UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Process-based Decomposition & Multicore Performance

Case Studies from Stringology

by
Marthinus David Strauss

Submitted in partial fulfilment of the requirements for the degree Philosophiae Doctor (Computer Science) in the Faculty of Engineering, Built Environment and Information Technology, University of Pretoria, Pretoria

2017

Process-based Decomposition and Multicore Performance: Case Studies from Stringology

by
Marthinus David Strauss

Abstract

Current computing hardware supports parallelism at various levels. Conventional programming techniques, however, do not utilise efficiently this growing resource. This thesis seeks a better fit between software and current hardware while following a hardware-agnostic software development approach. This allows the programmer to remain focussed on the problem domain. The thesis proposes process-based problem decomposition as a natural way to structure a concurrent implementation that may also improve multicore utilisation and, consequently, run-time performance.

The thesis presents four algorithms as case studies from the domain of string pattern matching and finite automata. Each case study is conducted in the following manner. The particular sequential algorithm is decomposed into a number of communicating concurrent processes. This decomposition is described in the process algebra CSP. Hoare's CSP was chosen as one of the best known process algebras, for its expressive power, conciseness, and overall simplicity.

Once the CSP-based process description has brought ideas to a certain level of maturity, the description is translated into a process-based implementation. The Go programming language was used for the implementation as its concurrency features were inspired by CSP. The performance of the process-based implementation is then compared against its conventional sequential version (also provided in Go).

The goal is not to achieve maximal performance, but to compare the run-time performance of an "ordinary" programming effort that focussed on a process-based solution over a conventional sequential implementation.

Although some implementations did not perform as well as others, some did significantly outperform their sequential counterparts. The thesis thus provides *prima facie* evidence that a process-based decomposition approach is promising for achieving a better fit between software and current multicore hardware.

Keywords: CSP, concurrency, Go, finite automata, regular expressions, process based decomposition
Degree: Ph.D (Computer Science)
Supervisors: D. G. Kourie and B. W. Watson

Publications

The following publications are included in parts or in an extended version in this thesis:

- T. Strauss, D. G. Kourie and B. W. Watson (2008a). 'A Concurrent Specification of an Incremental DFA Minimisation Algorithm'. In: *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*. Ed. by J. Holub and J. Zdárek. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, pp. 218–226.
- T. Strauss, D. G. Kourie and B. W. Watson (2008b). 'A Concurrent Specification of Brzozowski's DFA Construction Algorithm'. In: *International Journal of Foundations of Computer Science* 19.1, pp. 125–135.
- T. Strauss, D. G. Kourie, B. W. Watson and L. Cleophas (2014). 'A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm'. In: *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 17–29.
- T. Strauss, D. G. Kourie, B. W. Watson and L. Cleophas (2015). 'Process-Based Aho-Corasick Failure Function Construction'. In: *Communicating Process Architectures 2015. Proceedings of the 37th WoTUG Technical Meeting, 23–26 August 2015, University of Kent, UK*. ed. by K. Chalmers et al. Open Channel Publishing Ltd., pp. 183–206. URL: `http://wotug.org/cpa2015/programme.shtml`.
- T. Strauss, D. G. Kourie, B. W. Watson and L. Cleophas (2017). 'CSP for Parallelising Brzozowski's DFA Construction Algorithm'. In: *The Role of Theory in Computer Science: Essays Dedicated to Janusz Brzozowski*. Ed. by S. Konstantinidis et al. To appear, pp. 217–243. URL: `http://www.worldscientific.com/worldscibooks/10.1142/10239`.
- B. W. Watson, D. G. Kourie and T. Strauss (2012). 'A Sequential Recursive Implementation of Dead-Zone Single Keyword Pattern Matching'. In: *Combinatorial Algorithms, 23rd International Workshop,*

IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers. Ed. by S. Arumugam and W. F. Smyth. Vol. 7643. Lecture Notes in Computer Science. Springer, pp. 236–248. URL: http://dx.doi.org/10.1007/978-3-642-35926-2.

Acknowledgements

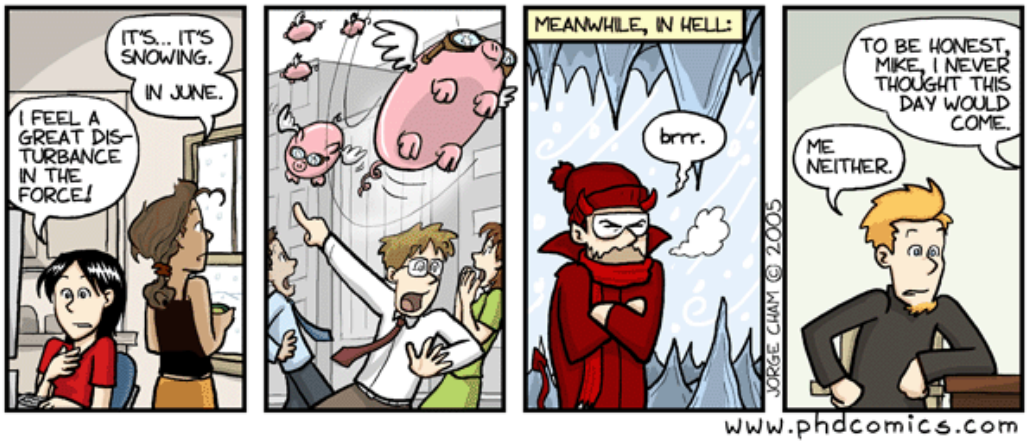
Naturally, a project of this nature does not happen without the support of others. I wish to thank the following people for their contribution to the completion of this project.

- The WoTUG group who organises the Communicating Process Architectures (CPA) conferences. I attended CPA2015 and received valuable input regarding the CSP and process-oriented aspects of my work.
- The friendly people at the Prague Stringology Club for their hospitality and welcoming conferences that I have attended more than once.
- The FASTAR research group, particularly Bruce Watson and Loek Cleophas.
- My supervisor Derrick Kourie for many years of guidance and conversation—not only academically, but also personally.
- My family for tolerating far too many years of "working on my PhD". Janet, thank you for your support, patience, and all important inspiration. Martiens and Isabella, you had to wait your entire lives for me to finish. Thank you for hanging in there!

I want to share with my family the two comic strips on the next page. The first is for Janet and Martiens who, I am sure, are very relieved that I have finally finished. To be honest, I also doubted whether the day would ever come.

The second strip is for Isabella who regularly joined me at my desk—also to work.

"Piled Higher and Deeper" by Jorge Cham www.phdcomics.com



"Piled Higher and Deeper" by Jorge Cham www.phdcomics.com

Contents

List of Figures

List of Tables

1 Introduction

> Concurrency is fundamental to the workings of the universe. It exists at all levels of granularity (e.g. nanoscale, human, astronomic). Complex, interesting and useful behaviour emerges from the concurrent actions of zillions of processes, each managing its own – *and only its own* – state, and synchronising and communicating to enable and/or constrain each others' individual behaviours.
>
> *P. H. Welch and J. B. Pedersen*

The term *impedance mismatch* has been used to describe the ill fit between two widely used technologies: relational databases for storing data in tables, and object orientation for building applications out of software objects (Ambler 2016). To alleviate the problem, object-relational frameworks have been developed to map between the different domain models, Red Hat's (2016) Hibernate for the Java language being one such example.

There appears to be an analogous impedance mismatch when running traditional sequential algorithms on hardware platforms that are increasingly multicore and parallel in nature (Lee 2006; McDougall 2005; Meade, Buckley and Collins 2011; Patterson 2010; Sutter 2005, 2012). Here too, there has been some effort at providing tools to bridge the gap between the different domain models, OpenMP (2016) being an example of an API to support parallel programming in C, C++ and Fortran. However, use of such a tool typically involves the alteration of an existing sequential implementation via a compiler directive and the invocation of library routines, rather than by reconsidering problems in a new light. Conventional programmers are thus screened from the impact of this mismatch by layers of systems software (compiler- and/or operating system) and this tends to leave them in the comfort zone of a sequential software paradigm. Nevertheless, a cursory examination of core activity on multicore machines will reveal that, invariably, processing load is unevenly distributed over the available cores when a sequential program is run.

The work reported here was inspired by this perception. The purpose is to seek ways of bringing about a better fit between software and current hardware by using process-based problem decomposition in the spirit of

process-oriented programming (Sampson 2008; Welch and Pedersen 2010). There are many underlying questions. Is it feasible to re-conceptualise, in a process-based fashion, solutions to problems that have traditionally been solved in a sequential manner? To what extent do available specification notations support such a re-conceptualisation? What are appropriate programming languages that can be used to implement such specifications? Do such process-based solutions result in better use of available cores? And, most importantly, can process-based solutions result in interesting speedups?

1.1 Scope

Naturally, in a limited-resource project such as this, one cannot provide comprehensive answers to these many questions.

Instead, practical boundaries have to be determined to match the time and resources available for the project, in the hope that future research may more deeply explore some of the issues suggested by the conclusions reached here. This section outlines those boundaries.

The main focus here is on gathering evidence to support the hypothesis that process-based decomposition enables one to better utilise modern hardware, yielding improved software run-time performance. For pragmatic reasons, such evidence is gathered subject to the following limitations:

- Only a limited number of problems commonly solved by sequential algorithms are considered. Section 1.3 provides more details regarding the specific domain.
- No formal attempt is made to develop a software development methodology. Nor is an existing software development process (such as Agile, RUP, etc) followed. Rather, concurrent software is developed in an intuitive, *ad hoc* manner as described in Section 1.3. Nevertheless, the experiences in doing this are recorded and could inform future research to develop an appropriate software methodology for this kind of software development.
- A single machine will be used for performance experiments. It is for future research to verify the results to various other hardware platforms.
- After a preliminary investigation of concurrency formalisms it was decided, as mentioned in Section 1.2, to use Communicating Sequential Processes (CSP). No attempt is made, therefore, to cross-compare

the advantages or disadvantages of such formalisms against one another.

- In a similar manner, after a preliminary investigation into languages that support concurrency, it was decided to use Go. No further comparison of different programming languages is made.
- Problem solutions are modelled in CSP and then implemented in Go. No specific CSP verification tools are used to check the CSP for correctness with respect to safety and liveness properties. Neither is a formal process followed in mapping from CSP to Go, although the experiences recorded here could inform future attempts to develop heuristics in this regard.
- The concern is not with fine-grained performance differences in various implementations. Hence statistical hypothesis testing is not done. Rather, descriptive statistics and graphs are used to highlight course-grained order-of-magnitude performance differences.
- Only data relating to run-time speedup of the process-based implementations against the sequential implementations is collected. Other performance-related data such as memory, core or cache utilisation, is not collected and no attempt is made to uncover deep explanations for performance differences in terms of cache effects or hardware configurations.
- No specific effort was made to maximally refine the process-based implementations so as to squeeze out the last drop of performance! Rather, the intent is to explore the run-time performance implications of an "ordinary" programming effort that is focussed *ab initio* on a process-based decomposition, over and against a conventional sequential implementation. The resulting process-based concurrent architecture need not be the only or best way in which the problem at hand may be formulated as a set of concurrent interacting processes. The aspiration is more modest: to achieve better core utilisation with at least moderate speedup and to do this with minimal refactoring of an initial process-based solution.

Before outlining the study's methodology, it is appropriate to contextualise the notion of concurrency and to indicate why CSP and Go are well-suited to be used as representative notations in the scope of the study.

1.2 Concurrent programming

Concurrency and *parallelism* are two related, but different ideas (Buhr and Harji 2005; Gerrand 2013; Harper 2011; Pike 2012a). This text follows Turon

(2013) and defines concurrency as the arbitrarily overlapped execution of processes whereas parallelism is seen as the simultaneous execution of computations. Overlapped execution does not entail simultaneous execution. Consider, for example, the multiprogramming that takes place on a uniprocessor computer. Here, execution is overlapped in the sense that each of several programs is allocated a slice of time to execute on the single processor, then swapped out for the next program to execute for its allocated time-slice, typically in a round-robin fashion.

These definitions (of concurrency versus parallelism) are congruent with the idea of Buhr and Harji (2005) that concurrency is the *logical* concept of actions happening at the same time and parallelism is the *physical* concept of actions happening at the same time. Concurrency is a system structuring mechanism and parallelism is a resource. A given machine has a certain capacity for parallelism and the goal is to maximise the throughput by intelligently utilising this resource.

In concurrent programming a problem is decomposed into a set of activities with synchronising properties. Brinch Hansen (2002) traces the origins of concurrent programming to Dijkstra (1965, 1968, 1971) and Hoare (1972). Dijkstra articulates the correctness criteria for processes sharing data and also introduces the problem of the *dining philosophers*. Hoare is the first to attempt extending programming languages with features for concurrent programming.

Since then various models or theories of concurrency have been developed. These include transition systems like *Petri nets* (Petri and Reisig 2008) and *input/output automata* (Lynch and Tuttle 1989). In the *Actor model* (Agha 1985; Hewitt 1977), a concurrent process passess messages asynchronously to a particular process. *Concurrent separation logic* (Brookes 2007; O'Hearn 2007; Reynolds 2002), allows for correctness-proofs of concurrent programs in which "ownership" of critical variables is transferred dynamically between concurrent processes. *Software transactional memory* (Shavit and Touitou 1995) offers an alternative to traditional lock-based synchronisation by employing the transaction concept of database systems. All modifications to memory appear to happen at the moment a transaction commits. Unsuccessful transactions are aborted and need to be reattempted. *Process algebras* provide notations for the description and analysis of the interaction, communication, and synchronisation of concurrent processes. Tools are available for several process algebras to assist in the analysis of the specified concurrent system. Some of the better known process algebras are CSP (see Section 2.5), Calculus of Communicating Systems (CCS) (Milner 1989), Algebra of Communicating Processes (ACP) (Bergstra and Klop 1984), and the π-calculus (Milner, Parrow and Walker 1992a,b).

In this thesis the process algebra CSP is used to describe the behaviour of

concurrent processes. CSP was chosen as one of the best known process algebras, for its expressive power, its conciseness, and overall simplicity.

Programming languages have also been developed to cater for concurrency. In the *shared memory* paradigm, processes have shared access to data and access is controlled via locking mechanisms. OpenMP, mentioned above, follows this approach. In contrast, processes in the *message passing* paradigm share data by sending messages to one another. A well-known example is the Message Passing Interface (MPI) (*MPI Forum* 2016) used in high performance computing.

For this research project, it was decided to use a programming language that supports message-passing concurrency. To select a suitable language, considerable experimentation was initially carried out with various candidate programming languages. An important consideration was to ensure that the selected language should be conceptually close to CSP. Erlang (Armstrong 2007; Erlang 2016) and D (Alexandrescu 2010; D 2016) were both considered since they support message passing in the language. However, they follow the Actor model in which processes send messages to other processes using the identity of the other process. In the CSP model on the other hand, messages are sent via channel end points. Whichever process is attached to the other end of the process receives the messages sent on the channel.

Since the choice was made to use CSP as the description language for processes, it is reasonable to use a programming language with concurrency features close to CSP. Library-based options include: JCSP (Welch 2002; Welch, Brown et al. 2007) for Java, C++CSP (Brown and Welch 2003) for C++, and CHP (Brown 2008) for Haskell. These were not considered because the requirement is for support within the existing parent language. Occam-π (Barnes and Welch 2016; Welch and Barnes 2005) was considered, but defining new recursive data structures proved to be a problem. In the end, the Go programming language (Go 2016) was chosen. It supports lightweight processes and message passing via synchronous channels.

1.3 Methodology

The research method that this research project has used falls under the heading of case study research. It should be noted that it is in the very nature of such a case study that choices have to be made, both with respect to the domain from which algorithms were selected for study and with respect to the range of specific algorithms within that domain. In principle, the algorithms chosen should, in some sense, be representative of the spectrum of algorithms generally encountered in software development. However, it is not immediately self-evident how to make such an

assessment in regard to the representivity. Neither was it considered critical in the present context to delve too deeply into this question. This is because the intent was to gather *prima facie* evidence for the conclusions reached, rather than to arrive at a final incontrovertible conclusion. Consequently, an initial choice of algorithms was made, based on pragmatic considerations, in a domain in which there was ready access to a local pool of domain expertise. Whether and how the conclusions reached from this present research are manifested in a wider range of algorithms, is a matter for future research.

The cases considered here are algorithms from the domain of Stringology, in particular: string pattern matching and finite automaton construction and minimisation. An attempt was made to select not only *embarrassingly parallel* problems (Moler 2013), but rather problems in which there may be significant inter-dependence among tasks.

An embarrassingly parallel problem is one that can naively be parallelised by running in parallel several instances of an existing sequential algorithm on partitions of the problem and then appropriately combining the results from each instance.

Of course the present project is only one of several efforts at developing concurrent algorithms for Stringological problems. Some previous efforts include Choi and Burgstaller (2013) and Ziadi and Champarnaud (1999) for finite automaton construction, Burgstaller et al. (2011), Holub and Štekr (2009) and Ko et al. (2012) for membership testing, and JáJá and Ryu (1996), Ravikumar and Xiong (1996) and Tewari, Srivastava and Gupta (2002) for minimisation. Hanneforth and B. W. Watson (2012) consider the parallelisation of finite automaton determinisation.

The case studies presented in the subsequent chapters have been conducted in the following manner.

First, the sequential algorithm was decomposed into communicating processes and described in CSP. In this project, the intention in developing a CSP description for the problem at hand has been limited to pragmatic considerations: specifically, the intention was to provide robust descriptions of concurrent, process-based, algorithms whose performance would scale with increasing numbers of processor cores. Although it would be relatively simple to use a model checker such as FDR3 (Gibson-Robinson et al. 2015) to assess correctness properties such as deadlock- and livelock-freedom, this has not been done in the research described below. Instead, CSP has been used primarily as a communication mechanism to interact with, to articulate, to refine, to challenge, and to evolve process-based design ideas.

Once the CSP-based specification process has brought ideas to a certain level of maturity, the specification is translated into a process-based Go implementation. It is then confirmed that the sequential and concurrent implementations produce the same results for a given input. The performance of the process-based algorithm is then compared against its conventional sequential implementation (also provided in Go). If variants of the

process-based implementation suggest themselves, they too may be articulated in CSP, implemented in Go, and further performance comparisons made.

1.4 Structure of thesis

The thesis is structured around four case studies. Before the case studies are presented, Chapter 2 provides background information. It provides a list of definitions from Stringology, and it also provides overviews of CSP, Go, and multiprocessor hardware.

The first case study is presented in Chapter 3 and considers the Aho-Corasick string pattern matching algorithm. The case focusses on one aspect of the initialisation phase of the algorithm, rather than the actual matching phase. This is followed by two case studies relating to finite automata. In Chapter 4 a construction algorithm is considered in which a deterministic finite automaton is constructed from an input regular expression. Minimisation of finite automata is the topic of Chapter 5. The algorithm under consideration determines whether pairs of states in a finite automaton are equivalent. The final case study returns to string matching and attempts to perform multiple match attempts concurrently. The latter example represents a problem that can be readily solved in an embarrassingly parallel fashion, and such an implementation is indeed also assessed.

The thesis closes in Chapter 7 with general observations and lessons to be learned from the case studies.

2 Background

> How complex or simple a structure is depends critically upon the way in which we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve this simplification, we must find the right representation.
>
> *Herbert A. Simon*

This chapter presents the necessary mathematical preliminaries as well as overviews of the languages used to describe algorithms in later chapters.

Section 2.1 introduces basic definitions such as quantification. Section 2.2 provides definitions relating to strings and formal languages, while Section 2.3 defines automata and related concepts. The material in these sections provides the theoretical underpinnings of, for example, the deterministic finite automaton (DFA) construction algorithm whose implementations are studied in Chapter 4. Readers already familiar with this material may wish to skip these sections. In reading subsequent chapters, use could be made of the backward references to the relevant definitions found here, whenever clarification is needed.

DFAs are defined in Def 2.28.

The remaining sections each provide a description of a notation that is employed in the thesis. Section 2.4 briefly overviews the Guarded Command Language (GCL)—the notation used in the thesis for specifying sequential algorithms. The process-based decompositions of these algorithms are presented in CSP, which is the theme of Section 2.5. Finally, Section 2.6 introduces the programming language Go that is used to implement the algorithms.

2.1 General definitions

In this section, some general notational definitions are presented.

Notation 2.1 (Quantification). *A basic understanding of the meaning of* quantification *is assumed. The following notation is used*

$$\langle \oplus\, a : R(a) : f(a) \rangle$$

where $\oplus$ *is an associative and commutative operator (to be quantified) with unit* $1_\oplus$*,* a *is a* dummy variable*,* R *is a* range predicate*, and* $f(a)$ *is the* quantification expression*.*

By definition, when the range is empty (i.e. when the range predicate $R(a)$ is *false*), then the entire quantification evaluates to the unit $1_\oplus$. For example, if $\oplus$ stands for $\forall$ quantification and if $R(a)$ is an empty range, then $\langle \forall\, a : R(a) : f(a) \rangle$ evaluates to *true*. This is because the unit, unit $1_\forall$ is *true*.

Notation 2.2 (Conditional Boolean operators). *Use* ***cand*** *and* ***cor*** *to refer to the* conditional *(also known as 'short circuit') equivalents of* $\wedge$ *and* $\vee$*, respectively.*

Notation 2.3 (Powerset). *For any set* A*, use* $\mathcal{P}(A)$ *to denote the set of all subsets (including the empty set,* $\emptyset$*) of* A*.*

2.2 Strings and languages

This section presents definitions and properties related to strings and languages. These definitions may be found in classical text such as Crochemore and Rytter (2003), Hopcroft, Motwani and Ullman (2007), Hopcroft and Ullman (1979), Smyth (2003) and B. W. Watson (1995).

Definition 2.4 (Alphabet). *An alphabet is a finite nonempty set of* symbols — *also known as* letters*.*

Whenever an alphabet is required, Σ will be used to denote the alphabet. In explicit form, it may be represented as $\{a_1, a_2, \ldots, a_n\}$.

Definition 2.5 (String). *A* string – *or* word – *is a finite sequence of symbols from* Σ*.*

Notation 2.6. *The length of a string* w *is the number of symbols in* w *and expressed* $|w|$*.*

Notation 2.7. *The symbol of string* w *in index position* $i \in [0, |w|)$ *is* $w_i \in \Sigma$*. Hence, the first symbol of* w *is* w_0 *and the last symbol is* $w_{|w|-1}$*.*

Notation 2.8 (Substring). *If* w *is a string, then, assuming valid indices,*

- $w_{[i,j)}$ *is the substring* $w_i w_{i+1} \cdots w_{j-1}$*,*

- $w_{[i,j]}$ *is the substring* $w_i w_{i+1} \cdots w_j$,
- $w_{(i,j]}$ *is the substring* $w_{i+1} w_{i+2} \cdots w_j$, *and*
- $w_{(i,j)}$ *is the substring* $w_{i+1} w_{i+2} \cdots w_{j-1}$.

Notation 2.9. Σ^* *denotes the set of all words, or strings, over* Σ *— including the empty string, written as* ε. *Furthermore,* $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.

Definition 2.10 (Language)**.** *L is a language over* Σ *if* $L \subseteq \Sigma^*$.

Definition 2.11 (Language concatenation)**.** *Language concatenation, denoted by the dot infix operator* $\cdot : \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \longrightarrow \mathcal{P}(\Sigma^*)$, *is defined as:*

$$L_0 \cdot L_1 = \{\, vw \mid v \in L_0 \wedge w \in L_1 \,\}$$

As a notational short-hand, the infix dot may be omitted and one may write $L_0 L_1$ *for the concatenation of* L_0 *and* L_1.

Definition 2.12 (Language exponentiation)**.** *Define language exponentiation recursively as follows:*

$$L^k = \begin{cases} \{\varepsilon\} & \text{if } k = 0 \\ L \cdot L^{k-1} & \text{if } k > 0 \end{cases}$$

Definition 2.13 (Closure of languages)**.** *The* Kleene closure, *also called the* star closure, *of language L is:*

$$L^* = \langle \cup\, i : 0 \leq i : L^i \rangle.$$

The plus closure *is:*

$$L^+ = \langle \cup\, i : 1 \leq i : L^i \rangle.$$

Note that $L^* = L^+ \cup \{\varepsilon\}$.

Definition 2.14 (Operations on languages)**.** *One may define the following operations on languages* L_0 *and* L_1 *over* Σ.

$$\begin{array}{rcll} L_0 \cap L_1 & = & \{\, v \mid v \in L_0 \wedge v \in L_1 \,\} & \textit{(intersection)} \\ L_0 \setminus L_1 & = & \{\, v \mid v \in L_0 \wedge v \notin L_1 \,\} & \textit{(relative difference)} \\ \neg L_0 & = & \Sigma^* \setminus L_0 & \textit{(negation)} \\ L_0^? & = & L_0 \cup \{\varepsilon\} & \textit{(optional)} \end{array}$$

Definition 2.15 (Regular languages)**.** *Define the set of regular languages REG over alphabet* Σ *inductively.*

$$\begin{array}{rcll} \emptyset & \in & REG & \\ \{\varepsilon\} & \in & REG & \\ \{a\} & \in & REG & \text{for } a \in \Sigma \end{array}$$

For languages L_0 and $L_1 \in REG$:

$$\begin{aligned} L_0^* &\in REG \\ L_0 \cdot L_1 &\in REG \\ L_0 \cup L_1 &\in REG \end{aligned}$$

Nothing else is in REG.

Definition 2.16 (String operators)**.** *Define string head and tail operators* $head : \Sigma^+ \longrightarrow \Sigma$ *and* $tail : \Sigma^+ \longrightarrow \Sigma^*$ *for* $a \in \Sigma, v \in \Sigma^*$ *as*

$$head(av) = a$$

and

$$tail(av) = v$$

Although the the two operators, *head* and *tail*, are defined above to operate on sequences of elements from Σ (i.e. on sequences of symbols) they generalise to act upon sequences of elements from an arbitrary set.

Definition 2.17 (String and language reversal)**.** *Given string w, define w^R to be the reversal of w, that is, the letters appear in reverse order. Inductively, for $a \in \Sigma$, $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$. For a language L, define $L^R = \{ w^R \mid w \in L \}$.*

Some algorithms require the *lexicographic* ordering (also known as the 'telephone book' ordering) on words in Σ^*. For this, assume a total ordering $\leq$ on alphabet Σ. (This is typically the ASCII ordering.)

Definition 2.18 (Lexicographic ordering of Σ^*)**.** *Ordering $\sqsubseteq_l$ is a total ordering on Σ^* that is defined inductively as follows. For all $a, b \in \Sigma$ and $v, w \in \Sigma^*$*

$$\varepsilon \sqsubseteq_l v$$

and

$$a \sqsubseteq_l b \equiv a \leq b$$

and

$$av \sqsubseteq_l bw \equiv \begin{cases} v \sqsubseteq_l w & \text{if } a = b \\ a \sqsubseteq_l b & \text{otherwise} \end{cases}$$

Example 2.19 (Lexicographic order)**.** *The words* `had`, `hard`, `he`, `head`, `heard`, `her`, `herd`, *and* `here` *are in lexicographic order.*

Definition 2.20 (Regular expressions)**.** *Let RE be the set of regular expressions over alphabet Σ. Define RE inductively as follows:*

$$\begin{aligned} \emptyset &\in RE \\ \varepsilon &\in RE \\ a &\in RE \quad \text{for all } a \in \Sigma \end{aligned}$$

For $E_0, E_1 \in RE$

$$\begin{array}{rcll} E_0 \cup E_1 & \in & RE & \textit{(union)} \\ E_0 \cdot E_1 & \in & RE & \textit{(concatenation)} \\ E_0^* & \in & RE & \textit{(Kleene closure)} \\ E_0^+ & \in & RE & \textit{(plus closure)} \\ E_0^? & \in & RE & \textit{(optional)} \\ \neg E_0 & \in & RE & \textit{(negation)} \\ E_0 \cap E_1 & \in & RE & \textit{(intersection)} \end{array}$$

The above is considered to be a definition of the *extended* regular expressions. The *basic* regular expressions only contain the union, concatenation, and Kleene closure operators. The symbol ε is used both for the empty string and for the regular expression representing the regular language comprising only the empty string. Similarly $\emptyset$ is used to represent the empty language as well as the regular expression for that language.

Note that the infix dot for regular expression concatenation may be omitted, so one may write E_0E_1 for $E_0 \cdot E_1$.

Definition 2.21 (Language of a regular expression)**.** *The language of a regular expression E is a set of strings $\mathcal{L}_{RE}(E) \subseteq \Sigma^*$ determined as follows. For $E_0, E_1 \in RE$:*

$$\begin{array}{rcl} \mathcal{L}_{RE}(\emptyset) & = & \emptyset \\ \mathcal{L}_{RE}(\varepsilon) & = & \{\varepsilon\} \\ \mathcal{L}_{RE}(a) & = & \{a\} \quad \textit{for } a \in \Sigma \\ \mathcal{L}_{RE}(E_0 \cup E_1) & = & \mathcal{L}_{RE}(E_0) \cup \mathcal{L}_{RE}(E_1) \\ \mathcal{L}_{RE}(E_0 \cdot E_1) & = & \mathcal{L}_{RE}(E_0) \cdot \mathcal{L}_{RE}(E_1) \\ \mathcal{L}_{RE}(E_0^*) & = & (\mathcal{L}_{RE}(E_0))^* \\ \mathcal{L}_{RE}(E_0^+) & = & (\mathcal{L}_{RE}(E_0))^+ \\ \mathcal{L}_{RE}(E_0^?) & = & (\mathcal{L}_{RE}(E_0))^?) \\ \mathcal{L}_{RE}(\neg E_0) & = & \neg \mathcal{L}_{RE}(E_0) \\ \mathcal{L}_{RE}(E_0 \cap E_1) & = & \mathcal{L}_{RE}(E_0) \cap \mathcal{L}_{RE}(E_1) \end{array}$$

Definition 2.22 (Nullability of a regular language)**.** *The nullability of a regular language $L \in REG$ is determined by the predicate $null_{REG}(L)$. The predicate's truth value is determined by the following equivalence*

$$null_{REG}(L) \equiv \varepsilon \in L$$

Definition 2.23 (Nullability of a regular expression)**.** *The nullability of a regular expression $E_0 \in RE$ is determined by the truth value of the predicate $null_{RE}(E)$. The predicate's truth value is specified by the following equivalences,*

where $E_0, E_1 \in RE$ represent arbitrary regular expressions

$$\begin{aligned}
null_{RE}(\emptyset) &\equiv false \\
null_{RE}(\varepsilon) &\equiv true \\
null_{RE}(a) &\equiv false \quad for\ a \in \Sigma \\
null_{RE}(E_0 \cup E_1) &\equiv null_{RE}(E_0) \vee null_{RE}(E_1) \\
null_{RE}(E_0 \cdot E_1) &\equiv null_{RE}(E_0) \wedge null_{RE}(E_1) \\
null_{RE}(E_0^*) &\equiv true \\
null_{RE}(E_0^+) &\equiv null_{RE}(E_0) \\
null_{RE}(E_0^?) &\equiv true \\
null_{RE}(\neg E_0) &\equiv \neg null_{RE}(E_0) \\
null_{RE}(E_0 \cap E_1) &\equiv null_{RE}(E_0) \wedge null_{RE}(E_1)
\end{aligned}$$

Derivatives of regular expressions were originally defined by Brzozowski (1964) and used *inter alia* for constructing DFAs (see below) from regular expressions. More recently Owens, Reppy and Turon (2009) report on derivatives' elegance for developing regular expression recognisers.

