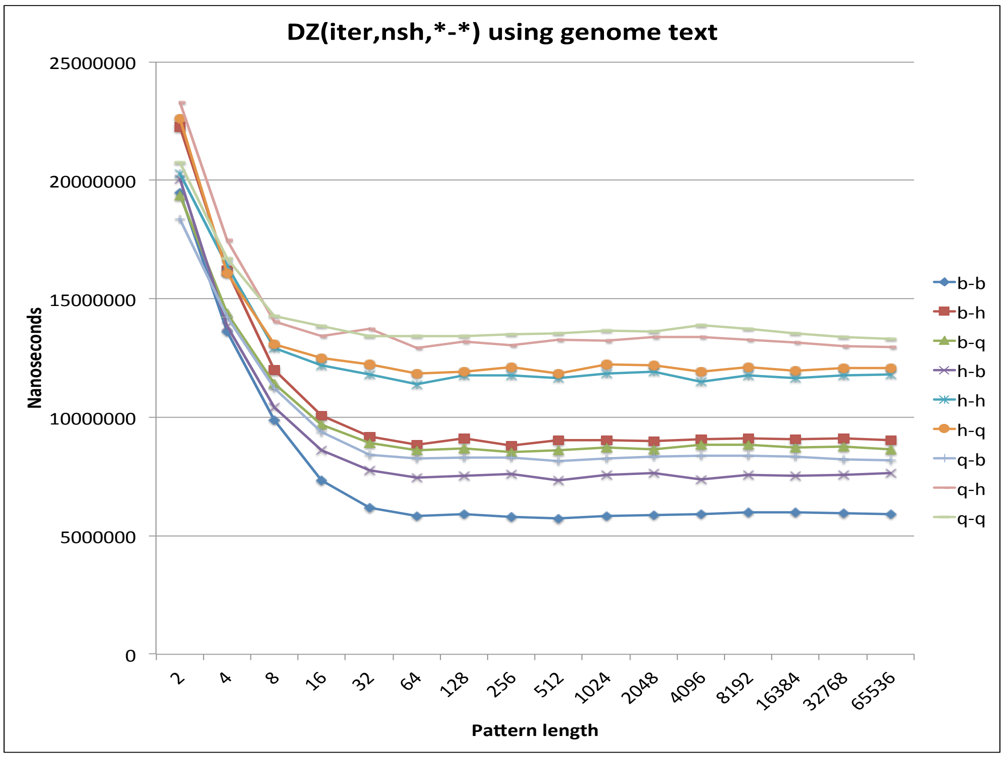
The graph in Figure 4.1 illustrates the general trend of the timing data. The minimum time of thirty runs with the same data averaged over one hundred different patterns of the same length is shown. Patterns were chosen randomly from the text.

Figure 4.1(a) depicts the time taken for each of the nine left-right shifter combinations of the *DZ(iter,sh,*-*)* algorithm to find all patterns in the natural language text. creffig:trendEcoliMultiShift shows related data corresponding to the genome text. All left-right shifter combinations perform better as pattern size increases, with the three *DZ(iter,nsh,*-b)* combinations being amongst the top performers. This is true for both alphabets. However, it should be noted that, although the general shape of the graphs are similar, the scales applied on the vertical axes of the respective subfigures are different. It is clear that the averaged minimum times of pattern matching using the genome alphabet (shown in Figure 4.1(b)) are significantly higher than the corresponding ones for natural language data (shown in Figure 4.1(a)). This is to be expected, as smaller alphabets typically result in smaller shifts.

The complete set of graphs for all multi shifter *DZ* variants can be found in the appendix.



(a) Natural Language: *DZ(iter,nsh,*-*)*



(b) Genomic Data: *DZ(iter,nsh,*-*)*

Figure 4.1: Illustrative raw averaged minimum time data of multiple shifter *DZ*

4.9 Assessing Berry-Ravindran Shifters

To determine the shift distance the *Horspool* and *Quick Search* algorithms both rely on one symbol of the search text and a one-dimensional array for the respective shift table [17][33]. Contrastingly, the *Berry-Ravindran* algorithm uses the two adjacent characters that occur just beyond the current window in the search text [8]. These two adjacent characters index into a two-dimensional array which implements its shift tables. It is therefore not surprising that DZ variants using *Berry-Ravindran* shifters generally deliver better time efficiency than their *Horspool* and *Quick Search* counterparts. However, the *Berry-Ravindran* shifters make use of a time and space trade-off—improved time efficiency is bought at the cost of space efficiency that depends quadratically rather than linearly on alphabet size.

In essence, the data relating to *b* shifters is essentially incommensurate with the data derived from the other two shifters, since the latter have not made the performance trade-off between time and space.

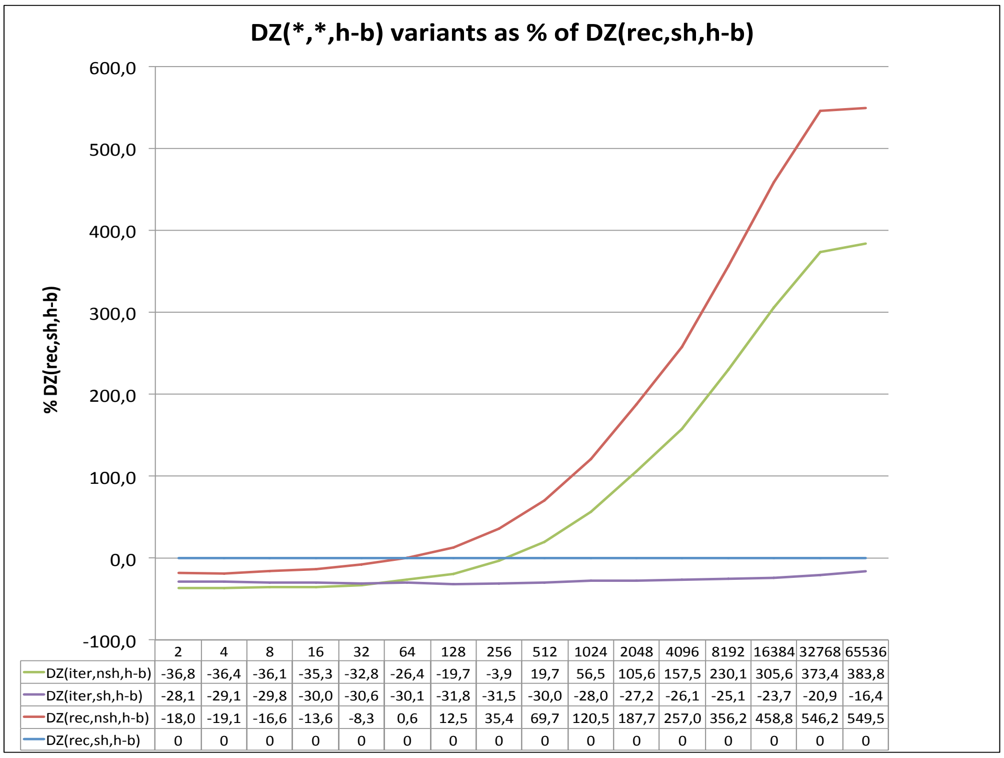
4.10 Cost of Recursion

The graphs for all the different shifter combinations are too numerous for all of them to be included here, however, it should be taken into account that they all have the same general trend. The graph for *DZ(*,*,h-b)* is shown in Figure 4.2(a), and is used to give an explanation of the data. It shows the impact of recursion using a natural language text, with *DZ(rec,sh,h-b)* used as the base line. Figure 4.2(b) illustrates equivalent data for the genome text. Note that the graphs for other shifter combinations can be found in the appendix.

When a natural language text is used, all variants of *DZ(iter,sh,*-*)* perform approximately 20% to 35% better than *DZ(rec,sh,*-*)*. The difference between *DZ(iter,nsh,*-*)* and *DZ(rec,nsh,*-*)* starts off relatively small at about 10%. However, as pattern size gets larger the difference increases to approximately 60% to 150%. Moreover, *DZ(iter,*,*-*)* variants are, in general, more efficient than *DZ(rec,*,*-*)* ones.

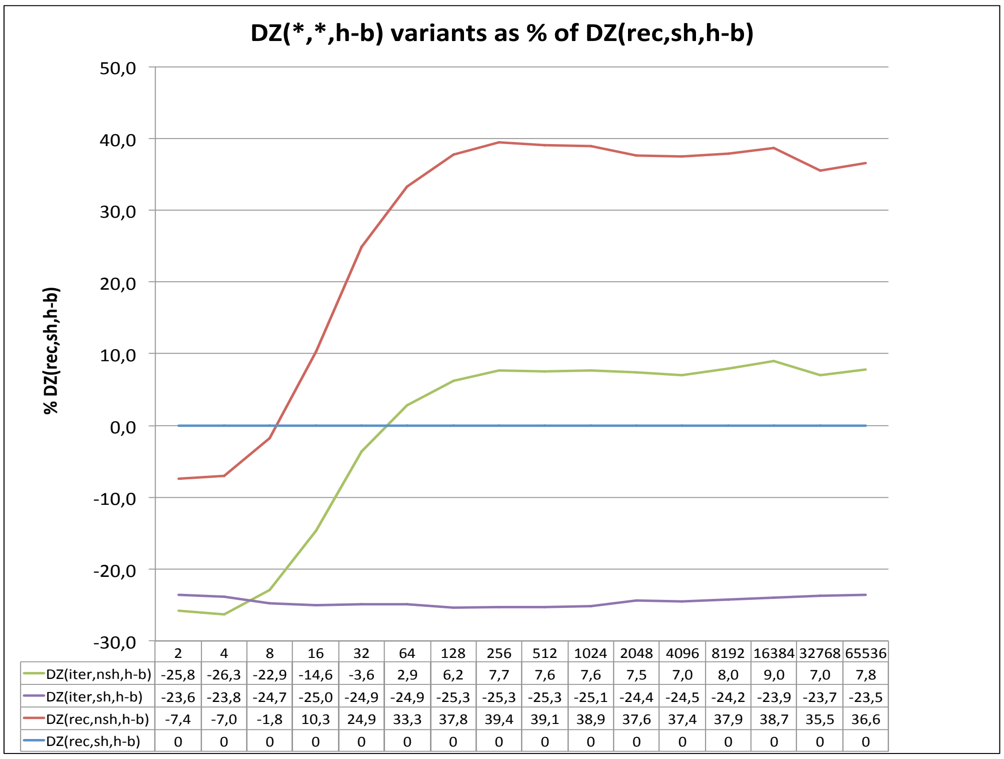
These findings are consistent with what is described in Section 3.11 (where many experiments were based on *DZ(*,*,h-h)* applied to a natural language text), despite the experimental design of the study described in Chapter 3 differing from this one in numerous ways—for example, different hardware was used. In both studies it was found that iterative *DZ* implementations are generally more efficient than recursive *DZ* implementations.

In an attempt to make sense of the vast amount of data that was gathered during the multiple shifter experiments, the genome data set was also analysed to compare iterative implementations with recursive implementations. Inter-



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,h-b)	-36,8	-36,4	-36,1	-35,3	-32,8	-26,4	-19,7	-3,9	19,7	56,5	105,6	157,5	230,1	305,6	373,4	383,8
DZ(iter,sh,h-b)	-28,1	-29,1	-29,8	-30,0	-30,6	-30,1	-31,8	-31,5	-30,0	-28,0	-27,2	-26,1	-25,1	-23,7	-20,9	-16,4
DZ(rec,nsh,h-b)	-18,0	-19,1	-16,6	-13,6	-8,3	0,6	12,5	35,4	69,7	120,5	187,7	257,0	356,2	458,8	546,2	549,5
DZ(rec,sh,h-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Cost of Recursion of *DZ(*,*,h-b)* using a natural language text



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,h-b)	-25,8	-26,3	-22,9	-14,6	-3,6	2,9	6,2	7,7	7,6	7,6	7,5	7,0	8,0	9,0	7,0	7,8
DZ(iter,sh,h-b)	-23,6	-23,8	-24,7	-25,0	-24,9	-24,9	-25,3	-25,3	-25,3	-25,1	-24,4	-24,5	-24,2	-23,9	-23,7	-23,5
DZ(rec,nsh,h-b)	-7,4	-7,0	-1,8	10,3	24,9	33,3	37,8	39,4	39,1	38,9	37,6	37,4	37,9	38,7	35,5	36,6
DZ(rec,sh,h-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Cost of Recursion of *DZ(*,*,h-b)* using a genome text

Figure 4.2: Cost of Recursion of *DZ(*,*,h-b)*

estingly, the results of the genome data set follow the same pattern as that of the natural language data set. All variants of *DZ(iter,*,*-*)* perform approximately 15% to 40% better than *DZ(rec,*,*-*)*. However, in the case of the genome data, the difference between *DZ(iter,nsh,*-*)* and *DZ(rec,nsh,*-*)* stays constant and does not become larger as pattern length increases.

This suggests that the cost of recursion is a characteristic of the *DZ* family of algorithms, and is irrespective of the alphabet and shifter being used.

4.11 Impact of Information Sharing

The impact of information sharing using a natural language text is also illustrated in Figure 4.2(a). Note that the sharing of data incurs a running-time penalty (discussed in Section 3.12). In Chapter 3, a pattern length of 64 characters was found to be the break-even point between sharing information and not sharing information. However, the results from this experiment imply that the break-even point is dependent on the left-shifter being used. The break-even points for the three left-shifter variants of *DZ(*,*,b-*)*, *DZ(*,*,h-*)* and *DZ(*,*,q-*)* are approximately 256, 64 and 32 characters respectively.

As mentioned in Section 4.10, the experimental design of this study differed from the one described in Chapter 3, yet these findings are also consistent with what is described in Section 3.12. For example, in both studies *DZ(iter,sh,h-h)* improves over *DZ(iter,nsh,h-h)* for pattern lengths greater than or equal to 64.

However, the same result does not apply to the genome data set, which is illustrated in Figure 4.2(b). Variants of *DZ(iter,sh,*-*)* generally perform better than *DZ(iter,nsh,*-*)* variants up to a pattern length of about 8 characters, although variants of *DZ(rec,sh,*-*)* and *DZ(rec,nsh,*-*)* do not follow a similar trend. Under certain circumstances *DZ(rec,nsh,*-*)* variants always perform better than *DZ(rec,sh,*-*)* variants by 5% to 15%. By contrast, at times *DZ(rec,sh,*-*)* performs worse than *DZ(rec,nsh,*-*)* for small pattern lengths but starts to perform better as pattern length increases, eventually beating *DZ(rec,nsh,*-*)* at a pattern length of about 8 characters.

This highlights a level of unpredictability of information sharing when matching is performed with a genome text and a recursive implementation of *DZ*. Nevertheless, the break even point between sharing information and not sharing information when using a genome text and an iterative *DZ* implementation is roughly expected to be at a pattern length of 8 characters.

4.12 Assessing Shifter Pairs

To get a sense of the trends and preferred options for the various shifter pair possibilities, we consider here the relative performance differences exhibited by the data for *DZ(iter,nsh,*-*)*, *DZ(iter,sh,*-*)*, *DZ(rec,nsh,*-*)* and *DZ(rec,sh,*-*)*. It should be noted that, as discussed in Section 4.9, the data relating to *b* shifters is incompatible with the data derived from the *h* and *q* shifters, since the Berry-Ravindran algorithm makes a performance trade-off between time and space. Therefore, the *b* shifter data has been excluded from this section.

1. **DZ(*,*,h-h) vs. DZ(*,*,h-q)**: Instead of using the *hh* shifter pair, it is always slightly better to use the *hq* shifter pair for all variants excluding *DZ(iter,nsh,*-*)*. The genomic data behaviour is nearly the same as the above and the differences between the two shifter pairs are also slight. However, for a pattern length of 2, the *hh* shifter pair should be used.

2. **DZ(*,*,h-h) vs. DZ(*,*,q-h)**: If pattern lengths are very short, the *qh* pairing has a marginal advantage over the *hh* pairing for all variants, although this advantage is lost as pattern length increases. The genomic data demonstrates a far more decisive picture: the *hh* pairing is significantly better than the *qh* pairing for all pattern lengths and all variants.

3. **DZ(*,*,h-h) vs. DZ(*,*,q-q)**: Neither of the shifter pairs has a clear advantage over the other for all patterns lengths. Rather, for short patterns, the *qq* shifter pair is better than the *hh* shifter pair, although this advantage is lost as pattern length increases. However, in the genome case the *hh* shifter pair is better than the *qq* shifter pair for all variants and for all patterns lengths (except for very short patterns in the case of non-sharing variants). Performance gains can be quite significant, sometimes manifesting differences in the region of 10 percentage points.

4. **DZ(*,*,h-q) vs. DZ(*,*,q-h)**: For all variants, excluding pattern lengths of 2, it is always slightly better to use the *hq* shifter pair instead of the *qh* shifter pair. The genomic data behaviour is almost identical to the above, except that the *hq* shifter pair is also superior for patterns of length 2, but the advantage is again marginal in terms of percentage point differences—the same order of magnitude as for the natural language data.

5. **DZ(*,*,h-q) vs. DZ(*,*,q-q)**: If pattern lengths are relatively short the *qq* pairing is the preferred option, but the *hq* pairing performs better on larger strings. In general, for a given variant, the performance differences between the two shifter pairs are relatively small, but the differences for *DZ(rec,nsh,*-*)* are fairly large. The genomic data reinforces these observations, manifesting larger percentage differences between the respective shifter pairs. For example, for long patterns, *DZ(iter,sh,q-q)* is more than 10 percentage worse off than *DZ(iter,sh,h-q)*.

6. **DZ(*,*,q-h) vs. DZ(*,*,q-q)**: Excluding *DZ(iter,nsh,*-*)* variants and also excluding the marginal case of *DZ(iter,sh,*-*)* with a pattern length of 2, it is always slightly better to use the *qq* shifter pair then the

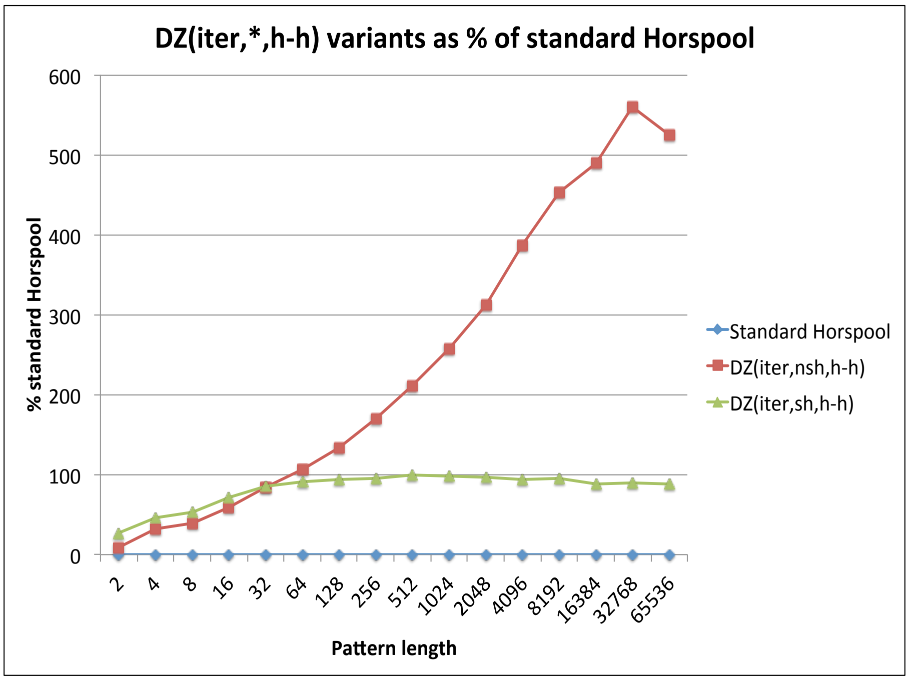
qh shifter pair. The genomic data behaviour is similar to the above in the case of recursive variants. However, for the iterative variants, the behaviour is only similar to the above for shorter patterns, while, for longer patterns, a *qh* shifter pair is slightly better—i.e. for $|p| > 4$ for *DZ(iter,sh,*-*)* and for $|p| > 64$ for *DZ(iter,nsh,*-*)*.

4.13 Comparison with Standard Versions

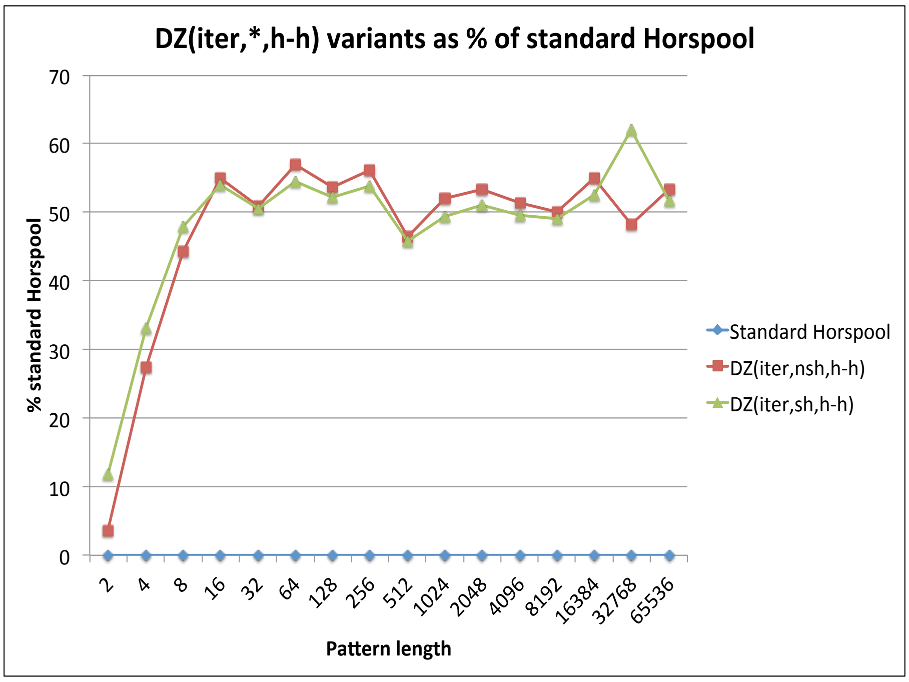
To determine whether using a *DZ* algorithm with the same shifter to shift both to the left and to the right in the text performs better than the equivalent standard version of the algorithm, we compare the performance of *DZ(iter,*,b-b)*, *DZ(iter,*,h-h)* and *DZ(iter,*,q-q)* to the performance of the standard implementation of the corresponding algorithm (i.e. the implementations of*BM*, *Horspool* and *QS* found in Appendix A. This can be seen in the graphs in Figure 4.3, which illustrates the performance of *DZ(iter,nsh,h-h)* and *DZ(iter,sh,h-h)* relative to performance of the standard *Horspool* implementation. It was shown in Section 4.10 that *DZ(rec,nsh,h-h)* and *DZ(rec,sh,h-h)* perform worse than their iterative counterparts, and have therefore been excluded from the graphs. Note that similar graphs for *BR* and *QS* can be found in the appendix.