The construction of a DFA from a regular expression, using derivatives is the theme of Chapter 4.

Definition 2.24 (Derivative of a language)**.** *For a language $L \in \mathcal{P}(\Sigma^*)$, the derivative with respect to symbol $a \in \Sigma$ is defined as:*

$$a^{-1}L = \{\, v \mid av \in L \,\}$$

Definition 2.25 (Derivatives of regular expressions)**.** *The derivative of a regular expression with respect to symbol $a \in \Sigma$ is defined inductively as follows. Let E_0 and E_1 be regular expressions.*

$$\begin{aligned}
a^{-1}\emptyset &= \emptyset \\
a^{-1}\varepsilon &= \emptyset \\
a^{-1}a &= \varepsilon \\
a^{-1}b &= \emptyset \quad for\ b \neq a \\
a^{-1}(E_0 \cup E_1) &= a^{-1}E_0 \cup a^{-1}E_1 \\
a^{-1}(E_0 \cdot E_1) &= \begin{cases} a^{-1}E_0 \cdot E_1 \cup a^{-1}E_1 & if\ null_{RE}(E_0) \\ a^{-1}E_0 \cdot E_1 & if\ \neg null_{RE}(E_0) \end{cases} \\
a^{-1}(E_0^*) &= a^{-1}E_0 \cdot E_0^* \\
a^{-1}(E_0^+) &= a^{-1}E_0 \cdot E_0^* \\
a^{-1}(E_0^?) &= a^{-1}E_0 \\
a^{-1}(\neg E_0) &= \neg(a^{-1}E_0) \\
a^{-1}(E_0 \cap E_1) &= a^{-1}E_0 \cap a^{-1}E_1
\end{aligned}$$

Definition 2.26 (Equivalence of regular expressions)**.** *Regular expressions $E_0, E_1 \in RE$ are equivalent when they define the same language:*

$$equiv_{RE}(E_0, E_1) \equiv \mathcal{L}_{RE}(E_0) = \mathcal{L}_{RE}(E_1)$$

Definition 2.27 (Similarity of regular expressions)**.** *Two regular expressions*

E_0 and E_1 are similar (written $E_0 \sim E_1$) if and only if they are identical *or one can be transformed into the other using the following rules:*

1. $E_0 \cup E_1 = E_1 \cup E_0$
2. $(E_0 \cup E_1) \cup E_2 = E_0 \cup (E_1 \cup E_2)$
3. $E_0 \cup E_0 = E_0$

Similarity is an equivalence relation on regular expressions. If $E_0 \sim E_1$ then $\mathcal{L}_{RE}(E_0) = \mathcal{L}_{RE}(E_1)$. However, if $\mathcal{L}_{RE}(E_0) = \mathcal{L}_{RE}(E_1)$, then it is not necessarily the case that $E_0 \sim E_1$.

Brzozowski also mentions that the derivatives of regular expressions may be reduced by applying the following identities.

$$E_0 \cup \emptyset = E_0$$
$$E_0 \cdot \emptyset = \emptyset \cdot E_0 = \emptyset$$
$$E_0 \cdot \varepsilon = \varepsilon \cdot E_0$$

This means that the regular expressions on the left-hand side and the right-hand side of the identities are equivalent.

2.3 Automata

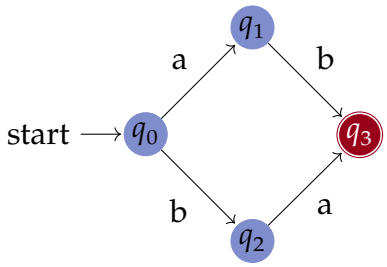
In this section, automata and related concepts are presented. These definitions are pertinent to both Chapter 4 and Chapter 5. In Chapter 4 the construction of automata from regular expressions, using derivatives of regular expressions, is considered. Likewise, Chapter 5 considers the so-called minimisation of automata.

Definition 2.28 (Deterministic finite automata)**.** *A DFA is a quintuple $(Q, \Sigma, \delta, s, F)$ where*

- *Q is a finite nonempty set of* states*.*
- *Σ is an alphabet.*
- *$\delta\colon Q \times \Sigma \longrightarrow Q \cup \{\perp\}$ is the transition function. The symbol $\perp$ is used to denote the invalid destination state of a transition.*
- *$s \in Q$ is the start state.*
- *$F \subseteq Q$ is the set of final states.*

In the literature, a common interpretation is to view $\perp$ as a *special* state, such as a sink state. Under such an interpretation, $\perp \in (Q \setminus F)$, $\langle \forall\ a : a \in \Sigma : \delta(\perp, a) = \perp \rangle$ and δ is a total function. Under an alternative interpretation, δ is a partial function, and $\delta(q, a) = \perp$ is used as a notational convenience to indicate that $\delta(q, a)$ is undefined—i.e. there is no transition from state q on symbol a.

Notation 2.29 (Drawing DFAs). *As shown below, automata are drawn in the standard way, whereby states are depicted as circles, a start state has an in-edge from nowhere and each final state is depicted as two concentric circles. Transitions are depicted as labelled directed edges. The state representing $\perp$, as well as the related transitions, are omitted from the drawing. Refer to this representation of δ as the* transition graph *of the DFA.*



Definition 2.30 (Acyclic DFA). *An* acyclic *DFA is a DFA with an acyclic transition graph.*

Definition 2.31 (Size of a DFA). *The size of DFA M, written $|M|$, is defined as $|Q|$.*

Notation 2.32. *For a state p, Σ_p denotes the subset of Σ on which p has out-transitions. That is,*

$$\Sigma_p = \{\, a \mid a \in \Sigma \wedge \delta(p, a) \neq \perp \,\}$$

Definition 2.33 (Confluence state). *A state p is a* confluence *state, written $Is\_confl(p)$, if and only if it has more than one in-transition. In Notation 2.29, state q_3 is a confluence.*

Definition 2.34. *A set of states X are confluence-free, written $Confl\_free(X)$, if and only if*

$$\langle \forall\ p : p \in X : \neg Is\_confl(p) \rangle$$

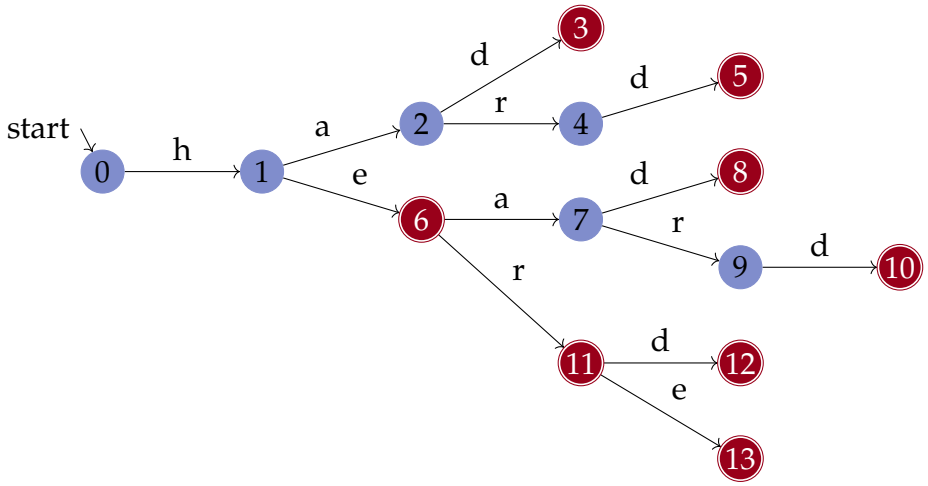
Definition 2.35 (Useless state). *A state p is useless if there is no path from the start state to p, or there is no path from p to a final state.*

Definition 2.36 (Trie). *The DFA, M, is a* trie, *written $Is\_trie(M)$, if and only if its transition graph is a tree rooted at start state s.*

The string matching algorithm in Chapter 3 utilises a trie.

Property 2.37 (Tries). *Tries have no confluence states.*

Example 2.38. *The following DFA is a trie:*



Property 2.39. *If a trie has no useless states then all leaves are final states.*

Definition 2.40 (The extension of δ)**.** *The extension of δ is defined as the function $\delta^* : Q \times \Sigma^* \longrightarrow Q \cup \{\perp\}$ where*

$$\delta^*(p, \varepsilon) = p$$

and for $a \in \Sigma, v \in \Sigma^*$

$$\delta^*(p, av) = \begin{cases} \delta^*(\delta(p,a), v) & \text{if } a \in \Sigma_p \\ \perp & \text{otherwise} \end{cases}$$

Definition 2.41 (Right language of a state)**.** *The right language of a state p, denoted by $\overrightarrow{\mathcal{L}}(p)$, is defined as*

$$\overrightarrow{\mathcal{L}}(p) = \{ w \mid \delta^*(p, w) \in F \}$$

That is, $\overrightarrow{\mathcal{L}}(p)$ is the set of strings traced out by the labels on paths from p to any final state.

Property 2.42. *Note that for $q \in F$, $\varepsilon \in \overrightarrow{\mathcal{L}}(q)$.*

Definition 2.43 (Left language of a state)**.** *The left language of a state p, denoted by $\overleftarrow{\mathcal{L}}(p)$, is defined as*

$$\overleftarrow{\mathcal{L}}(p) = \{ w \mid \delta^*(s, w) = p \}$$

Property 2.44. *For a state p that is not useless, we have $\overleftarrow{\mathcal{L}}(p) \neq \emptyset$ and $\overrightarrow{\mathcal{L}}(p) \neq \emptyset$.*

Example 2.45. *Referring to the illustrative DFA example depicted in Notation 2.29, $\overrightarrow{\mathcal{L}}(q_0) = \overleftarrow{\mathcal{L}}(q_3) = \{ab, ba\}$.*

Property 2.46 (Recursive definition of $\overrightarrow{\mathcal{L}}$)**.** *The recursive definition of δ^* can be used to give a recursive definition for $\overrightarrow{\mathcal{L}}$ as follows:*

$$\overrightarrow{\mathcal{L}}(q) = \left(\bigcup_{a \in \Sigma_q} \{a\} \overrightarrow{\mathcal{L}}(\delta(q,a)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

Phrased differently, a string v is in $\overrightarrow{\mathcal{L}}(q)$ if and only if

- v is of the form aw where $a \in \Sigma$ is a label of an out-transition from q to $\delta(q,a)$ (i.e. $a \in \Sigma_q$) and w is in the right language of $\delta(q,a)$, or
- $v = \varepsilon$ and q is a final state.

A recursive definition of $\overleftarrow{\mathcal{L}}$ is not required in this thesis.

Definition 2.47 (Language of a DFA)**.** *The language accepted by DFA M, denoted $\mathcal{L}(M)$, is defined by*

$$\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(s)$$

Note that one could alternatively define $\mathcal{L}(M)$ using left languages as

$$\mathcal{L}(M) = \langle \cup f : f \in F : \overleftarrow{\mathcal{L}}(f) \rangle$$

Definition 2.48 (Path through a *DFA*)**.** *For state p and $w \in \Sigma^*$,*

$$[p \overset{w}{\rightsquigarrow}]$$

is the sequence of states $\delta^*(p,\varepsilon), \ldots, \delta^*(p,v)$ where v is the longest prefix of w such that $\delta^*(p,v) \neq \perp$. Refer to $[p \overset{w}{\rightsquigarrow}]$ as the single "w-path from state p."

The standard parentheses notation is used to denote state sequences that are open at the beginning or end—for example, $(p \overset{w}{\rightsquigarrow}]$ does not include p but *does* include the rest of $[p \overset{w}{\rightsquigarrow}]$. One may pass a path $[p \overset{w}{\rightsquigarrow}]$ as a parameter to a predicate or function that expects a *set*, thereby implicitly treating the path as a set of states.

Property 2.49. *Define $[p \overset{w}{\rightsquigarrow}]$ recursively as:*

$$[p \overset{\varepsilon}{\rightsquigarrow}] = p$$

and, for all $a \in \Sigma, w \in \Sigma^*$ (where $\cdot$ is sequence concatenation and ε is the empty sequence which some authors write as $[\,]$)

$$[p \overset{aw}{\rightsquigarrow}] = p \cdot \begin{cases} [\delta(p,a) \overset{w}{\rightsquigarrow}] & \text{if } a \in \Sigma_p \\ \varepsilon & \text{otherwise} \end{cases}$$

Notation 2.50. *Write $Succ(p)$ for the set of states that are direct successors of state $p \in Q$.*

$$Succ(p) = \{\, \delta(p,a) \mid a \in \Sigma_p \,\}$$

The same notation may be used for the successors of a set of states $P \subseteq Q$,

$$Succ(P) = \langle \cup\, p : p \in P : Succ(p) \rangle$$

Definition 2.51 (Longest right word length function)**.** *For an* acyclic *DFA only, function $\overrightarrow{\mathcal{L}}_{|max|} : Q \longrightarrow \mathbb{N}$ is defined as*

This restriction is placed because a cyclic *DFA* may have infinitely long paths from a state to a final state.

$$\overrightarrow{\mathcal{L}}_{|max|}(p) = \langle \boldsymbol{MAX}\ w : w \in \overrightarrow{\mathcal{L}}(p) : |w| \rangle$$

$\overrightarrow{\mathcal{L}}_{|max|}(p)$ is the length of the longest path from p to a final state in an acyclic DFA. Revuz (1992) calls $\overrightarrow{\mathcal{L}}_{|max|}(p)$ the 'height' of p.

Definition 2.52 (Height levels)**.** *In an acyclic DFA, for $k \in \mathbb{N}$, define a set of states*

$$HL_k = \{\, p \mid p \in Q \wedge \overrightarrow{\mathcal{L}}_{|max|}(p) = k \,\}$$

to be the set of states at height level k.

Property 2.53. *$\{HL_0, HL_1, \ldots, HL_n\}$ is a* partition *of Q, where $n \in \mathbb{N}$ is the largest value for which $HL_n \neq \emptyset$.*

Example 2.54. *In the trie of Example 2.38,*

$$\begin{aligned} HL_0 &= \{3,5,8,10,12,13\} \\ HL_1 &= \{4,9,11\} \\ HL_2 &= \{2,7\} \\ HL_3 &= \{6\} \\ HL_4 &= \{1\} \\ HL_5 &= \{0\} \end{aligned}$$

Definition 2.55 (State depth function)**.** *Function $\overleftarrow{\mathcal{L}}_{|min|} : Q \longrightarrow \mathbb{N}$ is defined as*

$$\overleftarrow{\mathcal{L}}_{|min|}(p) = \langle \boldsymbol{MIN}\ w : w \in \overleftarrow{\mathcal{L}}(p) : |w| \rangle$$

$\overleftarrow{\mathcal{L}}_{|min|}(p)$ is the length of the shortest path from s to p. In the literature, $\overleftarrow{\mathcal{L}}_{|min|}(p)$ is also known as the 'depth' of p.

Definition 2.56 (Depth levels)**.** *In an acyclic DFA, for each $k \in \mathbb{N}$, define a set of states*

$$DL_k = \{\, p \mid p \in Q \wedge \overleftarrow{\mathcal{L}}_{|min|}(p) = k \,\}$$

to be the set of states at depth level k.

Property 2.57. *$\{DL_0, DL_1, \ldots, DL_n\}$ is a partition of Q, where $n \in \mathbb{N}$ is the largest value for which $DL_n \neq \emptyset$.*

Example 2.58 (Depth levels)**.** *In the trie of Example 2.38,*

$$\begin{aligned} DL_0 &= \{0\} \\ DL_1 &= \{1\} \\ DL_2 &= \{2, 6\} \\ DL_3 &= \{3, 4, 7, 11\} \\ DL_4 &= \{5, 8, 9, 12, 13\} \\ DL_5 &= \{10\} \end{aligned}$$

Definition 2.59 (Minimality of a DFA)**.** *DFA M is minimal – written $Min(M)$ – if, and only if, it is the smallest DFA accepting $\mathcal{L}(M)$. Any other such DFA M' may not have fewer states.*

Smallest with respect to the number of states in M as given in Definition 2.31

$$Min(M) \equiv \langle \forall\ M' : M' \in DFA \wedge \mathcal{L}(M) = \mathcal{L}(M') : |M| \leq |M'| \rangle$$

Property 2.60. *A minimal DFA is unique up to isomorphism—see Hopcroft and Ullman (1979, §3.4).*

Definition 2.61 (State equivalence)**.** *Define E as an equivalence relation on states where*

$$E(p, q) \equiv (\vec{\mathcal{L}}(p) = \vec{\mathcal{L}}(q))$$

If two states p, q are equivalent under E, states p and q can be merged. For example, to merge p into q, remove all inbound transitions into p, replace them by inbound transitions into q and remove p.

Property 2.62. *Assuming no useless states, start state s is unique—that is, it is not equivalent to any other state.*

Property 2.63 (Recursive definition of E)**.** *The recursive definition of $\vec{\mathcal{L}}$ (Property 2.46) gives rise to a recursive definition of E as follows (from B. W. Watson 2010, Property 2.71)*

$$\begin{aligned} & E(p, q) \\ \equiv & \quad \text{“ definition of } E \text{ ”} \\ & \vec{\mathcal{L}}(p) = \vec{\mathcal{L}}(q) \\ \equiv & \quad \text{“ definition of language equality ”} \\ & \langle \forall\ v : v \in \Sigma^* : v \in \vec{\mathcal{L}}(p) \equiv v \in \vec{\mathcal{L}}(q) \rangle \\ \equiv & \quad \text{“ split domain } \Sigma^* \text{ into } \{\varepsilon\} \cup \Sigma^+ \text{ ”} \\ & \langle \forall\ v : v \in \{\varepsilon\} \cup \Sigma^+ : v \in \vec{\mathcal{L}}(p) \equiv v \in \vec{\mathcal{L}}(q) \rangle \end{aligned}$$

$$
\begin{aligned}
\equiv \quad & \text{“ split quantification ”} \\
& \langle \forall\, v : v \in \{\varepsilon\} : v \in \vec{\mathcal{L}}(p) \equiv v \in \vec{\mathcal{L}}(q) \rangle \wedge \\
& \langle \forall\, v : v \in \Sigma^+ : v \in \vec{\mathcal{L}}(p) \equiv v \in \vec{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{“ one-point rule on the first universal quantification ”} \\
& (\varepsilon \in \vec{\mathcal{L}}(p) \equiv \varepsilon \in \vec{\mathcal{L}}(q)) \wedge \\
& \langle \forall\, v : v \in \Sigma^+ : v \in \vec{\mathcal{L}}(p) \equiv v \in \vec{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{“ introduce dummies } a \in \Sigma, w \in \Sigma^* \text{ such that } v = aw \text{ in second quantification ”} \\
& (\varepsilon \in \vec{\mathcal{L}}(p) \equiv \varepsilon \in \vec{\mathcal{L}}(q)) \wedge \\
& \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : aw \in \vec{\mathcal{L}}(p) \equiv aw \in \vec{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{“ } \varepsilon \in \vec{\mathcal{L}}(r) \equiv r \in F \text{ ”} \\
& (p \in F \equiv q \in F) \wedge \\
& \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : aw \in \vec{\mathcal{L}}(p) \equiv aw \in \vec{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{“ in context, } aw \in \vec{\mathcal{L}}(p) \equiv (a \in \Sigma_p \textbf{ cand } w \in \vec{\mathcal{L}}(\delta(p,a))) \text{ ”} \\
& (p \in F \equiv q \in F) \wedge \\
& \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : (a \in \Sigma_p \textbf{ cand } w \in \vec{\mathcal{L}}(\delta(p,a))) \equiv (a \in \Sigma_q \textbf{ cand } w \in \vec{\mathcal{L}}(\delta(q,a))) \rangle \\
\equiv \quad & \text{“ split universal quantifier; } \textbf{cand} \text{ no longer needed with } a \in \Sigma_p \cap \Sigma_q \text{ in quantifier range ”} \\
& (p \in F \equiv q \in F) \wedge \langle \forall\, a : a \in \Sigma : a \in \Sigma_p \equiv a \in \Sigma_q \rangle \wedge \\
& \langle \forall\, a, w : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \vec{\mathcal{L}}(\delta(p,a)) \equiv w \in \vec{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{“ definition of alphabet equality ”} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \\
& \langle \forall\, a, w : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \vec{\mathcal{L}}(\delta(p,a)) \equiv w \in \vec{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{“ definition of language equality ”} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \\
& \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : \vec{\mathcal{L}}(\delta(p,a)) = \vec{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{“ definition of } E \text{ ”} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : E(\delta(p,a), \delta(q,a)) \rangle
\end{aligned}
$$

Definition 2.64 (Pairwise inequivalent states)**.** *Define predicate*

$$Inequiv(X) \equiv \langle \forall\, p, q : p \neq q \wedge p, q \in X : \neg E(p,q) \rangle$$

Property 2.65 (Minimality of a DFA)**.** *Min is equivalent to:*

- *all states in* $Q \setminus \{s\}$ *are useful, and*
- $Inequiv(Q)$.

This is shown by Hopcroft and Ullman (1979, §3.4).

2.4 Guarded Command Language

Abstract algorithms are presented in this thesis in Dijkstra's GCL (Dijkstra 1975, 1976; Gries 1980; Kourie and B. W. Watson 2012). In this section, the aim is to provide sufficient detail to enable someone familiar with programming to comprehend the notation. It is therefore not considered necessary to provide a formal specification of the semantics of the various language constructs. The syntax is described and some semantic matters are addressed informally. Formal semantics—based one pre- and post-conditions—may be found in the references above.

GCL relies on the following basic constructs. The empty command (**skip**), assignment (:=), composition (;), selection (**if**), and repetition (**do**). A discussion of each of these commands now follow.

Although program constructs are commonly called *statements*, Dijkstra preferred the term *command*.

Empty command The **skip** command is simply a command that does nothing. As it will be seen below, it is sometimes mandatory to use **skip** as part of a selection command.

Assignment GCL allows for both single and multiple assignment. In both cases assignment is denoted by the := symbol. Multiple assignment is not a necessary construct, but is does allow for shorter and more elegant code specification. The example of swapping two variables $x, y := y, x$ is commonly used to illustrate this.

Composition The composition of two code segments $S1$ and $S2$ is denoted $S1; S2$. Composition assists in breaking up a task into a sequence of smaller, more manageable, tasks.

Selection The syntax of the selection command appears similar to the `switch`-statement of other languages and is as follows.

if $G_1 \rightarrow S_1$
[] $G_2 \rightarrow S_2$
...
[] $G_n \rightarrow S_n$
fi

The G_i are predicates, called guards, and the S_i are GCL commands that could possibly be the composition of multiple commands. Each $G_i \rightarrow S_i$ constitutes a so-called guarded command—hence the name *guarded command* language. The semantics of the GCL selection command differs from that of typical `switch`-statements. The first step in executing the command it to evaluate all the guards. From the set of guarded commands that evaluate to *true*, one is non-deterministically selected and executed. If no guard evaluates to *true*,

then the selection command may exhibit any behaviour. The intention behind this decision is that it should be made explicit what happens in every circumstance—there is no default behaviour. For this reason the union of the guards should cover the entire universe of possibilities.

When a selection command has the form

if $G \to S$
$[]$ $\neg G \to$ **skip**
fi

some authors abbreviate this to

as $G \to S$ **sa**.

This abbreviation will not be used in this thesis.

Repetition The general syntax of the repetition command is:

do $G_1 \to S_1$
$[]$ $G_2 \to S_2$
$\cdots$
$[]$ $G_n \to S_n$
od

As in the selection command above, the G_i are predicates and the S_i are GCL commands. The repetition command may execute zero or more times. At the start of each iteration, all the guards are evaluated and one of the S_i for which a guard evaluated to *true* is nondeterministically selected for execution. The command iterates until all the guards evaluate to *false*. In that case the command terminates successfully and control is passed to the next command in the sequence.

The above form may appear strange at first glance and may be changed into a repetition with a single guard:

do $G \to S$ **od**

where $G = G_1 \vee G_2 \vee \cdots \vee G_n$ and S is the select command:

if $G_1 \to S_1$
$[]$ $G_2 \to S_2$
$\cdots$
$[]$ $G_n \to S_n$
fi

This latter form is typically used in the algorithms presented later in the text. Another form of repetition that is used frequently is as follows:

for $i : I \to S$ **rof**

This describes a repetition in which i is a loop variable that assumes, in each iteration, a value from the set I. If I is an ordered collection, the elements are selected in order. If, on the other hand, the collection is unordered, the elements are selected in an arbitrary order. If this command is used, the contents of the set I should not be changed by S.

Procedures and functions Sequences of commands may be encapsulated into a procedure or a function with a given name. The following defines a procedure P that accepts two parameters.

proc $P(a, b) \to$
    ...
corp

The next definition is a function that returns a tuple $\langle x, y \rangle$.

func $F(a, b) \to$
    ...
    $x, y := ...$
    ...
    return $\langle x, y \rangle$
cnuf

Comments Comments may be added to GCL code by enclosing them in a pair of braces. For example, { this is a comment } is a comment.

2.5 Communicating Sequential Processes

Baeten, Basten and Reniers (2010) describes the *field of process algebra* as the field that studies parallel or distributed systems by *algebraic* means. It allows one to describe or *specify* the behaviour of such systems and thus has a means to refer to parallel composition. It also makes it possible to reason about these systems using algebra which then allows for *verification* of the systems.

According to Baeten (2005), the term *process algebra* was first defined by Bergstra and Klop (1982). Over the years a number of process algebras have been developed. Of these CCS (Milner 1989), ACP (Bergstra and Klop 1984), and CSP (Hoare 1985) are probably the best known.

CSP is used to describe processes in subsequent chapters. The original language called CSP was created by Hoare (1978). This language was not a process algebra, but rather an imperative programming language very

similar to the GCL, but with the addition of point-to-point communication. Theoretical work by Brooks, Hoare, and Roscoe (Brookes 1983; Roscoe 1982; Roscoe and Barrett 1989; Roscoe and Brookes 1985; Roscoe, Brookes and Hoare 1984) led to the abstraction and generalisation of the language. This new language – now a process algebra and also called CSP – is described in Hoare (1985, 2004). Development of CSP continued and the version used here correponds to the version described by Roscoe (2010) and Roscoe, Hoare and Bird (1997).

2.5.1 Overview

CSP is concerned with the studying of processes that interact with one another and their environment by means of communication. Hence, the most fundamental object in CSP is the communications event. These events are assumed to be drawn from a set Σ that contains all possible communications for processes in the universe under consideration. CSP abstracts away from time in the sense that events are deemed to be atomic. An event therefore occurs either before or after some other event. One way of describing a process is to specify all possible event sequences in which the process may engage, that is, to specify the process's set of traces. Various operators are available to describe the sequence in which events may occur, as well as to connect processes. Table 2.1 briefly outlines the main operators used in this thesis.

Note that Σ is also used for a string alphabet. The context should make clear which set is meant.

Primitive processes The simplest process of all is *STOP*. It is the process that never engages in any events. It is essentially a deadlocked state. This is different from the notion of a process that reaches a state where its execution is completed. This is modelled by process *SKIP* which does nothing apart from indicating that it has terminated successfully.

Prefixing Let $a \in \Sigma$ and P be a process. Then $a \rightarrow P$ is the process that is initially willing to communicate event a to a process in its environment and then behaves as described by P. This operation is known as *prefixing* since it changes P into $a \rightarrow P$.

Recursion Recursion is used to specify repeating processes. Using a recursively defined process's name on the right-hand side of the equation means the same as the whole. For example, $P = a \rightarrow P$ describes the process that can indefinitely engage in event a. Instead of using a single equation to describe a process, it is also possible to use mutual recursion. For example, if $P_1 = a \rightarrow P_2$ and $P_2 = b \rightarrow P_1$, then P_1 behaves the same as the single recursive process $P_3 = a \rightarrow b \rightarrow P_3$.

Guarded alternative With prefixing and recursion one may describe processes with a single thread of execution. It is, however, possible in

Σ	The set of all possible communication events.
$SKIP$	Successful termination.
$STOP$	Deadlock.
$a \rightarrow P$	Prefixing. Event a then process P.
$a \rightarrow P \mid b \rightarrow Q$	Guarded alternative. a then P choice b then Q.
$?x : A \rightarrow P(x)$	Prefix choice. Choice of x from set A then $P(x)$.
$a.b.c$	Compound or multipart event.
$c!e$	On channel c output event e.
$c?x$	From channel c input to variable x.
$\{\!\|a, b\|\!\}$	The set of events associated with channels a and b.
if b then P else Q	Conditional choice. If b then process P else process Q.
$P \sqcap Q$	Non-deterministic choice between process P and process Q.
$b \& P$	Conditional guard. If b then P else $STOP$.
$P \parallel Q$	Synchronous parallel. Synchronise on all events.
$P \underset{X}{\parallel} Q$	Generalised parallel. Synchronise on events in set X.
$P \mid\mid\mid Q$	Interleaving parallel. Do not synchronise.
$P \setminus X$	Hiding. The events in X are not observable.
$P[\![a/b]\!]$	Renaming. All events b is changed to a.
$l.P$	Process naming. Communicates $l.x$ whenever P communicates x.
$P; Q$	Sequential composition. Process P followed by process Q.
$P \Box Q$	External choice. Process P choice process Q.
$P \, [a \leftrightarrow b] \, Q$	Link parallel. Synchronise on events over channels a and b.

Table 2.1: Selected CSP notation used in the thesis.

CSP to describe processes that offer a choice of events to their environments. The simplest way in which to do this is to make use of the guarded alternative notation. Process $(a_1 \rightarrow P_1 \mid \cdots \mid a_n \rightarrow P_n)$ offers $a_1, \ldots, a_n$ to its environment and then behaves as P_i if a_i was the first event interaction that occurred with the environment.

Prefix choice Prefix-choice generalises the guarded alternative. If $A \subseteq \Sigma$ then $?x : A \rightarrow P(x)$ describes the process that offers initially any $x \in A$ and then behaves as $P(x)$.

External choice The external choice operator is similar to the guarded alternative operator. However, here the operands are processes and not events. The process $P \,\Box\, Q$ offers the environment the initial events of P and of Q and then behaves according to the choice. If the first event chosen is from P only, then $P \,\Box\, Q$ behaves as P, and if the first event is from Q only, $P \,\Box$ behaves as Q. If P and Q have first events in common, and one of those common events is selected for interaction by the environment, then the process behaves *nondeterministically*—i.e its subsequent behaviour is non-deterministically described by either P or Q. An important property of external choice is that $P \,\Box\, STOP = P$.

If $S = \{1, 2, \ldots, n\}$ is a finite set that indexes a finite collection of processes, then $\Box_{i \in S} P_i$ is short-hand for $P_1 \,\Box\, P_2 \,\Box \cdots \Box\, P_n$. If S is empty, then $\Box_{i \in S} P_i$ behaves as $STOP$.