Figure 4.3(a) shows the results when a natural language text is used. It is evident that *DZ(iter,nsh,h-h)* is approximately 10% slower than the standard *Horspool* implementation when patterns are of length 2. However, the performance of *DZ(iter,nsh,h-h)* gets worse as pattern length increase, until it peaks at 560% slower at a pattern length of 32768. With a pattern length of 65536, it performs marginally better, but is still 520% slower than standard *Horspool*. Similarly, *DZ(iter,sh,h-h)* is 30% slower than the standard *Horspool* implementation with a pattern length of 2 and performs worse as pattern length increases, until patterns reach a length of 256 characters. Then, its performance stays consistent at approximately 100% slower than standard *Horspool*.

Although the performance of the two *DZ(iter,*,h-h)* variants when matching a genome text, which can be seen in Figure 4.3(b), is not quite as poor, it is still disappointing. *DZ(iter,nsh,h-h)* is about 4% slower than the standard *Horspool* implementation at a pattern length of 2, and is approximately 28% slower at a pattern length of 4. It performs consistently between 45% and 55% slower than standard *Horspool* for patterns longer than 4 characters. Similarly, *DZ(iter,sh,h-h)* is 11% slower than standard *Horspool* for a pattern length of 2, and 23% slower for patterns with a length of 4 characters. Then, for pattern lengths larger than 4, it has a similar behaviour to *DZ(iter,nsh,h-h)*, and performs consistently between 46% and 61% slower than the standard *Horspool* implementation.



(a) *DZ(iter,*,h-h)* compared to standard *Horspool* using a natural language text



(b) *DZ(iter,*,h-h)* compared to standard *Horspool* using a genome text

Figure 4.3: *DZ(iter,*,h-h)* compared to standard *Horspool*

4.14 Conclusion

For all of the algorithms used in this study, the standard implementation of the algorithm performed better than the corresponding *DZ* implementation with the same left and right shifter. In fact, there is only one case when the standard implementation does not beat the corresponding *DZ* implementation: For a pattern length of 65536, *DZ(iter,sh,b-b)* performs about 8% better than the standard *BR* implementation.

Although the standard implementations of the algorithms are generally faster than the multiple shifter *DZ* implementations, it is evident, however, that certain combinations of shifters perform better than others.

Furthermore, this study established that the cost of recursion and the impact of information sharing are intrinsic *DZ* characteristics that are, in general, independent of the shifter being used.

This study also taught us about the importance of choosing the correct algorithms for the experiment. In particular, selecting the *Berry-Ravindran* algorithm for this study meant that a large chunk of our data was not compatible with the rest of the data. It was significant to realise that not all pattern matching algorithms are comparable.

Chapter 5

Parallel Dead-Zone

5.1 Introduction

Chapter 2 explains that the recursive version of the *DZ* algorithm has two recursive calls: one call spawns a left live-zone, and a later call spawns a right live-zone. There is no interaction between these two recursive calls. This presents the recursive *DZ* algorithm as a good candidate for parallelisation. According to Amdahl's Law [2], if S is the fraction of a calculation that is serial and $1 - S$ the fraction that can be parallelised, then the greatest speedup that can be achieved using P processors is: $\frac{1}{(S+(1-S)/P)}$. Therefore, we know that the greatest speedup that can be achieved using 2 processors on code that is 20% sequential is 1.6.

This chapter aims to determine whether a parallel version of *DZ* can be implemented, and also, if it can be implemented, then what performance increase can be achieved.

First, two types of parallel implementations of *DZ* are introduced and the details of their experiments discussed, including the data set used as well as the implementation details. Then, the data resulting from the CUDA benchmark experiments are given in a table. The results of the Pthreaded experiments are presented, which is followed by a discussion of the CUDA benchmark results.

5.2 Experimental Design

The parallelism experiments were carried out in two distinct stages that made use of different parallelisation techniques, each producing unique parallel *DZ* implementations.

Because a C implementation of *DZ* exists, it was possible to use POSIX threads (Pthreads)[27], one of the most efficient models of parallelisation in C [7]. In the first stage, the C implementation of *DZ(rec,nsh)* discussed in Chapter 3 was modified to create three threaded versions of *DZ*. This was achieved by utilising POSIX threads to parallelise the two recursive calls of *DZ(rec,nsh)* in the following ways:

- Both recursive calls are parallelised. Two threads get spawned to perform pattern matching in the new left live-zone and the new right live-zone.
- The left recursive call is parallelised. A thread is spawned that performs pattern matching in the new left live-zone. The recursive call to create a right live-zone remains unchanged, and pattern matching in the right live-zone continues with the current thread.
- The right recursive call is parallelised. A thread is spawned that performs pattern matching in the new right live-zone. The recursive call to spawn a left live-zone remains unchanged, and pattern matching the left live-zone continues with the current thread.

When recursion is used, the parent process has to wait for its children processes to complete. Rather than having the parent wait for its children, it seemed sensible to spawn a new thread and let the parent thread continue with some processing. In the cases where one recursive call was modified to spawn a new thread, the parent thread would continue pattern matching the live-zone that did not get processed by the new thread. Thus, thread re-use was achieved.

The Pthreaded experiment was carried out on a 2011 model MacBook Pro with the following specifications:

- Operating System: Mac OS X version 10.7.4
- Processor: Intel Core i7
- Processor speed: 2.8 GHz
- Number of cores: 2
- L2 Cache (per core) : 256 KB
- L3 Cache (per core): 4 MB
- Memory: 4 GB, 1333 MHz, DDR3

The executables for the Pthreaded benchmark experiment were compiled with Xcode 4.2 into Release builds, which corresponds to -O3 optimisation on GCC and most other compilers. The benchmark tests were performed on two CPU cores with hyper-threading disabled.

It should be noted that the benchmark experiments involving the Pthreaded versions of *DZ(rec,nsh)* occurred prior to the experiments examined in Chapter 2 and Chapter 4, thus it was not possible to apply what was learned in those studies.

The second stage involved developing an implementation of *DZ(rec,nsh)* that would execute on a Graphics Processing Unit (GPU) using the CUDA parallel computing platform [28]. As discussed in Chapter 3, the DZ algorithm was implemented in C code and benchmarked in a C++ environment, therefore the original code and testing harness would be compatible with CUDA [20]. To execute CUDA code, a computer must have an NVIDIA GeForce, NVIDIA Quadro or NVIDIA Tesla graphics card [29].

A hybrid version of *DZ* that makes use of both iteration and recursion was implemented with C code and CUDA. This hybrid DZ was benchmarked against a CUDA implementation of the *Horspool* algorithm as well as the C implementation of *Horspool* from Chapter 3, executed on the CPU.

The CUDA implementation was developed iteratively, with the results of each benchmark test bringing about code changes in an attempt to optimise the CUDA code. Therefore, two rounds of experimentation (each consisting of multiple benchmark tests) occurred in order to benchmark the CUDA implementations. First, the experiments were performed on a 2012 model MacBook Pro with the following configuration:

- Operating System: Mac OS X version 10.9.5
- Processor: Intel Core i7
- Processor speed: 2.6 GHz
- Number of cores: 4
- L2 Cache (per core): 256 KB
- L3 Cache (per core): 6 MB
- Memory: 8 GB, 1600 MHz, DDR3

This MacBook Pro had the CUDA driver version 6.0 and the CUDA runtime version 6.0 installed, as well as an NVIDIA GeForce GT 650M graphics card with these specifications:

- CUDA cores: 384 cores
- Multiprocessors: 2
- Global memory: 1024 MB
- Constant memory: 65536 bytes
- Shared memory per block: 49152 bytes
- Warp size: 32
- Maximum number of threads per processor: 2048
- Maximum number of threads per block: 1024

The performance of the CUDA version of DZ was disappointing when executed on the MacBook Pro. Therefore, a second round of experiments were conducted on a desktop computer with a better GPU and the following specifications:

- Operating System: Fedora 20
- Processor: Intel Core i3
- Processor speed: 3.3 GHz
- Number of cores: 2
- L2 Cache (per core): 256 KB
- L3 Cache (per core): 3 MB
- Memory: 8 GB, 1333 MHz, DDR3

The desktop computer had the CUDA driver version 6.0 and the CUDA runtime version 6.0 installed, as well as an NVIDIA GeForce GTX 460 graphics card with these specifications:

- CUDA cores: 336 cores
- Multiprocessors: 7
- Global memory: 1073 MB
- Constant memory: 65536 bytes
- Shared memory per block: 49152 bytes
- Warp size: 32
- Maximum number of threads per processor: 1024
- Maximum number of threads per block: 1024

The executables were compiled with the NVCC compiler driver, which calls the GCC compiler for C code and the NVIDIA PTX compiler for the CUDA code. Optimisation level O3 was specified for the C code.

5.3 The Data

The data set used to conduct the Pthreaded DZ pattern matching experiments was similar to that used in the experiments discussed in Chapter 2 and Chapter 4. Two texts from the SMART corpus, a natural language text (the Bible) and a genome text, were utilised for the benchmark experiments. Patterns were randomly selected from the text for pattern lengths of 2^n characters, where $n = 2, \ldots, 12$.

The natural language and genomic data sets were also used during the CUDA benchmark experiments. Patterns were randomly selected from the text for pattern lengths of 2^n characters, where $n = 1, \ldots, 10$. Furthermore, one supplementary text was used to additionally determine the performance of the algorithms in the best case and in the worst case. This text consists of 4 MB worth of the letter *a*. Because a pattern consisting entirely of the letter *b* cannot be found in a text containing only the letter *a*, the best case involved matching patterns that were only comprised of the letter *b*. Conversely, a pattern consisting entirely of the letter *a* will match every character of a text containing only the letter *a*, thus the worst case involved matching patterns that were only comprised of the letter *a*.

5.4 Implementation

In the Pthreaded *DZ* version, all threads other than the main thread are explicitly created and passed the parameters necessary to process the new live-zone—i.e. the boundary information. The C `struct` data structure is used to setup and pass multiple parameters to the threads, and each thread receives a unique instance of the `struct`. Moreover, for each thread to receive the correct parameters it was important to keep track of the thread IDs with a thread ID counter.

All threads perform the same function, therefore all threads could potentially spawn other threads. One consequence of this it that a mutex lock is needed to manage thread creation, which could negatively affect performance due to lock contention [34]. The thread ID counter is locked whenever a `struct` for a new thread is constructed with the correct parameters and while a new thread is being created. Likewise, the mutex lock is unlocked after the newly created thread has been created with the correct parameters. This prevents two threads from creating new threads at the same time and their initialisation parameters getting mixed up.

Whenever a new thread is spawned it is created using the `pthread_create` subroutine provided by the Pthreads standard API [13]. Furthermore, no upper bound is given to the number of threads that could be spawned.

All threads have read access to the pattern being matched and the text to be searched for possible matches. During the experiments discussed in Chapter 3 with the non-threaded *DZ(rec,nsh)*, the compiler optimised out all code that does not produce a side effect, and we assumed that this compiler optimisation would also be apparent in the threaded version. Therefore, all threads have write access to a global variable that keeps track of the number of matches across all the threads for a given run. However, this meant that an additional mutex lock would be necessary to lock the global count variable whenever a thread wants to read from or write to the count variable. Similarly, the count variable also has to be unlocked when the thread is done reading from or writing to the variable. The addition of the extra lock could possibly result in even greater lock contention.

POSIX makes use of the most common style of parallel programming, the Single Program, Multiple Data (SPMD) parallel programming model [31]. This is in contrast to CUDA, which uses the Single Instruction, Multiple Data (SIMD) parallel programming model [28]. This means that a single function (known as a kernel in CUDA) performs the same action simultaneously on multiple pieces of data. Consequently, the CUDA programming paradigm is different to that of POSIX, and the two implementations of parallel *DZ* are unique.

The CUDA implementation is a hybrid version of *DZ* in that the solution makes use of both the recursive and iterative *DZ* algorithms. The C implementation of *DZ(rec,nsh)* from Chapter 3 is reused, and only *DZ(iter,nsh)* and *DZ(iter,sh)* are implemented as CUDA kernels that get executed in parallel on the GPU. Pattern matching is performed on the entire text using a divide and conquer method.

When pattern matching begins, the C function of *DZ(rec,nsh)* is called. However, instead of recursively pattern matching the whole text, the recursion stops once a certain recursion depth has been reached. This results in the text being broken up into smaller chunks of text (all of which are live-zones still to be searched). The entire text is copied to the GPU, as well as the live-zone boundary information. A CUDA kernel function is called that will spawn the specified number of threads to execute the iterative *DZ* function on the GPU. To determine which chunk of the text to match, each thread uses the boundary information copied to the GPU. Pattern matching the whole text has concluded once all threads have completed pattern matching their chunk of text.

In addition to the *DZ(rec,nsh)* partitioning function, a method that simply divides the text into equal-sized sections was also implemented in C code such that the best technique for dividing the text into chunks could be determined. In Section 5.8 this method is referred to as the standard partitioning technique. It should be noted that partitioning the text with *DZ(rec,nsh)* is the default partitioning technique used, unless otherwise stated.

Using the `cudaMalloc` function, GPU memory is allocated for the `int` array of live-zone indices, the global match count `int` array (each thread stores its count in one entry in the array), the `unsigned char` array storing the text and the `unsigned char` array for the pattern. The data is then transferred to the GPU and stored in the corresponding GPU memory with `cudaMemcpy`, which copies the data to global memory on the GPU. The left shift table and the right shift table are copied to constant memory on the GPU with the `cudaMemcpyToSymbol` function. Once pattern matching has completed, the GPU copies the global match count data back to the host program running on the CPU. This serves as a sanity check to ensure that all algorithms find the same number of matches.

The high resolution timer discussed in Section 3.6 was used for benchmarking the Pthreaded experiments as well as the recursive step executed on the CPU in the CUDA experiments conducted on the MacBook Pro. However, the `mach_timebase_info` API is only compatible with Apple computers and could not be used for the experiments conducted on the desktop computer running a Fedora operating system. Consequently, a Linux version of a high resolution timer was implemented using a monotonic clock and `clock_gettime()` [30]. The high resolution CPU timer is started immediately before the first recursive function is called and is stopped once the desired recursion depth has been reached and the recursive stack is empty.

In addition to the high resolution CPU timer, a `cudaEventRecord` timer was implemented to capture the time taken for all the threads in one run to execute the parallel iterative step on the GPU. Using the specialised CUDA event API ensures that host-device synchronisation time is not recorded [16]. The `cudaEventRecord` timer is started immediately prior to the CUDA kernel being called, and is stopped as soon as all the threads on the GPU associated with this kernel have finished executing.

During one run with the same pattern, recursively splitting the text with the same depth of recursion will always result in the chunks of text being identical. Therefore, the recursive function is only called once per run and the CUDA implementations of *DZ(iter,nsh)* and *DZ(iter,sh)* both make use of the same chunks of text produced by the recursive function. The time taken to recursively split the text is recorded and added to the CUDA kernel timing data for both iterative *DZ* implementations. The overall time taken to perform pattern matching is the sum of the time taken to perform the recursive step and the time taken for all threads to perform the iterative step, excluding the the preprocessing step.

5.5 Test Procedure

The benchmarking platform that is described in Section 3.4 and used to benchmark the non-threaded *DZ(rec,nsh)* was also used for benchmarking the version of *DZ(rec,nsh)* with Pthreads. The benchmark tests compared the C implementation of *DZ(rec,nsh)* as described in Chapter 2 to the three threaded variants of *DZ(rec,nsh)*. The benchmark time is the average of the 30 minimums over 100 runs.

Benchmark tests for the Pthreaded version were also conducted to determine the total time taken to create a new thread. Furthermore, the Pthreaded version was benchmarked with the high resolution timer detailed in Section 3.6, which uses the Mach absolute time unit. Consequently, over 30 runs and for every thread, the `mach_absolute_time` was captured immediately before and immediately after a Pthread was created. Additionally, the number of active threads for a given a run were also recorded. This was achieved by recording the number of currently active threads every time a thread was created. In this context, an active thread is a thread that is currently alive, either performing pattern matching or waiting for a child thread to complete.

The same benchmarking platform was also used to benchmark the CUDA implementation. Initially, the benchmark tests compared the performance of a CUDA implementation of *DZ(iter,nsh)* executing on the GPU to a CUDA implementation of the Horspool algorithm also running on the GPU as well as a C implementation of Horspool running on the CPU. However, in subsequent tests a CUDA implementation of *DZ(iter,sh)* was also benchmarked. The benchmark time is the average of the 30 minimums over 100 runs.

For the benchmark experiments executed on the MacBook pro, the recursive step executed until a recursion depth of 10 was reached, splitting the text up into 1024 live-zone chunks. The total size of all the texts used in the benchmark experiment is approximately 4 MB. Because this was divided into 1024 chunks, each chunk was about 4 KB depending on the dead-zone that was generated. The CUDA architecture is built around Streaming Multiprocessors (SMs). When a program running on the CPU host calls a CUDA kernel, the blocks of the grid are distributed to unused multiprocessors. One block is executed on a multiprocessor at a time, and the threads in the block execute concurrently on the same multiprocessor. Each thread was allocated one live-zone chunk to pattern match, thus a total of 1024 threads were executed on the GPU. The threads were divided into two blocks of 512 threads each and each block was executed on a different multiprocessor, thus utilising both multiprocessors of the GPU.

The benchmark experiments executed on the more powerful computer executed the recursive step until a recursion depth of 12 was reached. This split the text up into 4096 chunks, each with a size of approximately 1 KB, with one thread per chunk. The threads were divided into four blocks of 1024 threads each, thus utilising four multiprocessors of the GPU and leaving three multiprocessors unutilised. Because a maximum of 1024 threads can be executed concurrently on a multiprocessor on this particular GPU, this is the maximum recursion depth and multiprocessor utilisation that can be achieved for this GPU. The optimal number of GPU threads and blocks are highly dependent on the algorithm and the GPU. For example, a recursion depth of 13 can be achieved on a GPU with 8 multiprocessors, which splits the text into 8192 chunks and, with one thread per chunk, 8 blocks of 1024 threads each could be executed simultaneously on 8 multiprocessors.

Lastly, multiple recursion depths were tested to determine the effect of recursion depth (and therefore the number of threads and the size of the chunks of text) on the performance of the hybrid *DZ* CUDA implementation. A benchmark experiment was conducted with recursion depths of 3, 6, 9 and 10.

5.6 Output Data

An overview of the benchmarking data that was captured for this study is presented in Table 5.1. Twelve different benchmark experiments were conducted using the testing harness. Each benchmark test generated a separate set of data that was captured in its own file. In total, 87 MB of raw data was stored and analysed. As described in the preceding sections, the resulting data was used to change the CUDA DZ implementations over the course of the study.

Benchmark Number	**Text**	**Max Pattern Length**	**Recursion Depth**	**Description of Data**
1	Bible and ecoli	65536	9 and 10	Initial tests with one block on one GPU multiprocessor.
2	Bible and ecoli	65536	9 and 10	Two blocks on two GPU multiprocessors with shifters in shared thread memory.
3	Bible and ecoli	65536	9 and 10	Two blocks on two GPU multiprocessors with shifters in thread local memory.
4	Bible and ecoli	65536	9 and 10	Two blocks on two GPU multiprocessors with shifters in constant thread memory.
5	Bible and ecoli	65536	9 and 10	Two blocks on two GPU multiprocessors with shifters in constant thread memory using the `cudaEventRecord` timer.
6	Bible and ecoli	512	10	Two blocks on two GPU multiprocessors with shifters in constant thread memory using the `cudaEventRecord` timer, preprocessing time included.
7	Bible and ecoli	512	10	Two blocks on two GPU multiprocessors with shifters in constant thread memory using the `cudaEventRecord` timer, excluding preprocessing time.
8	Bible and ecoli	512	10	The same as 7 executed on the desktop computer.

9	Bible and ecoli	2048	10	Unoptimised desktop computer experiments with larger patterns.
10	Bible	256	3, 6, 9, 10	Unoptimised desktop computer experiments with different recursion depths.
11	Bible and ecoli	256	10	Unoptimised desktop computer best case and worst case experiments.
12	Bible and ecoli	1024	12	Optimised desktop computer with best case, worst case and average case experiments.