Nondeterministic choice CSP provides two ways of representing explicitly the nondeterministic choice of processes: $P \sqcap Q$ or $\bigsqcap S$. Here P and Q are processes and S is a *non-empty* set of processes. The process $P \sqcap Q$ can behave like either process P or process Q. If S is a set of processes then $\bigsqcap S$ can behave like any member of S. The choice of behaviour is determined by the processes itself, without influence from its environment. Thus, $P \sqcap Q$ may at one time only permit engagement with the environment via one of the first events of P, ignoring any environmental offering to engage in one of the first events of Q. At another time it may do the opposite—nondeterministically only engage via one the first events of Q and ignore environmental offerings of one of the first events in P.

Similarly to external choice, an indexed version of the internal choice operator will also be used. Thus, if I is a finite set that indexes a finite collection of processes, then $\bigsqcap_{i \in I} P_i$ means the same as $\bigsqcap \{ P_i \mid i \in I \}$.

Conditional choice Another form of choice is based on a Boolean expression and is also required when describing processes. CSP provides an if ... then ... else ... construct. Hence, when b holds in (if b then P else Q), then the process behaves as described by P, but like Q when b does not hold. As an aside, note that CSP also provides a binary

operator to express conditional choice: $P \triangleleft b \triangleright Q$ means the same as if b then P else Q. This notation is not employed in this thesis.

Conditional guard When there is a conditional choice with *STOP*, one may use a notational short-hand and write $(b \,\&\, P)$ for (if b then P else *STOP*).

Multi-part events It is often useful to specify an event as a compound object using an infix dot to combine a qualifier and an element of an alphabet Σ. If c is the name of a channel then $c.x$ indicates that x is to be communicated over channel c, and if T is the type of the event communicated over the channel c, then $c.T = \{c.x \mid x \in T\} \subseteq \Sigma$ denotes the set of all such events. Furthermore, one may specify a process receives input and/or sends output of type T over a channel c. In this case, input of type T over channel c is described using prefix choice like this: $?y{:}c.T \rightarrow P(y)$. Another form that allows one to state explicitly the element x from T, is as follows. $c?x{:}T \rightarrow P'(x)$. This latter form makes it clear that some element of T is communicated over channel c. When the type of communication allowed is clear from the context, one may omit T and simply write $c?x \rightarrow P'(x)$. In this case an exclamation mark (!) is used to designate output. For example, a process that inputs some value on channel *in* and outputs the same value on channel *out* may be described as $P = in?x \rightarrow out!x \rightarrow P$.

In general it is permissible to have events with any finite number of parts separated by infix dots. A channel is characterised by its name and a finite sequence of data types. Σ then contains events of the form $c.x_1.x_2.\cdots.x_n$ with c the channel name, $T_1, \ldots, T_n$ the data types, and $x_i \in T_i$.

The most common case for using such multipart channel types is when an array of channels is used to communicate between similarly indexed processes. Let P_i and Q_i with $i \in I \subset \mathbb{N}$ be such indexed processes. They need to communicate objects of type T. Then c may be a channel of type $\mathbb{N}.T$ and the event $c.i.x$ might represent the communication of value x from P_i to Q_i.

Another useful notation that is used often in the thesis is the $\{\!|\ldots|\!\}$ notation. It allows one to specify the set of events on a channel by merely mentioning the channel name. Thus $\{\!|c_1, c_2|\!\}$ is the set of events of channels c_1 and c_2.

Synchronous parallel Processes in CSP interact by agreeing – or *synchronising* – on events. Synchronization is symmetric and instantaneous, and occurs only when both participants engage in it simultaneously. Like most process algebras, CSP has a parallel composition operator that specifies how processes interact. In fact, CSP has a number of forms of parallel composition. The simplest form requires

that processes agree on *all* events that occur and is written $P \parallel Q$. Hence, if $P = ?x{:}A \to P'$ and $Q = ?x{:}B \to Q'$ then

$$P \parallel Q = (?x{:}A \to P') \parallel (?x{:}B \to Q') = ?x{:}A \cap B \to (P' \parallel Q').$$

The $\parallel$ operator is symmetric, associative and distributive.

Interleaving Parallel composition by interleaving ($|||$) is another form of parallel composition. In this case the processes need not agree on *any* of their events—they execute independently. Any event communicated by $P ||| Q$ arose in exactly one of P and Q. If both could have communicated an event, the ambiguity is resolved by selecting one nondeterministically as shown in its step law: Let $P = ?x{:}A \to P'$ and $Q = ?x{:}B \to Q'$, then

They do, however, synchronise on termination. $P ||| Q$ terminates only when both P and Q terminate.

$$\begin{aligned} P ||| Q = ?x{:}A \cup B \to \quad & \text{if } x \in A \cap B \text{ then} \\ & \quad P' ||| Q \sqcap P ||| Q' \\ & \text{else if } x \in A \text{ then} \\ & \quad P' ||| Q \\ & \text{else} \\ & \quad P ||| Q' \end{aligned}$$

The operator is also symmetric, associative, and distributive.

Generalised parallel The behaviour of the previous parallel composition operators may be achieved by a single operator in which the *interface* of interaction is specified. Process $P \underset{X}{\parallel} Q$ is the process in which all the events in X need to be synchronised and the events not in X may proceed independently. Hence it may be seen as a combination of synchronous parallel and interleaving parallel. So $P ||| Q = P \underset{\emptyset}{\parallel} Q$ and $P \parallel Q = P \underset{\Sigma}{\parallel} Q$. Let $P = ?x{:}A \to P'$ and $Q = ?x{:}B \to Q'$. Then the set of initial events of $P \underset{X}{\parallel} Q$ is $C = (X \cap A \cap B) \cup (A \setminus X) \cup (B \setminus X)$. The step law is quite complex since events may be synchronised; independent, but ambiguous; or from one process only.

$$\begin{aligned} P \underset{X}{\parallel} Q = ?x{:}C \to \quad & \text{if } x \in X \text{ then} \\ & \quad P' \parallel Q' \\ & \text{else if } x \in A \cap B \text{ then} \\ & \quad P' \underset{X}{\parallel} Q \sqcap P \underset{X}{\parallel} Q' \\ & \text{else if } x \in A \text{ then} \\ & \quad P' \underset{X}{\parallel} Q \\ & \text{else} \\ & \quad P \underset{X}{\parallel} Q' \end{aligned}$$

The operator is symmetric and distributive. It is associative when the interface is the same for all instances of the operator.

Hiding Sometimes it is better to hide internal events so that they become invisible and uncontrollable by the environment. Given a process P and a set of events X, the process $P \setminus X$ behaves like P except that the events in X have been hidden. That means that the traces of $P \setminus X$ are the same as the traces of P, except that all events in X are removed from the traces of $P \setminus X$.

Renaming and alphabet transformation It is possible to rename the events in which a process engages. This is also called an alphabet transformation. There are various ways in which this may be accomplished in CSP, but the two forms used in this text are shown here. A common form is called *process naming*. A process P may be labelled – or named – $l.P$. It then communicates $l.x$ whenever P communicates x. Some authors write $l{:}P$ instead of $l.P$. Process labelling is typically used where one requires multiple copies of a process in a system but with disjoint alphabets.

In the second form of renaming, one may write $P[\![a/b]\!]$ to mean that the event or channel b in P is replaced by a. This is often used to enable synchronisation with another process.

Linking parallel operator It is often desirable to hide the synchronising action of the parallel composition of processes. A combination of hiding and renaming with parallel composition may be used to achieve this. Since this is used often, a succinct way of writing it down exists. $P\,[a \leftrightarrow b]\,Q$ is the process in which P's communications on channel a are synchronised with Q's communications on channel b, and then hidden. One may think of this as linking the two processes with a wire. If c is an unused channel name of the same type as a and b, then

$$P\,[a \leftrightarrow b]\,Q = (P[\![c/a]\!] \underset{\{\!|c|\!\}}{\|} Q[\![c/b]\!]) \setminus \{\!|c|\!\}.$$

This can be extended to any number of pairs of channels (a_i, b_i) as in $P\,[a_1 \leftrightarrow b_1, \ldots, a_n \leftrightarrow b_n]\,Q$. Each pair must be of the same type, but different pairs may be of different types.

If the channel pairs are distinct and the visible alphabets of the processes are disjoint, the operator is associative:

$$(P\,[a \leftrightarrow b]\,Q)\,[c \leftrightarrow d]\,R = P\,[a \leftrightarrow b](Q\,[c \leftrightarrow d]\,R)$$

To help reduce typesetting and aid the reader, the following notational style is used. Instead of writing $(P\,[a \leftrightarrow b]\,Q)\,[c \leftrightarrow d]\,R$, a vertical style is used:

$$
\begin{array}{l}
P \\
\qquad [a \leftrightarrow b] \\
Q \\
\qquad [c \leftrightarrow d] \\
R
\end{array}
$$

Note that the parentheses are omitted in the vertical style—the convention is that the association is from the top. When a different grouping is required, parentheses will be used to make the grouping explicit.

Sequential composition $P; Q$ is the process that behaves as P until it terminates successfully and then behaves as Q.

Sequences The empty sequence is represented as $\langle\rangle$ and $\langle a_1, a_2, \ldots, a_n\rangle$ is the sequence containing $a_1, a_2, \ldots, a_n$, in that order. If s and t are finite sequences then $s^\frown t$ is the concatenation of s and t. The length of a sequence s is usually written as $\#s$. In this thesis, a functional form $len(s)$ will be used instead. Functions *head* and *tail* from Definition 2.16 may also be applied to CSP sequences. $head(\langle a\rangle^\frown s) = a$ and $tail(\langle a\rangle^\frown s) = s$.

CSP traces are not often used explicitly in this thesis. The notation for tuples also use the same brackets. It should be clear from the context which interpretation is intended.

In deploying CSP, the following assumptions relating to atomic execution were made. First, if an event maps to a function call, then that function is assumed to be a sequence of code in the original sequential algorithm that runs uninterruptedly to completion on some processor. Furthermore, in the interest of conciseness and without loss of generality, it will sometimes be convenient to subsume certain assignment operations of the sequential program into the actual parameter list of a process invocation. For example, instead of specifying some recursive parameterised process $P(D)$ as $P(D) = \ldots \rightarrow (D := D \cup \{q\}); P(D)$, regard the specification $P(D) = \ldots \rightarrow P(D \cup \{q\})$ as equivalent. This means that operations that are needed to compute the actual parameters for a process invocation are regarded as taking place atomically, that is they cannot be interrupted by any other process's activity.

Similarly, where the CSP syntax for a conditional is used, as in if b then P else Q it will be assumed that the computation of the condition, b, takes place atomically and prior to the activation of any first event possible in the constituent processes, P and Q.

These instances of atomic activity are highlighted, not because they deviate from CSP syntax, but because they represent potential opportunities for more fine-grained specifications of the algorithms than is given in the

various specifications to follow. These fine-grained specifications have not been provided because they were deemed unnecessary in the context of the current work.

2.5.2 Buffer example

The following example serves two purposes. The first is to illustrate how CSP processes are presented in the thesis, and the second is to define two buffer processes that will be used in later chapters.

Define n processes as follows. $P_i = in.i?x \rightarrow out.i!f(x) \rightarrow P_i$. Each P_i process accepts input on an input channel $in.i$ and emits on the output channel $out.i$, the value of function f applied to the input. These n processes may run independently and concurrently as $|||_{i=1..n} P_i$. Say one wants to provide an interface to these processes via a single channel, such that a process may send data to any one of the P_i processes via this channel. To achieve this – and to also add buffering capabilities – define a process *DBUFF* as follows. Its CSP definition is adapted from buffer definitions given in classical texts such as Roscoe (2010).

The name *DBUFF* stands for 'distributing buffer'.

DBUFF is parameterised by its size, the number of output channels, and a sequence representing its current contents. It receives elements from a channel *left*, provided that it is not full. If not empty, it may emit the least recently stored element on any available one of an array of *right.i* channels with . If more than one such channel is available, the choice of channel may be made arbitrarily.

$$
\begin{array}{l}
DBUFF(s, N, n) = \\
\quad \text{if } s = \langle\,\rangle \text{ then} \\
\qquad left?x \rightarrow DBUFF(\langle x \rangle, N, n) \\
\quad \text{else if } len(s) < N \text{ then} \\
\qquad left?x \rightarrow DBUFF(s \hat{} \langle x \rangle, N, n) \\
\qquad \Box\ right?i{:}\{1, 2, \ldots, n\}!head(s) \rightarrow DBUFF(tail(s), N, n) \\
\quad \text{else} \\
\qquad right?i{:}\{1, 2, \ldots, n\}!head(s) \rightarrow DBUFF(tail(s), N, n)
\end{array}
$$

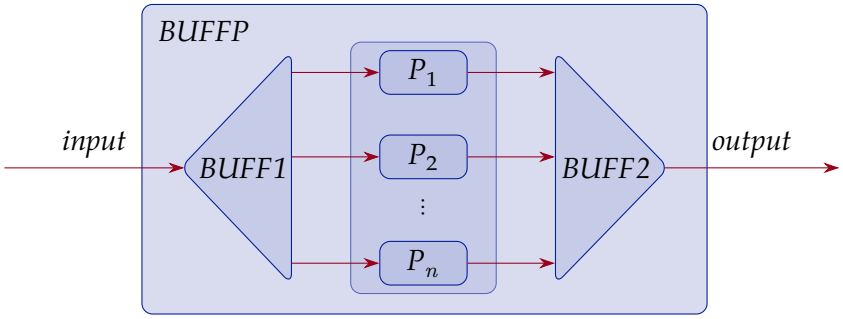
One may define a similar buffer process *MBUFF* that receives elements from an array of *left.i* channels. These elements are then emitted in FIFO order on a single *right* channel.

The name *MBUFF* stands for 'multiplexing buffer'.

$$
\begin{array}{l}
MBUFF(s, N, n) = \\
\quad \text{if } \ s = \langle\,\rangle \text{ then} \\
\qquad left?i{:}\{1, 2, \ldots, n\}?x \rightarrow MBUFF(\langle x \rangle, N_1) \\
\quad \text{else if } len(s) < N \\
\qquad left?i{:}\{1, 2, \ldots, n\}?x \rightarrow MBUFF(s \hat{} \langle x \rangle) \\
\qquad \Box\ right!head(s) \rightarrow MBUFF(tail(s), N_1)
\end{array}
$$

else
$right!head(s) \rightarrow MBUFF(tail(s), N_1)$

Let $PROC = |||_{i=1..n} P_i$. One may compose $PROC$ with instances of $DBUFF$ and $MBUFF$ to form a process $BUFFP$ that receives input on a single channel and emits results on a single channel. This idea is depicted in the following diagram.



The buffer, $BUFF1$ in the diagram, may be described by an instance of $DBUFF$. Let

$$BUFF1 = a.DBUFF(\langle\,\rangle, N_a, n)[\![^{input}/_{a.left}]\!].$$

The buffer is initially empty $\langle\,\rangle$, has capacity N_a, and may synchronise on n output channels. The labelling by a causes every event in $DBUFF$ to be prefixed with a and the renaming $[\![^{input}/_{a.left}]\!]$, causes the input channel $a.left$ to become $input$. One may then connect the output channels of $BUFF1$ and the input channel of $PROC$ by using the link parallel operator:

$$T = BUFF1\,[a.right \leftrightarrow in]\,PROC.$$

In a similar fashion,

$$BUFF2 = b.MBUFF(\langle\,\rangle, N_b, n)[\![^{output}/_{b.right}]\!].$$

This buffer may then be linked to process T to form

$$BUFFP = T\,[out \leftrightarrow b.left]\,BUFF2.$$

In the 'vertical style', the same may be expressed in one process definition as:

$BUFFP =$
$\quad a.DBUFF(\langle\,\rangle, N_a, n)[\![^{input}/_{a.left}]\!]$
$\quad\quad [a.right \leftrightarrow in]$
$\quad PROC$
$\quad\quad [out \leftrightarrow b.left]$
$\quad b.MBUFF(\langle\,\rangle, N_b, n)[\![^{output}/_{b.right}]\!]$

This style is employed in the later chapters.

Note, therefore, that the overall effect of *BUFFP* is to accept data on channel *input*, to store it in a buffer whose state is reflected in the first parameter of the *DBUFF* process, to have that data passed on via P_i processes to the buffer whose state is reflected in the first parameter of the *MBUFF* process, and then to subsequently have that data removed by some external process interacting through the *output* channel.

2.6 Go overview

Go is a compiled, statically typed language providing concurrency features and garbage collection. According to the Go FAQ (2016), Go was conceived at Google by Robert Griesemer, Rob Pike, and Ken Thompson in September 2007. Russ Cox joined in 2008 and Go was publicly released as an open-source project in November 2010.

Pike (2012b) states that the main design goal of the new language was to eliminate cumbersomeness in systems programming. He mentions, among other things, the following issues: long compilation times, uncontrolled dependencies, poor source code documentation, and cost of updates. Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. They wanted a language that works at scale, is familiar, and modern—meaning that is should support networked computing and concurrency.

To keep the language familiar, Go's syntax is C-like. Declarations are, however, in the style of Pascal in that the name goes before the type and there is a keyword **`var`**. It also contains ideas from languages that were inspired by CSP, namely the languages Limbo (Kernighan 2005; Ritchie 2005) and Newsqueak (Pike 1990).

From the Go Language Specification (2016) it can be seen that the language contains only 25 keywords. The grammar is also fairly small, allowing for a simple parser.

Go's approach to visibility of identifiers is a bit different from current conventions. Instead of keywords like `public` and `private`, the visibility information is contained in the name of the identifier. If the first character of the identifier is a capital, then the identifier is exported (public). If not, then the identifier is not exported.

Go semantics is also very C-like. However, many small changes have been made in the service of robustness. These include:

- There is no pointer arithmetic,

- there are no implicit numeric conversions,
- array bounds are always checked,
- there are no type aliases (after `type T int`, `T` and `int` are distinct types and not aliases),
- `++` and `--` are statements not expressions,
- an assignment is not an expression, and
- it is legal to take the address of a stack variable.

Go has no explicit memory-freeing operation: the only way allocated memory returns to the pool is through the garbage collector.

Go takes an unusual approach to object-oriented programming, allowing methods on any type, not just classes, but without any form of type-based inheritance like subclassing. This means there is no type hierarchy. It uses structs rather than classes.

The concurrency features of Go are inspired by CSP. Hence Go includes first class channels that may be synchronous or asynchronous. Processes interact via channels which allows for the concurrent composition of independently executing functions of regular procedural code. Concurrently running functions are called *goroutines*. Goroutines are multiplexed onto threads by the Go runtime.

Unlike pure functional languages, Go does not take a write-once approach to value semantics in the context of concurrent computation.

The remainder of the section provides examples of Go code that illustrates the syntax and semantics of the language. These examples should be sufficient to cover the features that are used by the code fragments in later chapters.

Hello world

The following program prints the string `"Hello world!"` on the console.

```
1 package main
2 import "fmt"
3 // This is a comment.
4 func main() {
5     fmt.Println("Hello world!")
6 }
```

The first line is the package declaration. Every Go file needs such a package declaration. The entry point to a Go program is the function `main` in package `main`. Line 2 imports the package `fmt`. Once imported, one may use any of the package's exported identifiers by prefixing the name of the package to the identifier. For example, calling `Println` on line 5 is prefixed by `fmt`.

Note that `Println` was exported by the `fmt` package since its first letter is in upper case.

Declarations

The following code fragment illustrates a number of declarations. Note how variable declarations follow the Pascal tradition of identifier before type.

```
1  var i int             // Integer
2  var pointer1 *float   // Pointer to a float
3  pointer2 := &i        // Pointer to i
```

Pointers types are indicated using asterisks and `&` is the "address-of" operator. Line 3 shows an example of Go's short-hand for declaring and initialising a variable in one step using the `:=` operator. The type of the variable is determined by the type of the right-hand operand.

The next code fragment illustrates an unusual Go feature—functions may return multiple values. In this case the function returns both an **`int`** and a **`bool`**. The example goes further to also show the use of multiple assignment and to show that Go's **`switch`** statement is more general than C's.

```
1  func f(a, b int) (int, bool) {  // Specify return types.
2      ret, flag := 0, false       // Declare and init variables.
3      switch {
4      case a == b:
5          ret, flag = 0, true     // Use multiple assignment.
6      case a < b:
7          ret, flag = a, false
8      case a > b:
9          ret, flag = b, false
10     }
11     return ret, flag            // Return the values.
12 }
13 min, equal := f(10, 5)          // Call the function.
14 _, equal = f(12,1)              // Ignore one return value.
```

In Go, the expression cases in a **`switch`** statement are evaluated top to bottom and the first one that matches is executed. There is no automatic fall-through.

In the final two lines of the fragment, the function is called. If one is not interested in one of the returned values, one may use the empty identifier _ to discard the value.

Arrays, slices, and maps

In Go, an *array* is a fixed-length numbered sequence of elements of the same type. The length of the array is part of the type. The declaration `var a [5]int` declares an array of 5 integers. The first element is `a[0]` and the last is `a[len(a) - 1]`.

A *slice* grants access to a contiguous segment of an underlying array. The length of a slice may change. To create a slice with a length of 5 integer elements `x := make([]int, 5)`. The underlying array is created and accessed through `x`. It is also possible to create a slice for which the underlying array is larger than the slice by passing a third parameter to `make`. For example, consider `y := make([]int, 5, 10)`. Here, the third parameter is the capacity of the underlying array. Another way to create a slice is by using the slice operator on a slice or an array. For example, `x := a[low:high]` creates a slice from `a[low]` to `a[high - 1]`. It is possible to omit one or both of the variables. `a[:high]` is the same as `a[0:high]` and `a[low:]` is the same as `a[low:len(a)]`.

The built-in function `append` is used to grow a slice by adding new elements.

To iterate through a slice one may use the following construct:

```
for i,e := range x {
    fmt.Println("x[", i, "] = ", e)
}
```

In each iteration, `i` is the index position of element `e`, that is `x[i] = e`.

Another structure that is often used is a *map*, also sometimes called an associative array or a hash table. It is an unordered collection of key-value pairs. The example below declares on line 1 a map from strings to integers.

```
1 x := make(map[string]int)
2 x["key"] = 10
3 fmt.Println(x["key"])
```

Line 2 assigns integer value 10 to the key `"key"`. The built-in function `delete` is used to remove elements from a map. If a key is not found, Go return a zero value for the value type. To distinguish between a zero value and an absent value, Go allows the programmer to test whether a key is present in the map as shown below.

```
if val, ok := x["Missing"]; ok {
    fmt.Println(val)
}
```

The statement between the **if** and the semicolon is executed before the **if**-statement is evaluated. If `x["Missing"]` is not present, the Boolean variable `ok` is **false** and `val` is the zero value. If the key is present, `ok` is **true** and `val` may be used.

Object orientation

Go supports object-oriented programming. One may create objects and call methods on them. However, Go does not provide classes and a class hierarchy. Instead structures are defined and methods may be defined on any type. The following code fragment defines a structure consisting of a slice of integers and an integer.

```
type Structure struct {
    data []int
    i    int
}
```

One may create instances of these structures in a number of ways. One possibility is to create a literal structure: `s := &Structure{make([]int,100),0}`. A more object-oriented approach would be to define a constructor for the new type.

```
func NewStructure(size int) *Structure {
    this := new(Structure)
    this.data = make([]int, size)
    this.i = 0
    return this
}
```

The constructor takes an integer `size` as argument and initialises the fields of the structure. The constructor returns a reference to the newly created object.

One may also define methods for the new type. A method declaration is essentially a function declaration, but, unlike a standard function declaration, a method declaration has a *receiver* that specifies the type to which this method belongs. In the code below, the receiver declaration is `(this *Structure)`.

```
func (this *Structure) Insert(e int) {
    this.data[this.i] = e
    this.i++
}
```

Here an instance of the structure is created and the method is invoked.

```
s := NewStructure(10)
s.Insert(1)
```

Concurrency features

Two important features here are *goroutines* and *channels*. A goroutine is a function that may run concurrently with other functions. To start a goroutine, the keyword **go** is placed before a function invocation: **go** `f()`. These goroutines are multiplexed onto operating system threads by the Go runtime engine. The maximum number of such threads may be set using `GOMAXPROCS`.

Channels allow goroutines to communicate and synchronise. A channel type is specified by the keyword **chan** and the type of the items to be passed over the channel. In the following, two channels for integers are created.

```
c1 := make(chan int)
c2 := make(chan int, 10)
```

The first channel is synchronous. That means that a send and a receive need to occur together. A send can only complete when a corresponding receive occurs. The sender or receiver will block until such time as the corresponding operation occurs.

The second channel is buffered and hence asynchronous. The capacity is given as the second argument to `make`. As long as there is space in the buffer, a send will succeed. However, when the buffer is full, the send will block until space becomes available. Similarly. a receive will succeed as long as there are elements in the buffer and block on an empty buffer.

To send and receive on a channel the `<-` operator is used. `out <- 5` means "send 5 on channel out". Receive is similar: `n = <- c` means receive a value from channel `c` and store it in variable `n`.

Go also provides a multi-way communications mechanism through the **select** statement. It is similar to a **switch** statement, but for channels. If more than one communication is possible the runtime arbitrarily selects a case. If no cases are ready, the statement blocks. To achieve non-blocking communication, a **default** case may be added. It is selected if no other case is ready for communication.

```
select {
    case a := <-chanA:
    // Received something into a
    case b := <-chanB:
    // Received something into b
    case chanC <- c:
    // Sent c
}
```

When declaring channel variables, it is possible to assign direction to the channel. Consider the following function definition.

```
func f(in <-chan int, out chan<- int) {
    for n := range in
        out <- n
    }
}
```

The function takes two channels as arguments. One may only be received from and the other may only be sent upon. One may use the **range** of the channel to repeatedly receive elements from the channel. The loop terminates when the channel is closed by calling `close`.

Go provides another form of synchronisation through a special `WaitGroup` structure. A `WaitGroup` waits for a collection of goroutines to finish executing. A driver goroutine calls `Add` to set the number of goroutines for which to wait. Each of the goroutines then runs and calls `Done` when finished. At the same time, `Wait` can be used to block the driver until all goroutines have finished.

The following code fragment illustrates the use of such a `WaitGroup`.

```
var wg sync.WaitGroup
for _, task := range taskList {
        // Increment the WaitGroup counter.
        wg.Add(1)
        // Launch a goroutine to do work.
        go func(n int) {
                // Decrement the counter when finished.
                defer wg.Done()
                // Do work.
                doWork(n)
        }(task)
}
```

```
13 // Wait for all goroutines to complete.
14 wg.Wait()
```

2.7 Parallel computing

This section gives a brief overview of the parallel computing landscape in general and provides more detail on shared memory multiprocessor computers in particular. Recall that concurrency refers to the logical concepts of overlapping processes and parallelism refers to the physical concept of simultaneous execution of computations. Consequently, this section focuses on hardware. One class of computer is discussed in more detail and the section closes with a brief overview of a particular Intel® Xeon® processor.

This processor is the same as the one used to run the performance experiments reported in the later chapters.

2.7.1 The hardware landscape

It is possible to classify computer hardware in many ways. Here, the presentation follows Hennessy and Patterson (2011) and first lists classes of computers and then different classes of parallelism. The following classes of computers may be identified. These range from small embedded devices to large collections of computers.

Personal mobile devices These are small portable devices such as cellular phones and tablet computers. Important factors to consider are cost and energy efficiency, since these devices are typically powered by batteries. Responsiveness is a valued performance measure.

Desktop computing Devices range from low-end netbooks to high-end workstations. The trend is moving towards more battery operated laptop computers. Here the price-performance ratio is important. High-end gaming stations require a lot of performance, but price is an important factor since these devices are usually used by a single person.

Servers In the 1980s a shift from mainframe computing to desktop computing occurred and the role grew of servers to provide larger-scale file and computing services. Nowadays servers form the backbone of large-scale enterprise computing although there is a movement towards so-called cloud computing. For servers, availability and scalability are critical requirements. It is important to be able to scale a server as demand increases. Responsiveness to individual requests remains important. However, overall transaction throughput and cost-effectiveness are key.

Clusters Clusters are collections of desktop computers or servers connected via a local area network to act as single larger computer. Each node in the cluster runs its own operating system and communicates with others via communication protocols. When the nodes are commodities such as headless workstations or blade servers, the cluster is called a *commodity cluster*. On the other hand, in *custom clusters*, the nodes and the interconnect are customised and more tightly integrated.

Very large clusters are called *warehouse scale computers*. These may be used to provide services in the "cloud". Large-scale multiprocessors for scientific applications are often built using the custom cluster approach.

Embedded computers are lodged in other devices such as microwaves, printers and even cars. These computers are purpose-specific and do not have the ability to run third-party software. Embedded computers cover a wide spread of processing power and cost.

Two basic kinds of parallelism are found in applications: data-level parallelism and task-level parallelism. In data-level parallelism many data items may be operated on at the same time, whereas in task-level parallelism tasks of work may be done independently and at the same time. Computer hardware exploits these two kinds of parallelism in four major ways:

1. *Instruction-level parallelism* exploits task-level parallelism at low levels by using techniques such as pipelining to overlap the execution of instructions in the processor. The hardware may discover and exploit parallelism dynamically or the compiler may identify parallelism statically.

In general, instructions are executed in different phases, e.g. fetch, decode, and execute. During pipelining, multiple instructions may then be executing at the same time, but in different phases.

2. *Vector-based* approaches exploit data-level parallelism by applying a single instruction to data in parallel. Vector instructions, graphical processing units (GPUs), and vector architectures use this approach to achieve parallelism.

3. *Thread-level parallelism* exploits data- or task-level parallelism in a tightly-coupled hardware model that allows for interaction among threads that execute simultaneously. These threads may be independent or cooperating.

4. *Request-level parallelism* exploits parallelism among decoupled tasks specified by the programmer or operating system.

Flynn (1966, 1972) classified computers according to the number of instruction streams and data streams that can be active in the processor during processing. He placed all computers into one of four categories: SISD, SIMD, MISD, and MIMD—each of which is described below.

Single instruction stream, single data stream (SISD) A uniprocessor machine conforms to this description. Programmers think of their programs as if they execute in sequential terms. The machine may, however, make use of instruction-level parallelism techniques such as superscalar execution (Smith and Sohi 1995).

A superscalar processor may execute more than one instruction in a clock cycle by dispatching multiple instructions to different ALUs.

Single instruction stream, multiple data streams (SIMD) The same instruction is executed by different processors using different data streams. These computers thus exploit data-level parallelism. Examples of the SIMD approach include vector architectures, multimedia extensions to standard instruction sets, and GPUs.

Multiple instruction streams, single data stream (MISD) Multiple instructions operate on a single data stream. According to Hennessy and Patterson (2011), no commercial multiprocessor of this type has been built.

Multiple instruction streams, multiple data streams (MIMD) Each processor fetches its own instructions and operate on its own data. These computers exploit task-level parallelism and are more flexible than SIMD computers, although at the expense of more overhead. *Tightly coupled* MIMD computers, such as multiprocessors and multicores, exploit thread-level parallelism with cooperating threads running in parallel. *Loosely coupled* MIMD computers—clusters—exploit request-level parallelism where tasks require little to no interaction.