Table 5.1: Overview of captured CUDA data

5.7 Pthreaded Dead-Zone Results

The performance gains that we expected to achieve by parallelising the *DZ* algorithm with Pthreads was not apparent in the benchmarking data. Figure 5.1 illustrates the benchmark time for finding all patterns in the genome text with the *DZ(rec,nsh)* algorithm as well as the three Pthreaded variants. It is evident that *DZ(rec,nsh)* always performs better than the Pthreaded implementations. The performance of using a thread for the left recursive call is similar to the performance when threading the right recursive call. The *DZ(rec,nsh)* variant with both sides threaded only performs better than the left-threaded and right-threaded *DZ* variants with a pattern size of 4 characters, and the performance deteriorates with larger patterns. Similar results were observed with the experiments involving the natural language text.

Removing the global match count and, consequently, one of the mutex locks from the code did not have much effect on the performance of the Pthreaded variants of *DZ(rec,nsh)*. Figure 5.2 shows the benchmark time for pattern matching the genome text with the *DZ(rec,nsh)* algorithm and the three Pthreaded variants of *DZ(rec,nsh)* without a global counter to keep track of the number of matches. Our assumption about code optimisation also being apparent in the threaded version was incorrect. Code that did not produce a side effect was not optimised, and there was no need to keep track of the number of matches.

Figure 5.3 shows the number of active threads during one benchmark run of *DZ(rec,nsh)* with both sides threaded. The number of active threads quickly increases to 2048 and then plateaus until threads start terminating as the pattern matching comes to an end. The number of active threads during a

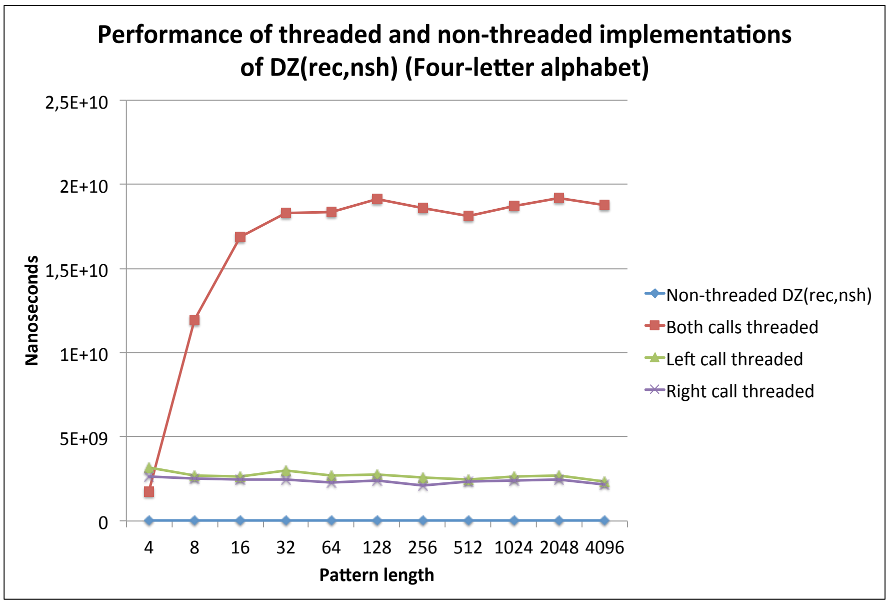


Figure 5.1: Raw averaged minimum time data with Pthreads and a genome text

benchmark run presented similar behaviour for all patterns sizes and for all three versions of Pthreaded *DZ(rec,nsh)*. In this study no upper bound for the number of threads that could be spawned was given. The maximum number of 2048 threads has been introduced by the hardware on which the benchmark tests were performed.

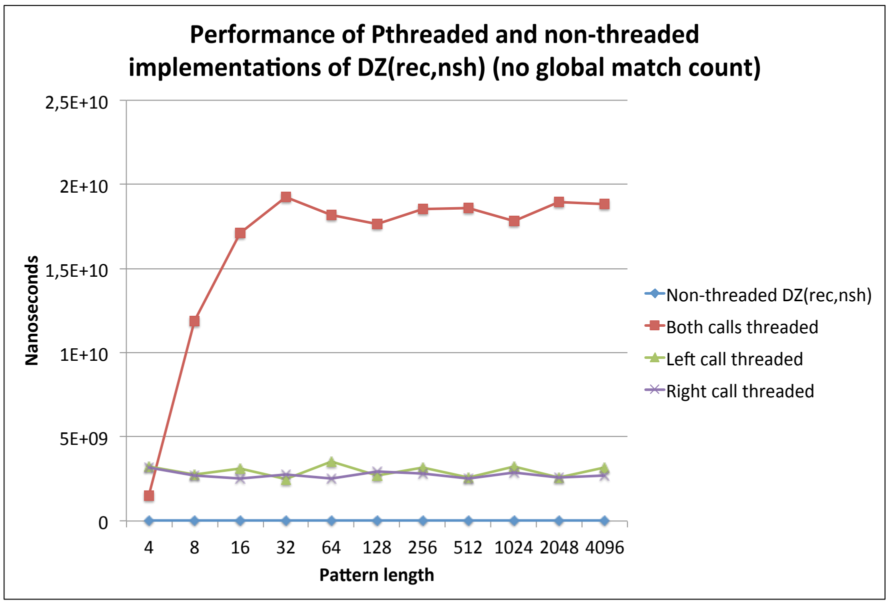


Figure 5.2: No compiler optimisations with Pthreads

For each run, although a maximum of 2048 threads were only ever active at one time, a very large number of threads were created—more than eighteen thousand, in fact. There is an overhead associated with thread creation, and our data shows a wide range of time taken to create threads in our benchmark runs. The minimum time required to create a thread was 1646 nanoseconds, whereas the maximum time was 31230780 nanoseconds. However, 50% of the recorded times fell between 3583 nanoseconds and 140144 nanoseconds. This extremely wide range of time required to create threads highlights the unpredictabilities brought on by the operating system that are associated with multithreading.

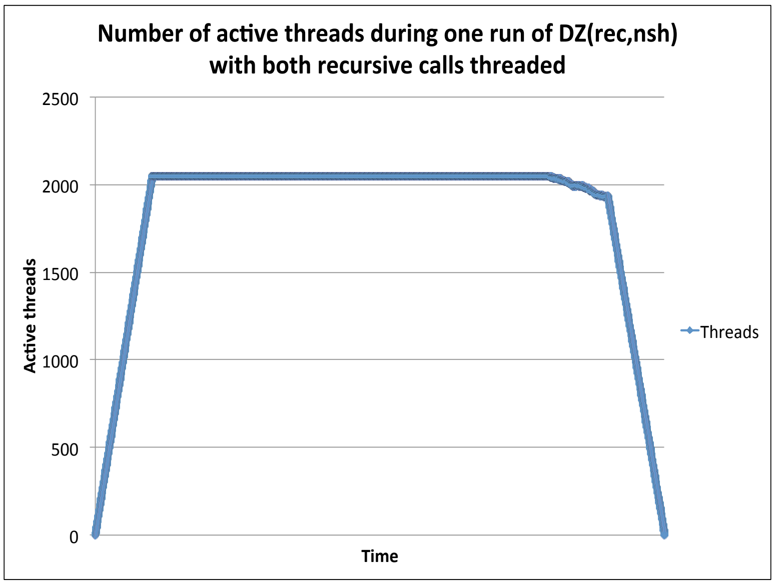


Figure 5.3: Number of active threads

5.8 CUDA Dead-Zone Results

Figure 5.4 displays the time taken for identical implementations of the *DZ(iter,nsh)* CUDA code to pattern match the natural language text on the GPU of the MacBook Pro and the GPU of the desktop PC. In both cases, a recursion depth of 10 was used and 1024 chunks of text were matched on two blocks of 512 threads. Note that exactly the same code was executed on both GPUs; the code made use of only two GPU multiprocessors and was not optimised for the desktop computer with better specifications, yet the two GPUs performed differently. The GPU of the MacBook Pro performed consistently worse than the GPU of the desktop computer. Consequently, subsequent benchmark tests were all executed on the desktop computer.

The performance of the hybrid *DZ(iter,nsh)* CUDA implementation with *DZ(rec,nsh)* partitioning with a recursive depth of 10 compared to the performance of the the hybrid *DZ* CUDA implementation using standard partitioning to split the text into 1024 chunks is shown in Figure 5.5. Our data shows that splitting the text with *DZ(rec,nsh)* takes slightly longer than standard partitioning, but results in quicker pattern matching performed on the GPU with the iterative

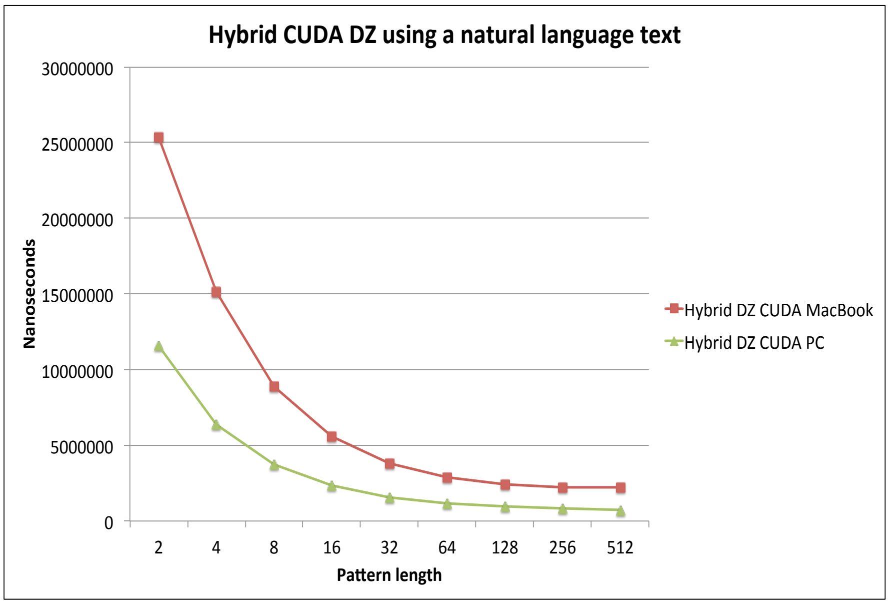


Figure 5.4: Raw averaged minimum time data of CUDA *DZ*

DZ CUDA kernel because dead-zones have already been created in the text. However, the total pattern matching times using both partitioning methods are extremely similar and both times are remarkably faster than the *Horspool* algorithm executed on the CPU. Similar results are observed for the genome data set.

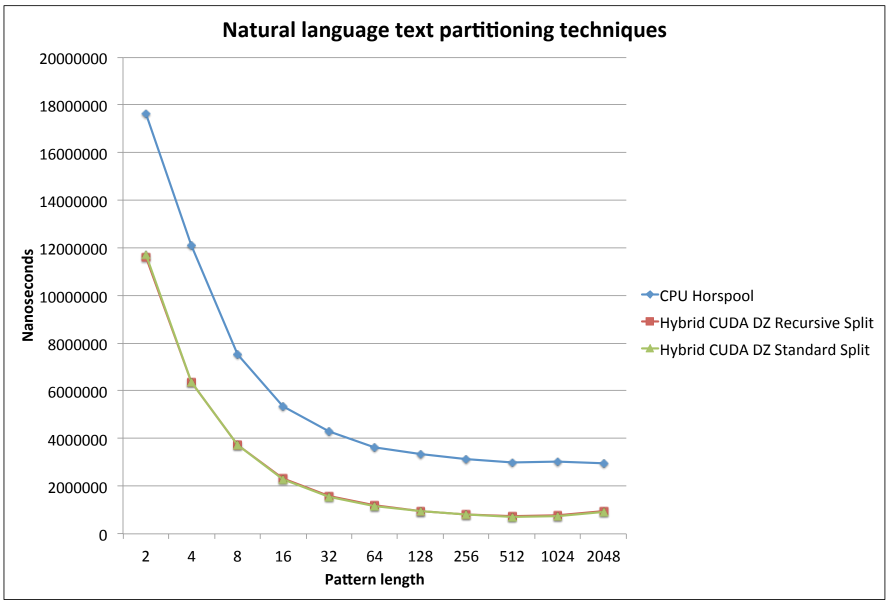


Figure 5.5: Splitting the text with *DZ(rec,nsh)* versus division into equal-sized chunks

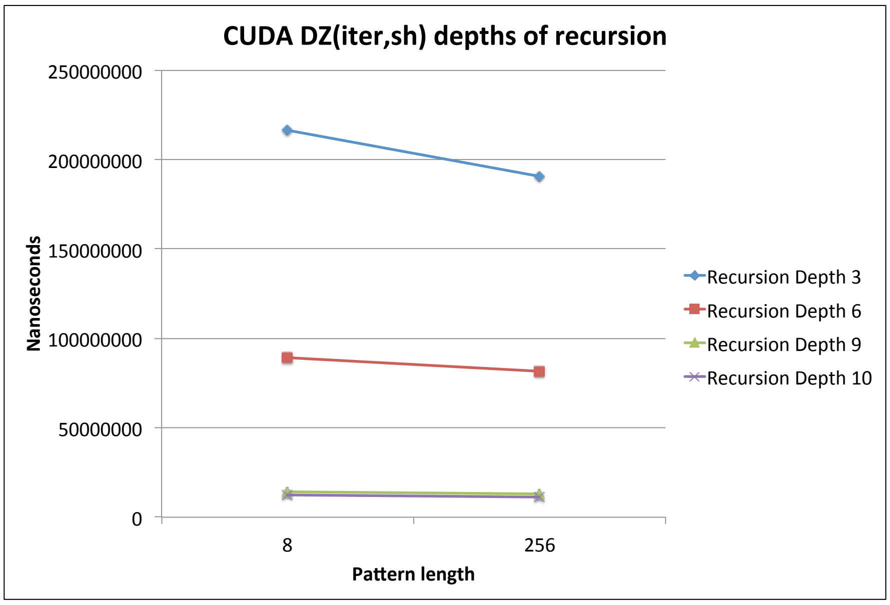
The performance of the hybrid *DZ(iter,nsh)* CUDA implementations with *DZ(rec,nsh)* partitioning and varying depths of recursion and a natural language text are illustrated in Figure 6.1. Different depths of recursion result in a different number of chunks of text to be search and, consequently, a different number of GPU threads performing the iterative pattern matching on the chunks of text. Figure 5.6(a) shows the performance of the CUDA implementation of *DZ(iter,sh)*, whereas Figure 5.6(b) shows the performance of the CUDA implementation of *DZ(iter,nsh)*. Due to the experiments taking a long time to complete, especially for shorter recursion depths, they were only conducted for shortish patterns (8 characters in length) as well as slightly longer patterns (256 characters in length). As recursion depth increases, it is evident that the performance of both iterative *DZ* CUDA implementations improves for both pattern lengths that were tested.

Figure 5.7 presents selected graphs showing the best case, worst case and average case performance of hybrid *DZ* CUDA implementations compared to a CUDA implementation of *Horspool* as well as a C implementation of *Horspool* executed on the CPU with a natural language text. The code has been optimised for the desktop PC such that it uses 4 multiprocessors of the GPU with a recursion depth of 12 and 4096 chunks of text running on 4096 GPU threads. It should be noted that the vertical axes of the subfigures make use of a logarithmic scale. Also, each subfigure is drawn to a different scale and should not be compared against one other.

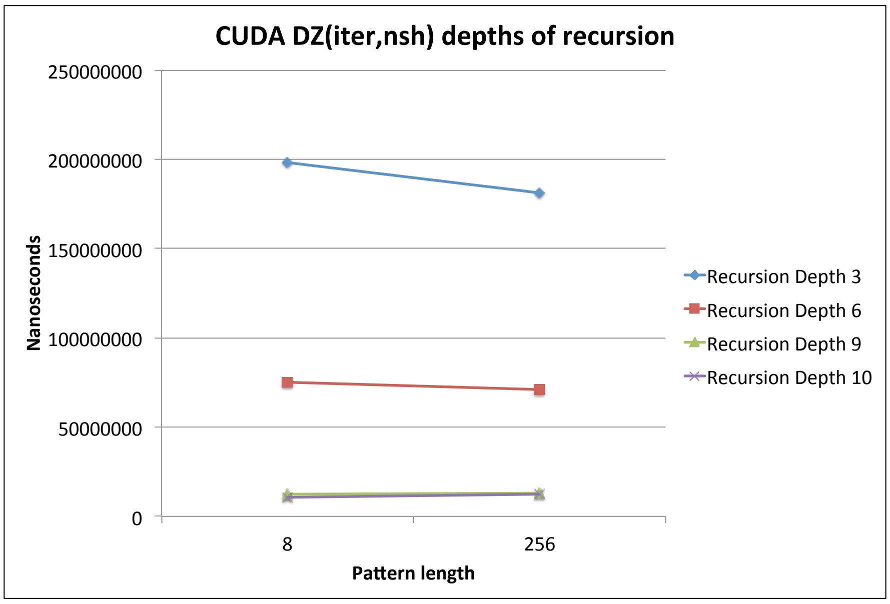
Shown in Figure 5.7(a), for a pattern length of 2, the CUDA *DZ* implementations have a very similar best case, worst case and average case performance. The CUDA implementation of *Horspool* is clearly the best performing algorithm, with its best case performance approximately 1.5x better than the best case performance of both CUDA *DZ* implementations. Also, the worse case performance of CUDA *Horspool* is almost the same as the best case performance of both CUDA *DZ* implementations. The CPU *Horspool* implementation performs the worst, with its best case performance more than 2x slower than the best case performance of the CUDA *DZ* implementations. Furthermore, the performance of the CPU *Horspool* and CUDA *Horspool* algorithms are not as consistent as the CUDA *DZ* implementations, as depicted by the larger box plots.

For patterns of length 8, Figure 5.7(b) illustrates that the performance of both CUDA *DZ* implementations is again similar, and the best case performance for both algorithms is slightly better than the best case performance of CUDA *Horspool* and almost 4x faster than the CPU *Horspool* implementation. However, the average case performance of CUDA *Horspool* is still better than the average case performance of both CUDA *DZ* implementations, although the average case CUDA *DZ* implementations are also 4x faster than the average case CPU *Horspool* implementations. Somewhat surprisingly, the worst case performance of the CPU *Horspool* implementation is the best. The range between the best case performance and worst case performance of all the algorithms are larger when compared to the results with patterns of length 2.

The range increase between the best case performance and worst case perfor-



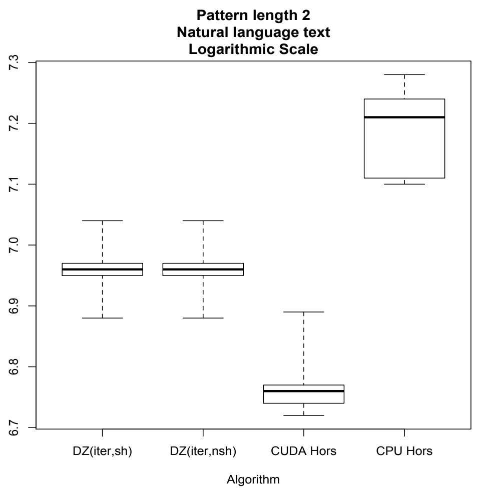
(a) Performance of *DZ(iter,sh)* CUDA implementation



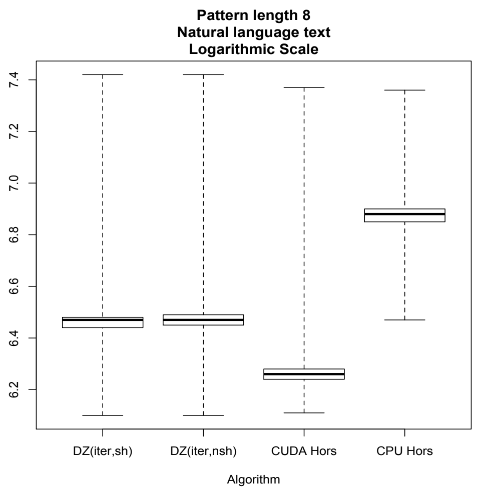
(b) Performance of *DZ(iter,nsh)* CUDA implementation

Figure 5.6: Impact of recursion depth on iterative *DZ* CUDA implementation

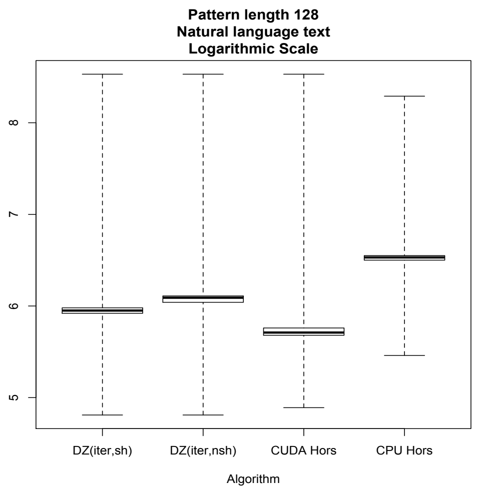
mance of all the algorithms continues when patterns of length 128 are used, as shown in Figure 5.7(c), where the difference between the best case or CUDA *DZ* implementations performs approximately 30 times better than the worst case. Again, the best case performance of both CUDA *DZ* implementations is marginally better than the best case performance of CUDA *Horspool*. The average case of both CUDA *DZ* implementations perform approximately 5x better than the CPU *Horspool* implementation, but are slightly slower than the CUDA *Horspool* implementation. Moreover, the average case performance of the *DZ(iter,sh)* CUDA implementation is slightly better than the average



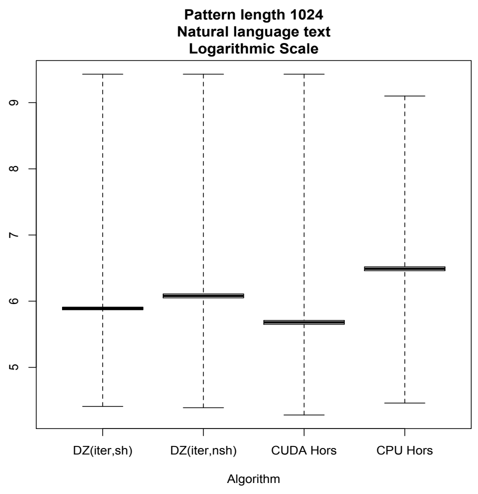
(a) Pattern length 2



(b) Pattern length 8



(c) Pattern length 128



(d) Pattern length 1024

Figure 5.7: Optimised CUDA implementations

case performance of the *DZ(iter,nsh)* CUDA implementation.

For patterns of length 1024, Figure 5.7(d) shows that the average case performance of the *DZ(iter,sh)* CUDA *DZ* implementation is slightly better than the average case performance of the *DZ(iter,nsh)* CUDA *DZ* implementation, and both are again approximately 5x better than the average case performance of CPU *Horspool*. The best case performance of the CPU *Horspool* implementation is almost as good as the best case performance of the CUDA *DZ* implementation, however the best case of the CUDA *Horspool* implementation performs the best. Again, the range increase between the best case performance and worst case performance of all the algorithms is evident when compared to smaller patterns.

It is interesting to note that, with a natural language text, the CUDA implementation of *DZ(iter,nsh)* performs better than the CUDA implementation of

DZ(iter,sh) when pattern lengths are small, but as pattern lengths increase the performance of the CUDA implementation of *DZ(iter,sh)* eventually surpasses that of CUDA *DZ(iter,nsh)*. This corresponds to the findings in the previous chapters, with a pattern length of about 32 characters being the break even point between sharing and non-sharing of information.

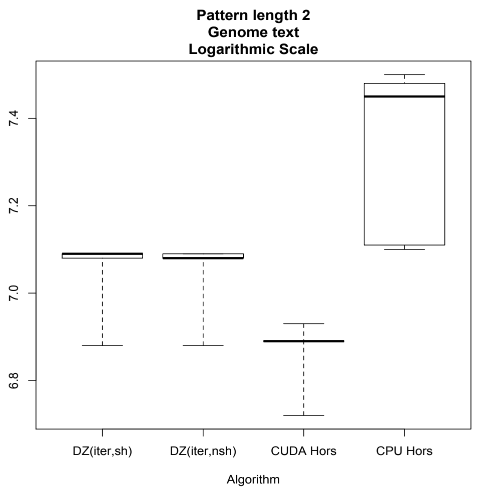
Figure 5.8 presents the same selection of graphs using the genome data set. Note that, again, the vertical axes of the subfigures make use of a logarithmic scale and that each subfigure is drawn to a different scale and should not be compared against one other.

For patterns of length 2, shown in Figure 5.8(a), the best case, worst case and average case performance of the two CUDA *DZ* implementations are extremely similar, although their best case performances are approximately as good as the average case performance of CUDA *Horspool*. Their worst case (and average case) performances are almost identical to the best case performance of the CPU *Horspool* implementation. The median line of the CUDA *DZ(iter,nsh)* implementation is slightly lower than the median line of CUDA *DZ(iter,sh)*, which means that at least half of the average case runs for CUDA *DZ(iter,nsh)* performed better than CUDA *DZ(iter,sh)*. Note the very large box plot representing the average case performance of CPU *Horspool*, which indicates the broad range of times obtained in the average case benchmark tests.

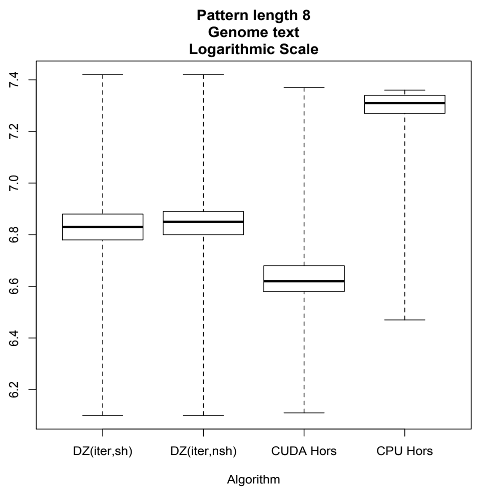
Shown in Figure 5.8(b), for a pattern length of 8, the performance of both CUDA *DZ* implementations is almost identical. The best case performance of both CUDA *DZ* implementations is approximately the same as the best case performance of CUDA *Horspool*, however, the average case performance of CUDA *Horspool* is 2x faster than that of the CUDA *DZ* implementations. Nonetheless, both CUDA *DZ* average case performances are about 5x faster than the average case of CPU *Horspool*, although the worst case performance of CPU *Horspool* is the best. The sharing *DZ* implementation performed better than the non-sharing *DZ* implementation because the median line of CUDA *DZ(iter,sh)* is lower than the median line of CUDA *DZ(iter,nsh)*.

For a pattern length of 128, illustrated in Figure 5.8(c), the best case performance of both CUDA *DZ* implementations performs marginally better than the best case performance of CUDA *Horspool*. The average case performance of both CUDA *DZ* implementations is approximately 5x faster than the CPU *Horspool* implementation, although the average case performance of CUDA *Horspool* is the fastest. The range between the best case and worst case performances is very large, with the best case performance of both CUDA *DZ* implementations about 35x faster than the worst case performance. The worst case performance of CPU *Horspool* is faster than the worst case performances for all the CUDA algorithms.

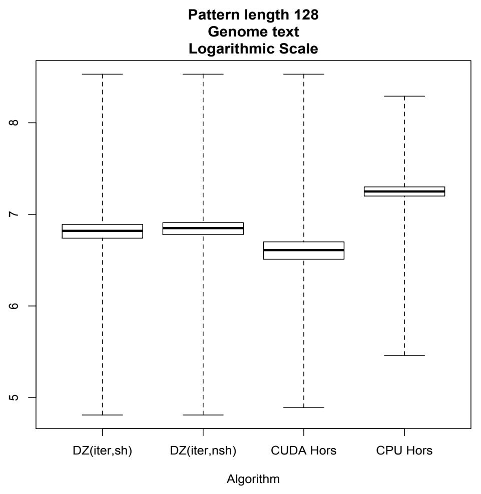
The range increase between the best case performance and worst case performance of all the algorithms continues when patterns of length 1024 are used, as shown in Figure 5.7(d), where the best case performance of both CUDA *DZ*



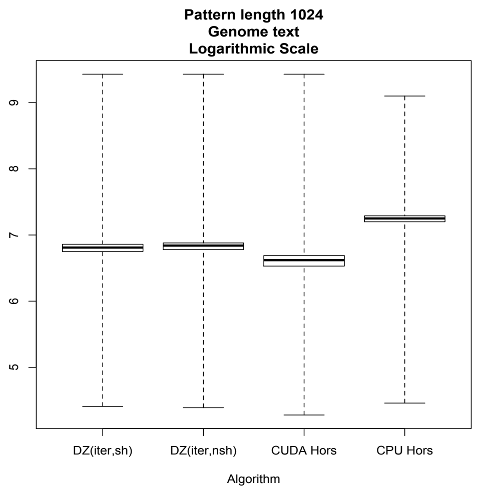
(a) Pattern length 2



(b) Pattern length 8



(c) Pattern length 128



(d) Pattern length 1024

Figure 5.8: Optimised CUDA implementations with a genome text

implementations is almost 50x faster than the worst case performance. The best case performances of all the algorithms are very similar, but the CUDA *Horspool* best case performs marginally better than the others. Again, the worst case performance of CPU *Horspool* is better than that of the other algorithms, but the average case performance of CPU *Horspool* is 5x slower than the average case performance of both CUDA *DZ* implementations.

When a genome text is used, the CUDA implementation of *DZ(iter,nsh)* performs better than the CUDA implementation of *DZ(iter,sh)* with small patterns. However, as pattern length increases, the performance of CUDA *DZ(iter,sh)* improves and performs better than CUDA *DZ(iter,nsh)*. The break even point between sharing and non-sharing information when using a genome text is approximately a pattern length of 16 characters.

5.9 Conclusions

Partly due to the vast magnitude and length of this study, several conclusions were obtained:

- Based on our findings on the non-threaded implementation of *DZ* in Chapter 3, we assumed that the compiler would also optimise the Pthreaded code that did not produce a side effect. A global match count variable was locked with a mutex lock and used by all threads to keep track of the number of matches. However, our initial assumption was incorrect: threaded code was not optimised like the non-threaded code and removing the global match count (and its mutex lock) from the algorithm had little effect on the performance. Thus, it seems probable that the compiler is not as proficient at optimising threaded code. Another possibility is that the compiler is just overly conservative, as it may not be as good at analysing parallel code.

- Although there was no limit to the maximum number of threads that could be spawned during the Pthreaded experiments, there were never more than 2048 threads running at a given time. It should be emphasised that this upper bound has been introduced by the hardware on which the benchmark tests were performed.

- It was observed in the Pthreaded experiments that an extremely wide range of time was required to create threads. This highlights the overhead and unpredictabilities brought on by the operating system that are associated with multithreading.

- The CUDA implementation was designed specifically for the GPUs that it would run on. Each algorithm should be tailored for the specific GPU, as different GPUs might require different block counts and distribution of the threads on those block that could be more efficient. As was evident in the CUDA study, the GPU specifications could determine whether or not a CUDA implementation improves the performance of an algorithm. Therefore, the GPU as well as the algorithm should be assessed to determine the most suitable implementation of the algorithm in CUDA, or if a CUDA implementation of the algorithm is even feasible.

- Further investigation of parallel implementations of *DZ* is a concern for the future. The Pthreaded study focused on a non-sharing implementation of *DZ*; future studies could implement a threaded version of *DZ(iter,sh)*. Moreover, different parallel variations, such as upper bounds on the number of threads as well as the coding language used for threading, should also be considered for future research.

- Finally, it was surprising to see that parallelisation does not always achieve the speedup that is expected. In fact, parallelising the *DZ* code with Pthreads decreased the performance of the *DZ* algorithm. Although an algorithm may appear to be a good candidate for parallelisation at first, careful consideration should be taken to determine whether this is actually true and what implementation of parallelisation will be used.

Chapter 6

Dead-Zone Skeletons

6.1 Introduction

It was shown in Chapter 4 that is it not immediately obvious whether a multiple shifter implementation of *DZ* is better than the standard implementation of the *Horspool*, *BR* and *QS* algorithms. This could be due to the structure of the *DZ* algorithm, which this chapter will refer to as the skeleton of the algorithm.

All of the DZ skeletons mentioned in this chapter represent one of the four *DZ* variants identified in Section 2.2.

- Recursive non-sharing implementations, whose code corresponds to the pseudocode in Algorithm 1.
- Recursive sharing implementations, whose code corresponds to the pseudocode in Algorithm 2.
- Iterative non-sharing implementations, whose code corresponds to the pseudocode in Algorithm 3.
- Iterative sharing implementations, whose code corresponds to the pseudocode in Algorithm 2 and uses the iterative loop from Algorithm 3.

Each of the four *DZ* variants has a corresponding basic *DZ* algorithm, discussed in Chapter 3:

- *DZ(rec,nsh)*,
- *DZ(rec,sh)*,
- *DZ(iter,nsh)*, and
- *DZ(iter,sh)*.

In this chapter, these four algorithms are referred to as the original *DZ* skeletons.

The aim of this chapter is to ascertain whether the *DZ* skeletons can be optimised such that they have a better performance.

This chapter does not introduce any new *DZ* variants; only new *DZ* skeletons produced by code changes made to the existing variants are discussed. Then, it explains how the benchmark experiments were conducted. This chapter also examines the performance of the four original skeletons of the *DZ* algorithm and compares it to the performance of the new *DZ* skeletons.

6.2 Code Adjustments

The code for the original *DZ* skeletons was sent to Professor Jorma Tarhio at Aalto University as well as to Dr David Gregg at Trinity College Dublin. They slightly tweaked the code of the original *DZ* skeletons, thus they generated nine new *DZ* skeletons:

1. *cache-friendly DZ*,
2. *stack-caching DZ*,
3. *stack-sentinel DZ*,
4. *stack-unrolled DZ*,
5. *2-gram DZ*,
6. *reduced-pop DZ*,
7. *byte-order DZ*,
8. *explicit 2-gram DZ*, and
9. *compressed-table DZ*.

Algorithms 1 to 4 were developed by Dr David Gregg, while algorithms 4 to 9 were developed by Professor Jorma Tarhio.

The new *DZ* skeletons are modifications of the iterative variants of *DZ*. Table 6.1 shows the *DZ* skeletons categorised by iterative implementation type (iterative non-sharing and iterative sharing).

Table 6.1: *DZ* skeletons grouped by iterative implementation type.

Iterative non-sharing *DZ* skeletons	**Iterative sharing *DZ* skeletons**
DZ(iter,nsh)	*DZ(iter,sh)*
2-gram DZ	*cache-friendly DZ*
reduced-pop DZ	*stack-caching DZ*
byte-order DZ	*stack-sentinel DZ*
explicit 2-gram DZ	*stack-unrolled DZ*
compressed-table DZ	

A detailed discussion on the four original *DZ* skeletons can be found in Section 2.2 and will not be discussed in this chapter. This section will accordingly focus on the new *DZ* skeletons. It should also be noted that the code of the four original *DZ* skeletons can be found in the appendix.

The following paragraphs are acquired from an email discussion with Dr David Gregg [15] and provide an overview of his *DZ* skeletons as well as how they were developed.

> Unlike *DZ(iter,sh)*, which divides the text array into equal sized parts, the *cache-friendly DZ* skeleton rather uses constant size chunks of text, and searches the chunks with the iterative sharing algorithm in *DZ(iter,sh)*. In preliminary experiments (searching for the pattern “camel” in the Bible text), this reduced cache misses by around 20%.
>
> *Stack-sentinel DZ* is similar to *DZ(iter,sh)*, but it eliminates a tiny inefficiency in checking for the stack being empty by putting a sentinel value on the bottom of the stack.
>
> The *stack-caching DZ* and *stack-unrolled DZ* skeletons consider branch mispredictions. There is a major decision to be made in the *DZ* algorithm, and that is whether the left segment is empty. If you profile your code, you will see that the left segment is empty around half the time. Therefore, we can expect to take a lot of branch mispredictions on the branch that determines whether the left segment is empty.
>
> Moreover, if the left segment is empty then the right segment is also usually empty. The reason is that whether a segment is empty or not depends very heavily on the depth of the recursion. As a result, if we are at a depth of recursion where the left segment is empty, we are probably also at a level where the right segment is empty.
>
> This means that when we are at a shallow level of recursion, the branch will mostly go the direction where the left segment is not empty. Furthermore, at deeper levels of recursion the left segment will generally be empty, so the branch will go the other way. If we could separate the branch into multiple branches — one for each level of recursion — then we could expect that each would be much more predictable than a single branch that is shared between all levels of recursion.[1]
>
> *Stack-unrolled DZ* unrolls the loop so that we have a different version of the code for each depth of the stack. The code uses a variable `tos` of type `int` to keep track of the depth of recursion, which starts out as `tos = 0`. When something gets pushed onto the stack, `tos` gets incremented, and `tos` gets decremented when

[1]It should be noted that, although these branch mispredictions are associated with the recursive *DZ* versions, they are also applicable to the iterative versions of *DZ* where recursion is implemented manually with the use of a stack (as discussed in Section 2.2)

something gets popped off of the stack.

Stack-caching DZ is a variant of stack-unrolled *DZ* that eliminates some of the stack manipulation operations and makes it easier for the compiler to promote stack locations to registers, should it choose to.

A summary of the other five *DZ* skeletons, based on email communication with Professor Jorma Tarhio [35], is given in the paragraphs that follow.

2-gram DZ applies the 2-gram shift to both directions. Testing an alignment window is made faster by first testing the leftmost 2-gram of the pattern (stored as variable `patstart`) with the leftmost 2-gram of the window, $x = (T[probe] << 8) + T[probe + 1]$. If it matches, the rest of the pattern is checked and x is further used in shifting to the left. If t denotes the text to be searched and $t_i, \ldots, t_{i+m-1}$ is the alignment window, the tested 2-grams for shifting are $t_i t_{i+1}$ and $t_{i+m-1} t_{i+m}$. The former could also be $t_{i-1} t_i$, which would provide longer shifts, but it is advantageous to use the same 2-gram for occurrence checking. Both the 2-grams could also be taken one position further (i.e. outside the alignment window), but that would lead to shorter shifts on average and the number of read characters will increase.

Reduced-pop DZ is a variation of *2-gram DZ* that reduces the number of POP operations. The loop `while ((TOP.first) >= (TOP.second)) POP;` has been removed and PUSH has been slightly modified: `if (kdright<hi) PUSH(kdright,hi);`.

Byte-order DZ is a variant of *2-gram DZ* with a different byte order in a 2-gram with $x = (T[probe]) + (T[probe + 1] << 8)$.

Explicit 2-gram DZ is a variant of *byte-order DZ* with explicit 2-gram reading with $x = *((uint16_t*)(T + probe))$.

Compressed-table DZ uses compressed shift tables with $x = ((T[probe] << 1) + T[probe + 1])\%256$. If x matches, the whole pattern is checked. The average shift is slightly shorter than in *2-gram DZ*.

6.3 Experimental Design

The experiment was carried out on a 2012 model MacBook Pro with the following specifications:

- Operating System: Mac OS X version 10.9.5
- Processor: Intel Core i7
- Processor speed: 2.6 GHz
- Number of cores: 4

- L2 Cache (per core): 256 KB
- L3 Cache (per core): 6 MB
- Memory: 8 GB, 1600 MHz, DDR3

All executables were compiled with the GCC compiler using the optimisation option -O3. Additionally, the -DNDEBUG compiler flag was specified because the code for some of the *DZ* skeletons made use of assertions.

6.4 The Data

The data set discussed in the preceding chapters is similar to the data set that was used to conduct this round of pattern matching benchmark experiments.

Two texts from the SMART corpus were used to perform the experiments, namely a genome text and a natural language text (the Bible).

In this experiment we yet again wanted to determine the effect of very short and very long patterns on the performance of the skeletons. Patterns of length 2^n were used, where $n = 1, \ldots, 16$. Patterns were selected randomly from the text with a pseudorandom number generator. Additionally, the pseudorandom number generator was seeded with the same number to ensure that different implementations always used the same randomly generated patterns for matching.

6.5 Implementation

Each of the *DZ* skeletons identified in Section 6.2 were implemented in C code by Jorma Tarhio and David Gregg. In order for their code to be executed on our benchmark platform we had to modify their code slightly by changing the signature of each algorithm's search function to match the following function signature: `int search(const unsigned char *P, int m, const unsigned char *T, int n)`. Thus, the C code of the *DZ* skeletons was executed in the C++ environment of the benchmark platform.

Although all the *DZ* skeletons use two shift tables (a left-shifter and a right-shifter), they do not all use the same preprocessing function to set up their shift tables. Hence there were several different preprocessing function implementations. We did not want the time of these preprocessing functions to be included as part of a *DZ* skeleton's pattern matching time. Therefore, the preprocessing functions were called prior to calling the corresponding search functions, and were excluded from the timing data.

6.6 Test Procedure

The same test harness used for the experiments conducted in Chapter 3 and Chapter 4 (discussed in Section 3.4) was applied to the 2 original *DZ* skeletons as well as each of the 9 new *DZ* skeletons, using both of the texts (bible and genome).

The search function of each of the skeletons gets called and the time taken for each skeleton to find all occurrences of a pattern p is recorded in nanoseconds using a high-resolution timer. This occurs for patterns of length 2^n where $n = 1, \ldots, 16$, $pnum = 100$ and $pmin = 5$.

It should be noted that the benchmark experiments from Chapter 3 and Chapter 4 had *pmin* set to thirty—i.e. thirty runs of the same pattern. Because we take the minimum value from these runs, statistically five runs with the same pattern is enough to minimise the impact of outliers caused by the operating system. In hindsight, $pmin = 30$ was needlessly large and caused unnecessarily long runtimes for the benchmark experiments.

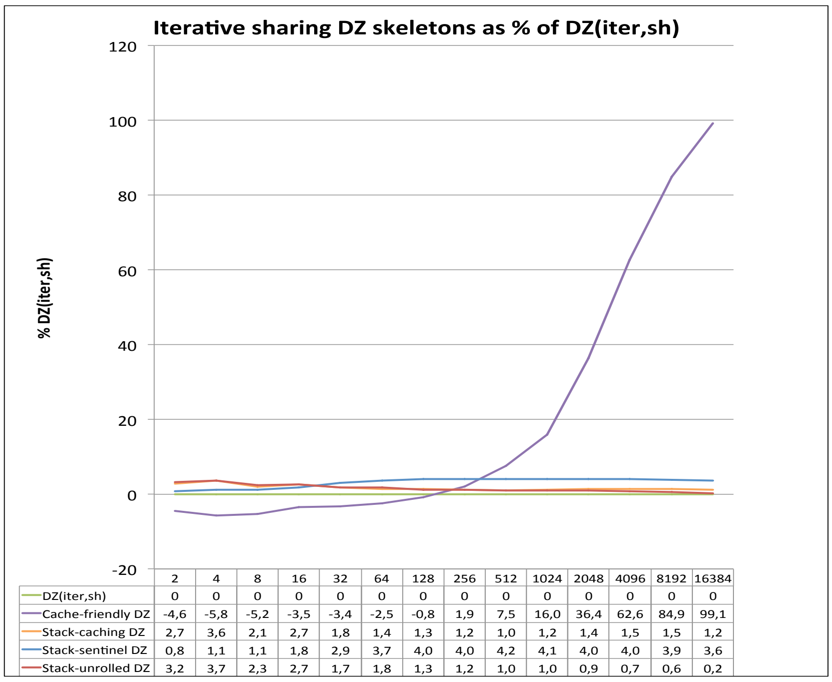
6.7 Results

The performance of the iterative sharing *DZ* skeletons is shown in Figure 6.1, relative to the original iterative sharing *DZ* skeleton, *DZ(iter,sh)*.