Modern computers do not fall neatly into one of the above classes—they are typically hybrids of one or more classes.

The focus in this thesis is on tightly coupled MIMD computers, in particular multicores, which are explored further below.

2.7.2 Multiprocessors and multicores

Hennessy and Patterson (2011) define a multiprocessor as a computer consisting of tightly coupled processors that share memory through a shared address space and that are under the coordination and control of a single operating system. Such computers range in size from two to dozens of processors, communicating and coordinating through shared memory. Multiprocessor computers also include single-chip systems with multiple cores, *multicores*, as well as systems with multiple chips, each of which may be a multicore.

Existing multiprocessors may be divided into two classes. The first class consists of so-called symmetric multiprocessors (SMPs). An SMP features a small number of processors and a single, centralised memory. Each of its

processors has equal access to this centralised memory. Such an architecture is only feasible if the number of processors is reasonably small. Since the processors all have uniform latency to memory, SMP architectures are also called uniform memory access (UMA) multiprocessors. Current multicores are typically examples of SMPs. A multi-chip system may also be an SMP.

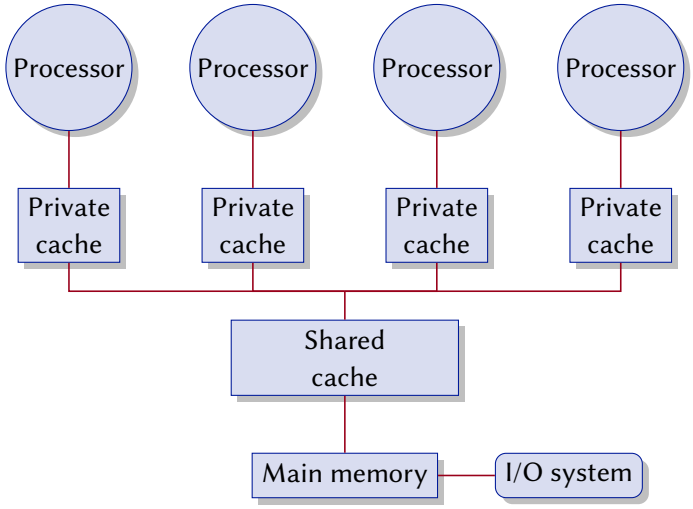


Figure 2.1: Basic structure of a symmetric multiprocessor as found in a multicore chip.

Figure 2.1 shows a typical SMP configuration in a multicore chip. For multi-chip systems, the shared cache would not be present and the interconnections from processors to memory would be between chips and not within a single chip.

The second class of multiprocessor systems has physically distributed memory and are called distributed shared memory (DSM) multiprocessors. Memory is distributed among processors in order to support larger processor counts. Because the distributed nature of such system requires fast interconnections, standard buses are replaced by more sophisticated interconnection networks. Figure 2.2 shows the basic structure of a DSM multiprocessor with four chips. Although each processor shares the entire memory, access times to local memory is faster than access times to remote memory. Hence these systems are also called non-uniform memory access (NUMA) machines.

Note that in both SMP and DSM cases all processors can access memory locations in *all* memories—it is only the access times that differ. This is different from clusters, where one processor cannot directly access the memory of another processor, but requires additional software support, like message-passing protocols.

Probably the most important performance factor in modern computer systems is the interaction between multiprocessing and the memory subsystem (Drepper 2007). In these modern systems, the central processing units (CPUs) are much faster than the random access memory (RAM). To ameliorate this performance disparity, *caches* are used to reduce memory latency. However, in multicore systems each core may have its own cache. This requires that cores should coordinate through a *cache coherence protocol*. A cache coherence protocol is a mechanism to propagate a newly written data value to a given core, thereby implementing the system's memory consistency model (Adve and Gharachorloo 1996). See Huang et al. (2013) and Molka et al. (2015) for investigations into cache and memory performance of two modern multiprocessors.

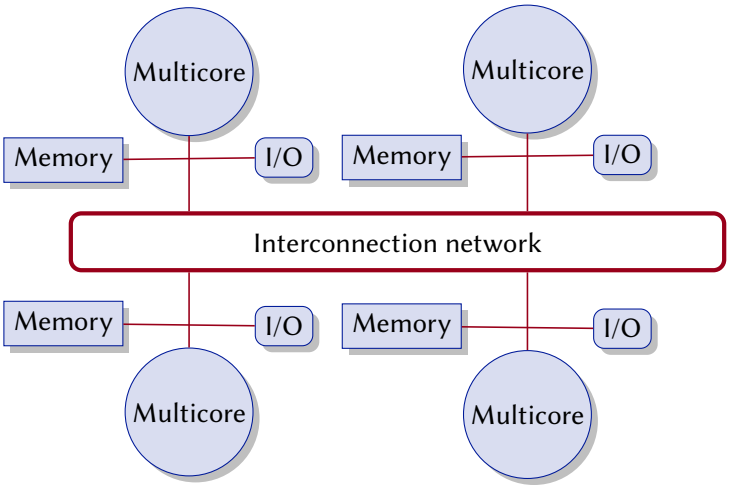


Figure 2.2: The basic structure of a distributed shared-memory multiprocessor. Each processor shares the entire memory, but access times to local memory is faster than access times to remote memory.

2.7.3 Intel Xeon E5 family

One example of DSM multiprocessors is the Intel® Xeon® E5 2600 family of processors (Farrel 2012; Syamalakumari 2013). This section gives an very brief overview of the processor family. This family was chosen because the computer on which the performance experiments were conducted, utilises one of these processors.

The processor is based on Ivy Bridge EP architecture (James 2012; Papazian et al. 2015) with 22 nm manufacturing size.

A conceptual view is given in Figure 2.3 of a six-core member of the family: the E5-2630 V2 (Intel 2014b, 2016b). This version allows for two sockets connected via Intel® QuickPath Interconnect (QPI) (Intel 2009) links. Each socket has four channels for local memory attachment and a local

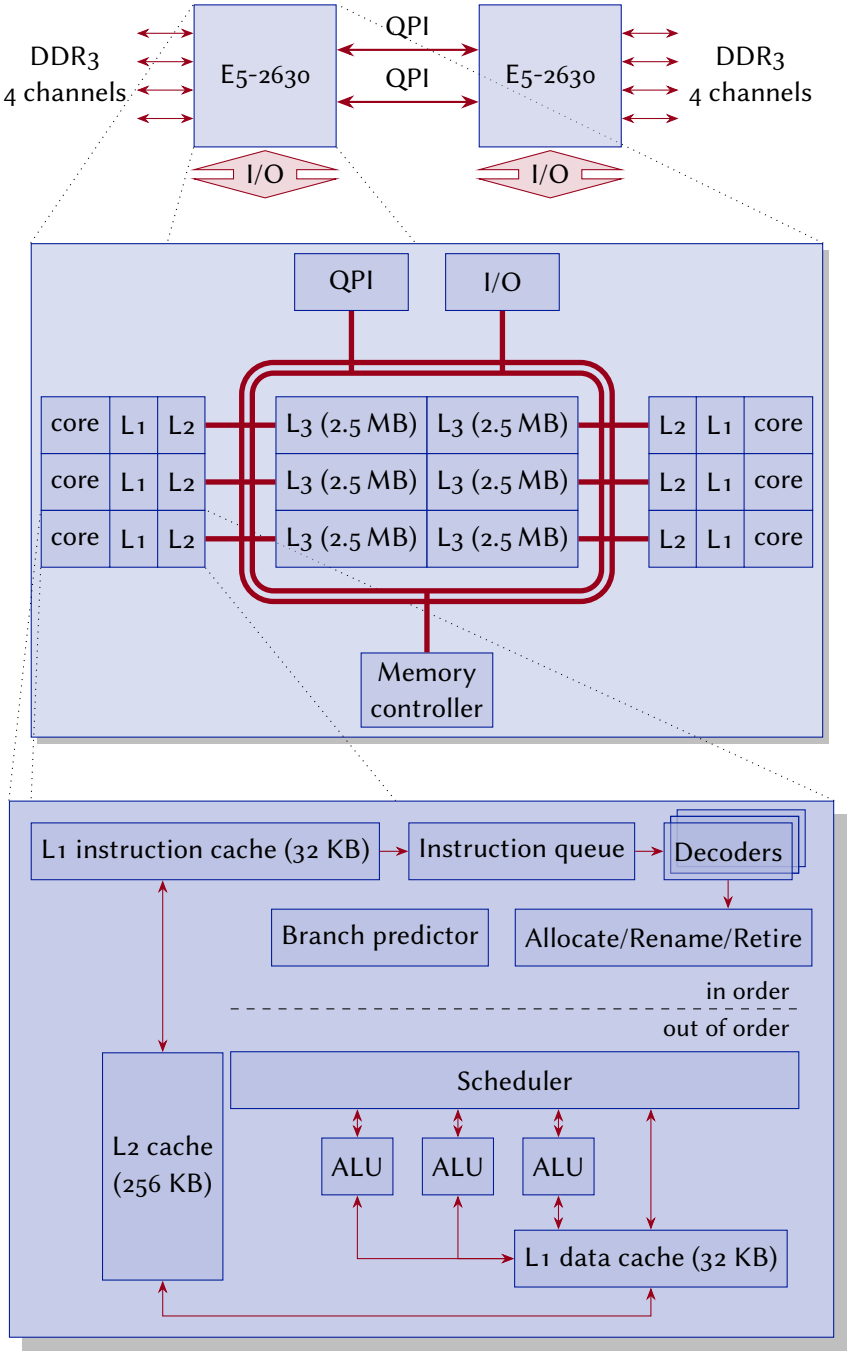


Figure 2.3: Conceptual view of a Xeon® E5-2630 V2 processor. The top level shows two chips in a DSM fashion. The next level zooms into one chip to show the multicore nature of the chip. The last level shows the simplified internals of a single core. (Adapted from Farrel (2012, Fig. 1, Fig. 5) and Intel (2016a, Fig. 2-5).)

input/output controller. Due to the distributed nature of the memory attachment, the system is an example of a NUMA system.

In a single socket there are six cores, each with local L1 and L2 cache. L3 cache is divided into slices, one for each core. All cores can, however, address all slices of L3 cache. In this case the total L3 cache available is $6 \times 2.5\,\text{MB} = 15\,\text{MB}$.

A ring-style interconnect links the cores, L3 cache, input/output controller, and integrated memory controller. The interconnect is made up of four independent rings—a data ring, request ring, acknowledge ring, and snoop ring.

Cache coherence is ensured using the MESIF protocol (Hum and Goodman 2005). More details on the caching system may be obtained from Huang et al. (2013) and Intel (2009, 2014a, 2016a).

Each core is a superscalar processor supporting pipelining. Multiple ALUs thus allows for instruction-level parallelism. SIMD is supported through vector-like instructions such as Intel® Advanced Vector Extensions. Thread-level parallelism is clearly supported via multiple cores. However, thread-level parallelism is also supported within a single core through simultaneous multi-threading (SMT), or hyperthreading in Intel® parlance. SMT allows a single core to run two threads in an interleaved manner. This results in better resource utilisation, since one thread my use available resources when the other is delayed. The threads do not truly run in parallel since they share almost all the processing resources in the core—it is only the state registers that are duplicated.

Intel® provides many features that may be enabled or disabled in the processor. The following performance-related features are *enabled* by default in an "out-of-the-box" configuration.

- Multicore support, i.e. all six cores are enabled.
- Intel® SpeedStep™. The processor's clock speed may be changed dynamically by software.
- C-States. Enables additional processor sleep states.
- Intel® TurboBoost™ allows for increasing the processor's clock rate depending on energy and thermal limits, as well as the number of active cores.
- Intel® HyperThreading enables SMT as described above.
- Cache prefetching. The hardware prefetcher automatically fetches data and code for the processor. When a cache line is requested the prefetcher also retrieves the adjacent cache line.

The preceding discussion demonstrates the architectural complexity of the multicore machine used for performance comparisons. No doubt, performance is affected by the way in which the various architectural components interact with respect to cacheing, hyperthreading, scheduling, etc. Such interactions are determined by the underlying hardware and operating system and lie largely outside of the control of the applications programmer. This holds irrespective of whether the applications are developed as conventional sequential programs or in a process-based fashion. While this thesis is concerned with comparing the performance of sequential solutions against process-based solutions, it was considered beyond the scope of the thesis to seek explanations for such differences in terms of the interaction of the architectural components of the multicore machine. Consequently, the system was left at its factory settings for the purposes of this study.

3 Aho-Corasick failure function construction

> I have no data yet. It is a capital mistake to theorise before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts.
>
> *Sherlock Holmes*

Multiple keyword pattern matching involves finding all occurrences in a text T of a set of keywords K over an alphabet Σ. One finite automaton-based solution to this problem is due to Aho and Corasick (1975). Their algorithm constructs a special kind of string matching automaton from K and then uses the automaton to find indices of T at which elements of K occur.

This chapter begins by outlining in Section 3.1 the classical sequential algorithm for computing the Aho-Corasick (AC) failure function as well as the sequential algorithm that uses this function for multiple keyword pattern matching. The purpose of providing the sequential algorithm is to indicate what parts were parallelised. It is not meant to describe or justify fully the AC algorithm. Full details may be found in Aho and Corasick (1975). Thereafter, Section 3.2 offers four alternative CSP descriptions to show how the failure function could be computed in a process-based fashion. Since the algorithm lends itself to a data-parallel implementation, Section 3.3 briefly mentions how this may be implemented. Section 3.4 then shows how these process-based descriptions are implemented in Go. Finally, Section 3.5 reports on the performance of the implementations and shows how a small refinement on the initial process-based implementations results in significant speedup improvements.

3.1 Sequential AC algorithm

To match the set of keywords $K = \{y_1, y_2, \ldots, y_k\}$, a classical algorithm for constructing an automaton from a given regular expression (B. W. Watson 1993) could be applied to the regular expression $\Sigma^*(y_1 \cup y_2 \cup \cdots \cup y_k)\Sigma^*$. The resulting automaton could then be used as the basis for matching the

multiple keywords in K. Aho and Corasick (1975, Section 6) and B. W. Watson (1995, Section 4.3) show such DFA-based approaches. The AC algorithm discussed here also uses an automaton-based approach. However, it deviates from the classical DFA approach in that it builds an automaton described by *two* transition functions. For this reason, this chapter employs the terminology of Aho and Corasick (1975) rather than that of Section 2.3.

In the description of the algorithm, the automaton's states are represented by numbers, zero being the initial state. Two transition functions are required: a so-called "goto" function, g, and a failure function, f. The g function is total, mapping every (state, character) pair either to a state or to a special label $\perp$. The failure function f is a state to state mapping. Both of these functions will be explained in more detail below. A function *output* associates a (possibly empty) set of keywords with each state. A state q for which $output(q) \neq \emptyset$ is an accepting state.

The function g corresponds with the transition function in a DFA, usually denoted with δ.

Before scanning the text T, the algorithm requires that the functions g, f and *output* be constructed (each typically implemented as a finite set of pairs from the respective function's domain and range). A brief explanation of their construction will be provided later. Assuming at this point that their construction has taken place, a left-to-right sweep through T takes place. For each character of T that is examined, the automaton goes from its current state to a next state via one of two kinds of transitions: a *goto* transition, where g determines the next state; or a *failure* transition, where f determines the next state.

Suppose that the current state is q and the current character to be examined is T_i. If $(g(q, T_i) = q') \wedge (q' \neq \perp)$, then a goto transition is made and q' becomes the new current state. On the other hand, if $g(q, T_i) = \perp$ then the failure function is used to determine the new current state. This new current state is provisionally set as $q' = f(q)$. However, both in theory and in practice, it is possible that in this new current state, $g(q', T_i) = \perp$ continues to hold. In such a case, it is necessary to move the next new current state to $q'' = f(q')$, etc. In other words, before arriving at a current state from which a goto transition can be made, it might be necessary to make a sequence of failure function transitions.

Pseudocode for the algorithm is shown in Algorithm 3.1. The goto and failure functions are computed respectively by the functions *computeG* and *computeF*. In each case, they augment the *output* function as needed. In the pseudocode, q represents the current state of the automaton and i represents the index of the current character of the text, T_i.

Here and elsewhere, pseudocode is provided in the GCL developed by Dijkstra. See Section 2.4 for details.

As will be seen in the algorithm, a **do**-loop is used to move along a sequence of failure function transitions. For as long as this loop's condition, $g(q, T_i) = \perp$, continues to hold, a failure transition is made, as expressed by the assignment statement in the loop's body, $q := f(q)$. Once $g(q, T_i) \neq \perp$

Algorithm 3.1 (AC pattern matching):

```
proc AC(Σ, K, T) →
    { Construct automaton. }
    ⟨g, output⟩ := computeG(K);
    ⟨f, output⟩ := computeF(Σ, g, output);
    { Use automaton to do matching. }
    q := 0;
    for (i : 0 .. |T| − 1) →
        do (g(q, T_i) = ⊥) → q := f(q) od;
        q := g(q, T_i);
        if (output(q) = ∅) → skip
        [] (output(q) ≠ ∅) → print('Match ending at ', i);
                             print(output(q))
        fi
    rof
corp
```

□

holds, the loop terminates and the transition $q := g(q, T_i)$ is made. At this point this new current state is tested by the *output* function to determine whether a match has been found.

3.1.1 How computeG computes the goto function

In Algorithm 3.1 it was seen that $computeG(K)$ returns both the goto function and the partially defined *output* function. Note that $computeF(\Sigma, g, output)$ is subsequently called and it, in turn, returns the fully-defined function *output*.

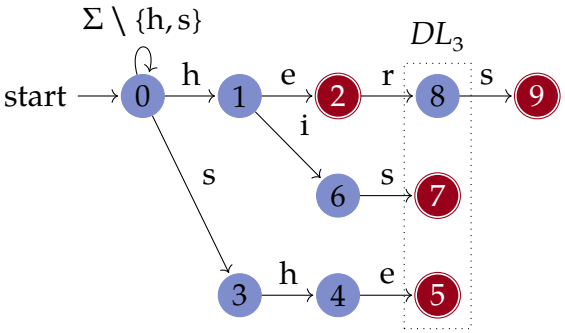
Instead of providing full algorithmic details of how $computeG(K)$ determines the goto function, g, and partially completed *output* function from a given keyword set, K, this subsection walks through a small example that sufficiently illustrates the broad algorithmic ideas for the purposes at hand.

The goto function, g, represents a trie such that $g(q, T_i) = q'$ if and only if an arc labelled by T_i exits state q and enters state q'. If, instead, $g(q, T_i) = \perp$ then there is no arc labelled T_i exiting state q.

Trie is the term used for a deterministic finite automaton graph that is a tree. See Def. 2.36.

$computeG(K)$ may be seen as incrementally building up this trie. Each new keyword is added into the trie to date by following the longest possible path in the trie that corresponds to a prefix of the current keyword. The suffix that remains of the keyword is then entered by adding a new branch

in the trie that starts at the state at which the longest prefix path of the keyword ended.



q	$output(q)$
2	{he}
5	{she}
7	{his}
9	{hers}

Figure 3.1: Trie for the keywords in $K = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$. $DL_3 = \{5, 7, 8\}$ is the set of states at depth 3.

Consider the example trie in Figure 3.1 representing the keyword set $K = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$. Initially the trie consists of the root state, 0, alone. When the first keyword, `he`, is entered, no prefix of `he` exists in the trie and the new states 1 and 2 are created with transitions on the corresponding symbols. The second keyword to be added is `she`. Again, since there is no prefix of `she` already in the trie, the new states 3, 4, and 5 are created to form a path from the root, 0. When `his` is entered, the prefix `h` is already in the trie and the rest of the keyword is appended to the trie with a split at state 1. When `hers` is added, the prefix `he` is already in the trie at state 2, so the rest of the keyword is appended there. After inserting all the keywords, a loop is added at the root to itself on all the alphabet characters for which there are no transitions out of the root. Note that the absence of transition on an alphabet character at a state indicates a transition to $\perp$. For example, $g(2, a) = \perp$.

Figure 3.1 shows the trie after the keywords in K have been inserted and the loop at the root added. At this stage the *output* function can be defined as shown on the right hand side of Figure 3.1. The function indicates for each state q the set of keywords in K that start at the root and end at q. Not shown in the figure is that $output(q) = \emptyset$ for $q = 0, 1, 3, 4, 6, 7$ and 8. Indeed, once the trie has been built, for any state, q, $output(q)$ will either be $\emptyset$ or a singleton set. It is only during later construction of the failure function that a state may possibly be identified as a final state for more than one keyword.

On adding the loop at the trie's root, the structure is strictly no longer a trie since the root is now a confluence. However, the term *trie* still will be used to refer in the text to this structure.

Recall from Definition 2.55 that the *depth* of a state q, $\overleftarrow{\mathcal{L}}_{|min|}(q)$, is the length of the *shortest* path from the start state to q. In Figure 3.1 one finds that $\overleftarrow{\mathcal{L}}_{|min|}(5) = \overleftarrow{\mathcal{L}}_{|min|}(7) = \overleftarrow{\mathcal{L}}_{|min|}(8) = 3$. Also, from Definition 2.56, refer to the set of states at depth d as DL_d. The figure shows that the states in $DL_3 = \{5, 7, 8\}$.

Shortest since a path that loops over the root before following a path to q is excluded.

Note also from the figure that one may uniquely associate a state with the string that is spelled out along the shortest path from root to that state. Hence one may legitimately speak, for example, of a prefix or suffix of a state's string. For example, `hi` is the string of state 6, and `s` is a prefix of the string of state 4, that string being `sh`. Examples of state suffices are `h` and `he`, which are suffices of state 3.

3.1.2 Computing the failure function

After Algorithm 3.1 has called *computeG* to compute the goto functin, g, the next step in constructing the AC machine is to construct the failure function, f. Notice in Algorithm 3.1 that f is returned by the function *computeF* and it is called immediately after *computeG* has run. It will be seen below that *computeF* does a level-order traversal of the trie constructed by *computeG*(K). The function f is designed to record that a suffix of a state's string is also a prefix of some other state's string. Thus, $f(q) = p$ means that a suffix of state q's string is also a prefix of state p's string. For example, in Figure 3.1, $f(5) = 2$. However, since there may be several suffixes of q's string and several states whose prefixes meet this criterion, the definition of f requires that the *longest possible* suffix of q's string should be used. This ensures that there is only one possible state, p, in the trie whose prefix corresponds to that suffix.

Algorithm 3.2 lists GCL pseudocode for the function *computeF*. This function constructs f and also updates *output*. (Note that before being passed to *computeF* as a parameter, *output(s)* would have been assigned initial values for each state, s, in the prior call to *computeG*.) The construction of f proceeds in two distinct phases.

In a first phase, *computeF* identifies DL_1, the states that directly descend from the root state, 0 (excluding the root itself). In the example, $DL_1 = \{1, 3\}$. These states are then enqueued so as to be processed in the second phase of the function. Additionally, all these states are assigned failure transitions back to the root.

The second phase of *computeF* processes the rest of the trie in a level-order fashion. This is achieved by using a FIFO queue to store states for processing. When a state is removed from the head of the queue, two actions take place.

- The first is to find the state's direct descendants, if any, and to enqueue them in the tail (i.e. to compute $s = g(r, a)$ and assign s to the queue provided $s \neq \perp$). As a consequence, *queue* starts off with states only from a given level, DL_d, but these are progressively dequeued

Algorithm 3.2 (Constructing the failure function):

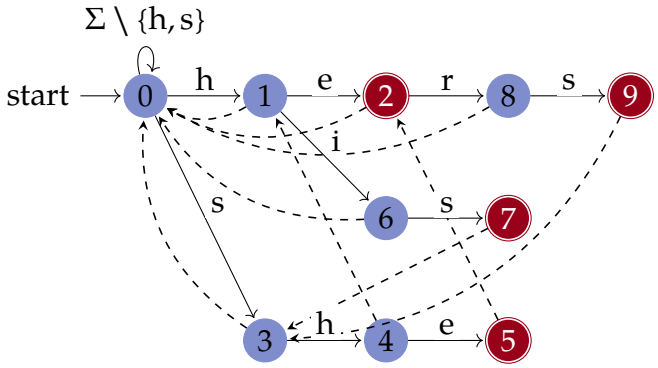
```
func computeF(Σ, g, output) →
    queue := ∅;
    { Phase 1: Build first level, i.e. DL_1 in queue. }
    for each (a ∈ Σ) →
        s := g(0, a);
        if (s = 0) → skip
        [] (s ≠ 0) → queue.enqueue(s);
                     f(s) := 0
        fi
    rof;
    { Phase 2: Determine DL_d from DL_{d−1}. }
    do (queue ≠ ∅) →
        { Invariant: ∀ℓ ∈ queue : f(ℓ) has been defined }
        r := queue.dequeue();
        for each (a ∈ Σ) →
            s := g(r, a);
            if (s = ⊥) → skip
            [] (s ≠ ⊥) → q := f(r);
                         do (g(q, a) = ⊥) → q := f(q) od;
                         { g(q, a) ≠ ⊥ }
                         f(s) := g(q, a);
                         queue.enqueue(s);
                         output(s) := output(s) ∪ output(f(s))
            fi
        rof
        { (r ∉ queue) ∧ (Succ(r) ∈ queue) ∧ Invariant }
    od;
    return ⟨f, output⟩
cnuf
```

□

from its head and states from DL_{d+1} are enqueued to its tail. Eventually *queue* will contain states only from DL_{d+1} and these will be processed from the front of *queue* with states from DL_{d+2} being added to the back, etc. In this way, states are processed one level at a time.

- The second is to update f and *output* respectively. When the queue is empty, f and *output* are returned.

From the perspectives of this study, the precise details of how f and *output* are constructed are of secondary importance. For completeness, Figure 3.2 shows their final values for the running example, f being represented by the dashed arrows in the graph and *output* being shown in the table in the figure. A formal derivation of *computeF* can be found in B. W. Watson and Zwaan (1993), albeit in slightly different notation to Algorithm 3.2.



q	$output(q)$
2	{he}
5	{she, he}
7	{his}
9	{hers}

Figure 3.2: Example trie with failure transitions included. Solid lines represent goto transitions and dashed arcs represent failure transitions.

What is important is to note the following features about the **for each** loop:

- First, the order of selection of elements from Σ is unimportant, indicating that the various iterations of the loop's body could execute concurrently, so long as concurrent updates to the queue are managed safely.
- Second, from Aho and Corasick (1975, Lemma 1) it can be shown that, in the loop's body, there is never a reference to f values of states still in the queue. Instead, only f values of states already dequeued are referenced. Put differently, f values of states at depth d depend only on f values of states at depth less than d. This indicates that it is possible to process the trie in level order, and to process concurrently the states *within* a level. In Section 3.2, CSP is used to describe various process-based architectures that exploit this opportunity for concurrency.

3.2 Process-based decomposition of the failure function construction

Section 3.1 describes that the failure function construction should happen one trie level at a time, but that the states within a level may be processed concurrently. This section describes an approach in which the algorithm is decomposed into communicating processes; hence the use of CSP to describe the implementations.

Assume that L_d represents the set of states at trie depth d and that $LAUNCHER(L_d)$ is a process responsible for launching worker processes associated with each of these states. As a first conceptual approximation, what one wants is the following sequential composition of n processes: $LAUNCHER(L_1); LAUNCHER(L_2); \cdots; LAUNCHER(L_{max})$ where L_{max} is the deepest level of the trie. Using $WORKER(s)$ to represent the process that does the work associated with an arbitrary state s, one could then model the concurrent and independently processed work associated with all states at level L_d as $|||_{s \in L_d} WORKER(s)$. However, since one does not know *ab initio* the states in level L_d, one cannot simply use $|||_{s \in L_d} WORKER(s)$ as a refined description of the $LAUNCHER(L_d)$ process. Before launching $|||_{s \in L_d} WORKER(s)$, a process has to execute that dynamically gathers the states of level d into L_d as they become known by the processing taking place at level $d-1$. Additionally, worker processes at level d have to pass on information about successor states in L_{d+1} before terminating. A description of a process-based solution that works through one trie level at a time is thus somewhat more complicated than the mere sequential composition of a process per level. The subsections that follow describe four alternative process-based implementations of the *computeF* function from Algorithm 3.1.

To streamline the notation and narrative, *level* will be used for *depth level* and L_d will mean DL_d.

These alternative implementations, however, only relate to phase 2 of *computeF*. Phase 1 is treated as an abstract process, *PHASE1*, that identifies the state elements of L_1 and enqueues them for subsequent processing. Thus, abstracting from parameters, see *computeF* as: $computeF = PHASE1; PHASE2$ where four variants of *PHASE2* are considered below.

3.2.1 Variant 1

In the first variant, the processing work associated with each state is performed in a separate process. Processes within a given trie level execute concurrently, but are only created and launched once all processes executing at one level earlier have completed.

The first variant is modelled as two concurrent processes, *LAUNCHER* and *GATHERER*, that interact through a channel called *result*.

$$VARIANT1 = LAUNCHER(L_1) \underset{\{\!|result|\!\}}{\|} GATHERER(\emptyset, |L_1| \times |\Sigma|))$$

LAUNCHER has as parameter a set of states. Initially the parameter is L_1 as shown above. It represents the set of states at level 1 of the trie, made available in *PHASE1* mentioned above. *LAUNCHER* is responsible for the processing associated with each state in its state set parameter. This processing corresponds with the body of the **for each** loop in Algorithm 3.2. Part of this processing entails that *LAUNCHER* should communicate to *GATHERER* the direct descendants of each state that it processes. Another part of this processing is that it should communicate whether the state that it is processing, say s, has no transition on a given symbol, say a. In other words, *LAUNCHER* has to communicate to *GATHERER* the value of $g(s, a)$. The communication takes place via channel *result*. Before specifying these actions of *LAUNCHER* in greater detail, the functioning of *GATHERER* is explained.

GATHERER takes two parameters. The first is a set of states and the second is an integer. The first parameter identifies the states passed to it to date via the *result* channel. Initially this is the empty set as shown above. The second parameter indicates the number of messages it still needs to receive along the *result* channel before terminating. Initially this is $|L_1| \times |\Sigma|$ since it needs to receive the outcome $g(s, a)$ for all $s \in L_1$ and for all $a \in \Sigma$. The task of *GATHERER*, therefore, is to accumulate all the states of the next level that will need to be processed by a new instance of *LAUNCHER*.

Let us now consider *GATHERER* in more detail, where the state set Q and a counter *Cnt* are its parameters.

$$\begin{array}{l}
GATHERER(Q, Cnt) = \\
\quad \text{if } (Cnt > 0) \text{ then} \\
\quad\quad result?r \rightarrow \\
\quad\quad\quad \text{if } (r \neq \perp) \text{ then} \\
\quad\quad\quad\quad GATHERER(Q \cup \{r\}, Cnt - 1) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad GATHERER(Q, Cnt - 1) \\
\quad \text{else} \\
\quad\quad \text{if } (Q \neq \emptyset) \text{ then} \\
\quad\quad\quad LAUNCHER(Q) \underset{\{\!|result|\!\}}{\|} GATHERER(\emptyset, |Q| \times |\Sigma|) \\
\quad\quad \text{else} \\
\quad\quad\quad SKIP
\end{array}$$

- Each *GATHERER* runs in parallel with a *LAUNCHER* instance that is responsible for processing a given level of the trie. If there are $|L_i|$ states in level i, then the associated *LAUNCHER* instance will send exactly $|\Sigma| \times |L_i|$ messages to *GATHERER*, using the *result* channel. The above specification assumes that *LAUNCHER* has exactly *Cnt* state

values left to send. When *LAUNCHER* has sent these *Cnt* values, it sends no more and should terminate.