Figure 6.1(a) displays the results for a natural language text. Evidently, *cache-friendly DZ* is the only new skeleton that performs better than *DZ(iter,sh)*. Despite the *cache-friendly DZ* skeleton reducing cache misses by 20%, the overall execution time only improves by up to 5.8%. It is therefore apparent that cache misses do not make up a major part of overall execution time and that the *DZ(iter,sh)* skeleton is already moderately cache efficient. However, the *cache-friendly DZ* skeleton only performs better than the original *DZ(iter,sh)* skeleton for pattern lengths up to approximately 128 characters. As patterns get longer, the *cache-friendly DZ* skeleton performs steadily worse until it is almost 100% slower than *DZ(iter,sh)* with a pattern length of 16384.

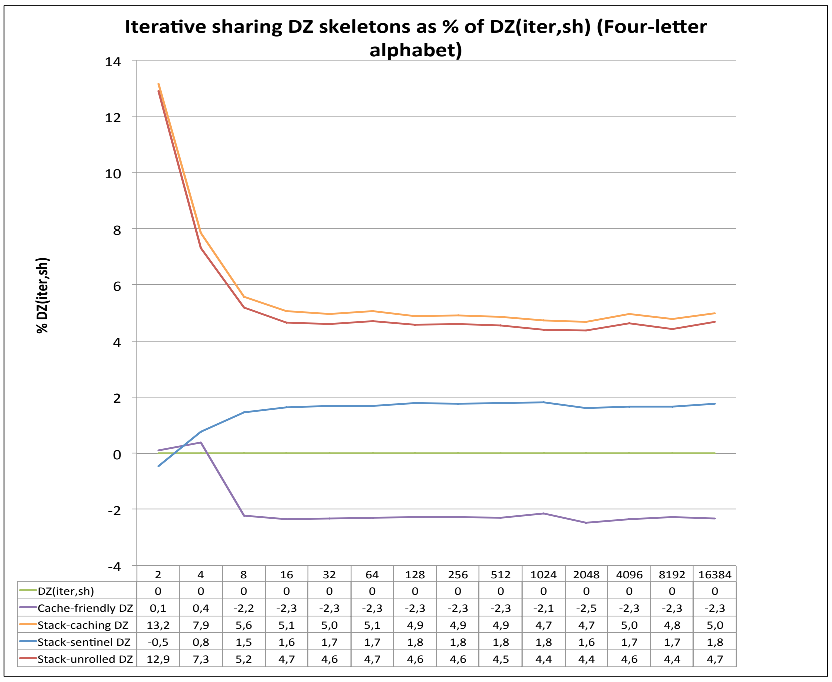
The use of a sentinel in *stack-sentinel DZ* has no effect on overall execution time. In fact, the performance is a few percentage points worse than the performance of *DZ(iter,sh)*.

The *stack-unrolled DZ* skeleton and *stack-caching DZ* skeleton reduce branch mispredictions by around 10%, yet both skeletons perform slightly worse than *DZ(iter,sh)*. In hindsight, the reason that branch mispredictions do not fall by more than 10% is obvious, but was not obvious when the code for the skeletons was written. The reason is that stack depth in the code version with the explicit stack (iterative *DZ* skeletons) is not the same thing as recursion depth in the simple recursive algorithm. In the simple recursive algorithm most empty left partitions will be found at quite deep levels of recursion, but that



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
— DZ(iter,sh)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
— Cache-friendly DZ	-4,6	-5,8	-5,2	-3,5	-3,4	-2,5	-0,8	1,9	7,5	16,0	36,4	62,6	84,9	99,1
— Stack-caching DZ	2,7	3,6	2,1	2,7	1,8	1,4	1,3	1,2	1,0	1,2	1,4	1,5	1,5	1,2
— Stack-sentinel DZ	0,8	1,1	1,1	1,8	2,9	3,7	4,0	4,0	4,2	4,1	4,0	4,0	3,9	3,6
— Stack-unrolled DZ	3,2	3,7	2,3	2,7	1,7	1,8	1,3	1,2	1,0	1,0	0,9	0,7	0,6	0,2

(a) Performance of iterative sharing skeletons using a natural language text



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
— DZ(iter,sh)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
— Cache-friendly DZ	0,1	0,4	-2,2	-2,3	-2,3	-2,3	-2,3	-2,3	-2,3	-2,1	-2,5	-2,3	-2,3	-2,3
— Stack-caching DZ	13,2	7,9	5,6	5,1	5,0	5,1	4,9	4,9	4,9	4,7	4,7	5,0	4,8	5,0
— Stack-sentinel DZ	-0,5	0,8	1,5	1,6	1,7	1,7	1,8	1,8	1,8	1,8	1,6	1,7	1,7	1,8
— Stack-unrolled DZ	12,9	7,3	5,2	4,7	4,6	4,7	4,6	4,6	4,5	4,4	4,4	4,6	4,4	4,7

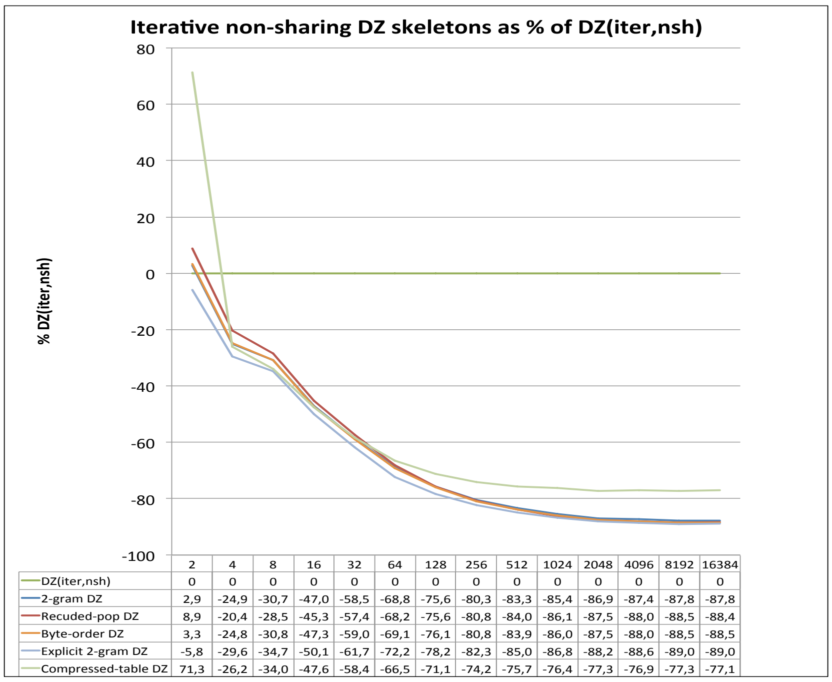
(b) Performance of iterative sharing skeletons using a genome language text

Figure 6.1: Performance of iterative sharing DZ skeletons

can correspond to a place in the iterative algorithm with a relatively shallow stack, because the stack is deeper when following left recursions, and shallower when following right recursions.

The results when using a genome text are shown in Figure 6.1(b). The performance of the *cache-friendly DZ* skeleton is almost identical to the performance of *DZ(iter,sh)* for patterns with a length of up to 8 characters. For very short patterns (pattern lengths less than 2 characters), *stack sentinel DZ* performs marginally better than *DZ(iter,sh)*. When pattern lengths are 8 characters and longer, the *cache-friendly DZ* skeleton is consistently about 2.3% faster than *DZ(iter,sh)*.

6.8 Impact of 2-grams



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
DZ(iter,nsh)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2-gram DZ	2,9	-24,9	-30,7	-47,0	-58,5	-68,8	-75,6	-80,3	-83,3	-85,4	-86,9	-87,4	-87,8	-87,8
Recuded-pop DZ	8,9	-20,4	-28,5	-45,3	-57,4	-68,2	-75,6	-80,8	-84,0	-86,1	-87,5	-88,0	-88,5	-88,4
Byte-order DZ	3,3	-24,8	-30,8	-47,3	-59,0	-69,1	-76,1	-80,8	-83,9	-86,0	-87,5	-88,0	-88,5	-88,5
Explicit 2-gram DZ	-5,8	-29,6	-34,7	-50,1	-61,7	-72,2	-78,2	-82,3	-85,0	-86,8	-88,2	-88,6	-89,0	-89,0
Compressed-table DZ	71,3	-26,2	-34,0	-47,6	-58,4	-66,5	-71,1	-74,2	-75,7	-76,4	-77,3	-76,9	-77,3	-77,1

Figure 6.2: Impact of 2-grams

The study described in Chapter 4 explores the use of *Berry-Ravindran (BR)* shift tables in *DZ* algorithms. It was concluded that the BR data was incompatible with the rest of the data because BR uses 2-grams to perform the shifts—i.e. two consecutive characters are looked at to determine the shift distance—they make a performance trade-off between time and space (see the discussion in Section 4.9).

Because *2-gram DZ*, *reduced-pop DZ*, *byte-order DZ*, *explicit 2-gram DZ* and *compressed-table DZ* also make a trade-off between space and time by using 2-grams to do the shifting, the data relating to these skeletons are, in essence, incommensurate with the data derived from the other *DZ* skeletons. However, to highlight the impact of utilising 2-grams in pattern matching algorithms, the graph comparing *DZ(iter,nsh)* to the *DZ* skeletons that make use of 2-grams has been included in Figure 6.2. It shows the results when a natural language text is used. The results for the genome text are similar, and can be seen in the appendix.

When pattern lengths are short, the advantages of using 2-grams are not entirely obvious. In fact, for very short pattern lengths ($|p| = 2$) only the *explicit 2-gram DZ* skeleton performs better than the *DZ(iter,nsh)* skeleton. For longer patterns, the performance benefits of using 2-grams in pattern matching becomes apparent. The *2-gram DZ* skeletons perform increasingly better than *DZ(iter,nsh)* as pattern length increases, until their performances peak at between 87% and 89% from a pattern length of about 2048 characters. The *compressed-table DZ* skeleton performs slightly worse than the other *2-gram DZ* skeletons—at its peak it performs 77% to 79% better than *DZ(iter,nsh)*.

6.9 Conclusion

Although the iterative *DZ* skeletons are near optimal, they can be optimised such that they have performance improvements. The use of *cache-friendly DZ* skeleton over the *DZ(iter,sh)* skeleton should be preferred, except when searching a natural language text for patterns with more than 128 characters, as well as when searching a genome text for very short patterns.

It is evident that, instead of looking at only one character to determine the shift distance, looking at two consecutive characters can improve the performance of *DZ(iter,nsh)* by up to 89%.

Chapter 7

Conclusion

7.1 Results

This text presented a study on a variety of *DZ* implementations and their performance during benchmark experiments. These implementations are all based on the four basic variants of the *DZ* family of algorithms as described in Section 2.2. The main objective of this study is to determine the empirical performance of new variants of the *DZ* algorithm and how they compare to existing pattern matching algorithms

Initial experiments involved comparing the performance of an object-oriented version of *DZ* implemented in C++ to versions of *DZ* with no object-orientation (essentially plain C code), thus a C++ benchmarking environment was developed for this purpose. Consequently, C implementations of the four basic variants of the *DZ* family of algorithms were benchmarked using the same C++ benchmarking platform. The iterative implementations performed favourably, with both algorithms beating traditional pattern matching algorithms when searching natural language and genome texts, particularly for short patterns. In fact, for both the natural language as well as the genome experiments, the *DZ(iter,nsh)* algorithm was the best performing algorithm for patterns up to 8 characters in length. A cost of recursion and the impact of information sharing in the *DZ* algorithms were also identified.

The benchmarks revealed that the general behaviour of the *DZ* algorithms was similar to that of the *Horspool* algorithm, thus they were extended such that various left shifters and right shifters were used in all four *DZ* variants. We hoped that significant changes in the behaviour of the *DZ* algorithms would be observed when different left and right shifters are used—this was the case, because certain combinations of shifters performed better than others during the experiment. Overall benchmark results were, however, disappointing, as there was only one case where our *DZ* implementation performed better than an implementation of the traditional algorithm: For a pattern length of 65536, *DZ(iter,sh,b-b)* performs about 8% better than the standard BR implementation. Nevertheless, it was observed that the cost of recursion and the impact of information sharing are intrinsic *DZ* characteristics that are independent of the shifter being used. The selection of the *Berry-Ravindran* algorithm as a shifter was, in hind sight, a poor choice, as it was essentially incommensurate with the other shifters because it makes a trade-off between space and time.

Because the recursive *DZ* implementations make use of double tail recursion, they were identified as good candidates for parallelisation and parallel implementations of the *DZ* algorithms were thus developed and benchmarked. Three implementations of *DZ(rec,nsh)* using Pthreads all performed poorly compared to the non-threaded version of *DZ(rec,nsh)*. This can be explained by the lack of upper bound given to the number of threads that can be spawned during the experiment. Consequently, more than eighteen thousand threads were created during every benchmark run, resulting in enormous overhead costs for the operating system when creating all the threads, and causing the performance to be even slower than the CPU implementation.

A second attempt was made at implementing and benchmarking a parallel version of *DZ* executing on the GPU with CUDA (instead of on the CPU). The solution was a hybrid *DZ* implementation that made use of both recursion and iteration. Benchmark results were poor during the initial experiments and did not beat a CPU implementation of *Horspool*. However, the performance greatly improved when a better GPU was used—i.e. the CUDA implementation was consistently 5x faster than a CPU implementation of *Horspool*.

Throughout the preceding experiments, the following question was asked: is it possible to modify the skeletons of the original *DZ* algorithms such that there is an improvement in performance? *DZ* code with slight tweaks was obtained from Dr David Gregg and Professor Jorma Tarhio and benchmark experiments were conducted. The benchmark tests revealed that when a natural language text is used the *cache-friendly DZ* skeleton beats all other *DZ* skeletons for pattern lengths up to approximately 128 characters. Furthermore, with a genome text the *cache-friendly DZ* skeleton beats all other *DZ* skeletons for all patterns with a length of 4 or more characters. Therefore, in general, the original implementation of *DZ* is near-optimal, but a *cache-friendly DZ* skeleton slightly improves the performance of the original *DZ(iter,sh)* implementation. Unfortunately, the skeletons received from Jorma Tarhio all had a space time trade-off by using 2-grams, and the data obtained from their benchmarks was incompatible with the data obtained from the other skeletons.

7.2 Potential Future Research

Although this dissertation reports on several implementations of *DZ*, there are still plenty of possible extensions. Possible future work includes:

- Further investigation of David Gregg's cache-friendly DZ skeleton to determine if it should become the standard skeleton for *DZ(iter,nsh)*. Additionally, the use of David Gregg's *cache-friendly DZ skeleton* in *DZ(iter,sh)*, *DZ(rec,sh)* and *DZ(rec,nsh)* is a key concern for future benchmarking.

- Designing a more efficient technique to define shifter combinations. In this research effort, C preprocessor macros were used to define the different shifter combinations in the multiple shifter *DZ* implementation. To determine whether this implementation was the reason for the poor performance of all the multiple shifters, a straightforward multiple shifter implementation without macros could be developed, although this will be time-consuming.
- Further investigation of parallel implementations of *DZ*, with particular regard to different parallelisation models and coding languages.
- An implementation of a version of *DZ* that performs approximate pattern matching.
- An implementation of a version of *DZ* that performs multiple keyword pattern matching.

Appendix A

Traditional Pattern Matching Algorithms Code

Code A.1: Boyer-Moore

```
#define SIGMA 256

// Shift tables:
static int brBc[SIGMA][SIGMA];

void preBR(const unsigned char *P, int m) {
    int a, b, i;
    for (a = 0; a < SIGMA; ++a)
        for (b = 0; b < SIGMA; ++b)
            brBc[a][b] = m + 2;
    for (a = 0; a < SIGMA; ++a)
        brBc[a][P[0]] = m + 1;
    for (i = 0; i < m - 1; ++i)
        brBc[P[i]][P[i + 1]] = m - i;
    for (a = 0; a < SIGMA; ++a)
        brBc[P[m - 1]][a] = 1;
}

int search(const unsigned char *P, int m, unsigned char *T, int n) {
    int i, j;
    int count;

    count =0;

    /* Searching */
    T[n + 1] = '\0';
    j = 0;
    while (j <= n - m) {
        for (i=0; i<m && P[i]==T[j+i]; i++);
        if (i>=m) count++;
        j += brBc[T[j + m]][T[j + m + 1]];
    }
    return count;
}
```

Code A.2: Horspool

```
#define SIGMA 256

//  Shift tables:
static int hbc[SIGMA];

void preHorspool(const unsigned char *P, int m) {
    int i;
    for (i=0;i<SIGMA;i++) hbc[i]=m;
    for (i=0;i<m-1;i++) hbc[P[i]]=m-i-1;
}

int search(const unsigned char *P, int m, unsigned char *T, int n) {
    int i, s, count;
    /* Searching */
    s = 0;
    count = 0;
    while(s<=n-m) {
        i=0;
        while(i<m && P[i]==T[s+i]) i++;
        if (i==m) count++;
        s+=hbc[T[s+m-1]];
    }
    return count;
}
```

Code A.3: Quick Search

```
#define SIGMA 256

//  Shift tables:
static int qsbc[SIGMA];

void preQS(const unsigned char *P, int m) {
    int i;
    for (i=0;i<SIGMA;i++) qsbc[i]=m+1;
    for (i=0;i<m;i++) qsbc[P[i]]=m-i;
}

int search(const unsigned char *P, int m, const unsigned char *T,
    int n) {
    int i, s, count;
    s = 0;
    count = 0;
    while(s<=n-m) {
        i=0;
        while(i<m && P[i]==T[s+i]) i++;
        if (i==m) count++;
        s+=qsbc[T[s+m]];
    }
    return count;
}
```

Appendix B

Dead-Zone Code

Code B.1: Recursive non-sharing DZ

```
static int searchrec(int lo, int hi) {
   int count = 0;
   int probe;
   {
      int i;
      probe = (lo+hi)>>1;
      assert(probe == (lo+hi)/2);
      assert(lo <= probe);
      assert(probe < hi);
      for (i=0; i<m && P[i] == T[probe+i]; i++) {
         // Intentionally empty;
      }
      if (i == m) {
         count = 1;
      }
   }
   {
      int kdleft = probe - shl[T[probe]] + 1;
      if (lo < kdleft) {
         count += searchrec(lo, kdleft);
      }
   }
   {
      int kdright = probe + shr[T[probe+m-1]];
      if (kdright < hi) {
         count += searchrec(kdright, hi);
      }
   }
   return count;
}

int search(const unsigned char *Patt, int emm, const unsigned char
    *Text, int enn) {
```

```
    //  These two should be set already.
    assert(P == Patt);
    assert(m == emm);
   T = Text;
   n = enn;
   if (n < m) {
      return 0;
   } else {
        return searchrec(0, n-(m-1));
    }
}
```

Code B.2: Recursive sharing DZ

```
static int searchrec(int lo, int hi) {
    if (lo >= hi) {
        d = lo;
        return 0;
    } else {
        int i;
        int count = 0;
        int probe = (lo+hi)>>1;
        assert(probe == (lo+hi)/2);
        assert(lo <= probe);
        assert(probe < hi);
        for (i=0; i<m && P[i] == T[probe+i]; i++) {
            // Intentionally empty;
        }
        count = (i==m);
        count += dzRecShareRec(lo, probe - shl[T[probe]] + 1);
        count += dzRecShareRec(MAX(d, probe + shr[T[probe+m-1]]), hi);
        return count;
    }
}

int search(const unsigned char *Patt, int emm, const unsigned char
    *Text, int enn) {
    P = Patt;
    m = emm;
    d = 0;

    T = Text;
    n = enn;
    if (n < m) {
        return 0;
    } else {
        return dzRecShareRec(0, n-(m-1));
    }
}
```