LAUNCHER does not quite terminate since, for simplicity of presentation, its internal buffer, described later in this section, has no termination code.

- If $Cnt > 0$ then *GATHERER* recurses *Cnt* number of times, decrementing *Cnt* at each recursion instance. Before recursing, it reads a state from channel *result* and adds it to the state set Q for the next recursion. If the special $\perp$ value is read, it is not inserted into Q.
- If $Cnt = 0$ and $Q \neq \emptyset$, the process's behaviour is described by $LAUNCHER(Q)$ concurrently communicating with $GATHERER(\emptyset, |Q| \times |\Sigma|)$ over the *result* channel. This concurrent composition of *LAUNCHER* and *GATHERER* describes the processing to take place at a level whose state set is Q. Note that this instance of *LAUNCHER* does not compete with the previous *LAUNCHER* for the *result* channel, since that previous *LAUNCHER* has terminated (or should have—see first bullet above).
- If $Q = \emptyset$ and $Cnt = 0$ then *GATHERER* terminates.

In this way the levels are processed sequentially. Consider now how the processing within a level can be done.

Recall that in *VARIANT1*, the processing associated with each state is carried out in a *separate* process. To identify the processes associated with states, map each state $s \in L_j$ to an integer in $1 \,..\, |L_j|$. Let $WORKER_i(s)$ denote the i^{th} worker process that is associated with some generic state, s. We can then model all the workers running independently and concurrently as follows.

$$WORKERS(L_j) = |||_{i=1..|L_j|} WORKER_i(s_i)$$

where s_i denotes the i^{th} state at level j of the trie.

$WORKER_i(s)$ is described below by a local process, $P_i(\Sigma, s)$. Here the two parameters are the text alphabet Σ and the generic state s. The generic description of this process for a non-empty symbol set S and generic state s is the non-deterministic choice between several sub-processes, one for each state, a, in S. Each such sub-process entails a trace of two events followed by a recursion to the same process with altered parameters.

- The first event reflects the update to the failure table based on symbol a, denoted for convenience by the event *updateF.a.s*.
- The last event is the communication of $g(s, a)$ on the channel *out.i*. (Recall that $g(s, a)$ is the descendant of state s on symbol a, if such a descendent exists. If there is no descendant of state s on symbol a, then $g(s, a)$ assumes a special value, $\perp$.)

After engaging in these events, local process P_i recurses, but with a symbol set reduced by the elimination of symbol a. When the symbol set has been reduced to the empty set, the local process, P_i, terminates, and hence also the $WORKER_i(s)$.

As a result, exactly $|\Sigma|$ events are communicated on $out.i$, some of which are $\perp$ events and the remainder, state descendant events. When all of these $|\Sigma|$ events have been communicated, the worker terminates. The following CSP specification defines the behaviour of $WORKER_i(s)$:

$$WORKER_i(s) = P_i(\Sigma, s)$$

$$P_i(S, s) =$$
$$\quad \text{if } (S \neq \emptyset) \text{ then}$$
$$\quad\quad \Box_{a \in S}\, updateF.a.s \rightarrow out.i!g(s,a) \rightarrow P_i(S \setminus \{a\}, s)$$
$$\quad \text{else}$$
$$\quad\quad SKIP$$

In this variant, the output from each $WORKER_i(s)$ is deposited via channel $out.i$ into a multiplexing buffer, *BUFF1*. As will be shown below, *BUFF1* is one of two components of *LAUNCHER*—the other being *WORKERS*, the interleaving of a set of workers defined above. *BUFF1* reads input from an array of *left* channels, one for each worker, and outputs values on the single *right* channel. *LAUNCHER*'s definition links each worker's output channel $out.i$ to $left.i$, the corresponding input channel of *BUFF1*. As shown below, it also renames *BUFF1*'s *right* channel to *result* to synchronise with *GATHERER*.

$$LAUNCHER(L) =$$
$$\quad (WORKERS(L)\ [out \leftrightarrow left]\ BUFF1)[\![^{result}/_{right}]\!]$$

The purpose of *BUFF1* is to smooth the data flow between the *WORKER*s and *GATHERER*. In order to ensure this, the capacity of *BUFF1*, namely N_1, has to be suitably large. $BUFF1 = MBUFF(\langle\,\rangle, N_1, |L|)$, where *MBUFF* is the multiplexing buffer defined as an example in Section 2.5.2.

Figure 3.3 illustrates how the processes interact for the states at depth j in the trie. $|L_j|$ workers are created and they send their results via *BUFF1* to a *GATHERER* process. This process should read $|L_j| \times |\Sigma|$ results and construct L_{j+1}.

3.2.2 Variant 2

In order to test whether or not an implementation of Variant 1 would be needlessly inefficient in associating a process with every single state of a level, a second variant was considered. In this second variant a *WORKER*

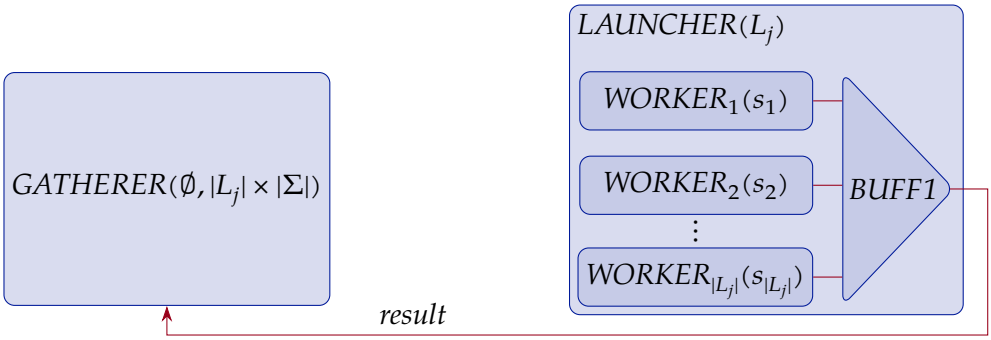


Figure 3.3: Processes involved in computing the failure function for states at depth j in the trie.

process handles more than one state and there are a fixed number of such processes. Instead of being parameterised by a state, a *WORKER* process now starts by reading in a state, s, from an input channel, *in*. It then behaves the same as the previously described local process in Variant 1, $P(\Sigma, s)$. Once $P(\Sigma, s)$ has completed, the *WORKER* process repeats itself, waiting to read another state from the *in* channel.

$$WORKER_i = in.i?s \rightarrow (P_i(\Sigma, s); WORKER_i)$$

In Variant 2, there are a fixed number (n) of such workers working independently. The composite process, *WORKERS*, describes the resulting overall process.

$$WORKERS = |||_{i=1..n} WORKER_i$$

Two buffer processes facilitate communication to *and* from the workers. The first is *BUFF1*, the same multiplexing buffer as in Variant 1. Recall that this buffer reads a state as input from any one of an array of *left* channels and then emits these state elements on a single *right* channel.

The second buffer, *BUFF2*, has capacity N_2 and also operates on a FIFO basis. Its behaviour is described by the process, *DBUFF*, from Section 2.5.2. It receives state elements from a channel *left*, provided that it is not full. While not empty, *DBUFF* may emit the least recently stored state on any available one of an array of *right* channels. If more than one such channel is available, the choice of channel may be made arbitrarily.

One may compose the workers with the buffers by chaining and renaming the relevant channels as shown below.

$$\begin{aligned} &BWORKERS = \\ &\quad BUFF2[\![^{work}/_{left}]\!] \\ &\qquad [right \leftrightarrow in] \\ &\quad WORKERS \\ &\qquad [out \leftrightarrow left] \\ &\quad BUFF1[\![^{result}/_{right}]\!] \end{aligned}$$

The right hand side of Figure 3.4 illustrates the chaining. The figure indicates that the number of *WORKER* processes is fixed at n and no longer given by $|L_j|$ (the number of states in the given level) as shown in Figure 3.3.

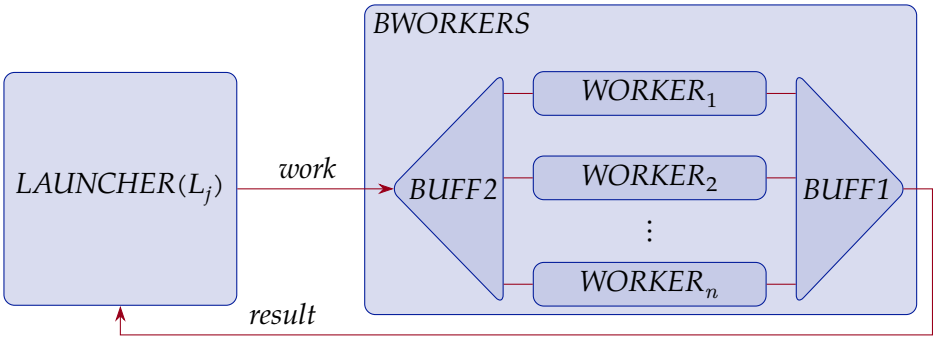


Figure 3.4: Processes active while processing the states in level L_j.

A process is needed to dispatch the states in the current level to these n workers. To do this we rely on a new *LAUNCHER* process that models the processing of states in the state set, L, of the current level. *LAUNCHER*'s actions are described by the sequential composition of a *SENDER* followed by a *GATHERER* process. We therefore model it as follows.

$$LAUNCHER(L) = SENDER(L); GATHERER(\emptyset, |L| \times |\Sigma|)$$

The *SENDER* process dispatches all the elements of L to the workers via the *work* channel. Its CSP description corresponds to that of the local process $P(\Sigma, s)$ described for Variant 1, namely:

$$\begin{aligned}
&SENDER(S) = \\
&\quad \text{if } (S \neq \emptyset) \text{ then} \\
&\qquad \textstyle\bigsqcap_{a \in S} work!a \rightarrow SENDER(S \setminus \{a\}) \\
&\quad \text{else} \\
&\qquad SKIP
\end{aligned}$$

SENDER thus transfers the elements of its parameterised set of states to the workers on the *work* channel and then terminates.

GATHERER then dynamically constructs the set containing the states of the next level based on information received from the workers via the *result* channel. *GATHERER* is slightly modified from Variant 1. As in Variant 1, the parameters of *GATHERER* are a set of states, Q and a counter Cnt. Its description is given below:

$$\begin{aligned}
&GATHERER(Q, Cnt) = \\
&\quad \text{if } (Cnt > 0) \text{ then} \\
&\qquad result?r \rightarrow \\
&\qquad\quad \text{if } (r \neq \perp) \text{ then}
\end{aligned}$$

$$
\begin{array}{l}
\quad\quad\quad\quad GATHERER(Q \cup \{r\}, Cnt - 1) \\
\quad\quad\quad\quad \text{else} \\
\quad\quad\quad\quad GATHERER(Q, Cnt - 1) \\
\quad\quad \text{else} \\
\quad\quad\quad \text{if } (Q \neq \emptyset) \text{ then} \\
\quad\quad\quad\quad LAUNCHER(Q) \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad SKIP
\end{array}
$$

As before, if $Cnt > 0$ then the process recursively reads the next level's states from *result*, at each recursive instance accumulating the most recently read state into Q and decrementing Cnt. Also as before, if the special $\perp$ signal is read, it is not inserted into Q. However, the description of $GATHERER(Q, Cnt)$ differs slightly from its description for Variant 1 when $Cnt = 0$ and $Q \neq \emptyset$. In this case it is simply described by $LAUNCHER(Q)$—that is, it behaves as the *LAUNCHER* process that deals with the set of states, Q.

As in Variant 1, assume that *PHASE1* has discovered L_1 so that Variant 2 of *PHASE2* is:

$$VARIANT2 = LAUNCHER(L_1) \underset{\{\!|work,result|\!\}}{\|} BWORKERS$$

Note that *VARIANT2* deadlocks if the buffers are not sufficiently large. For example, suppose that states from level L are being processed, and that $|L|$ is very large. *SENDER(L)* emits states on *work* and the workers' results are buffered in *BUFF1*. Eventually *BUFF1* fills up and the workers cannot make progress. *SENDER* then fills up *BUFF2* and waits for more space to become available. Space will be made available when *GATHERER* starts to consume elements via *result*. Unfortunately *GATHERER* is waiting for *SENDER* to complete before it can start—a classic wait-for-cycle. A necessary and sufficient condition to guarantee that the variant is deadlock-free in terms of buffer usage is that: $\langle \forall\ i : i \in [1, max + 1) : |L_i| \leq N_1 + N_2 + n \rangle$ should hold.

Recall that N_1 and N_2 are the capacities of the two buffers and that n is the number of *WORKER* processes.

3.2.3 Variant 3

Variant 3 is very similar to Variant 2, but the potential deadlock issue is addressed. Instead of sending out all the state elements of the current level L and then reading in the elements of the new level, there is a choice between sending out elements of the current level and reading in elements of the next level. This means that the *LAUNCHER* process is no longer the sequential composition of the *SENDER* and *GATHERER* processes as in Variant 2. Instead, these two processes are reformulated in such a way that each next trace event of *LAUNCHER* can either be the sending of a

state element along channel *work* or the reception of a state indicator along channel *result*. To this end, the *LAUNCHER* process is defined in terms of a local process, *SAG* parameterised by the set of states still to send on the *work* channel, the set of states received to date on the *result* channel and the number of transmissions still to be made on the *result* channel. Initially, for $LAUNCHER(L)$ these are L, $\emptyset$ and $|L| \times |\Sigma|$ as shown below:

SAG is an acronym for *Sender And Gatherer.*

$$LAUNCHER(L) = SAG(L, \emptyset, |L| \times |\Sigma|)$$

The CSP elaboration of the *SAG* process given below shows that the status of its parameters has to be evaluated in order to describe the next possible events. If there are still states to be transmitted on the *work* channel and still the possibility of receiving a state indicator message on the *result* channel, then there is a choice between the initial events of the *SENDER* and *GATHERER* processes. If the parameters of the *SAG* process are such that one, but not both, of the *SENDER* or *GATHERER* processes may engage in a next event, then the appropriate process is invoked. If neither of these processes can engage in a next event (because all states at the current level have been transmitted down the *work* channel and all state indicator messages have been received from the *result* channel), then *LAUNCHER* is invoked with the set of states received to date—the states of the next level. Of course, if there are no states at the next level, then the process terminates.

$$\begin{array}{l} SAG(S, Q, Cnt) = \\ \quad \text{if } (S \neq \emptyset \wedge Cnt > 0) \text{ then} \\ \quad\quad SENDER(S, Q, Cnt) \,\square\, GATHERER(S, Q, Cnt) \\ \quad \text{else if } (Cnt > 0) \text{ then} \\ \quad\quad GATHERER(S, Q, Cnt) \\ \quad \text{else if } (S \neq \emptyset) \text{ then} \\ \quad\quad SENDER(S, Q, Cnt) \\ \quad \text{else if } (Q \neq \emptyset) \text{ then} \\ \quad\quad LAUNCHER(Q) \\ \quad \text{else } SKIP \end{array}$$

The definitions of *SENDER* and *GATHERER* are practically the same as before. However, they now recurse to the local *SAG* process, rather than to a new instance of themselves, as in Variant 2.

$$SENDER(S, Q, Cnt) = \bigsqcap_{a \in S} work!a \rightarrow SAG(S \setminus \{a\}, Q, Cnt)$$

$$\begin{array}{l} GATHERER(S, Q, Cnt) = \\ \quad result?r \rightarrow \\ \quad\quad \text{if } (r \neq \perp) \text{ then} \\ \quad\quad\quad SAG(S, Q \cup \{r\}, Cnt - 1) \\ \quad\quad \text{else} \\ \quad\quad\quad SAG(S, Q, Cnt - 1) \end{array}$$

3.2.4 Variant 4

Instead of a repeated choice between sending a node and receiving a node, one could allow the sending and receiving of nodes to continue independently and concurrently, yielding the fourth and final variant.

$$LAUNCHER(L) = SENDER(L) \;|||\; GATHERER(\emptyset, |L| \times |\Sigma|)$$

SENDER and *GATHERER* are the same as in *VARIANT2*.

3.2.5 Process termination

It should be pointed out that termination has not been addressed fully in the above process descriptions. A special sentinel could be used to terminate the processes when there are no more states to process. *GATHERER* could be modified to send out a special term event when $Q = \emptyset$ and *WORKER* could be modified to terminate when it receives term after sending it along. The *BUFF2* processes would need to be modified to emit term on all its *right* channels only after all the existing elements have been emitted. Similarly, *BUFF1* would need to accept a term from all its *left* channels before it may terminate when empty. A full elaboration of this "poisoning" approach in CSP is beyond the scope of this thesis as it would not add insight into the focus here, namely to explore process-based architectural alternatives for implementing phase 2 of *computeF* in Algorithm 3.2.

Note that the same termination approach may be used in the CSP models found in subsequent chapters. To keep the models concise, termination is also not addressed completely in those models. Terminating completely is treated as an implementation matter.

3.3 Concurrency through data partitioning

The multiple keyword string matching problem also lends itself to a more direct concurrent solution that is purely data-driven: the keyword set may be arbitrarily partitioned and multiple instances of the AC algorithm's construction phases (*computeG* and *computeF*) may be run on each of those partitions. The result is a set of disjoint AC automata, each of which could then be used to search, concurrently or sequentially, for keywords in a text. For completeness, such a data-parallel approach has also been implemented and its performance will be compared to the process-based implementations.

The implementation is labelled "Split" in Figure 3.6.

3.4 Implementation

The sequential algorithm of Section 3.1 as well as the four variants of Section 3.2 were implemented in the programming language Go. As mentioned in Section 2.6, Go was chosen because it supports light-weight processes called goroutines and it also contains typed channels as part of the language.

The CSP descriptions are mapped to Go in the following manner. Concurrent CSP processes are mapped to concurrent goroutines and CSP external choice is mapped to Go's **`select`** statement. A buffered Go channel shared by concurrent goroutines was used to communicate states between processes. This means that in Variant 1's implementation a buffered channel `results` was used to communicate the output of the *WORKER* goroutines to *GATHERER*. Similarly were two buffered channels (`work` and `results`) used in the implementations of Variants 2, 3, and 4. Consequently it is not required to implement explicit Go processes for *BUFF1* and *BUFF2*.

Extracts from the Go implementations for Variants 1 to 4 are given below. Please note that the listings are not complete, since they are meant to illustrate how the processes interact. Ellipses indicate omitted code. Complete source code will be made available at `http://fastar.org`. Section 2.6 provides an overview of Go in which enough of the syntax and semantics of the language are covered to comprehend the following code fragments.

3.4.1 Variant 1

The Go outline for the first variant of a worker is given in Go code 3.1. Recall that in the first variant, a worker only processes a single state and then terminates. It accepts the state as a parameter and then, for each symbol in Σ, updates the failure and output functions and communicates on channel `out` the state's direct descendants. These descendants are the elements of the next level. The process terminates once it has iterated though Σ.

Go code 3.1: AC worker code, Variant 1.

```
1 func worker(curState State, out chan<- State) {
2     for _, ch := range this.sigma {
3         curDest, ok := this.gotoTable[deltaPair{curState, ch}]
4         if ok {
5             // Communicate next level state.
6             out <- curDest
7             // Update failure function and output function.
8             ...
```

```
9           } else {
10              // Communicate failure.
11              out <- FAIL
12          }
13      }
14  }
```

Go code 3.1, line 3 shows how the worker obtains $g(s,a)$, where s is the state being processed and $a \in \Sigma$ is the current alphabet symbol. The goto function, g, is implemented in Go as a map from a `deltaPair` to a `State`. Here the destination `curDest` is looked up using a struct literal as key. If the key is present, then variable `ok` is **`true`** and `curDest` holds the destination state. This state is sent out on channel `out` and the failure and output functions are updated. If, however, the key is not present, `ok` will be **`false`** and `curDest` will be the zero value of the `State` type. In this case no updates are made and a special value `FAIL` is emitted on channel `out`.

Recall from Section 2.6 that a map is an associative array that maps a key to a value.

The driver goroutine for computing the failure function in shown in Go code 3.2.

Go code 3.2: Implementation of Variant 1 driver routine.

```
func (this *AC_Matcher) computeFVar1() {
    todoQ := make([]State, 0)
    // Phase 1: find level 1 states, enqueue them on todoQ
    //          and set their failure state to START.
    ...
    // Phase 2:
    for len(todoQ) > 0 {
        nodes := len(todoQ)
        results := make(chan State, nodes*len(this.sigma))
        // Launch workers for current level.
        for i := nodes; i > 0; i-- {
            cS := todoQ[0]
            todoQ = todoQ[1:]
            go worker(cS, results)
        }
        // Gather all results from workers.
        var tmp State
        for k := 0; k < nodes*len(this.sigma); k++ {
            tmp = <-results
            if tmp != FAIL {
                todoQ = append(todoQ, tmp)
            }
        }
        close(results)
```

```
    }
}
```

The function `computeFVar1` performs Phase 1 and then starts the level-order traversal of the trie, launching goroutines (`workers`) for each state in a level, before proceeding to gather and process their `results`. This takes a small licence with the concurrency in the CSP specification (Section 3.2.1) in that termination of the *LAUNCHER* and *GATHERER* components is not synchronised—the *forking* of the *LAUNCHER* `worker` processes is not matched by a corresponding *join*. This does not matter here because nothing is specified to happen afterwards. The benefit from this licence is that the recursion in *GATHERER* becomes *tail-recursion* and can be implemented with loops. The other simplification in this code is that the multiplexed buffering of communications between the `worker` processes and the *GATHERER* (Figure 3.3) is managed automatically by a single Go channel `results`.

3.4.2 Variant 2

Recall that the number of worker processes is fixed in the second variant. A given worker repeatedly receives work on an input channel and emits the results on an output channel. The remainder of the worker code is the same as in Variant 1.

Go code 3.3: Implementation of Variant 2 worker

```
func worker(in <-chan State, out chan<- State) {
    for curState := range in {
        // Implementation as in Variant 1
        ...
    }
}
```

As shown in Go code 3.4, all the states for a given level are first sent out on channel `work`, and after that, all the responses are gathered from the workers via channel `results`.

Go code 3.4: Driver routine for Variant 2.

```
func (this *AC_Matcher) computeFVar2(numW int, b1S, b2S int) {
    todoQ := make([]State, 0)
    // Phase 1: find level 1 states, enqueue them on todoQ
    //          and set their failure state to START.
    ...
    // Phase 2:
```

```
    work := make(chan State, b1S)
    results := make(chan State, b2S)
    // Start up workers
    for i := 0; i < numW; i++ {
        go worker(work, results)
    }
    for len(todoQ) > 0 {
        // Send out current level
        nodes := len(todoQ)
        for i := nodes; i > 0; i-- {
            work <- todoQ[0]
            todoQ = todoQ[1:]
        }
        // Gather the elements of the next level.
        var tmp State
        cnt := 0
        for k := 0; k < nodes*len(this.sigma); k++ {
            tmp = <-results
            if tmp != FAIL {
                todoQ = append(todoQ, tmp)
                cnt++
            }
        }
    }
    close(work)
    close(results)
}
```

3.4.3 Variant 3

In this variant, the `workers` are implemented exactly as in the previous variant. The driver goroutine is modified to send out a state of the current level on channel `work` or receive on channel `results` a state of the next level. This choice between alternatives is implemented with a Go `select` statement.

Go code 3.5: Driver routine for Variant 3.

```
func (this *AC_Matcher) computeFVar3(numW int, b1S, b2S int) {
    todoQ := make([]State, 0)
    // Phase 1: find level 1 states, enqueue them on todoQ
    //          and set their failure state to START.
    ...
    // Phase 2:
```

```
    // Channels for communication.
    work := make(chan State, b1S)
    results := make(chan State, b2S)
    // Start up workers
    for i := 0; i < numW; i++ {
        go worker(work, results)
    }
    for len(todoQ) > 0 {
        nodes := len(todoQ)
        pending := nodes * len(this.sigma)
        var tmp, cS State
        cS = FAIL
        for nodes > 0 {
            if cS == FAIL {
                cS = todoQ[0]
                todoQ = todoQ[1:]
            }
            // Choice between sending and receiving
            select {
                case work <- cS:
                cS = FAIL
                nodes--
                case tmp = <-results:
                if tmp != FAIL {
                    todoQ = append(todoQ, tmp)
                }
                pending--
            }
        }
        // Only need to receive since all nodes have been sent.
        for pending > 0 {
            tmp = <-results
            if tmp != FAIL {
                todoQ = append(todoQ, tmp)
            }
            pending--
        }
    }
    close(work)
    close(results)
}
```

3.4.4 Variant 4

In the final variant, shown in Go code 3.6, a `sender` process sends out all the elements of the current level on channel `work`. This `sender` process runs concurrently with the driving goroutine which gathers the elements of the next level from channel `results`. Line 19 shows where the `sender` is defined for the current level and line 24 shows where the goroutine is started, causing it to run concurrently with the receiving code on line 27.

Go code 3.6: Driver routine for Variant 4.

```
1  func (this *AC_Matcher) computeFVar4(numW int, b1S, b2S int) {
2      // Phase 1: find level 1 states, enqueue them on todoQ
3      //          and set their failure state to START.
4      ...
5      // Phase 2:
6      work := make(chan State, b1S)
7      results := make(chan State, b2S)
8      // Start up workers
9      for i := 0; i < numW; i++ {
10         go worker(work, results)
11     }
12     for len(todoQ) > 0 {
13         numnodes := len(todoQ)
14         nodes := todoQ
15         todoQ = todoQ[numnodes:]
16         pending := numnodes * len(this.sigma)
17         var r State
18         // Send the current level in a new goroutine
19         sender := func() {
20             for _, cS := range nodes {
21                 work <- cS
22             }
23         }
24         go sender()
25         // Gather the next level
26         for pending > 0 {
27             r = <-results
28             if r != FAIL {
29                 todoQ = append(todoQ, r)
30             }
31             pending--
32         }
33     }
34     close(work)
```

```
35      close(results)
36  }
```

3.5 Performance analysis

Section 3.2, has described four related ways of concurrently implementing the failure function construction in the AC algorithm. Naturally, it is of interest to determine whether such a process-based decomposition can result in performance improvement over a purely sequential algorithm. In order to test this, one needs to compare the runtimes of the different Go implementations.

3.5.1 Experimental setup

The aim of the experiment described here was simply to determine whether the concurrent implementations can construct the AC failure function faster that the sequential version. No attempt was made to investigate completely the performance characteristics of the concurrent implementations, nor was any attempt made to further refine or refactor the Go implementations to maximise possible speedups in comparison with the sequential implementation.

Consider first the input to the problem. To construct the AC automaton, a set of keywords is required. In order to observe the performance of the implementations over a range of input, one needs to vary the keyword sets. It was decided to vary in terms of two aspects: the number of keywords and the shape of the trie as explained below.

The set $\{10, 100, 1000, 10\,000, 100\,000\}$ represents the different sizes of keyword sets that were used in the experiments.

Varying the shape of the trie was achieved by using three sources for keywords. The first source is simply a text file with 200 000 lines, where each line is the concatenation of a single symbol a. The text on line i of the file is a^i, that is, a repeated i times. To construct a keyword set of k elements, pseudo-randomly select a line r in the text file and then use the words on lines r to $r + k - 1$ as the keyword set. In this manner a number of different keyword sets may be generated. In this case the alphabet size was two symbols—one symbol used in the keywords and another symbol on which to make failure transitions.

Essentially the keyword sets in this case only vary in terms of the length of the keywords.

The second source is derived from an English word list by ScrapMaker.com (2015). The 213 557 words from the list were randomly ordered and written

to a text file, one word per line. Finally, the third source is simply the same list of words, but lexicographically ordered, one word per line. The same approach as for the single-symbol case was used to generate keyword sets from these sources: For a keyword set of k elements, randomly select a line r in the text file and then use the words on lines r to $r + k - 1$ as the set of keywords. The alphabet size in these two cases was 256 symbols. This provides for the possibility that the text to be searched might contain more than just the 26 letters used for English words.

The single symbol keyword data ensures that each keyword that is inserted has a maximal proper prefix in the trie. This then results in a deep, narrow trie since only one symbol is used and the longest possible keyword is 200 000 symbols long.

In the shuffled English source, the probability of including, in the same keyword set, words with common prefixes, is small. This should cause the trie to branch out early, resulting in a wide but shallow tree. On the other hand, the sorted English case has a high probability of including words with common prefixes. This should result in a trie with some branching, but also some reuse of states. The trie should be fairly shallow (limited to the length of the longest keyword) and narrow close to the root.

Fifty keyword sets were randomly generated for each keyword set size and type combination. In the case of the single symbol keywords, the very largest keyword set size of 100 000 was not tested since the runtimes were expected to be excessive. For all the keyword sets, the AC automaton was generated using the sequential algorithm as well as the four concurrent variants. In each case, the time to construct the failure function was recorded. In order to mitigate transient operating system effects, each execution was repeated five times and the minimum of these was recorded as the data point for that case.

In the implementation of Variant 1, the capacity of the buffered channel `results` was $|L| \times |\Sigma|$, large enough to contain all the results for a level $|L|$. In Variants 2 to 4, the number of concurrent worker goroutines `numW` was set to 24. Very large buffers were used in order to avoid blocking due to buffer-space limitations. Buffer `work` had a capacity of 100 000 elements and `results` had a capacity of `numW` × 100 000.

Setting `numW` to 24 corresponds with twice the number of virtual cores in the machine used for the experiments. The number was chosen to ensure that all the cores have work.

Go version 1.5.1 was used to compile the implementations on a machine running Linux 3.10.17. The experiments were run on the same machine. It had a single six-core CPU with two hardware threads per core and 16 GB of RAM. The Go runtime was configured to run at most twelve operating system threads concurrently. The goroutines are then internally scheduled onto these threads.

Specifically an Intel® Xeon® E5-2630 v2 at 2.60 GHz. See Section 2.7.3 for more details.

3.5.2 Observations

In order to gain an appreciation of the size of the input and to assess whether the generated trie structures do vary between the different text types, the following characteristics of the generated tries were recorded: the *depth* of the trie, the *total* number of states in the trie, and the number of states for *each* level in the trie. Table 3.1 shows a summary of the observation grouped by text type and keyword set size. Note that an entry under "Level" is a median number of states *per level* over all the tries for that particular text type and keyword set size combination.

Table 3.1: Observed sizes of the generated tries grouped by text type and size of the keyword set. "Depth" contains the median number of levels in the tries and "States" the median total number of states in the tries. "Level" shows the median number of states per level over all the levels in all the tries in a particular group.

		Depth		States		Level	
Pattern	$\|K\|$	median	s.d.	median	s.d.	median	s.d.
Single Symbol	10	116 127	63 470	116 127	63 470	1	—
	100	99 863	63 769	99 863	63 769	1	—
	1000	102 151	45 585	102 151	45 585	1	—
	10 000	93 390	52 321	93 390	52 321	1	—
	100 000	—	—	—	—	—	—
English Unsorted	10	15	2	94	9	7	4
	100	18	2	856	26	38	37
	1000	21	1	7348	91	182	347
	10 000	24	1	59 489	222	1001	2948
	100 000	25	0	415 520	415	3324	20 980
English Sorted	10	15	2	38	7	2	2
	100	17	2	336	35	12	20
	1000	21	2	3392	431	64	195
	10 000	24	1	34 496	1773	433	1795
	100 000	25	0	350 172	1994	3152	17 569

From the table it can be seen that the single symbol tries do not branch, but that they are typically very deep because of the large keywords. This is clearly seen from the fact that the depth and the number of states are the same. The natural language cases, however, are different in that they are not very deep—only up to 25 levels—but they are wide as can be seen from the large level sizes. The depth is bound by the length of the longest keyword, which is not very large for the English word list. Comparing the sorted and shuffled cases, one sees that the sorted case tries tend to be "narrower"—their levels have fewer elements.

Earlier it was reasoned that the sorted-keyword tries would have smaller levels earlier in the trie since there are more common prefixes in a sorted keyword set. To confirm this, the relative number of states per level was calculated. That is, for each level, the number of states in the level was divided by the total number of states in the trie, giving a fraction of how many of the trie's states are in a given level. The data is visualised in Figure 3.5. From the figure it can be seen that it is indeed the case that the tries for the non-sorted keywords branch out earlier than those of the sorted case, since the levels closer to the root contain a larger fraction of the tries' states. This difference is most pronounced for small keyword sets, but reduces as $|K|$ increases. Recall that the two sources from which keywords are selected are fixed at ≈200 000 words each. As the keyword sets grow larger, the probability of having many keywords in common between the sorted and unsorted cases increases. In the extreme case, where each of the fifty keyword sets contain approximately half of the source words, there is clearly very little difference between the sorted an unsorted cases.

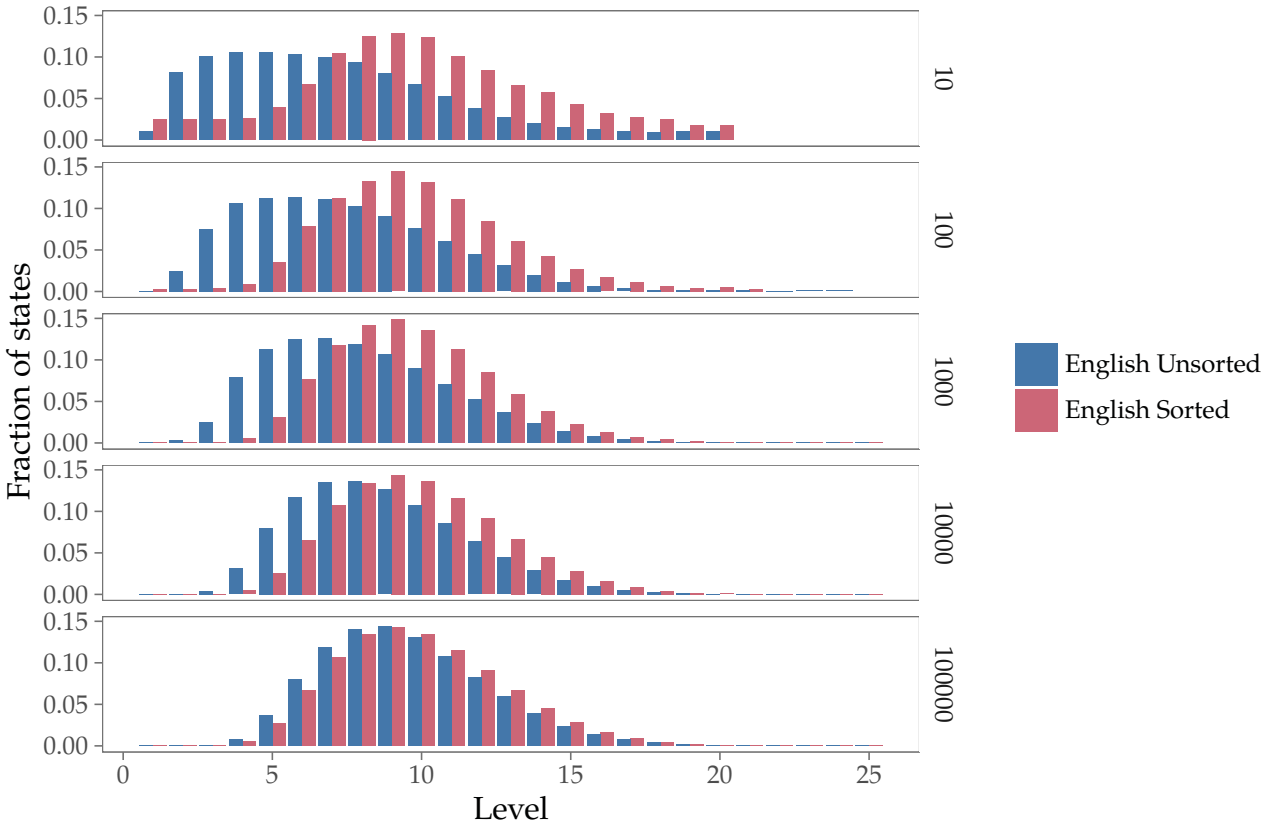


Figure 3.5: The relative number of states per trie depth level, grouped by text type and size of the keyword set.

Since the scope of the present work is to determine whether the concurrent implementations can construct the failure function faster than the se-

quential implementation, we do not consider the raw run times. Rather consider the speedup of the concurrent implementations relative to the sequential implementation. This speedup is defined as the execution time of the sequential algorithm divided by the execution time of the concurrent algorithm. Clearly a speedup greater than one is desired.

Table 3.2: Observed mean speedup for the different implementations grouped by text type and keyword set size. The table also shows the run-time of the sequential algorithm. Each entry is the mean over 50 observations.

Type	$\|K\|$	T_s (ms)	Variant 1	Variant 2	Variant 3	Variant 4
Single Symbol	All	67.83	0.38 ± 0.18	0.35 ± 0.19	0.32 ± 0.18	0.30 ± 0.18
	10	33.29	0.27 ± 0.01	0.24 ± 0.03	0.21 ± 0.02	0.20 ± 0.02
	100	31.75	0.28 ± 0.01	0.24 ± 0.04	0.21 ± 0.03	0.19 ± 0.03
	1000	33.82	0.28 ± 0.01	0.26 ± 0.01	0.23 ± 0.01	0.20 ± 0.01
	10 000	172.47	0.67 ± 0.12	0.66 ± 0.12	0.62 ± 0.10	0.59 ± 0.12
	100 000	—	—	—	—	—
English Unsorted	All	1699.18	0.18 ± 0.03	0.17 ± 0.03	0.16 ± 0.04	0.17 ± 0.03
	10	1.06	0.16 ± 0.00	0.12 ± 0.00	0.11 ± 0.00	0.12 ± 0.00
	100	9.49	0.14 ± 0.00	0.14 ± 0.00	0.13 ± 0.00	0.14 ± 0.00
	1000	102.49	0.17 ± 0.01	0.17 ± 0.01	0.17 ± 0.01	0.18 ± 0.01
	10 000	956.66	0.21 ± 0.00	0.20 ± 0.00	0.20 ± 0.00	0.20 ± 0.00
	100 000	7426.22	0.23 ± 0.00	0.20 ± 0.00	0.20 ± 0.00	0.20 ± 0.00
English Sorted	All	1193.87	0.18 ± 0.03	0.15 ± 0.03	0.15 ± 0.03	0.15 ± 0.03
	10	0.45	0.21 ± 0.03	0.10 ± 0.01	0.10 ± 0.01	0.10 ± 0.01
	100	3.73	0.14 ± 0.00	0.13 ± 0.00	0.13 ± 0.00	0.13 ± 0.00
	1000	45.76	0.16 ± 0.02	0.16 ± 0.02	0.16 ± 0.02	0.17 ± 0.02
	10 000	477.74	0.18 ± 0.00	0.17 ± 0.00	0.17 ± 0.00	0.17 ± 0.00
	100 000	5441.67	0.20 ± 0.00	0.18 ± 0.00	0.18 ± 0.00	0.18 ± 0.00

Table 3.2 shows the speedup obtained by the implementations. For each variant, the mean speedup is grouped by keyword type and size of the keyword set. Each entry in the table is the mean of fifty observations. To give an idea of absolute run-times, column T_s contains the run-times for the sequential algorithm.

It is clear from the table that none of the variants outperformed the sequential algorithm. Moreover, the concurrent implementations typically took 5 to 10 times longer than the sequential version. Clearly these results were not expected.

In order to understand better the reason for the poor performance, Go's built-in profiling tools were used to capture a CPU profile. When CPU profiling is enabled, the executing program is stopped about a 100 times per second and the program counters of all the goroutines are recorded. Inspecting the profile suggested that excessive channel communication may

be the reason for the concurrent variants' poor performance. Section 3.5.3 describes how this issue is addressed.

3.5.3 Reducing communication

Recall from Section 3.2.1 that a worker process dealing with state s finds $g(s,a)$ (the successor state of s on symbol a) and then communicates $g(s,a)$ on an *out* channel. It does this repeatedly for each $a \in \Sigma$ and consequently sends out $|\Sigma|$ messages on the *out* channel. A simple tactic to reduce communication would be to change a worker so that it does not communicate every single $g(s,a)$ directly on the channel, but stores these successor states in a collection instead, only communicating the entire collection on *out* when the worker has dealt with all $|\Sigma|$ symbols. This reduces the communication on the *out* channel by a factor of $|\Sigma|$. The process below captures the modified worker behaviour. R is used to collect all the $g(s,a)$ instances for a given state and then the process communicates the collection R on the *out* channel before terminating.

$$
\begin{aligned}
&WORKER_i(s) = P_i(\Sigma, s, \emptyset)\\
&P_i(S, s, R) =\\
&\quad \text{if } (S \neq \emptyset) \text{ then}\\
&\qquad \textstyle\bigsqcap_{a \in S} updateF.a.s \rightarrow P_i(S \setminus \{a\}, s, R \cup \{g(s,a)\})\\
&\quad \text{else}\\
&\qquad out.i!R \rightarrow SKIP
\end{aligned}
$$

Naturally the other processes also need to be modified in order to accept the collection and to process the elements. This means that *BUFF1* enqueues collections and that *result* transmits these collections. Additionally, *GATHERER* needs to unpack the collection and process the states individually. The *BUFF2* process and the *work* channel remain unchanged. These changes were implemented in Go and the modified Variant 1 becomes Variant 1a. The other variants were similarly modified and renamed. In the Go implementation R was implemented as an array of states and a reference to the array was communicated over the channel. Go code 3.7 shows the Go implementation of the modified worker process for Variant 1a.

Go code 3.7: Modified worker to reduce communication.

```
func worker(curState State, out chan<- []State) {
    calc := make([]State, len(this.sigma))
    i := 0
    for _, ch := range this.sigma {
        curDest, ok := this.gotoTable[deltaPair{curState, ch}]
        if ok {
            // Collect for processing as next level.
```

```
            calc[i] = curDest
            // Update failure function and goto function.
        ...
        } else {
            calc[i] = FAIL
        }
        i++
    }
    out <- calc
}
```

Table 3.3: Observed mean speedup for the modified variants where communication was reduced. The table is grouped by text type and size of the keyword set.

Type	$\|K\|$	T_s (ms)	Variant 1a	Variant 2a	Variant 3a	Variant 4a
Single	All	67.83	0.34 ± 0.16	0.29 ± 0.14	0.27 ± 0.14	0.25 ± 0.14
Symbol	10	33.29	0.25 ± 0.01	0.21 ± 0.05	0.18 ± 0.04	0.17 ± 0.03
	100	31.75	0.25 ± 0.01	0.21 ± 0.05	0.19 ± 0.04	0.17 ± 0.04
	1000	33.82	0.26 ± 0.01	0.23 ± 0.02	0.20 ± 0.01	0.18 ± 0.01
	10 000	172.47	0.60 ± 0.09	0.52 ± 0.07	0.49 ± 0.06	0.47 ± 0.06
	100 000	—	—	—	—	—
English	All	1699.18	3.78 ± 1.53	2.45 ± 1.75	2.44 ± 1.75	2.44 ± 1.74
Unsorted	10	1.06	1.27 ± 0.08	0.09 ± 0.01	0.09 ± 0.01	0.09 ± 0.01
	100	9.49	2.98 ± 0.16	0.65 ± 0.02	0.65 ± 0.02	0.65 ± 0.02
	1000	102.49	5.51 ± 0.10	3.29 ± 0.09	3.21 ± 0.38	3.27 ± 0.09
	10 000	956.66	4.99 ± 0.34	3.94 ± 0.16	3.91 ± 0.20	3.91 ± 0.18
	100 000	7426.22	4.17 ± 0.12	4.29 ± 0.07	4.34 ± 0.07	4.29 ± 0.05
English	All	1193.87	3.28 ± 1.48	1.92 ± 1.59	1.87 ± 1.63	1.87 ± 1.60
Sorted	10	0.45	1.03 ± 0.09	0.04 ± 0.01	0.04 ± 0.01	0.04 ± 0.01
	100	3.73	2.17 ± 0.19	0.23 ± 0.09	0.25 ± 0.08	0.25 ± 0.08
	1000	45.76	4.93 ± 0.18	2.11 ± 0.31	1.75 ± 0.66	1.84 ± 0.58
	10 000	477.74	4.32 ± 0.51	3.18 ± 0.14	3.23 ± 0.13	3.17 ± 0.14
	100 000	5441.67	3.93 ± 0.12	4.04 ± 0.06	4.07 ± 0.06	4.05 ± 0.06

The earlier experiments were repeated and the new speedup numbers can be found in the relevant columns in Table 3.3. Figure 3.6 plots the speedup against keyword set size, grouped by pattern set type. Note that the data set labelled "Split" will be discussed in Section 3.5.4.

Here are the main observations that can be made about these new measurements.

1. In the single symbol case the buffer modification did not improve the speedup. This is to be expected, since the alphabet in this case has only two symbols, and the modification was essentially aimed at

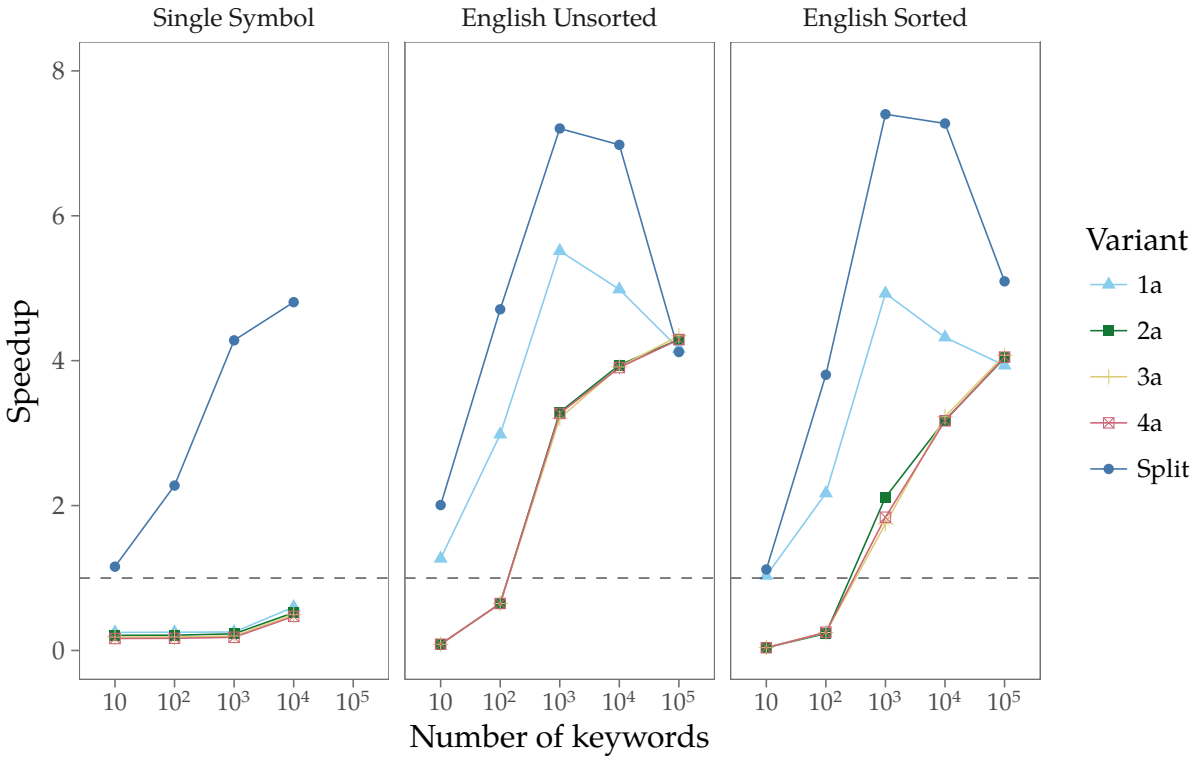


Figure 3.6: Observed speedup per keyword set size grouped by text type for the reduced communication. Each point is the mean over 50 observations.

ameliorating the effect of a large number of channel messages generated by a large number of alphabet symbols being processed per state. Additionally, the trie only has a single node per level since all the keywords comprise only a single repeated character. Consequently this scenario offers no opportunities for concurrency but incurs all the overhead costs of setting up and taking down processes. The observations that follow do not refer to the single symbol case.

2. In the other pattern types with a large alphabet and where the underlying trie could therefore contain many states per level, one does indeed observe speedup, sometimes even coming close to the number of physical cores, six.

3. For all variants, speedup is low for small keyword sets and then improves as keyword set size increases. This supports the findings seen in the single symbol keyword case: that a certain threshold of concurrency opportunities is needed to amortise the initial overhead cost of supporting concurrency. Nevertheless, Variant 1a manages a speedup of more than one for the smallest keyword set size of ten. On the other hand, Variant 2a to 4a achieve speedups greater than one only with a thousand or more keywords.

4. Across all scenarios, speedup increases monotonically as keyword set size increases, with one notable exception: the speedup attained by Variant 1a peaks at a thousand keywords and then degenerates. It starts off as being the best of the four variants but becomes the worst for the largest keyword set. It would seem that eventually the cost of setting up and taking down processes accumulates, thereby reducing performance.
5. Since the trie for the unsorted English keywords is typically wider than that for the sorted keywords, the potential for concurrency in the former case is greater. This is reflected throughout in the speedup data: the unsorted speedups are consistently greater than those in the sorted case. The relative impact is greatest on small keyword set sizes.
6. Buffer sizes could undoubtedly influence speedup. Not only could delays (and perhaps even livelock or deadlock) arise because processes have to wait for full buffers to have contents consumed by other processes; cache-related delays could also make an impact. In this study, buffers were selected to be sufficiently large to not only guarantee deadlock freedom, but also to eliminate the possibility of delays caused by full buffers. However, an exploration of the effect of the size of buffers on speedup was beyond the scope of the present study.

3.5.4 The data-driven solution

In order to compare this process-based approach to a data-driven approach that is built directly on the sequential solution, the following simple strategy was also implemented. The keyword set was split into m number of subsets, where $m = 5$ for the keyword set size of 10, and $m = 12$ otherwise. Each subset was then used as a keyword set to construct an AC string matching automaton. The idea is to have m smaller AC automata processing the text concurrently. The regular sequential algorithm was used to construct concurrently these smaller machines.

The number was chosen as it equals the number of virtual cores of the platform. This was deemed to be enough to keep the cores busy.

In this case it is not feasible to capture the failure function construction times alone, so the total construction time was recorded instead—building the trie and constructing the failure function—of the m automata and used this with the total construction time for the regular (non-split) sequential algorithm to compute speedup. The resulting speedups are shown as "Split" in Figure 3.6. The split performance follows a similar curve to Variant 1a, but at an improved speedup level.

The speedup increases beyond six (the number of physical cores) in four of the experiments could be associated with the hyperthreading implemen-

ted by Intel® on the CPU or perhaps it could be due to favourable caching. However, the details of exactly how this occurs have not been pursued in this study.

3.6 Conclusion and Future Work

This case study exploited opportunities for concurrency that are not quite as obvious as merely splitting up data to run multiple instances of a given algorithm. Four ways in which to do this for the problem at hand were tested. This enabled one to exploit the concurrency potential available on multicore platforms that typically remain unused for conventional sequential solutions. In general, the speedup gain was highly significant for larger data sets, but performance degenerates significantly for very small data sets and collapses for degenerate examples such as a set of single-symbol keywords. Moreover, it has been shown that for larger alphabets and keyword set sizes, speedups are comparable to those obtained by following a more traditional approach to obtaining concurrency, namely by partitioning the input data. Indeed there is even a slight hint in the speedup measurements that process-based solutions may eventually outperform the traditional approach for data sets larger than those used to date.

Additional work regarding the present case study may include

- studying the effect of buffer size on performance and determining heuristics for optimising buffer size based on trie structure;
- finding ways in which to reduce communication that does not depend on the alphabet, allowing for performance gains in the single symbol case;
- determining whether the speedups of Variants 2a and 3a decline when $|K| > 10^5$; and
- experimenting with other aspects of the AC algorithm such as constructing the trie and failure function concurrently.

Of course, there are numerous alternatives to- and refactorings of the three architectural variants proposed here. One suggestion is to consider whether it is possible and effective to move beyond the breadth-first traversal of the pre-constructed trie implicit in the sequential algorithm. Recall that the sequential algorithm of Algorithm 3.2 requires that the trie levels have to be processed in sequential order and all concurrent variants to date have respected this requirement. The requirement ensures that when the assignment $q := f(r)$ in the sequential code is executed, the value

of $f(r)$ is available from an earlier level traversal, as required by the invariant of the loop. It may be possible to ignore the constraints of a breadth-first sweep in a refactored concurrent implementation, provided that the assignment $q := f(r)$ in the code is blocked until $f(r)$ is indeed defined. The details of how this might be done and whether it would result in significant speedup is left for future research.

4 Brzozowski's DFA construction

> Increasingly, people seem to misinterpret complexity as sophistication, which is baffling—the incomprehensible should cause suspicion rather than admiration. Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user.
>
> *Niklaus Wirth*

The second case study focuses on the problem of developing concurrent versions of Brzozowski's (1964) algorithm for constructing a deterministic finite automaton (DFA) from an arbitrary regular expression. The language of the resulting DFA is, of course, the same as that represented by the regular expression.

The chapter starts with a discussion of Brzozowski's classical sequential DFA construction algorithm. Section 4.2 then presents two process-based decompositions of the algorithm that are suitable for concurrent execution. This is followed by a performance comparison of a number of implementations based on these decompositions in Section 4.4.

4.1 Sequential algorithm

The DFA construction algorithm by Brzozowski (1964) employs derivatives of regular expressions to construct a DFA. The algorithm takes a regular expression E as input and constructs a DFA M such that $\mathcal{L}(M) = \mathcal{L}(E)$.

Regular expressions and related concepts are defined in Section 2.2. DFAs are defined in Def. 2.28.

The algorithm identifies with each DFA state a regular expression. Elements of M may therefore interchangeably be referred to either as regular expressions or as states, depending on the context of the discussion. The start state corresponds to the input regular expression, E. Each remaining state is identified with a regular expression, say d, such that if $\delta(a, q) = d$, then d corresponds with the derivative $a^{-1}q$. In fact, it can be shown that the language of each state's associated regular expression is also the right language of that state.

The rules for finding the derivative of a regular expression may be found in Def. 2.25.

Algorithm 4.1 (Brzozowski's DFA construction algorithm):

```
func BrzS(E, Σ) →
    δ, s, F := ∅, E, ∅;
    D, T := ∅, s;
    do (T ≠ ∅) →
        let q be some state such that q ∈ T;
        D, T := D ∪ {q}, T \ {q};
        { build out-transitions from q on all alphabet symbols }
        for (a : Σ) →
            { find derivative of q with respect to a }
            d := a⁻¹q;
            if d ∉ (D ∪ T) → T := T ∪ {d}
            [] d ∈ (D ∪ T) → skip
            fi;
            { make a transition from q to d on a }
            δ(q, a) := d
        rof;
        if ε ∈ ℒ(q) → F := F ∪ {q}
        [] ε ∉ ℒ(q) → skip
        fi
    od;
    return (D, Σ, δ, s, F)
cnuf
```

□

The well-known sequential version of the algorithm is given in Dijkstra's GCL in Algorithm 4.1. The notation assumes that the set operations ensure uniqueness of the elements at the level of regular expression equivalence (Def. 2.26), that is, $a \in A$ implies that there is no $b \in A$ such that a and b are equivalent regular expressions.

As in the previous chapter, GCL is employed as notation for the sequential algorithms. An overview may be found in Section 2.4.

The algorithm maintains two sets of regular expressions (or states): a set T ('to do') containing the regular expressions for which derivatives need to be calculated; and another set D ('done') containing the regular expressions for which derivatives have been found already. When the algorithm terminates, T is empty and D contains the states of the automaton that recognises $\mathcal{L}(E)$.

The algorithm iterates through all the elements $q \in T$, finding derivatives with respect to all the alphabet symbols and depositing these new states (regular expressions) into T in those cases where no equivalent regular expression has already been deposited into $T \cup D$.

Each q, once processed in this fashion, is then removed from T and added into D.

In each iteration of the inner **for**-loop (i.e. for each alphabet symbol), the δ function is updated to contain the mapping from state q to its derivative with respect to the relevant alphabet symbol.

Finally, if state q represents a regular expression whose language contains the empty string ε, that state is included in the set of final states F.

Since computing equivalence of regular expressions is expensive (Aho, Hopcroft and Ullman 1974), in practice the set membership tests $d \in (D \cup T)$ and $d \notin (D \cup T)$ in the algorithm may use a weaker notion of equivalence—that of *similarity* as defined in Definition 2.27. Two regular expressions are similar if they are identical, or if one can be turned into the other using the rules given in the definition. Brzozowski (1964) shows that every regular expression has only a finite number of dissimilar derivatives. Hence, the algorithm above is guaranteed to terminate when similarity is used as an approximation for equivalence. Furthermore, since two dissimilar regular expressions may be equivalent, the resulting DFA is not guaranteed to be minimal.

The forthcoming section presents two process-based decompositions of the algorithm. In each case the CSP description is structured as a number of communicating sequential processes that may be executed concurrently.

4.2 Concurrent DFA construction

When one considers the sequential algorithm shown in Algorithm 4.1, a number of independent activities can be identified: derivatives of regular expressions are computed, the transition function δ is populated, finality is determined, and the sets T and D are updated until no more regular expressions need to be processed. These activities can be modelled as processes in CSP.

The next section describes two alternative ways in which these processes may be arranged. The main difference between the two configurations is in the way in which the computation of derivatives is split among processes. In the first configuration, a single CSP process is associated with a symbol a_i from the alphabet Σ. This process then computes derivates of regular expressions with respect to this symbol. Consequently one requires $|\Sigma|$ such processes to compute derivatives over the entire alphabet. In the second configuration, a single CSP process computes the derivatives of regular expressions with respect to every $a_i \in \Sigma$. An arbitrary number of such processes may be executed concurrently. This number is determined by the implementer.

4.2.1 Configuration A

The first obvious process to model is that of computing derivatives of regular expressions. Let $DERIV_{a_i}$ be the process that calculates the derivative of a regular expression with respect to a symbol a_i from the alphabet Σ. Each $DERIV_{a_i}$ repeatedly reads a regular expression re from its input channel $in.i$, calculates the derivative with respect to a_i and then sends out the result as a triple $\langle re, a_i, a_i^{-1}re\rangle$ on channel $out.i$.

$$DERIV_{a_i} = in.i?re \rightarrow out.i!\langle re, a_i, a_i^{-1}re\rangle \rightarrow DERIV_{a_i}$$

These processes may execute independently and concurrently and are thus modelled in process $DERIVS$ as the interleaving of $|\Sigma|$ processes. Additionally, $DERIVS$ multiplexes the output of the individual $DERIV_{a_i}$ processes onto a single output channel. This multiplexing is achieved by composing the $DERIV_{a_i}$ processes with $MBUFF$ from Section 3.2.1. Recall that $MBUFF$ accepts input from an array of $left.i$ channels and outputs the input in FIFO fashion on a single output channel $right$.

$$DERIVS = \left(|||_{i:1..|\Sigma|} DERIV_{a_i}\right)[out \leftrightarrow left]\, MBUFF(\langle\,\rangle, N_1)$$

$DERIVS$ is thus modelled as the interleaving of $|\Sigma|$ processes, each responsible for calculating the derivative with respect to a given $a_i \in \Sigma$. Each $DERIVS$ output channel $out.i$ is chained to a corresponding input channel $left.i$ of $MBUFF$.

The next task to model is updating the transition function. This is the responsibility of $UPDATED$. It is modelled as a repeating process that reads a triple $\langle re, a, d\rangle$ from its input channel in and records $\delta(re, a) = d$. It also sends one element of the triple, d, on via channel out. This d is potentially a new state from which transitions should be calculated and hence it should be added into T if a similar node has not been processed before.

$$UPDATED(\delta) = in?\langle re, a, d\rangle \rightarrow out!d \rightarrow UPDATED(\delta \cup \{\langle re, a, d\rangle\})$$

The actions performed in the outer loop of the pseudo-code in Algorithm 4.1 are modelled as process $OUTER$. It is responsible for maintaining the sets D, T, and F. It sends out regular expressions for which derivatives are computed and it receives these new regular expressions. The parameter p is used for termination as explained shortly.

$$\begin{aligned}
&OUTER(T, D, F, p) = \\
&\quad \text{if } |T| > 0 \vee p > 0 \text{ then} \\
&\qquad SENDER(T, D, F, p) \,\square\, RECEIVER(T, D, F, p) \\
&\quad \text{else} \\
&\qquad SKIP
\end{aligned}$$

- In the sequential algorithm, the outer loop continues while T is not empty. In the concurrent version, however, the termination condition needs to be modified. The process should not terminate immediately when T is empty, since it may happen that the last element of T has been sent out by *SENDER*, but that some regular expression is still to be received by *RECEIVER*. For this reason p is used to model the number of pending results. Every time a regular expression is sent out, p is incremented and every time a regular expression is received, p is decremented. When both T is empty and $p = 0$, then the process terminates.
- While *OUTER* has not terminated it is modelled as a choice between the following behaviours. It may choose some $q \in T$ and write it to a channel (*SENDER*) and repeat the choice, or it may receive a new regular expression d from its input channel (*RECEIVER*) and repeat the choice. The details of these two processes are described next.

It seems sensible to let *OUTER* and *SENDER* run concurrently using $|||$. However, in the present model they both manipulate T and p. Consequently, they cannot run concurrently by simply changing $\Box$ into $|||$. A larger refactoring is required and left as future work.