Code B.3: Iterative non-sharing DZ

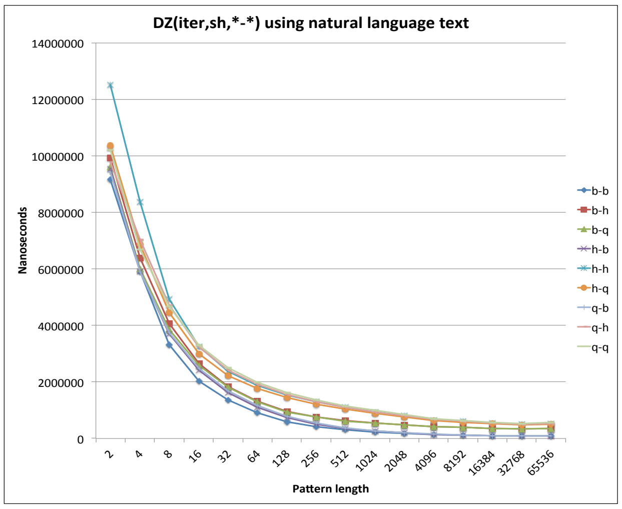
```
int search(const unsigned char *P, int m, const unsigned char *T,
    int n) {
    struct {
        int first, second;
    } todo[32];
    int i, count=0;
    int tos=0;
    int lo=0, hi=n-(m-1);
    int kdleft, kdright;
    int probe;

    if (n < m) {
        return count;
    }

#define EMPTY (tos==0)
#define POP (--tos)
#define TOP (todo[tos])
#define PUSH(x,y) ++tos; TOP.first=(x); TOP.second=(y)

    PUSH(0,INT_MAX);

    for(;;) {
        probe=(lo+hi)>>1;
        assert(probe == (lo+hi)/2);

        assert(lo<hi);
        assert(lo<=probe && probe<hi);

        for (i=0; i<m && P[i]==T[probe+i]; i++) {
            // Intentionally empty
        }
        if (i==m) {
            count++;
        }
        // Do shifts
        {
            kdleft = probe - shl[T[probe]] + 1;
            kdright = probe + shr[T[probe + m - 1]];
            if (lo < kdleft) {
                // Left is good, so enstack right, regardless if it's
                    good.
                PUSH(kdright, hi);
                hi = kdleft;
            } else {
                // Left is empty.
                assert(lo >= kdleft);
                // ...consider using the right...
                if ((lo=kdright) >= hi) {
                    // Right is also bad...
                    assert(lo >= hi);
                    assert(!EMPTY);
                    while ((TOP.first) >= (TOP.second)) {
                        assert(!EMPTY);
                        POP;
                    }
                    if (TOP.second == INT_MAX) {
                        return count;
                    } else {
                        lo = TOP.first;
                        hi = TOP.second;
                        POP;
                    }
                }
            }
            assert(lo < hi);
        }
    }
#undef EMPTY
#undef POP
#undef TOP
#undef PUSH
}
```

Code B.4: Iterative sharing DZ

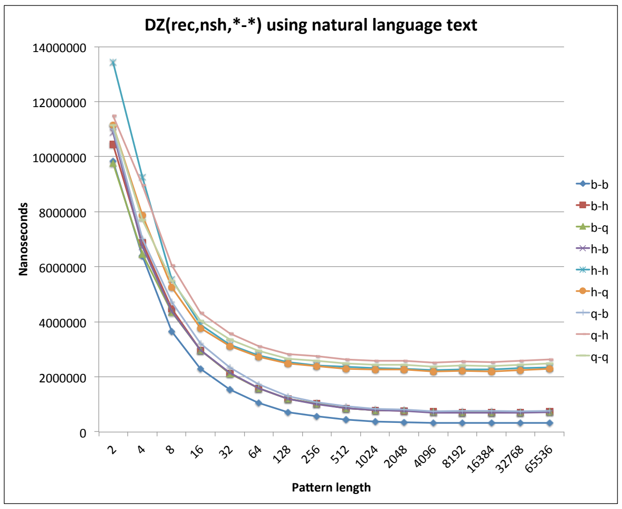
```
int search(const unsigned char *P, int m, const unsigned char *T,
    int n) {
    struct {
        int first, second;
    } todo[32];
    int i, count=0;
    int tos=0;
    int lo=0, hi=n-(m-1);
    int kdleft, kdright;
    int probe;

    if (n < m) {
        return count;
    }

#define EMPTY (tos==0)
#define POP (--tos)
#define TOP (todo[tos])
#define PUSH(x,y) ++tos; TOP.first=(x); TOP.second=(y)

    for(;;) {
        probe=(lo+hi)>>1;
        assert(probe == (lo+hi)/2);

        assert(lo<hi);
        assert(lo<=probe && probe<hi);

        for (i=0; i<m && P[i]==T[probe+i]; i++) {
            // Intentionally empty
        }
        if (i==m) {
            count++;
        }
        // Do shifts
        {
            kdleft = probe - shl[T[probe]] + 1;
            kdright = probe + shr[T[probe + m - 1]];
            if (lo < kdleft) {
                // Left is good, so enstack right, regardless if it's
                   good.
                PUSH(kdright, hi);
                hi = kdleft;
            } else {
                // Left is empty.
                assert(lo >= kdleft);
                // ...consider using the right...
                lo = kdright;
                if (kdright >= hi) {
                    // Right is also bad...
                    assert(kdright >= hi);
                    while (!EMPTY && !((lo=(MAX(TOP.first, lo))) <
                        (hi=TOP.second))) {
                        assert(!EMPTY);
                        POP;
                    }
                    if (EMPTY) {
                        return count;
                    } else {
                        POP;
                    }
                }
            }
            assert(lo < hi);
        }
    }
#undef EMPTY
#undef POP
#undef TOP
#undef PUSH
}
```

Appendix C

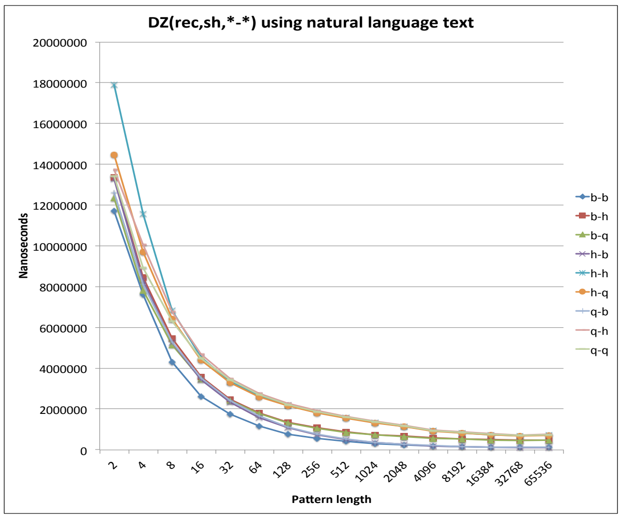
Multiple Shifters Benchmark Figures



(a) *DZ(iter,sh,*-*)*

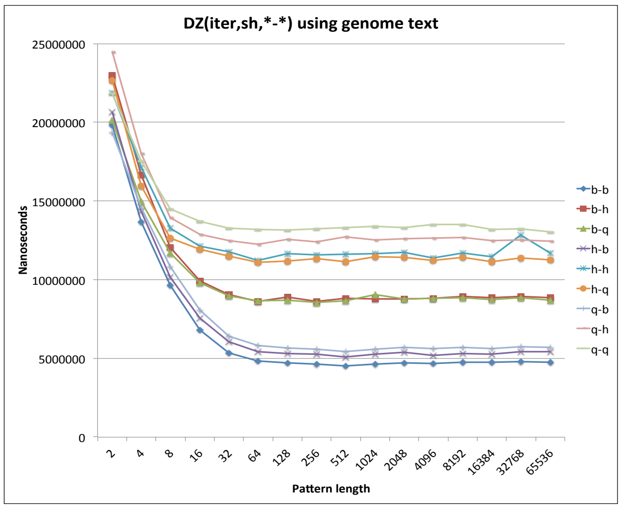


(b) *DZ(rec,nsh,*-*)*

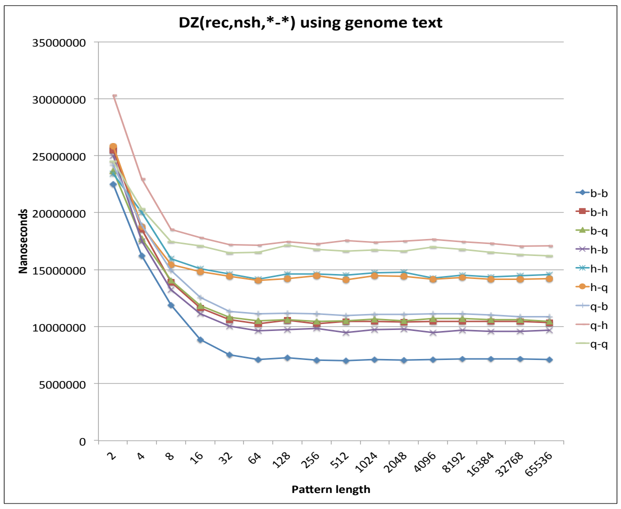


(c) *DZ(rec,sh,*-*)*

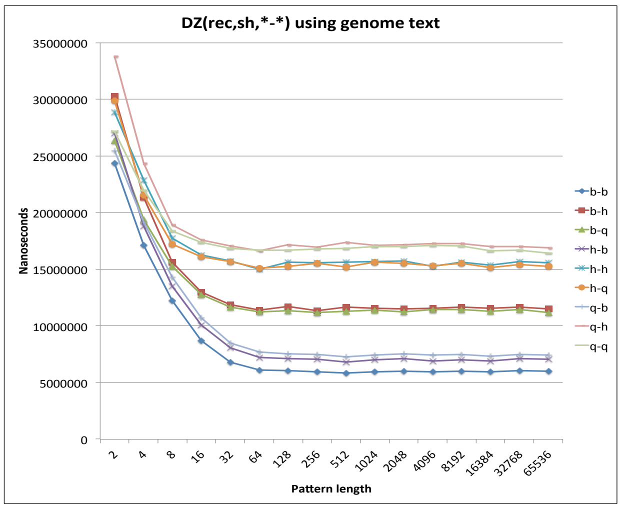
Figure C.1: Illustrative raw averaged minimum time data of multiple shifter DZ using a natural language text



(a) *DZ(iter,sh,*-*)*

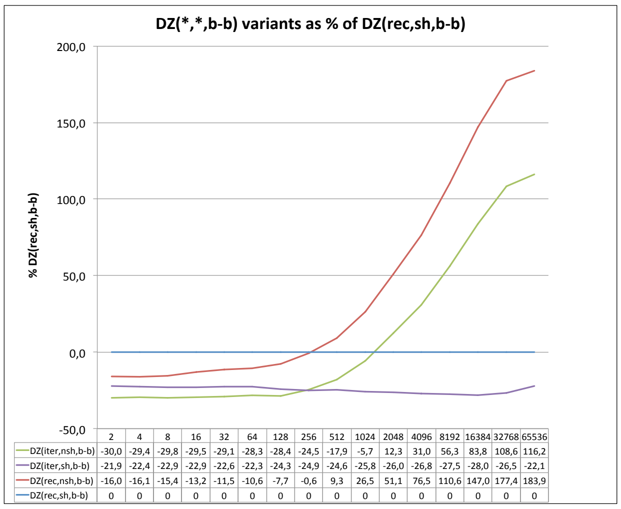


(b) *DZ(rec,nsh,*-*)*



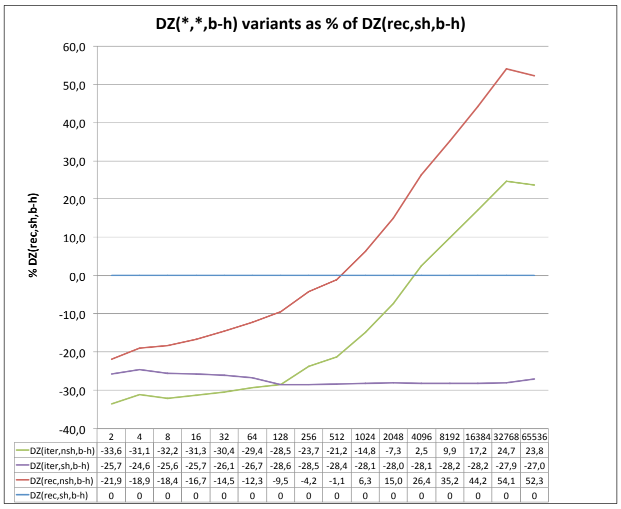
(c) *DZ(rec,sh,*-*)*

Figure C.2: Illustrative raw averaged minimum time data of multiple shifter DZ using a genome text



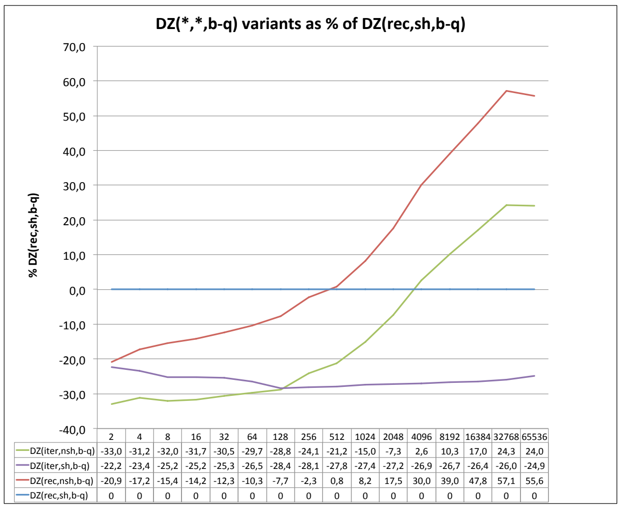
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-b)	-30,0	-29,4	-29,8	-29,5	-29,1	-28,3	-28,4	-24,5	-17,9	-5,7	12,3	31,0	56,3	83,8	108,6	116,2
DZ(iter,sh,b-b)	-21,9	-22,4	-22,9	-22,9	-22,6	-22,3	-24,3	-24,9	-24,6	-25,8	-26,0	-26,8	-27,5	-28,0	-26,5	-22,1
DZ(rec,nsh,b-b)	-16,0	-16,1	-15,4	-13,2	-11,5	-10,6	-7,7	-0,6	9,3	26,5	51,1	76,5	110,6	147,0	177,4	183,9
DZ(rec,sh,b-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Cost of Recursion of *DZ(*,*,b-b)*



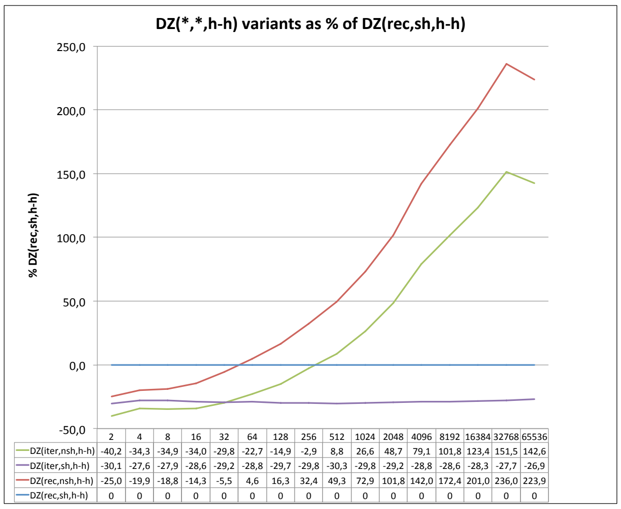
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-h)	-33,6	-31,1	-32,2	-31,3	-30,4	-29,4	-28,5	-23,7	-21,2	-14,8	-7,3	2,5	9,9	17,2	24,7	23,8
DZ(iter,sh,b-h)	-25,7	-24,6	-25,6	-25,7	-26,1	-26,7	-28,6	-28,5	-28,4	-28,1	-28,0	-28,1	-28,2	-28,2	-27,9	-27,0
DZ(rec,nsh,b-h)	-21,9	-18,9	-18,4	-16,7	-14,5	-12,3	-9,5	-4,2	-1,1	6,3	15,0	26,4	35,2	44,2	54,1	52,3
DZ(rec,sh,b-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Cost of Recursion of *DZ(*,*,b-h)*



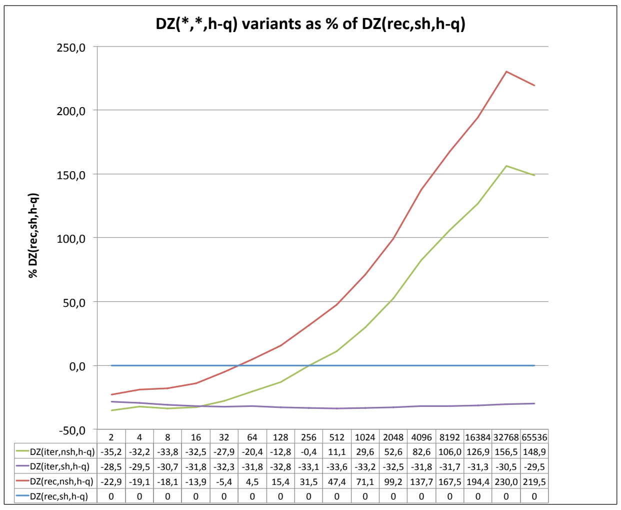
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-q)	-33,0	-31,2	-32,0	-31,7	-30,5	-29,7	-28,8	-24,1	-21,2	-15,0	-7,3	2,6	10,3	17,0	24,3	24,0
DZ(iter,sh,b-q)	-22,2	-23,4	-25,2	-25,2	-25,3	-26,5	-28,4	-28,1	-27,8	-27,4	-27,2	-26,9	-26,7	-26,4	-26,0	-24,9
DZ(rec,nsh,b-q)	-20,9	-17,2	-15,4	-14,2	-12,3	-10,3	-7,7	-2,3	0,8	8,2	17,5	30,0	39,0	47,8	57,1	55,6
DZ(rec,sh,b-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(c) Cost of Recursion of *DZ(*,*,b-q)*



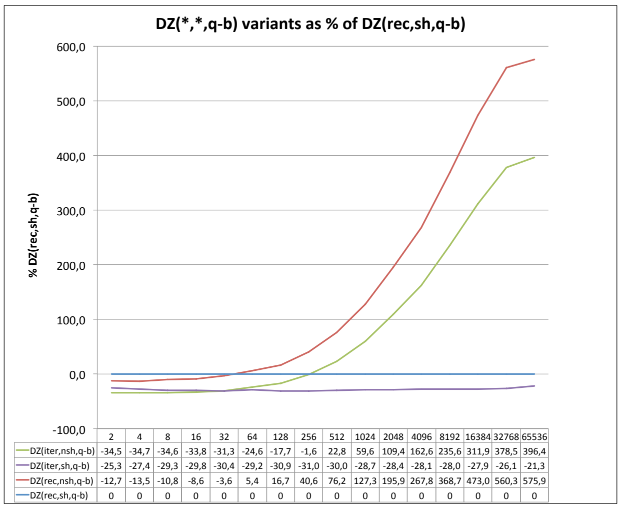
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,h-h)	-40,2	-34,3	-34,9	-34,0	-29,8	-22,7	-14,9	-2,9	8,8	26,6	48,7	79,1	101,8	123,4	151,5	142,6
DZ(iter,sh,h-h)	-30,1	-27,6	-27,9	-28,6	-29,2	-28,8	-29,7	-29,8	-30,3	-29,8	-29,2	-28,8	-28,6	-28,3	-27,7	-26,9
DZ(rec,nsh,h-h)	-25,0	-19,9	-18,8	-14,3	-5,5	4,6	16,3	32,4	49,3	72,9	101,8	142,0	172,4	201,0	236,0	223,9
DZ(rec,sh,h-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(d) Cost of Recursion of *DZ(*,*,h-h)*



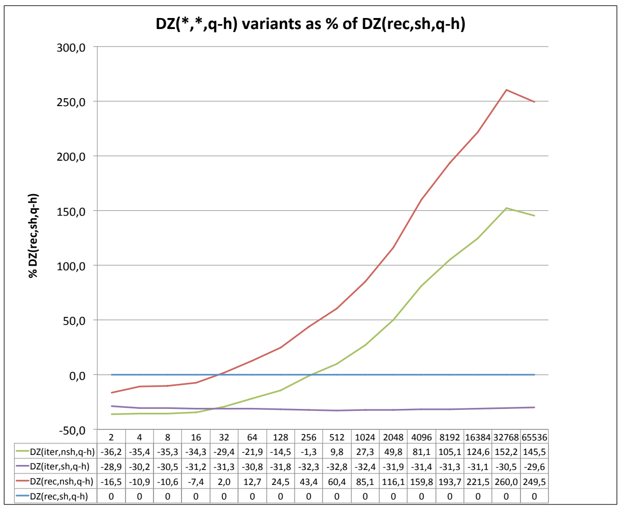
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,h-q)	-35,2	-32,2	-33,8	-32,5	-27,9	-20,4	-12,8	-0,4	11,1	29,6	52,6	82,6	106,0	126,9	156,5	148,9
DZ(iter,sh,h-q)	-28,5	-29,5	-30,7	-31,8	-32,3	-31,8	-32,8	-33,1	-33,6	-33,2	-32,5	-31,8	-31,7	-31,3	-30,5	-29,5
DZ(rec,nsh,h-q)	-22,9	-19,1	-18,1	-13,9	-5,4	4,5	15,4	31,5	47,4	71,1	99,2	137,7	167,5	194,4	230,0	219,5
DZ(rec,sh,h-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(e) Cost of Recursion of *DZ(*,*,h-q)*



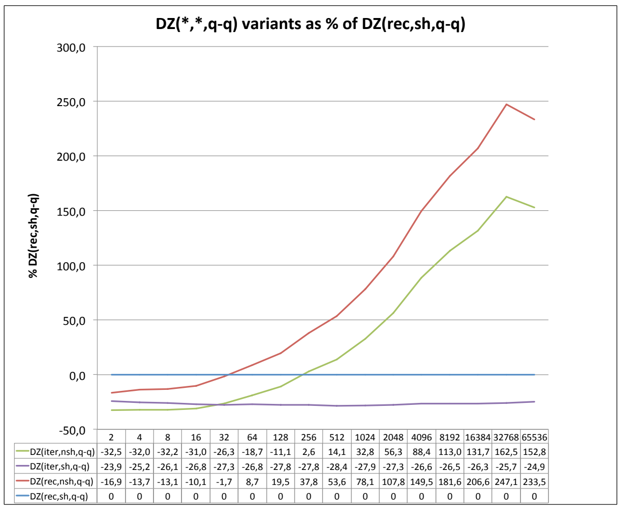
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,q-b)	-34,5	-34,7	-34,6	-33,8	-31,3	-24,6	-17,7	-1,6	22,8	59,6	109,4	162,6	235,6	311,9	378,5	396,4
DZ(iter,sh,q-b)	-25,3	-27,4	-29,3	-29,8	-30,4	-29,2	-30,9	-31,0	-30,0	-28,7	-28,4	-28,1	-28,0	-27,9	-26,1	-21,3
DZ(rec,nsh,q-b)	-12,7	-13,5	-10,8	-8,6	-3,6	5,4	16,7	40,6	76,2	127,3	195,9	267,8	368,7	473,0	560,3	575,9
DZ(rec,sh,q-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(f) Cost of Recursion of *DZ(*,*,q-b)*