RECEIVER describes the part of *OUTER* that receives the derivatives and updates T as necessary. The process receives a regular expression d on channel *results* and then deposits d in T if no similar regular expression is found in $T \cup D$. The process also decrements p since there is one fewer derivative pending.

$$
\begin{array}{l}
RECEIVER(T, D, F, p) = \\
\quad results?d \rightarrow \\
\qquad \text{if } d \notin T \cup D \text{ then} \\
\qquad\quad OUTER(T \cup \{d\}, D, F, p - 1) \\
\qquad \text{else} \\
\qquad\quad OUTER(T, D, F, p - 1)
\end{array}
$$

The part of *OUTER* that sends out regular expressions is modelled as process *SENDER*. It repeatedly sends out an arbitrary element of T on channel *deriv*. This arbitrary choice is modelled as the internal choice over $|T|$ subprocesses, one for each element of T. When *OUTER* sends out q, q is removed from T and added to D and, since $|\Sigma|$ more derivatives need to be computed, p is incremented by $|\Sigma|$. Further, if $\varepsilon \in \mathcal{L}(q)$, then q is added into F.

$$
\begin{array}{l}
SENDER(T, D, F, p) = \\
\quad \bigsqcap_{q \in T} (deriv!q \rightarrow \\
\qquad\quad \text{if } \varepsilon \in \mathcal{L}(q) \text{ then} \\
\qquad\qquad OUTER(T \setminus \{q\}, D \cup \{q\}, F \cup \{q\}, p + |\Sigma|) \\
\qquad\quad \text{else} \\
\qquad\qquad OUTER(T \setminus \{q\}, D \cup \{q\}, F, p + |\Sigma|))
\end{array}
$$

All the actions of the sequential algorithm are now described by CSP processes. However, since *SENDER* sends out regular expressions to be processed by *DERIVS*, an additional process is needed to distribute the single regular expression to each of the $DERIV_{a_i}$ processes. This process is called *FANOUT*. It repeatedly reads a regular expression from its input channel *deriv* and concurrently replicates it to the $|\Sigma|$ output channels $out.i$.

$$FANOUT = (deriv?re \rightarrow |||_{i:\Sigma}(out.i!re \rightarrow SKIP)); FANOUT$$

When *FANOUT* is composed below with *DERIVS*, one obtains a process with a single input channel and a single output channel.

$$DERIVERS = FANOUT\,[out \leftrightarrow in]\,DERIVS$$

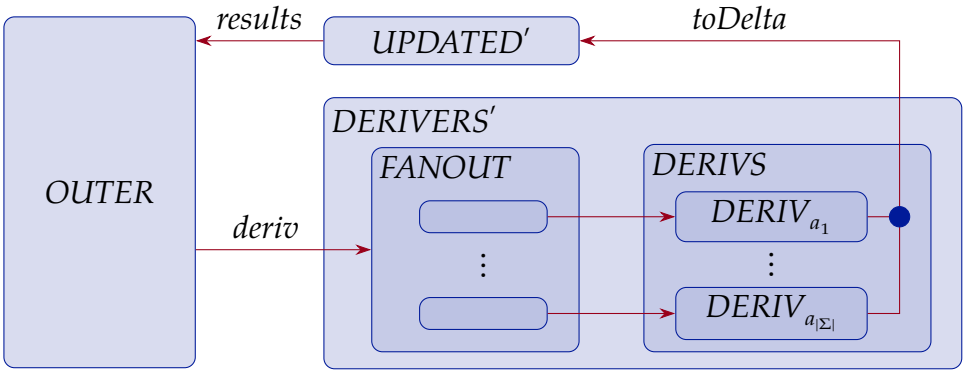


Figure 4.1: The communications network of the BRZ_A process. Note that there is a $DERIV_{a_i}$ process for each $a_i \in \Sigma$. A single regular expression is sent to all $DERIV_{a_i}$ for processing. The *MBUFF* process is represented by ●.

The final step is to get the processes to interact by connecting them as shown in Figure 4.1. Since interaction takes place through shared channels, it means that one needs to rename certain channels to be common among the interacting processes. First let $UPDATED'(\delta) = UPDATED(\delta)[\![^{toDelta, results}/_{in, out}]\!]$. Here the original *in* and *out* channels are renamed to *toDelta* and *results*, respectively. Second, let $DERIVERS' = DERIVERS[\![^{toDelta}/_{right}]\!]$. In this case channel *right* is renamed *toDelta*.

The concurrent version of the algorithm can be modelled as a process BRZ_A that is the parallel composition of the previously defined processes. The processes only interact through shared channel events as shown below.

$$BRZ_A(E) = OUTER(\{E\}, \emptyset, \emptyset, 0) \underset{\{\!|deriv, results|\!\}}{\|} (DERIVERS' \underset{\{\!|toDelta|\!\}}{\|} UPDATED'(\emptyset))$$

The regular expression E for which a DFA is to be constructed, is passed as input parameter to the process. Note the modelling decision of not passing in Σ as a parameter. Rather, it is seen as a global constant known by the relevant processes. The initial values of the process paramaters, correspond with the initialisation in Algorithm 4.1. *OUTER*'s T consists only of E, the first regular expression for which to find derivatives. D and F are both empty and $p = 0$. The transition function δ is also empty.

One could, of course, make Σ a parameter, but that would increase the number of parameters in the process descriptions, making them slightly more complex.

Before considering the implementation and performance of the present process-based decomposition, a second, alternative, decomposition is presented next.

4.2.2 Configuration B

The *FANOUT* process of the previous section sends out from T one single regular expression, q, at a time to $|\Sigma|$ processes. These processes concurrently compute the derivatives of q, each process dealing with one specific symbol from Σ. Only when all of these $|\Sigma|$ processes have run to completion can *FANOUT* send out the next regular expression. This seems unnecessarily restrictive.

Instead, define a second process configuration in which the derivatives of an arbitrarily predetermined number of regular expressions, n, can be computed in n separate processes, each process finding the derivatives of its current regular expression with respect to *all* $a \in \Sigma$. This configuration thus relies on the implementer preselecting n, the number of separate processes to be run. One may define $DERIVS(n)$ as the concurrent composition of n independent $DERIV_i$ processes.

$$DERIVS(n) = |||_{i=1..n} DERIV_i$$

Each of these $DERIV_i$ processes reads a regular expression from an input channel $in.i$ and then computes the derivative with respect to each $a \in \Sigma$. Once all these derivatives have been computed and communicated, the process repeats. Another minor change to the process is that it now has two output channels: $outd.i$ and $out.i$. The process sends out triples to populate the transition function on $outd.i$ and $out.i$ is used to send out the potentially new regular expressions back to *OUTER*.

$$DERIV_i = in.i?re \rightarrow (P_i(\Sigma, re); DERIV_i)$$
$$P_i(S, re) =$$
$$\quad \text{if } S \neq \emptyset \text{ then}$$
$$\quad\quad \Box_{a \in S}\, outd.i!\langle re, a, a^{-1}re\rangle \rightarrow out.i!a^{-1}re \rightarrow P_i(S \setminus \{a\}, re)$$
$$\quad \text{else}$$
$$\quad\quad SKIP$$

Other minor modifications are also present in this configuration. The first is to remove updating the set of final states F from *OUTER* and to create a new process *UPDATEF* that is responsible for this. This change allows for the nullability checking to occur concurrently with the other operations. *UPDATEF* reads a regular expression s from an input channel *toFinal* and then inserts s into F when $\varepsilon \in \mathcal{L}(s)$.

$$
\begin{aligned}
&UPDATEF(F) = \\
&\quad toFinal?s \rightarrow \\
&\qquad \text{if } \varepsilon \in \mathcal{L}(s) \text{ then} \\
&\qquad\quad UPDATEF(F \cup \{s\}) \\
&\qquad \text{else} \\
&\qquad\quad UPDATEF(F)
\end{aligned}
$$

The addition of *UPDATEF* necessitates a revision of the *OUTER* process. The references to F should be removed meaning that F should no longer be a parameter of *OUTER* and its subprocesses, *SENDER* and *RECEIVER* and *OUTER* should communicate regular expressions to *UPDATEF*. *OUTER* is still described as the choice between *SENDER* and *RECEIVER*.

$$
\begin{aligned}
&OUTER(T, D, p) = \\
&\quad \text{if } |T| > 0 \vee p > 0 \text{ then} \\
&\qquad SENDER(T, D, p) \,\square\, RECEIVER(T, D, p) \\
&\quad \text{else} \\
&\qquad SKIP
\end{aligned}
$$

The behaviour of *RECEIVER* remains unchanged as shown below.

$$
\begin{aligned}
&RECEIVER(T, D, p) = \\
&\quad results?d \rightarrow \\
&\qquad \text{if } d \notin T \cup D \text{ then} \\
&\qquad\quad OUTER(T \cup \{d\}, D, p - 1) \\
&\qquad \text{else} \\
&\qquad\quad OUTER(T, D, p - 1)
\end{aligned}
$$

SENDER's behaviour, however, does change because it does not need to determine whether $\varepsilon \in \mathcal{L}(q)$ for some $q \in T$. It only needs to send q out to *UPDATEF* on channel *toFinal*.

$$
\begin{aligned}
&SENDER(T, D, p) = \\
&\quad \bigsqcap_{q \in T} (toFinal!q \rightarrow deriv!q \rightarrow OUTER(T \setminus \{q\}, D \cup \{q\}, p + |\Sigma|))
\end{aligned}
$$

Since each $DERIV_i$ process communicates the derivatives directly to *OUTER*, *UPDATED* does not need to do that anymore. Hence it repeatedly reads a triple from its input channel and updates the transition function.

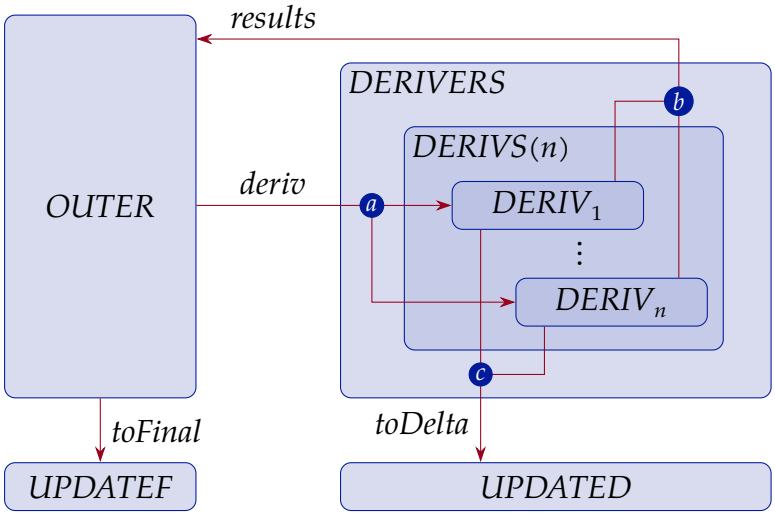


Figure 4.2: Configuration B process diagram. Note that the number of *DERIV* processes is determined by the implementation and not by $|\Sigma|$.

$$UPDATED(\delta) = toDelta?\langle re, a, d\rangle \rightarrow UPDATED(\delta \cup \{\langle re, a, d\rangle\})$$

Now the processes can be connected as shown in Figure 4.2. Since each $DERIV_i$ has an input and two output channels, $DERIVS(n)$ has $3n$ channels that need connecting. In order to simplify this, $DERIVS(n)$ is composed with instances of *DBUFF* and *MBUFF*. Recall from Section 2.5.2 that these processes provide not only buffering, but also a way to link a single channel to multiple channels. The CSP description of the composite process is shown next. An explanation follows the definition.

$$\begin{aligned}
&DERIVERS(n) = \\
&\quad (a.DBUFF(\langle\,\rangle, N_a, n) \\
&\qquad [a.right \leftrightarrow in] \\
&\quad DERIVS(n) \\
&\qquad [out \leftrightarrow b.left] \\
&\quad b.MBUFF(\langle\,\rangle, N_b, n) \\
&\qquad [outd \leftrightarrow c.left] \\
&\quad c.MBUFF(\langle\,\rangle, N_c, n))[\![^{deriv, results, toDelta}/_{a.left, b.right, c.right}]\!]
\end{aligned}$$

The first step in connecting the processes is to connect all the input channels of the $DERIV_i$ processes to an instance of *DBUFF*. This *a.DBUFF* receives elements on a single input channel *a.left* and emits an element on any *one* of an array of output channels *a.right.i*. In the present case *a.DBUFF* is used to move a regular expressions from *OUTER* to *one* of the $DERIV_i$ processes. Contrast this with *FANOUT* from Configuration A in which a given regular expression is delivered to *all* the $DERIV_i$ processes.

The first parameter is the state of the buffer, initially empty. The second is the capacity of the buffer, while the third is the number of output channels.

Next, the output channels of the $DERIV_i$ processes are multiplexed onto two channels using two instances of *MBUFF*. The first, *b.MBUFF*, aggregates the elements from all the *outd.i* channels to deliver to *UPDATED*. The second, *c.MBUFF*, does the same for the *out.i* channels and connects to *OUTER*.

Now, rename channels so that *DERIVERS* may interact with *OUTER* and *UPDATED* via common channel names. *a.DBUFF*'s input channel *a.left* is renamed to *deriv*, the output channel of *b.MBUFF* is renamed from *b.right* to *results*, and *c.MBUFF*'s output channel is renamed from *c.right* to *toDelta*.

The final step is to define BRZ_B in which all the processes are interacting via the channels as shown in Figure 4.2.

$$BRZ_B(E,n) = \\ (UPDATEF(\emptyset) \underset{\{|toFinal|\}}{\|} OUTER(\{E\},\emptyset)) \underset{\{|results,deriv|\}}{\|} (DERIVERS(n) \underset{\{|toDelta|\}}{\|} UPDATED(\emptyset))$$

These two process-based decompositions can now be used to guide a number of implementations.

4.3 Implementation

Regular expressions are implemented as expression trees, following the same approach as B. W. Watson (1994). A total ordering is defined over the regular expressions and, to assist in similarity checking, the expressions are kept in a canonical form. To test whether two regular expressions are similar, one may then simply consider their ordering. If they are equal then they are similar. Full details are not included here, but the source code will be made available at `http://fastar.org`.

The two CSP configurations from the previous section were used to guide a number of Go implementations. As before, processes are mapped to goroutines. The synchronous channels of CSP are mapped to Go channels. However, in Go, channels provide buffering and may be shared among goroutines. This eliminates the need for buffering and multiplexing processes (e.g. *MBUFF*, *DBUFF*) in the implementations.

In addition to the sequential version of the algorithm, five process-based variants of the algorithm were implemented. Three (BrzA1, BrzA2 and BrzA3) are based on Configuration A and two (BrzB1 and BrzB2) are based on Configuration B. The sequential version and variants may be described briefly as follows.

BrzS — A direct sequential implementation of Algorithm 4.1. This implementation serves as the baseline against which the concurrent implementations are measured. In this implementation, as in the others, no attempt was made to optimise performance. ‘

BrzA1 — An implementation based on configuration A in which channels are synchronous, that is the channels do not provide buffering.

BrzA2 — This implementation is the same as BrzA1, but this time the channels are buffered. The buffer sizes used in the performance comparison may be found in Table 4.1.

BrzA3 — This is the final implementation based on configuration A. However, a small modification is made that should potentially reduce channel communication. The $DERIV_{a_i}$ processes send out results only when $a_i^{-1}q \neq \emptyset$. This means that *OUTER* cannot anymore rely on counting the derivatives to decide when to terminate. Instead an alternative scheme is employed.

When *OUTER* has no more regular expressions to send, i.e. $T = \emptyset$, then it may terminate only if there are no more regular expressions on their way from the $DERIV_{a_i}$ processes. Since regular expressions are processed by the $DERIV_{a_i}$ processes in the order in which they are received, the following scheme may be adopted to ensure that *OUTER* does not terminate while there are states “in transit”.

When $T = \emptyset$, then *OUTER* sends out a special token on channel *deriv* to the $DERIV_{a_i}$ processes. Upon receiving the token, each of them simply sends the token on. Once $|\Sigma|$ tokens, one for each $DERIV_{a_i}$ process, have arrived consecutively back at *OUTER*, it may terminate. If, however, a regular expression arrives at *OUTER* before all $|\Sigma|$ tokens were received, *OUTER* aborts the termination attempt by resetting the token counter and inserts the regular expression into T.

BrzB1 — An implementation of Configuration B, with buffers sizes as shown in Table 4.1.

BrzB2 — An alternate Configuration B implementation in which communication is reduced by changing the way in which the $DERIV_i$ processes emit their results. Recall that each process computed the derivative over all alphabet symbols. Instead of communicating each such derivative, a $DERIV_i$ collects all $|\Sigma|$ derivatives and communicates this collection over its output channel.

The next section reports on the performance of these implementations.

4.4 Performance comparison

One of the objectives of the present study is to determine whether an implementation of a process-based decomposition may yield improvements in run time over the sequential algorithm. For this purpose the following experiments were conducted.

4.4.1 Experimental setup

Regular expressions were pseudo-randomly generated via a simple recursive procedure $gen(\Sigma, d)$. The procedure takes as input two parameters: an alphabet Σ and an integer d. If $d = 0$ the procedure returns a random symbol from Σ. If $d > 0$ then $gen(\Sigma, d)$ randomly chooses a regular expression operator and then recursively generates the required operands for the operator by calling $gen(\Sigma, d-1)$. The size of the regular expression is thus controlled by d since d defines the depth of the expression tree for the regular expression. The upper bound for the number of operators in the tree is $2^d - 1$ and for the number of symbols (leaves) it is 2^d. Many generated regular expressions will be smaller since some regular expression operators are unary operators which result in a tree that is smaller than a complete binary tree. It should be mentioned that more flexible generation is possible using the method by Héam and Nicaud (2011).

Regular expression were generated with depths $d = 5, 6, \ldots, 10$. In order to cover a wide range of alphabet sizes, alphabets of sizes, 4, 95, 256 and 512 were considered. For each of the elements in $\{4, 95, 256, 512\} \times \{5, 6, \ldots, 10\}$ a hundred regular expressions were generated. Each of these regular expressions was used as input to each of the six implementations described above. A given regular expression and implementation combination was executed twenty times and the minimum run time was taken as the data point for that particular combination. The reason for this is that very large run times occasionally were observed for the sequential implementation. However, when repeating the runs, these cases would record reasonable run times, while others would suddenly run very long. This suggests that the long run times are not due to the particular input test cases, but rather due to other factors. These factors could be, *inter alia*, operating system effects and the Go garbage collector. Instead of trying to identify and, if possible, remove the factors, it was decided to mitigate the effect of these by repeating each input twenty times. In each execution the number of states in the generated automaton was also recorded.

Go 1.5.3 was used to compile the implementations. They were executed on the same Linux 3.10.17 machine with a six-core hyperthreaded CPU as in the previous chapter. The Go runtime was configured to run twelve

Intel® Xeon® CPU E5-2630 v2 @ 2.60 GHz. See Section 2.7.3 for more information.

processes concurrently and the number of $DERIV_i$ processes in Configuration B was 24. The buffer sizes are listed in Table 4.1. The sizes of the buffers were chosen to allow the goroutines to communicate in an asynchronous fashion without using very large buffers. The approach was to have enough buffer space to allow for a message per process sharing the buffer.

Table 4.1: Buffer sizes for the different channels. Note that $n = 24$ is the number of $DERIV_i$ processes in Configuration B. A dash implies that the channel is not present in the implementation.

Channel	BrzA1	BrzA2	BrzA3	BrzB1	BrzB2
outNode	0	1	1	$2n$	$2n$
inNode	0	$\|\Sigma\|$	$\|\Sigma\|$	$\|\Sigma\|n$	n
toDelta/deriv	0	$\|\Sigma\|$	$\|\Sigma\|$	$\|\Sigma\|n$	n
outf.i	0	1	5	—	—
tofinal	—	—	—	$2n$	$2n$

4.4.2 Observations

To gain insight in the size of the output, first consider the sizes of the DFAs constructed by the algorithm. Figure 4.3 contains box plots of the number of states per automaton for each of the different alphabet sizes. It is clear that the average size of the automata increases as the depth of the regular expressions increases. This is expected since a greater depth means that the tree representing the regular expression is larger, potentially yielding more states in the DFA. Furthermore, it can be seen that the four-symbol alphabet regular expressions resulted in automata with the most states. The exact reason for this has not been investigated because it is not of immediate concern to the objectives of this study.

It is clear from the plots that the automata are fairly small, since most have fewer than fifty states. It is only the four-alphabet regular expressions that yielded automata with more than a hundred states.

Recall that for a given regular expression, each of the implementations was used to construct the corresponding DFA. Hence each regular expression yields six data points—one per implementation. In order to visually assess the performance of a concurrent implementation against the sequential benchmark implementation, one may plot a point for each pair of concurrent and sequential observations. Let the x-coordinate of the point be the sequential run time and let the y-coordinate be the concurrent run time. If the run times are always equal, the points should form a line with slope of one. If, however, the concurrent run times are smaller than the sequential

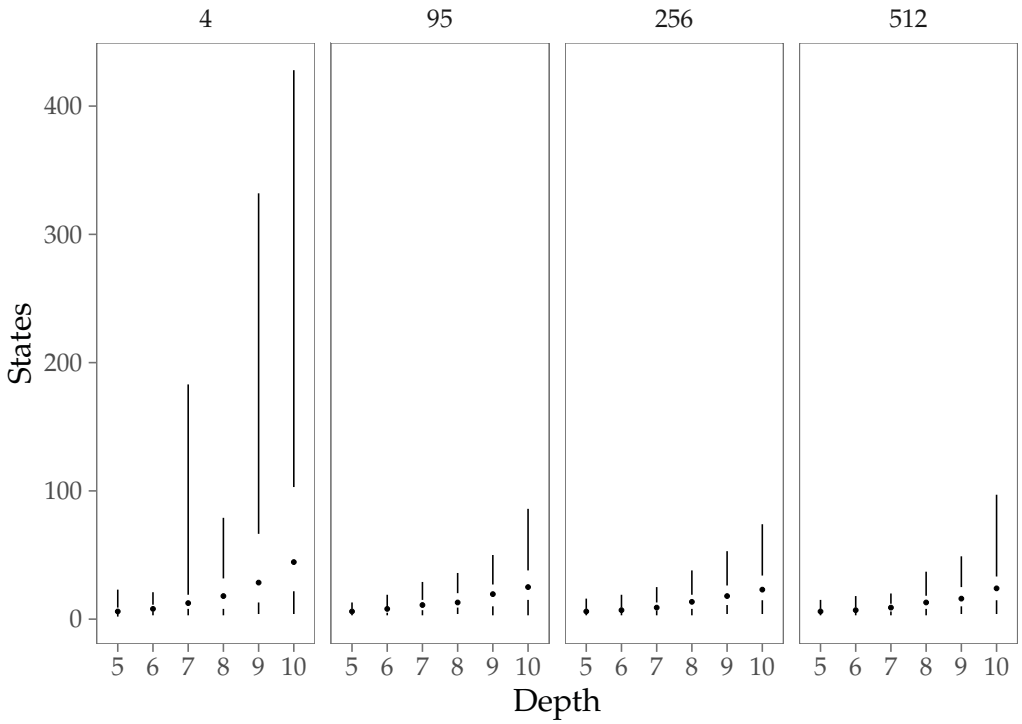


Figure 4.3: Box plots of the number of states per DFA against the regular expression depth, for each of the four alphabet sizes.

run times, the points will be below this line and, conversely, if the concurrent run times are greater than the sequential run times, the points will lie above this line.

Figure 4.4 shows the paired observations for each the concurrent implementations. Each plot shows the observations for all the alphabet sizes and depths: $4 \times 6 \times 100 = 2400$ points. The plots confirm that the concurrent run times tend to be lower than the corresponding sequential run times since the y-coordinates of the points tend the be smaller than the x-coordinates. Also there seems to exist a linear relationship between the concurrent and sequential times.

To verify the linear relationship, a regression line was fitted to each of the groups of observations. The fitted lines are shown in Figure 4.4 together with the parameters for the lines and the correlation coefficients.

Since all the correlation coefficients are close to 1, one may conclude with a high level of confidence that the linear regression lines fit the data well. Further, the slope parameters confirm that the concurrent implementations tend to run faster than the sequential implementations. The slope of a line gives an indication of how fast the concurrent run times grow relative to the sequential run times. From the data in Figure 4.4 one can see that the concurrent run times tend to be about half the sequential run times. According

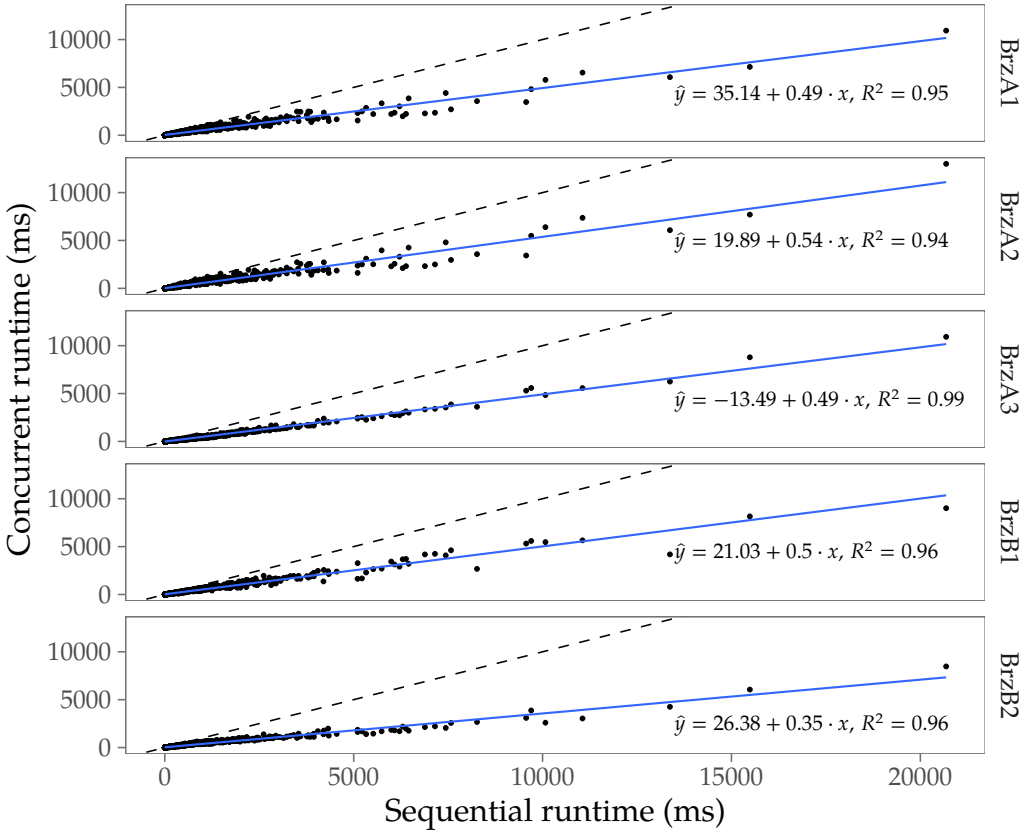


Figure 4.4: Scatter plot of concurrent times against sequential times. The solid line represents a regression line with parameters as shown in the plot. The dashed line represents the line with slope 1. The plots show that the concurrent implementations tend to outperform the sequential implementation.

to this measure, BrzB2 is the best performer with a slope of approximately a third.

Let us now consider the performance of the implementations based on speedup relative to the sequential implementation. As in Chapter 3, the speedup of the concurrent implementations was calculated by dividing the sequential run time by the concurrent run time. From the slopes above, one would expect mean speedup numbers around two. The observed mean speedup numbers together with the mean sequential run times are shown in Table 4.2.

The table provides mean speedup grouped by $|\Sigma|$ and regular expression depth. Recall that there are 100 observations in each such group. The table's first row shows the mean speedup over all alphabet sizes and all regular expression depths. Figure 4.5 plots the mean speedup against the regular expression depth, grouped by alphabet size.

The following are the main observations from the speedup data.

Table 4.2: Observed mean speedup and standard deviation for the concurrent implementations grouped by alphabet size and regular expression depth. The mean sequential run time is also shown in the column T_S. Each entry is the mean over 100 observations.