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,q-h)	-36,2	-35,4	-35,3	-34,3	-29,4	-21,9	-14,5	-1,3	9,8	27,3	49,8	81,1	105,1	124,6	152,2	145,5
DZ(iter,sh,q-h)	-28,9	-30,2	-30,5	-31,2	-31,3	-30,8	-31,8	-32,3	-32,8	-32,4	-31,9	-31,4	-31,3	-31,1	-30,5	-29,6
DZ(rec,nsh,q-h)	-16,5	-10,9	-10,6	-7,4	2,0	12,7	24,5	43,4	60,4	85,1	116,1	159,8	193,7	221,5	260,0	249,5
DZ(rec,sh,q-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

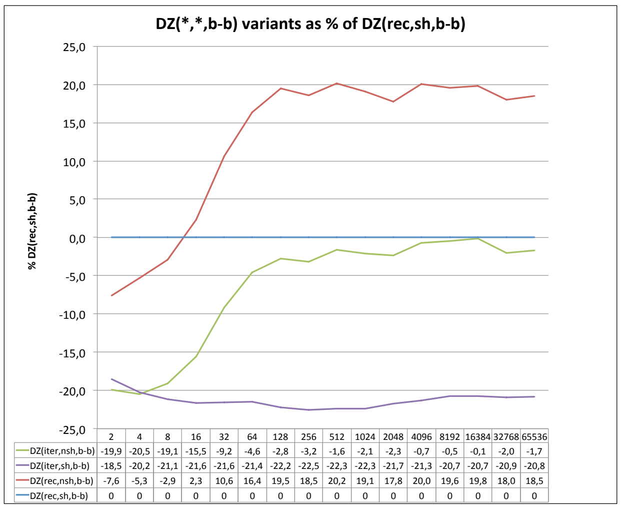
(g) Cost of Recursion of *DZ(*,*,q-h)*



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,q-q)	-32,5	-32,0	-32,2	-31,0	-26,3	-18,7	-11,1	2,6	14,1	32,8	56,3	88,4	113,0	131,7	162,5	152,8
DZ(iter,sh,q-q)	-23,9	-25,2	-26,1	-26,8	-27,3	-26,8	-27,8	-27,8	-28,4	-27,9	-27,3	-26,6	-26,5	-26,3	-25,7	-24,9
DZ(rec,nsh,q-q)	-16,9	-13,7	-13,1	-10,1	-1,7	8,7	19,5	37,8	53,6	78,1	107,8	149,5	181,6	206,6	247,1	233,5
DZ(rec,sh,q-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

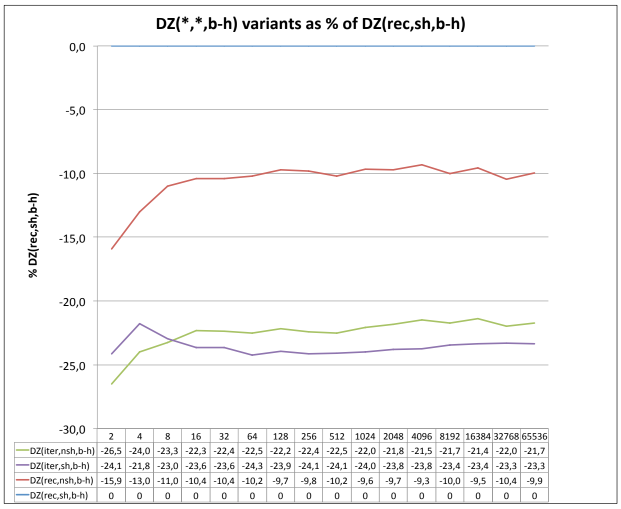
(h) Cost of Recursion of *DZ(*,*,q-q)*

Figure C.3: Cost of Recursion of multiple shifter DZ variants using a natural language text



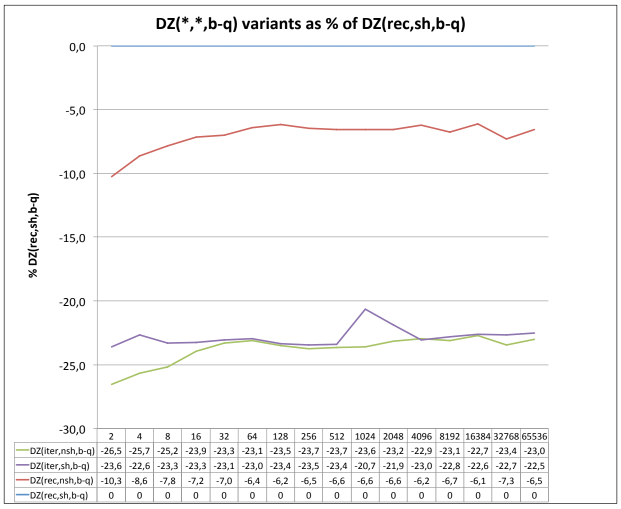
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-b)	-19,9	-20,5	-19,1	-15,5	-9,2	-4,6	-2,8	-3,2	-1,6	-2,1	-2,3	-0,7	-0,5	-0,1	-2,0	-1,7
DZ(iter,sh,b-b)	-18,5	-20,2	-21,1	-21,6	-21,6	-21,4	-22,2	-22,5	-22,3	-22,3	-21,7	-21,3	-20,7	-20,7	-20,9	-20,8
DZ(rec,nsh,b-b)	-7,6	-5,3	-2,9	2,3	10,6	16,4	19,5	18,5	20,2	19,1	17,8	20,0	19,6	19,8	18,0	18,5
DZ(rec,sh,b-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(a) Cost of Recursion of *DZ(*,*,b-b)* using a genome text



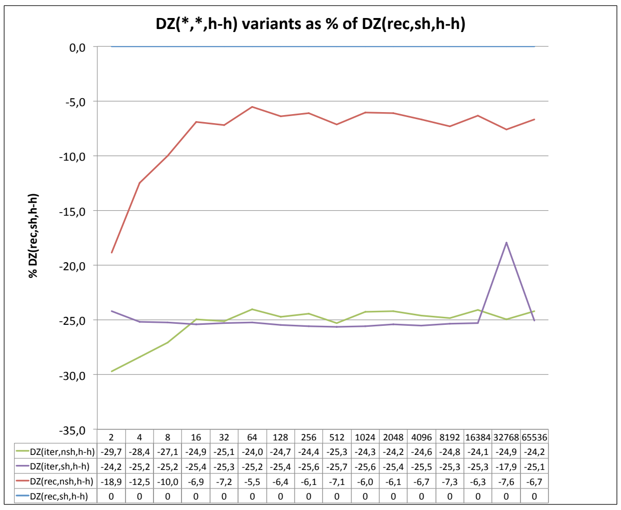
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-h)	-26,5	-24,0	-23,3	-22,3	-22,4	-22,5	-22,2	-22,4	-22,5	-22,0	-21,8	-21,5	-21,7	-21,4	-22,0	-21,7
DZ(iter,sh,b-h)	-24,1	-21,8	-23,0	-23,6	-23,6	-24,3	-23,9	-24,1	-24,1	-24,0	-23,8	-23,8	-23,4	-23,4	-23,3	-23,3
DZ(rec,nsh,b-h)	-15,9	-13,0	-11,0	-10,4	-10,4	-10,2	-9,7	-9,8	-10,2	-9,6	-9,7	-9,3	-10,0	-9,5	-10,4	-9,9
DZ(rec,sh,b-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b) Cost of Recursion of *DZ(*,*,b-h)* using a genome text



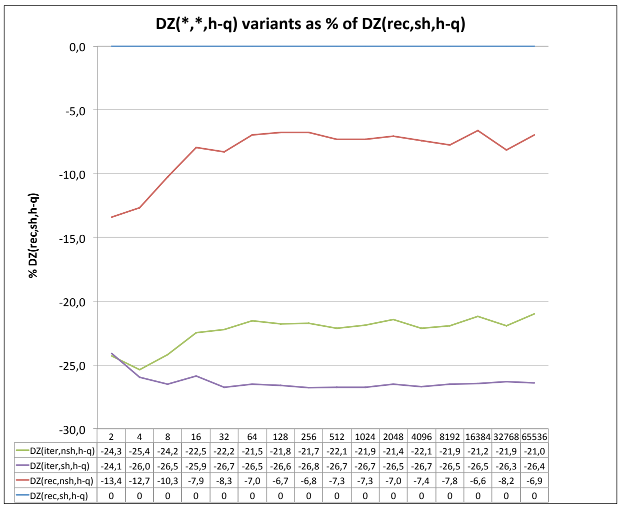
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,b-q)	-26,5	-25,7	-25,2	-23,9	-23,3	-23,1	-23,5	-23,7	-23,7	-23,6	-23,2	-22,9	-23,1	-22,7	-23,4	-23,0
DZ(iter,sh,b-q)	-23,6	-22,6	-23,3	-23,3	-23,1	-23,0	-23,4	-23,5	-23,4	-20,7	-21,9	-23,0	-22,8	-22,6	-22,7	-22,5
DZ(rec,nsh,b-q)	-10,3	-8,6	-7,8	-7,2	-7,0	-6,4	-6,2	-6,5	-6,6	-6,6	-6,6	-6,2	-6,7	-6,1	-7,3	-6,5
DZ(rec,sh,b-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(c) Cost of Recursion of *DZ(*,*,b-q)* using a genome text



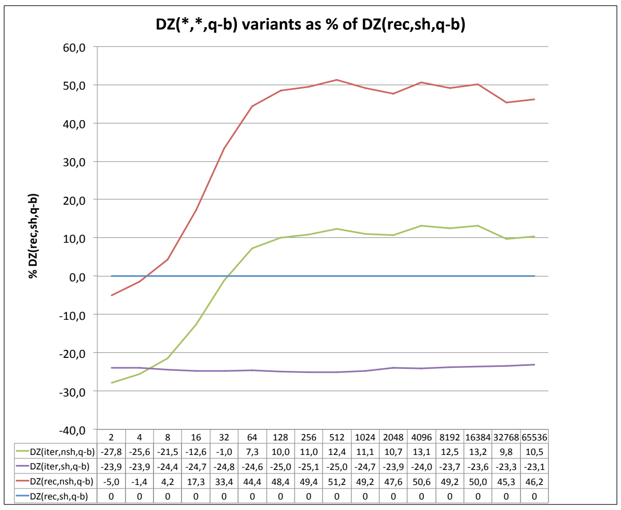
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,h-h)	-29,7	-28,4	-27,1	-24,9	-25,1	-24,0	-24,7	-24,4	-25,3	-24,3	-24,2	-24,6	-24,8	-24,1	-24,9	-24,2
DZ(iter,sh,h-h)	-24,2	-25,2	-25,2	-25,4	-25,3	-25,2	-25,4	-25,6	-25,7	-25,6	-25,4	-25,5	-25,3	-25,3	-17,9	-25,1
DZ(rec,nsh,h-h)	-18,9	-12,5	-10,0	-6,9	-7,2	-5,5	-6,4	-6,1	-7,1	-6,0	-6,1	-6,7	-7,3	-6,3	-7,6	-6,7
DZ(rec,sh,h-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(d) Cost of Recursion of *DZ(*,*,h-h)* using a genome text



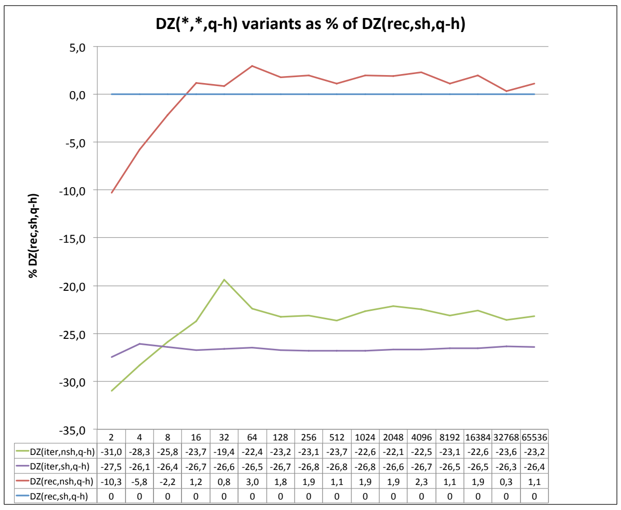
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
🟩 DZ(iter,nsh,h-q)	-24,3	-25,4	-24,2	-22,5	-22,2	-21,5	-21,8	-21,7	-22,1	-21,9	-21,4	-22,1	-21,9	-21,2	-21,9	-21,0
🟪 DZ(iter,sh,h-q)	-24,1	-26,0	-26,5	-25,9	-26,7	-26,5	-26,6	-26,8	-26,7	-26,7	-26,5	-26,7	-26,5	-26,5	-26,3	-26,4
🟥 DZ(rec,nsh,h-q)	-13,4	-12,7	-10,3	-7,9	-8,3	-7,0	-6,7	-6,8	-7,3	-7,3	-7,0	-7,4	-7,8	-6,6	-8,2	-6,9
🟦 DZ(rec,sh,h-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(e) Cost of Recursion of *DZ(*,*,h-q)* using a genome text



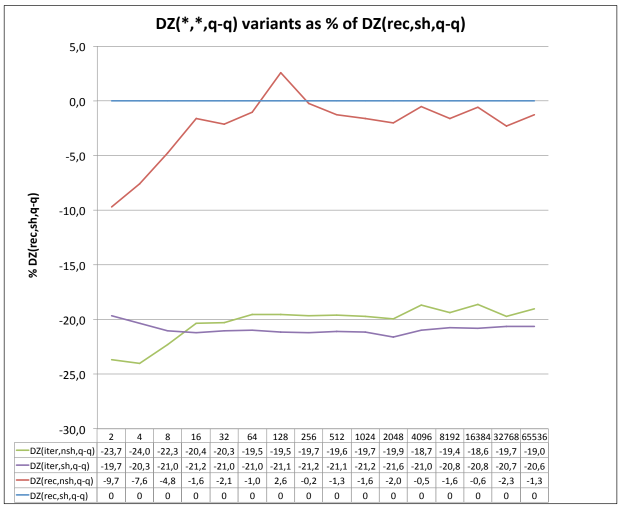
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
🟩 DZ(iter,nsh,q-b)	-27,8	-25,6	-21,5	-12,6	-1,0	7,3	10,0	11,0	12,4	11,1	10,7	13,1	12,5	13,2	9,8	10,5
🟪 DZ(iter,sh,q-b)	-23,9	-23,9	-24,4	-24,7	-24,8	-24,6	-25,0	-25,1	-25,0	-24,7	-23,9	-24,0	-23,7	-23,6	-23,3	-23,1
🟥 DZ(rec,nsh,q-b)	-5,0	-1,4	4,2	17,3	33,4	44,4	48,4	49,4	51,2	49,2	47,6	50,6	49,2	50,0	45,3	46,2
🟦 DZ(rec,sh,q-b)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(f) Cost of Recursion of *DZ(*,*,q-b)* using a genome text



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
🟩 DZ(iter,nsh,q-h)	-31,0	-28,3	-25,8	-23,7	-19,4	-22,4	-23,2	-23,1	-23,7	-22,6	-22,1	-22,5	-23,1	-22,6	-23,6	-23,2
🟪 DZ(iter,sh,q-h)	-27,5	-26,1	-26,4	-26,7	-26,6	-26,5	-26,7	-26,8	-26,8	-26,8	-26,6	-26,7	-26,5	-26,5	-26,3	-26,4
🟥 DZ(rec,nsh,q-h)	-10,3	-5,8	-2,2	1,2	0,8	3,0	1,8	1,9	1,1	1,9	1,9	2,3	1,1	1,9	0,3	1,1
🟦 DZ(rec,sh,q-h)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

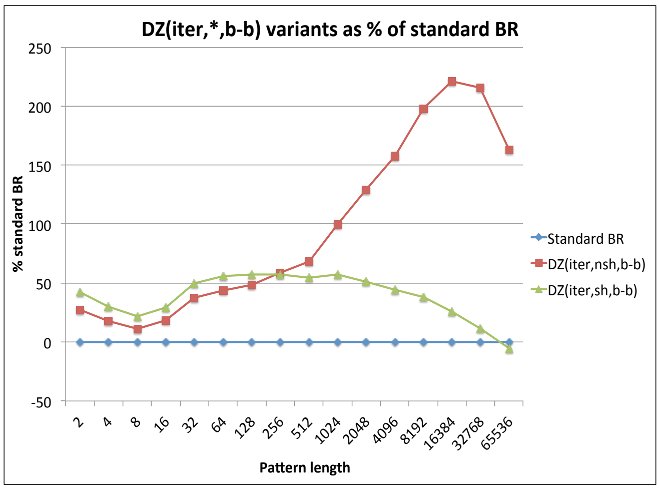
(g) Cost of Recursion of *DZ(*,*,q-h)* using a genome text



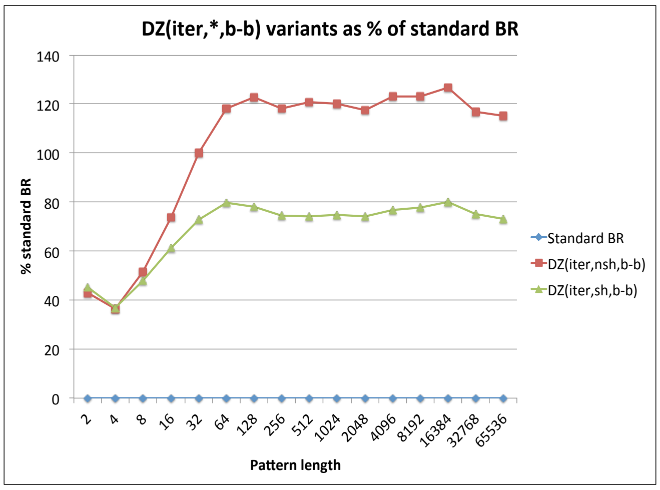
	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
DZ(iter,nsh,q-q)	-23,7	-24,0	-22,3	-20,4	-20,3	-19,5	-19,5	-19,7	-19,6	-19,7	-19,9	-18,7	-19,4	-18,6	-19,7	-19,0
DZ(iter,sh,q-q)	-19,7	-20,3	-21,0	-21,2	-21,0	-21,0	-21,1	-21,2	-21,1	-21,2	-21,6	-21,0	-20,8	-20,8	-20,7	-20,6
DZ(rec,nsh,q-q)	-9,7	-7,6	-4,8	-1,6	-2,1	-1,0	2,6	-0,2	-1,3	-1,6	-2,0	-0,5	-1,6	-0,6	-2,3	-1,3
DZ(rec,sh,q-q)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(h) Cost of Recursion of *DZ(*,*,q-q)* using a genome text ecoli

Figure C.4: Cost of Recursion of multiple shifter DZ variants using a genome text

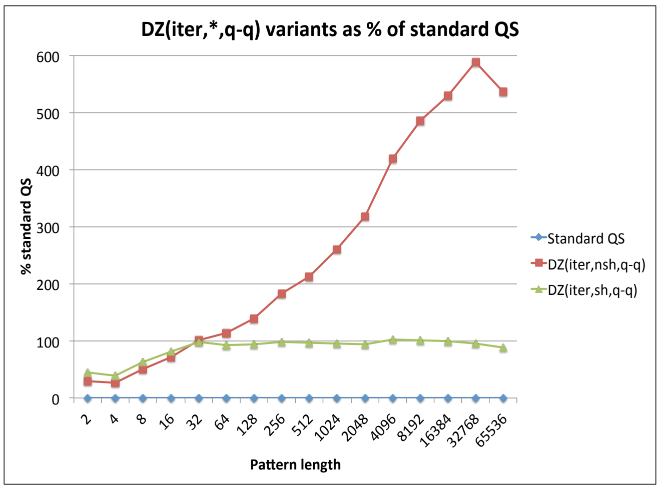


(a) *DZ(iter,*,b-b)* compared to standard Berry-Ravindran using a natural language text

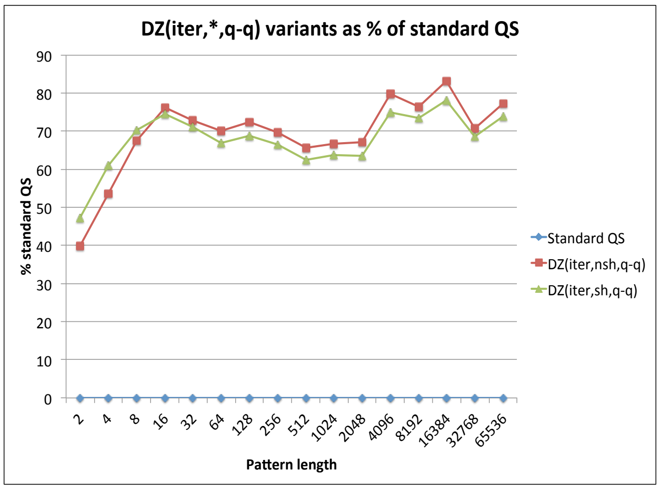


(b) *DZ(iter,*,b-b)* compared to standard Berry-Ravindran using a genome text

Figure C.5: *DZ(iter,*,b-b)* compared to the standard Berry-Ravindran



(a) *DZ(iter,*,q-q)* compared to standard Quick Search using a natural language text



(b) *DZ(iter,*,q-q)* compared to standard Quick Search using a genome text

Figure C.6: *DZ(iter,*,q-q)* compared to standard Quick Search

Appendix D

Parallel Benchmark Figures

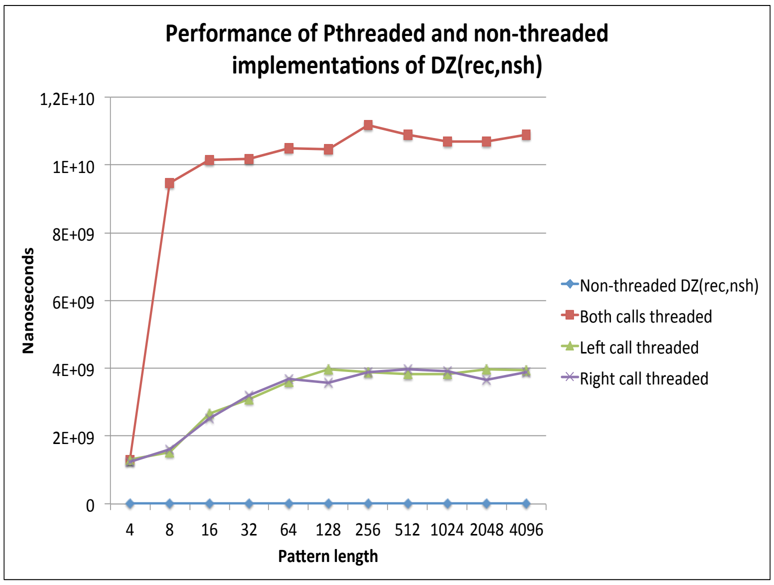


Figure D.1: Raw averaged minimum time data with Pthreads and a natural language text

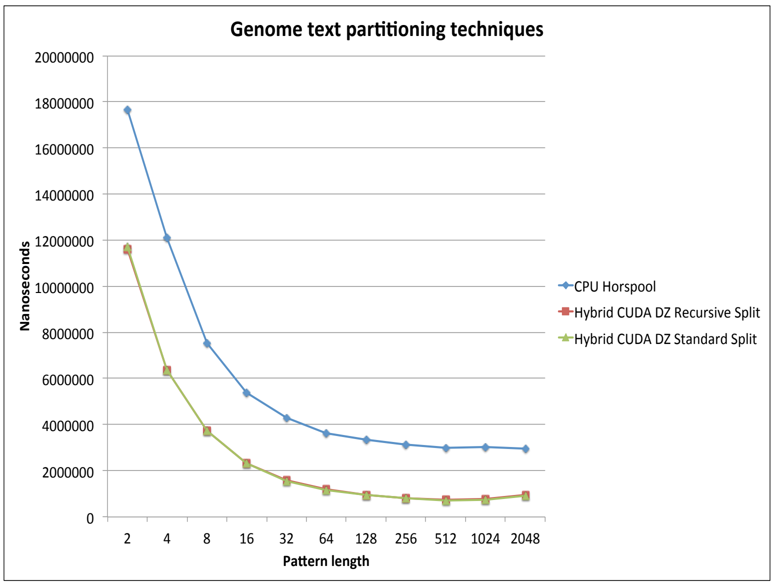
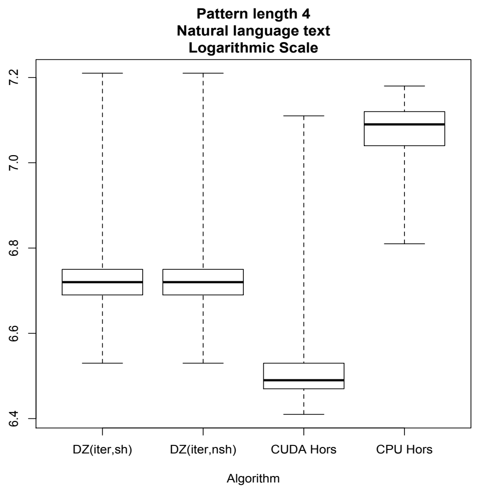
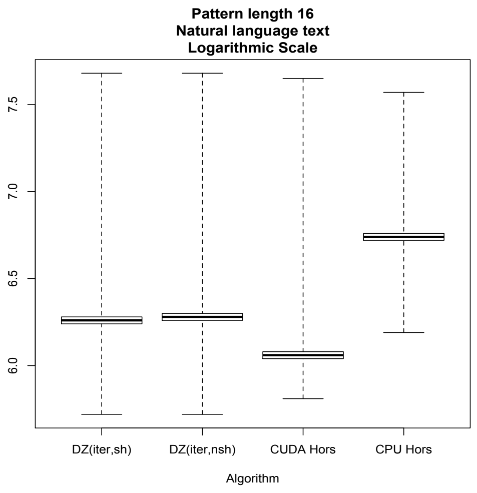


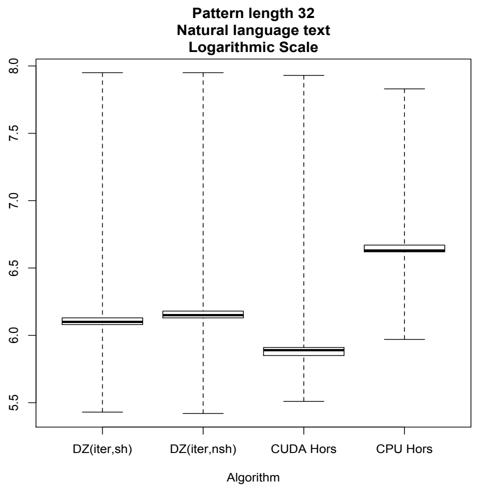
Figure D.2: Splitting a genome text with *DZ(rec,nsh)* versus division into equal-sized chunks



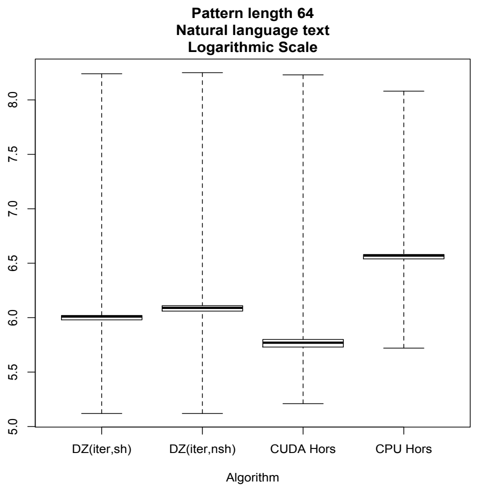
(a) Pattern length 4



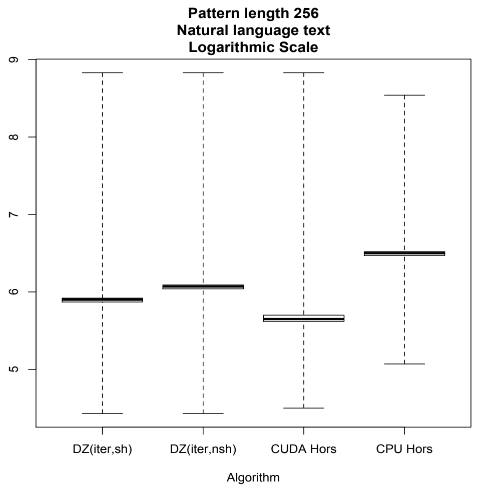
(b) Pattern length 16



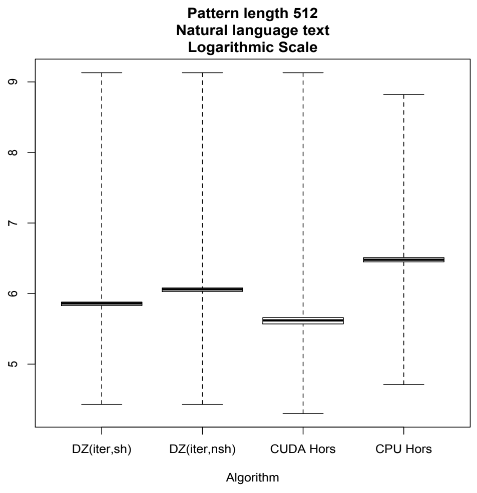
(c) Pattern length 32



(d) Pattern length 64

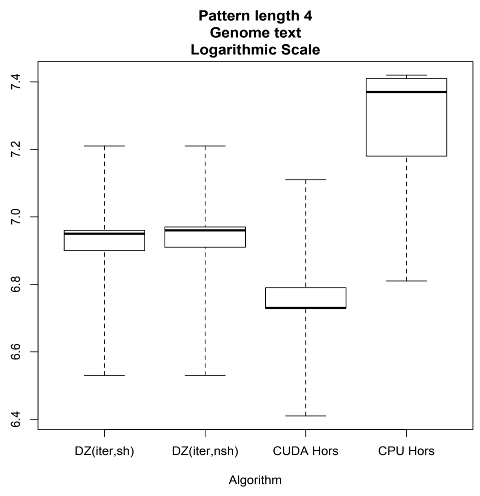


(e) Pattern length 256

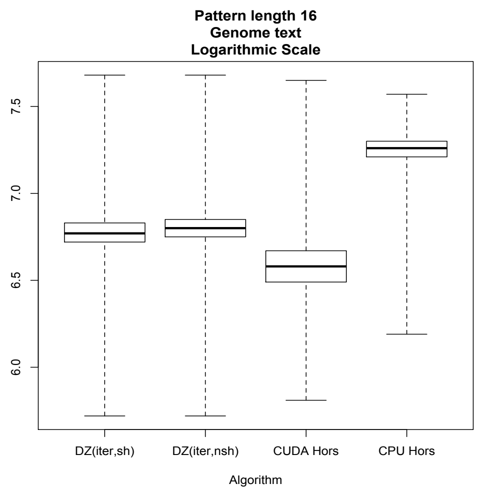


(f) Pattern length 512

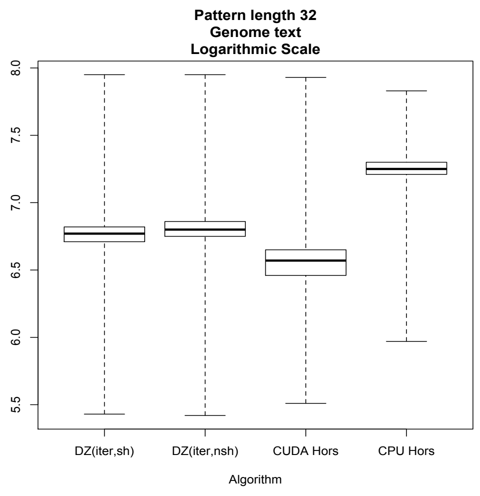
Figure D.3: Optimised CUDA implementations with a natural language text



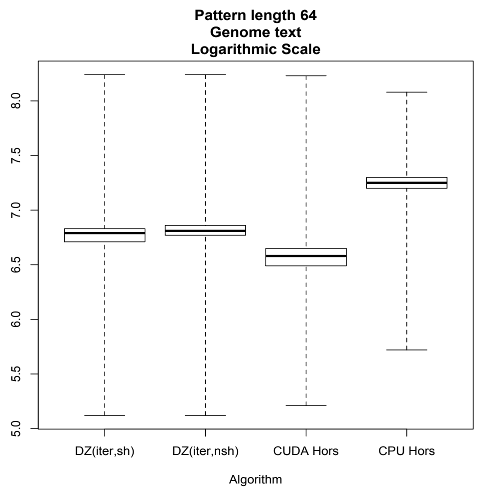
(a) Pattern length 4



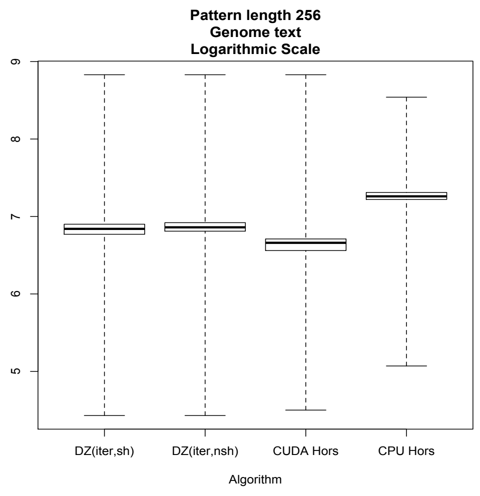
(b) Pattern length 16



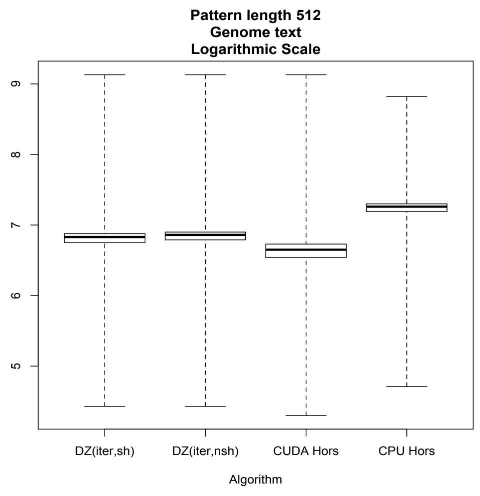
(c) Pattern length 32



(d) Pattern length 64



(e) Pattern length 256

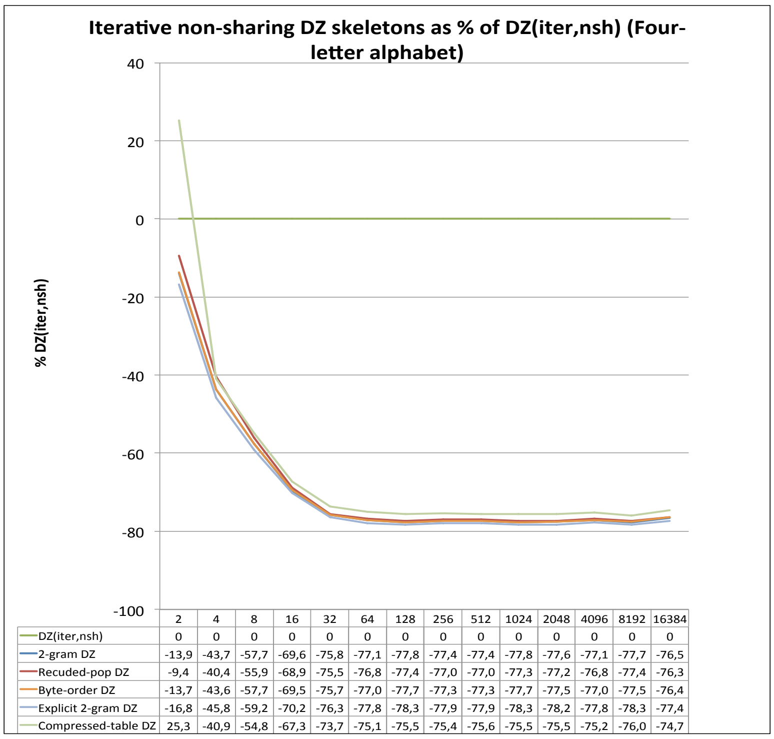


(f) Pattern length 512

Figure D.4: Optimised CUDA implementations with a genome text

Appendix E

Dead-Zone Skeletons Benchmark Figures



	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
DZ(iter,nsh)	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2-gram DZ	-13,9	-43,7	-57,7	-69,6	-75,8	-77,1	-77,8	-77,4	-77,4	-77,8	-77,6	-77,1	-77,7	-76,5
Recuded-pop DZ	-9,4	-40,4	-55,9	-68,9	-75,5	-76,8	-77,4	-77,0	-77,0	-77,3	-77,2	-76,8	-77,4	-76,3
Byte-order DZ	-13,7	-43,6	-57,7	-69,5	-75,7	-77,0	-77,7	-77,3	-77,3	-77,7	-77,5	-77,0	-77,5	-76,4
Explicit 2-gram DZ	-16,8	-45,8	-59,2	-70,2	-76,3	-77,8	-78,3	-77,9	-77,9	-78,3	-78,2	-77,8	-78,3	-77,4
Compressed-table DZ	25,3	-40,9	-54,8	-67,3	-73,7	-75,1	-75,5	-75,4	-75,6	-75,5	-75,5	-75,2	-76,0	-74,7

Figure E.1: Impact of 2-grams when matching a genome text

Bibliography

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. Commun. ACM, 18(6):333–340, June 1975.

[2] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[3] Apple. caffeinate(8) Mac OS X Manual Page, 2012. Available at `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man8/caffeinate.8.html`.

[4] Apple. Kernel Programming Guide: Mach Overview, 2012. Available at `https://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/Mach/Mach.html`.

[5] Apple. High Precision Timers in iOS / OS X, 2013. Available at `https://developer.apple.com/library/ios/technotes/tn2169/_index.html`.

[6] Mikhail J. Atallah and Marina Blanton. Algorithms and Theory of Computation Handbook. Chapman & Hall/CRC, 2nd edition, 2009.

[7] K. Berlin, J. Huan, M. Jacob, G. Kochhar, J. Prins, B. Pugh, P. Sadayappan, J. Spacco, and C.W. Tseng. Evaluating the Impact of Programming Language Features on the Performance of Parallel Applications on Cluster Architectures. Languages and Compilers for Parallel Computing, pages 194–208, 2004.

[8] Thomas Berry and Somasundaram Ravindran. A fast string matching algorithm and experimental results. In Jan Holub, editor, Proceedings of the Prague Stringology Club Workshop '99, pages 16–26, Czech Technical University in Prague, Czech Republic, 1999.

[9] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. Commun. ACM, 20(10):762–772, October 1977.

[10] Adam Drozdek. Data Structures and Algorithms in Java. Delmar Learning, 3rd edition, 2008.

[11] Simone Faro and Thierry Lecroq. The exact string matching problem: a comprehensive experimental evaluation. arXiv preprint arXiv:1012.2547, 2010.

[12] Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. ACM Comput. Surv., 45(2):13:1–13:42, March 2013.

[13] International Organization for Standardization (ISO Working Group 15 of Subcommittee SC 22). Portable operating system interface (POSIX) base specifications. ISO/IEC/IEEE 9945, 7, 2009.

[14] International Organization for Standardization (ISO Working Group 21 of Subcommittee SC 22). Technical report on C++ performance. ISO/IEC TR 18015, E, 2006.

[15] David Gregg. personal communication, June 2012. Trinity College Dublin, Ireland.

[16] Mark Harris. How to implement performance metrics in cuda c/c++. `http://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/`, 2012.

[17] R. Nigel Horspool. Practical fast searching in strings. Software Practice and Experience, 10:501–506, 1980.

[18] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In Proceedings of the 2013 International Symposium on Memory Management, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[19] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. IBM J. Res. Dev., 31(2):249–260, March 1987.

[20] David B. Kirk and Wen-mei W. Hwu. Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[21] D. Knuth, J. Morris, Jr., and V. Pratt. Fast pattern matching in strings. SIAM Journal on Computing, 6(2):323–350, 1977.

[22] Derrick G. Kourie, Bruce W. Watson, Tinus Strauss, Loek Cleophas, and Melanie Mauch. Empirically assessing algorithm performance. In Proceedings of the Southern African Institute for Computer Scientist and Information Technologists Annual Conference 2014 on SAICSIT 2014 Empowered by Technology, SAICSIT '14, pages 115:115–115:125, New York, NY, USA, 2014. ACM.

[23] Thierry Lecroq and Simone Faro. SMART: a string matching algorithm research tool, 2011. Available at `http://www.dmi.unict.it/~faro/smart/`.

[24] Melanie Mauch, Derrick G. Kourie, Bruce W. Watson, and Tinus Strauss. Performance assessment of dead-zone single keyword pattern matching. In

Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT '12, pages 59–68, New York, NY, USA, 2012. ACM.

[25] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[26] Gonzalo Navarro and Mathieu Raffinot. Flexible Pattern Matching in Strings: Practical On-line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, New York, NY, USA, 2002.

[27] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. " O'Reilly Media, Inc.", 1996.

[28] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. NVIDIA Corporation, 2007.

[29] NVIDIA Corporation. CUDA parallel computing platform, 2015. Available at http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/clock_getres.2?query=clock_gettime.

[30] OpenBSD. OpenBSD manual pages, 2015. Available at http://www.nvidia.com/object/cuda_home_new.html.

[31] Vreda Pieterse and Paul E. Black. Algorithms and theory of computation handbook. In Dictionary of Algorithms and Data Structures. CRC Press LLC, 1999. Available at http://xlinux.nist.gov/dads//HTML/singleprogrm.html.

[32] Vreda Pieterse and David Flater. The ghost in the machine: don't let it haunt your software performance measurements. Technical Note 1830, National Institute of Standards and Technology, Doi http://dx.doi.org/10.6028/NIST.TN.1830, April 2014.

[33] Daniel M. Sunday. A very fast substring search algorithm. Commun. ACM, 33(8):132–142, August 1990.

[34] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pages 269–280, New York, NY, USA, 2010. ACM.

[35] Jorma Tarhio. personal communication, May 2015. Aalto University, Helsinki, Finland.

[36] Jan Vitek and Tomas Kalibera. R3: Repeatability, reproducibility and rigor. SIGPLAN Not., 47(4a):30–36, March 2012.

[37] Bruce W. Watson, Derrick G. Kourie, and Tinus Strauss. A sequential recursive implementation of dead-zone single keyword pattern matching. In W. F. Smyth, editor, Proceedings of the International Workshop on Combinatorial Algorithms (IWOCA 2012), Tamil Nadu, India, July 2012.

[38] Bruce W. Watson and Richard E. Watson. A new family of string pattern matching algorithms. South African Computer Journal, 30:34–41, June 2003. For rapid access, a reprint of this article appears on `www.fastar.org`. This journal remains the appropriate citation reference.