$\|\Sigma\|$	Depth	T_S(ms)	BrzA1	BrzA2	BrzA3	BrzB1	BrzB2
All	All	357.63	1.38 ± 0.55	2.05 ± 0.96	2.39 ± 0.86	1.66 ± 0.53	1.77 ± 0.57
4	All	187.28	1.77 ± 0.57	1.94 ± 0.59	1.78 ± 0.56	1.87 ± 0.82	1.82 ± 0.77
	5	0.29	1.03 ± 0.38	1.23 ± 0.47	1.06 ± 0.36	0.95 ± 0.35	0.96 ± 0.35
	6	1.17	1.47 ± 0.50	1.73 ± 0.58	1.51 ± 0.50	1.39 ± 0.55	1.37 ± 0.52
	7	12.91	1.95 ± 0.47	2.22 ± 0.48	1.94 ± 0.45	1.97 ± 0.72	1.88 ± 0.64
	8	18.33	2.07 ± 0.45	2.27 ± 0.47	2.10 ± 0.44	2.25 ± 0.76	2.13 ± 0.72
	9	195.77	2.18 ± 0.29	2.25 ± 0.31	2.19 ± 0.26	2.44 ± 0.56	2.37 ± 0.55
	10	895.21	1.90 ± 0.36	1.94 ± 0.40	1.91 ± 0.37	2.24 ± 0.73	2.20 ± 0.68
95	All	136.97	1.22 ± 0.31	2.71 ± 1.08	2.61 ± 0.62	1.80 ± 0.40	1.71 ± 0.38
	5	2.82	0.93 ± 0.28	2.75 ± 0.71	2.48 ± 0.68	1.65 ± 0.47	1.46 ± 0.39
	6	7.82	1.11 ± 0.29	3.21 ± 0.85	2.81 ± 0.65	1.85 ± 0.45	1.59 ± 0.40
	7	24.53	1.20 ± 0.29	3.36 ± 1.29	2.65 ± 0.80	1.84 ± 0.58	1.70 ± 0.37
	8	62.89	1.32 ± 0.28	3.12 ± 1.07	2.60 ± 0.59	1.81 ± 0.27	1.77 ± 0.27
	9	193.30	1.39 ± 0.24	2.24 ± 0.68	2.64 ± 0.51	1.79 ± 0.26	1.81 ± 0.33
	10	530.44	1.35 ± 0.26	1.58 ± 0.45	2.49 ± 0.32	1.85 ± 0.23	1.93 ± 0.32
256	All	331.98	1.05 ± 0.42	1.53 ± 0.63	2.56 ± 0.74	1.57 ± 0.30	1.74 ± 0.47
	5	10.52	0.64 ± 0.28	1.52 ± 0.70	2.35 ± 0.75	1.46 ± 0.43	1.45 ± 0.43
	6	23.91	0.73 ± 0.28	1.67 ± 0.89	2.42 ± 0.91	1.49 ± 0.36	1.52 ± 0.36
	7	70.98	0.94 ± 0.22	1.65 ± 0.71	2.61 ± 0.90	1.52 ± 0.24	1.59 ± 0.35
	8	177.81	1.05 ± 0.16	1.46 ± 0.56	2.67 ± 0.72	1.58 ± 0.20	1.77 ± 0.33
	9	500.02	1.26 ± 0.18	1.26 ± 0.33	2.66 ± 0.37	1.66 ± 0.19	1.95 ± 0.41
	10	1208.67	1.67 ± 0.31	1.60 ± 0.26	2.63 ± 0.57	1.71 ± 0.22	2.16 ± 0.51
512	All	774.28	1.49 ± 0.56	2.02 ± 1.03	2.60 ± 1.12	1.41 ± 0.26	1.81 ± 0.58
	5	25.21	1.08 ± 0.45	2.50 ± 1.47	2.90 ± 1.56	1.19 ± 0.36	1.42 ± 0.56
	6	79.41	1.28 ± 0.47	2.27 ± 1.20	2.96 ± 1.33	1.34 ± 0.19	1.53 ± 0.34
	7	158.25	1.37 ± 0.51	1.97 ± 0.99	2.95 ± 1.38	1.37 ± 0.15	1.67 ± 0.33
	8	401.47	1.36 ± 0.21	1.61 ± 0.72	2.40 ± 0.63	1.44 ± 0.12	1.79 ± 0.32
	9	922.60	1.56 ± 0.28	1.55 ± 0.59	2.29 ± 0.55	1.48 ± 0.18	1.98 ± 0.49
	10	3058.73	2.27 ± 0.47	2.20 ± 0.53	2.14 ± 0.17	1.63 ± 0.25	2.46 ± 0.67

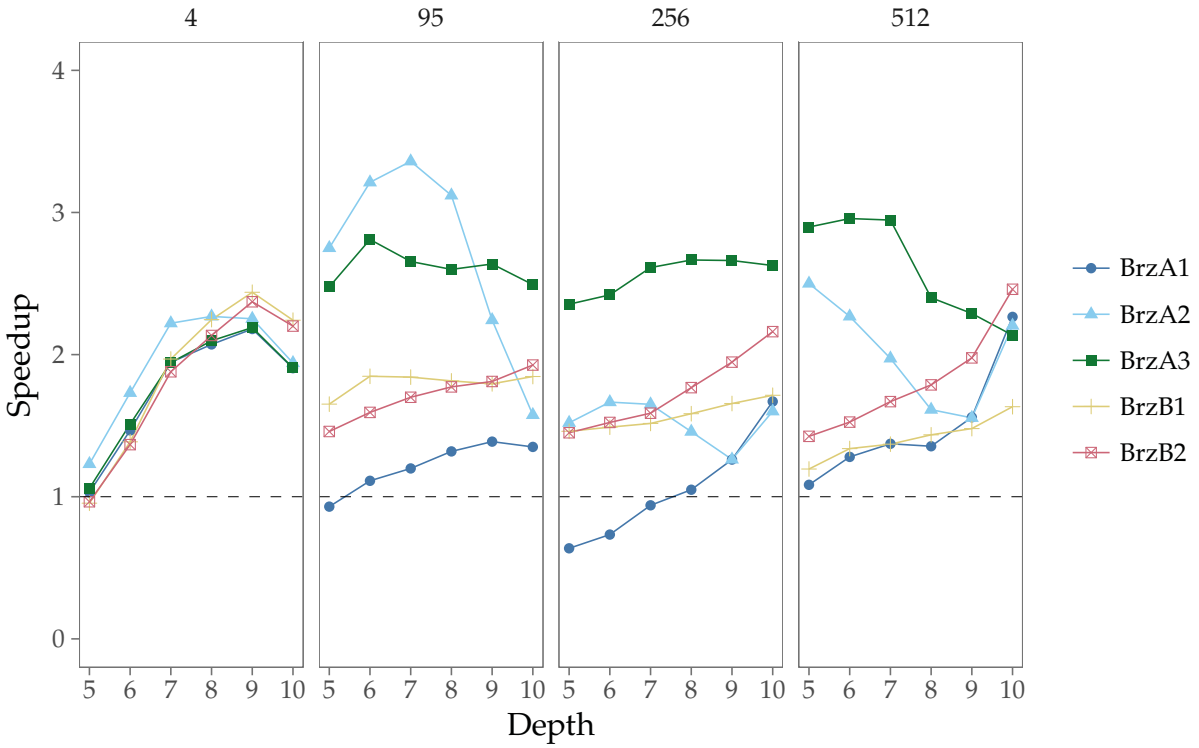


Figure 4.5: Mean observed speedup against regular expression depth, grouped by alphabet size. Each point is the mean over 100 observations. The numbers, together with standard deviations are shown in Table 4.2.

1. Based on the mean speedup over all alphabet sizes and regular expression depths, BrzA3 performed the best with a mean speedup of 2.39. This is different from the earlier measure where BrzB2 was the best, based on the slope of the regression line. This difference is revisited below when the actual run times are discussed.

2. All but one of the concurrent implementations achieved mean speedup above one in all the subgroups. Only BrzA1 failed to improve on the sequential implementation in some of the subgroups.

3. The best speedup (3.36) was obtained by BrzA2 at depth 7 and $|\Sigma| = 95$. The lowest speedup (0.64) was by BrzA1 at depth 5 and $|\Sigma| = 256$.

4. From Figure 4.5 one notices that when $|\Sigma| = 4$, the different implementations behave similarly. Speedup grows as regular expression depth grows, but when the depth reaches 10, the speedup drops. This suggests that when larger regular expressions are used, memory management may affect performance. For a definitive explanation, however, more tests are required and this was deemed to be outside the scope of the present work.

5. In the cases where $|\Sigma| > 4$, the implementations obtained significantly different speedups. The unbuffered BrzA1 implementation

typically performed the worst. Its buffered equivalent BrzA2, however, often performed well for smaller regular expressions.

6. From the performance of BrzA3 in which communication is reduced by not communicating the empty regular expression, it is clear that this modification is indeed an improvement.

7. The speedup of the two implementations based on Configuration B tend to improve as regular expression depth grows. This effect gets more pronounced as the alphabet size grows. BrzB2 is best with large alphabets and large regular expression depths.

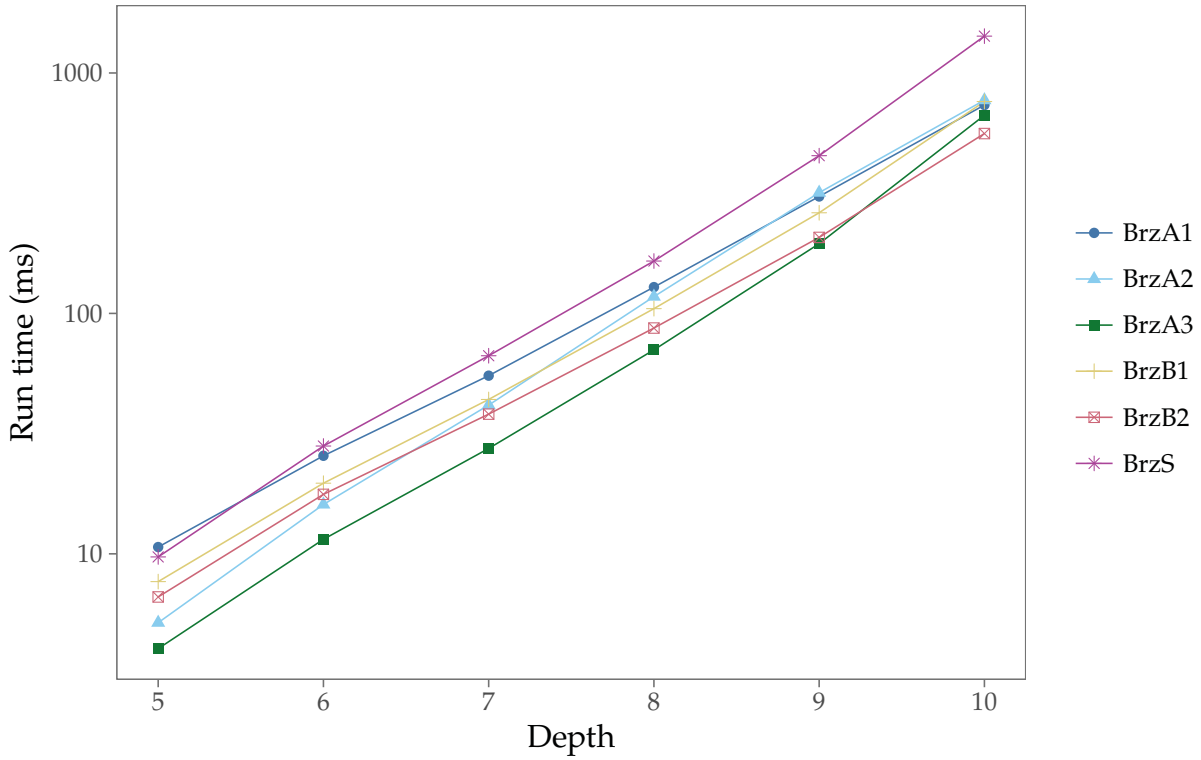


Figure 4.6: Mean run times of the different Brzozowski implementations. The run time is plot against the depth of the regular expression. Note the logarithmic scale on y-axis.

Finally, consider the run times of the different implementations. The mean run time of the implementations are shown in Figure 4.6. It plots the mean run time of each implementation against the regular expression depth. Note the logarithmic scale on the y-axis. It is clear from the plot that the run times grow exponentially as the depth increases. It is also evident that the sequential implementation (BrzS) is the slowest on average in most cases and BrzA3 the fastest. This is consistent with the observation earlier, where BrzA3 achieved the greatest mean speedup. But how does one reconcile this with the observation that BrzB2 is the best based on the slope of the regression line in Figure 4.4?

The lower slope of a regression line does not state that the average run time of the implementation is better than that of another. It simply asserts that the run time of the concurrent implementation grows at a slower rate relative to the run time of the sequential implementation. Compare the lines of the mean run times of BrzA3 and BrzB2 in Figure 4.6. The line for BrzB2 is above the line for BrzA3 for all depths up to nine. The lines cross just after depth nine. For the cases under consideration BrzA3 achieved the best average performance, but BrzB2 seems to be more scalable for larger regular expressions. For smaller regular expressions, however, the overhead of BrzB2 puts it at a disadvantage compared to BrzA3.

4.5 Conclusion

This case study differs from the previous in that there is no obvious way to employ data-level parallelism. The algorithm starts with a single regular expression and more are computed as the algorithm progresses. The processes-based approach allowed one to exploit some task-level parallelism.

The decomposition exercise enabled one to identify opportunities for alternatives in the implementation. The resulting implementations did outperform the plain sequential implementation, but significant speedup was not achieved. Speedup of around two on a six-core CPU suggests room for improvement.

The present work may be extended in a number of ways. The performance analysis could also consider, for example, the effects of different buffer sizes. As mentioned earlier, a better regular expression generation method (Héam and Nicaud 2011) could be employed to create larger regular expressions. The current regular expression implementation could also be improved. The current implementation creates many objects when performing operations on regular expressions. This puts strain on the Go runtime and garbage collector.

5 Incremental DFA minimisation

> Design is the art of separation, grouping, abstraction, and hiding. The fulcrum of design decisions is change. Separate those things that change for different reasons. Group together those things that change for the same reason.
>
> *Robert C. Martin*

The third case study considers the minimisation of finite automata. As in the earlier chapters, a particular sequential algorithm will be the starting point for a process-based implementation of DFA minimisation. The problem of minimising a finite state automaton has been studied quite extensively over the years and many sequential algorithms have been proposed to address this problem. See B. W. Watson (1995, Chap. 7), Almeida, Moreira and Reis (2012) and Berstel et al. (2010) for a comprehensive coverage of the area.

For this chapter, the algorithm under consideration is due to B. W. Watson (2001). Before the algorithm is presented, Section 5.1 summarises the relevant mathematical preliminaries from Chapter 2. Section 5.2 then describes the sequential DFA minimisation algorithm that is the focus of this chapter. Three CSP decompositions are presented in Section 5.3 and the performance of the implementations based on these decompositions is studied in Section 5.4.

5.1 Preliminaries

The purpose of this section is to present the necessary terminology and definitions from Section 2.3 in a convenient manner. For full details, links to the relevant definitions in Chapter 2 are provided in the margin.

Throughout this chapter, consider a specific DFA $(Q, \Sigma, \delta, q_0, F)$. Recall that Q is the finite set of states, Σ is the input alphabet, $\delta\colon Q \times \Sigma \longrightarrow Q \cup \{\bot\}$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of *final* states. Assume further that no state in the automaton is *useless*. That

A formal definition of a DFA can be found in Def. 2.28.

means that, for every state $q \in Q$, there is a path from the start state to q as well as a path from q to a final state. The size of a DFA, $|(Q, \Sigma, \delta, q_0, F)|$, is defined as the number of states, $|Q|$.

The shorthand Σ_q is used to refer to the set of all alphabet symbols that appear as out-transition labels from state q. When it is the case that $\Sigma_p = \Sigma_q$, it will be written as Σ_{pq} instead of Σ_p or Σ_q to emphasise their equality.

Recall from Definition 2.40 that $\delta^* : Q \times \Sigma^* \longrightarrow Q \cup \{\perp\}$ is the extension of δ and maps a state to another state, based on a word.

The *right language* of a state q, written $\overrightarrow{\mathcal{L}}(q)$, is the set of all words spelled out on paths from q to a final state. Formally, $\overrightarrow{\mathcal{L}}(q) = \{ w \mid \delta^*(q, w) \in F \}$. Using the recursive definition of δ^*, one can give a recursive definition of $\overrightarrow{\mathcal{L}}(q)$:

See Def. 2.41.

See Pty. 2.46.

$$\overrightarrow{\mathcal{L}}(q) = \left(\bigcup_{a \in \Sigma_q} \{a\} \overrightarrow{\mathcal{L}}(\delta(q, a)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

Predicate E indicates 'equivalence' of states:

$$E(p, q) \equiv \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$$

With the inductive definition of $\overrightarrow{\mathcal{L}}$, one may rewrite E as follows:

See Pty. 2.63 for a full derivation.

$$E(p, q) \equiv (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : E(\delta(p, a), \delta(q, a)) \rangle$$

The language accepted by DFA M is simply the right language of its initial state.

See Def. 2.47.

$$\mathcal{L}(M) = \overrightarrow{\mathcal{L}}(q_0)$$

The primary definition of minimality of a DFA M is:

See Def. 2.59.

$$Min(M) \equiv \langle \forall\, M' : M' \in DFA \wedge \mathcal{L}(M) = \mathcal{L}(M') : |M| \leq |M'| \rangle$$

Using right languages, minimality can also be written as the following predicate:

See Pty. 2.65.

$$\langle \forall\, p, q : p, q \in Q \wedge p \neq q : \neg E(p, q) \rangle$$

E indicates whether two states are interchangeable. If they are, then one can be eliminated in favour of the other. Of course, in-transitions to the eliminated state are redirected to the equivalent remaining one. This reduction step is not addressed here.

5.2 Sequential algorithm

This section briefly explains the sequential algorithm, Algorithm 5.2, as presented by B. W. Watson (2001). The algorithm takes a DFA as input and returns a set of pairs of states that are equivalent—that is, it returns an equivalence relation on the states. Note that in the text below *ComputeEquiv* refers to the name of the function in Algorithm 5.2, *Equiv* refers to the name of the equivalence relation, and *equiv* is the name of the function that computes the truth value of the predicate $E(p, q)$ for given values of p and q.

Algorithm 5.2 is different from most traditional DFA minimisation algorithms in the sense that it is *incremental*. By this is meant that the algorithm may be halted at any time before *all* equivalent pairs of states have been found. (Almeida, Moreira and Reis (2014) also developed an incremental algorithm, but that algorithm is not considered in the present work.) Because each equivalent pair of states in the relation may be collapsed into a single state, the equivalence relation attained to date can be used to reduce the size of the automaton. The resulting automaton will only be guaranteed to be minimal if the algorithm has run to completion, assuring that all equivalent state pairs have been identified. Optimising improvements were made by B. W. Watson and Daciuk (2003), but these are not considered here.

In essence, Algorithm 5.2 examines pairs of states for equivalence. The equivalence of two states is determined by the recursive functional program, *equiv*, shown in Algorithm 5.1 that is a translation the recursive definition of E. An invocation of the function *equiv* with state parameters p and q returns via the local boolean variable *eq* the truth value of $E(p, q)$.

Note, however, that if the definition of E were to be used directly as a functional program, then there is the possibility of non-termination when the input DFA is cyclic. In order for the functional program *equiv* to terminate, it takes a third parameter, S, along with the two states. S is the set of state pairs visited to date in the recursion. During the recursion, it is assumed that the two states are equivalent (by placing the pair of states in S) until shown otherwise.

It is known (B. W. Watson 1995, §7.3.3) that the depth of recursion is bounded by the larger of $|Q| - 2$ and 0 without affecting the result. Hence one may add a fourth parameter, k, to the function *equiv* to count down the number of recursive calls. An invocation $equiv(p, q, \emptyset, (|Q| - 2)\ \mathbf{max}\ 0)$ returns $E(p, q)$ after no more than $(|Q| - 2)\ \mathbf{max}\ 0)$ recursions.

The expression $a\ \mathbf{max}\ b$ evaluates to the greater of a and b.

The function *equiv* is used by *ComputeEquiv* to compute the relation (i.e. set of state pairs) *Equiv*.

Algorithm 5.2 maintains in variable G a set consisting of the pairs of states known to be inequivalent (*distinguished*), while in H, it accumulates pairs of

Algorithm 5.1 (Pointwise computation of $E(p,q)$):

```
func equiv(p,q,S,k) →
    if k = 0 → eq := (p ∈ F ≡ q ∈ F)
    [] k ≠ 0 ∧ {p,q} ∈ S → eq := true
    [] k ≠ 0 ∧ {p,q} ∉ S →
       eq := (p ∈ F ≡ q ∈ F) ∧ (Σ_p = Σ_q);
       for a : a ∈ Σ_p ∩ Σ_q →
           eq := eq ∧ equiv(δ(p,a), δ(q,a), S ∪ {{p,q}}, k − 1)
       rof
    fi;
    return eq
cnuf
```

□

Algorithm 5.2 (Computing the set $Equiv$):

```
func ComputeEquiv((Q,Σ,δ,s,F)) →
    G,H := ((Q \ F) × F) ∪ (F × (Q \ F)), {(q,q) | q ∈ Q};
    { invariant: G ⊆ ¬Equiv ∧ H ⊆ Equiv }
    do (G ∪ H) ≠ Q × Q →
        let p,q : (p,q) ∈ ((Q × Q) \ (G ∪ H));
        if equiv(p,q,∅,(|Q| − 2) max 0) →
            H := H ∪ {(p,q),(q,p)};
            H := H+
        [] ¬equiv(p,q,∅,(|Q| − 2) max 0) →
            G := G ∪ {(p,q),(q,p)}
        fi
    od; { H = Equiv }
    return H
cnuf
```

□

states belonging to the set *Equiv*. To initialise G and H, note that final states are never equivalent to non-final ones, and that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, the algorithm ensures that H is transitive at each step. This is indicated by $H := H^+$. The repetition in this algorithm can be interrupted and the partially computed H can safely be used to merge states.

5.3 Process-based decomposition

In the sections that follow, the sequential algorithm of the previous section is decomposed into a number of sequential processes. These processes then interact to form concurrent systems that compute the equivalence relation H.

In these decompositions as well as in the later implementations (sequential and concurrent), two deviations are made from Algorithm 5.2: G, the set of distinguished states, is not computed and the transitive closure H^+ is not computed. The consequence of this simplification is that all equivalence pairs will have to be computed explicitly. Also, if the program is interrupted, then H will not be an equivalence relation since it is not transitive. However, the elements in H may still be used to reduce the size of the automaton.

Three decomposition variants are presented. These variants evolved in an iterative fashion after reflecting on possible process-based architectures to implement the above sequential algorithms. In the first variant (Section 5.3.1), a predetermined number of processes compute pairwise equivalence while the transition function is accessed through a server process. In the second variant (Section 5.3.2), this server process is eliminated. In the third and final variant (Section 5.3.3), there is no server and also no longer a fixed, predetermined number of processes computing pairwise equivalence—the processes are created dynamically, one for each pair of states to compare. Each of these variants was implemented and their run time performance is compared against the sequential implementation in Section 5.4.

5.3.1 Access to transition function via server process

In the first decomposition, shown in Figure 5.1, the process network comprises the following processes. *SENDER* has access to a set of state pairs. It sends out one randomly chosen pair of states at a time for which pairwise equivalence is then determined by any one of a number of *EQUIV* processes. Each *EQUIV* process sends the verdict about whether or not its

input pair of states is equivalent to the *GATHERER* process which records the equivalent states. A $SERVER_S$ process encapsulates the transition function δ. The various *EQUIV* processes interact with $SERVER_S$ in order to do their computation—they do not directly access δ.

The subscript S stands for *sequential*. An alternate *concurrent* server is defined later.

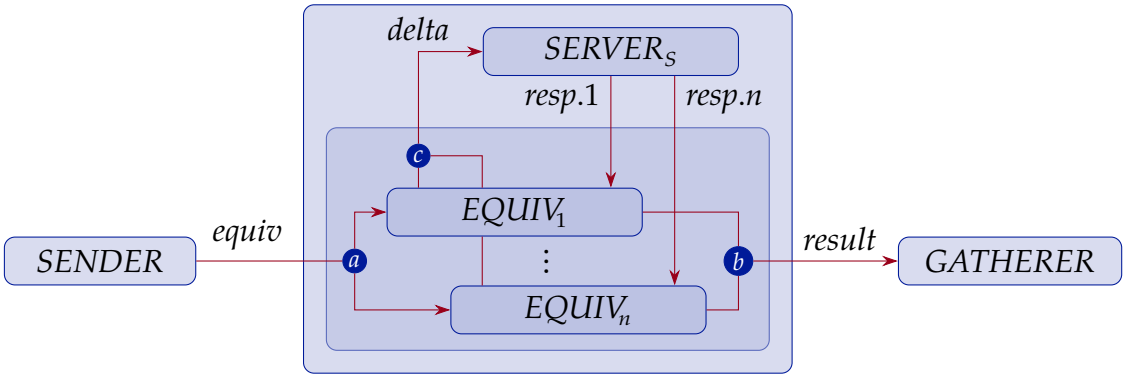


Figure 5.1: Process network structure for the first variant in which a server process interacts with the $EQUIV_i$ processes for access to the transition function. Requests are multiplexed onto channel *delta*, but responses are returned on dedicated channels, *resp.i*.

The first process described is the *SENDER* process. It is responsible for sending out all the pairs for which equivalence is to be determined.

$$
\begin{array}{l}
SENDER(T) = \\
\quad \text{if } |T| = 0 \text{ then} \\
\qquad SKIP \\
\quad \text{else} \\
\qquad \bigsqcap_{(p,q)\in T} (\text{if } p \in F \equiv q \in F \text{ then} \\
\qquad\qquad equiv!(p,q) \rightarrow SENDER(T \setminus \{(p,q),(q,p)\}) \\
\qquad\quad \text{else} \\
\qquad\qquad SENDER(T \setminus \{(p,q),(q,p)\}))
\end{array}
$$

The process is parameterised by a set T containing the pairs of states. It arbitrarily selects a pair (p,q) from T. If $p \in F \equiv q \in F$, meaning both are elements of F or both are not elements of F, then the pair are possibly equivalent. To check whether or not this is the case, the pair is sent out on channel *equiv*. Both (p,q) and (q,p) are removed from T and *SENDER* is re-invoked. If it is not the case that $p \in F \equiv q \in F$, then p and q are definitely not equivalent. Once again, both (p,q) and (q,p) are removed from T and *SENDER* is re-invoked. Once all the pairs in T have been processed, *SENDER* terminates.

The process responsible for updating the equivalence relation H is *GATHERER*.

$$
\begin{aligned}
&GATHERER(H) = \\
&\quad result?\langle p, q, eq\rangle \rightarrow \\
&\qquad \text{if } eq \text{ then} \\
&\qquad\quad GATHERER(H \cup \{(p,q),(q,p)\}) \\
&\qquad \text{else} \\
&\qquad\quad GATHERER(H)
\end{aligned}
$$

It repeatedly reads triples $\langle p, q, eq\rangle$ from its input channel *result*. The triple $\langle p, q, eq\rangle$ represents $E(p,q) = eq$ for some $p, q \in Q$. If p and q are equivalent, the equivalence relation, H, is appropriately updated. If one were to follow Algorithm 5.2, one should also compute the transitive closure of H. For simplicity, this step is omitted in the current presentation.

Note that the computation of the transitive closure is omitted in all implementations, including the sequential implementation.

The next process encapsulates the transition function δ. One may think of it as a server that receives queries for lookups into the implemented transition function structure and then responds with the relevant value. In the process definition the actual data is abstracted away. Rather, the model only captures the behaviour that a query is received and then responded to. In principle, such a sever could operate in either a sequential or concurrent fashion. In the former case, it would handle one item of incoming data at a time to completion before reading in the next item. In the concurrent case, it would read in the next item data at any stage, even while processing earlier data items.

A sequential server $SERVER_S$ may be described as follows. It reads a query from channel *delta* and then replies to the query on the channel dedicated to the process that performed the query. After responding, the server is ready to receive a new query.

$$SERVER_S = delta?id \rightarrow REPLY(id); SERVER_S$$

In the CSP model a query is modelled as an event on channel *delta*. The event identifies the process that issued the query through *id*. This index is used to identify the channel on which to send the response, as shown in process *REPLY* below.

$$REPLY(id) = resp.id \rightarrow SKIP$$

It is fairly simple to describe a concurrent server. Here the query is read and the *REPLY* process executes concurrently with the server process.

Section 5.4.1 shows that the two server versions resulted in two implementations for this variant. One in which the sequential server is utilised, and another in which the concurrent server is utilised.

$$SERVER_C = delta?id \rightarrow (REPLY(id) \;|||\; SERVER_C)$$

The only behaviour remaining to be modelled is that of the $EQUIV_i$ processes. Very abstractly, each of these processes repeatedly reads in a pair of states and then emits whether or not they are equivalent. A rough description might be as follows:

$$P = in?(p,q) \rightarrow out!E(p,q) \rightarrow P$$

This description, however, does not capture the interactions between the $EQUIV_i$ processes and $SERVER_S$ that are required to compute $E(p,q)$.

Below a description of $EQUIV_i$ is given that attempts to model the communication behaviour of the execution of Algorithm 5.1. The actual computation of $E(p,q)$, however, is not modelled, but abstracted away. Instead, CSP nondeterministic choice is used to model the possible paths of the computation. Also, to reduce clutter, *mrd* is used for expression $(|Q|-2)$ **max** 0.

This is a mnemonic for *maximum recursion depth.*

Initially $EQUIV_i$ reads in a pair of states (p,q) from its input channel $in.i$. The process should now behave like an invocation of $equiv(p,q,\emptyset,mrd)$. Inspection of the pseudo-code in Algorithm 5.1 shows that such an invocation may either return a result, or it may recurse after looking up entries in the transition function structure. Consequently, the next step in $EQUIV_i$ is modelled as the choice between $OUTPUT_i$ and $INTERACT_i$. This choice is modelled as CSP internal choice to highlight that the choices are not determined by the environment, but rather by the computation and thus internal to the process.

$$\begin{aligned} &EQUIV_i = \\ &\quad in.i?(p,q) \rightarrow (OUTPUT_i(p,q) \sqcap INTERACT_i(p,q,mrd)) \end{aligned}$$

$OUTPUT_i$ describes that part of $EQUIV_i$ that produces output. It simply emits either *true* or *false* for the given pair (p,q) on the output channel $out.i$. These results are then received by *GATHERER* as described earlier.

$$\begin{aligned} &OUTPUT_i(p,q) = \\ &\quad (out.i!\langle p,q,true\rangle \rightarrow EQUIV_i) \sqcap (out.i!\langle p,q,false\rangle \rightarrow EQUIV_i) \end{aligned}$$

$INTERACT_i$ is used to describe the interaction between $EQUIV_i$ and $SERVER_S$. It also models the bounded recursive nature of $equiv(p,q,\emptyset,mrd)$. For this reason an extra parameter k is used. When k reaches zero, the recursion may not continue and a result will be emitted through the process $OUTPUT_i$. While $k > 0$ the process has a choice between repeating or emitting output. Note that the query and the corresponding response are abstracted away—only the events of sending some query ($qry.i!i$) and receiving a response ($ans.i$) are modelled.

$$\begin{aligned} &INTERACT_i(p,q,k) = \\ &\quad \text{if } k > 0 \\ &\qquad qry.i!i \rightarrow ans.i \rightarrow (OUTPUT_i(p,q) \sqcap INTERACT_i(p,q,k-1)) \\ &\quad \text{else} \\ &\qquad OUTPUT_i(p,q) \end{aligned}$$

A predetermined number (n) of independent $EQUIV_i$ processes may run concurrently:

$$EQUIVS(n) = |||_{i=1..n} EQUIV_i$$

To simplify the interface to the array of $EQUIV_i$ processes as well as to provide buffering, *EQUIVS* is linked to one instance of *DBUFF* of size N_a and two instances *MBUFF* of sizes N_b and N_c respectively, as shown below.

Recall that *DBUFF* receives input from a single *left* channel and emits output on any of an array of *right* channels. Here the input channel is renamed *equiv* to allow interaction with *SENDER* and the output channels are linked to the input channels of *EQUIVS*.

The definitions of *DBUFF* and *MBUFF* my be found in Section 2.5.2.

The output of the $EQUIV_i$ processes are multiplexed onto a single channel by *b.MBUFF*. This process reads input from an array of *b.left.i* channels and multiplexes them onto *b.right*. Hence the output channels of *EQUIVS* are linked to the input channels of *b.MBUFF* and *b.right* is renamed *result*. A second multiplexing process *c.MBUFF* is used to multiplex the queries of the $EQUIV_i$ processes onto a single channel for input to $SERVER_S$. Thus the *qry.i* channels are linked to the *c.left* channels.

$$\begin{aligned}
&BEQUIVS(n) = \\
&\quad a.DBUFF(\langle\,\rangle, N_a, n)[\![^{equiv}/_{a.left}]\!] \\
&\qquad [a.right \leftrightarrow in] \\
&\quad EQUIVS(n) \\
&\qquad [out \leftrightarrow b.left] \\
&\quad b.MBUFF(\langle\,\rangle, N_b, n)[\![^{result}/_{b.right}]\!] \\
&\qquad [qry \leftrightarrow c.left] \\
&\quad c.MBUFF(\langle\,\rangle, N_c, n)
\end{aligned}$$

Next, one may link *BEQUIVS* and $SERVER_S$ to form *BEQUIVSERVER*. This is done by linking *a.right* to *delta* and the *ans* channels to the *resp* channels.

$$\begin{aligned}
&BEQUIVSERVER = \\
&\quad BEQUIVS(n)\,[c.right \leftrightarrow delta, ans \leftrightarrow resp]\,SERVER_S
\end{aligned}$$

The complete system may then be modelled as three top-level processes interacting through two channels as shown in Figure 5.1.

$$MIN1 = (SENDER(Q \times Q) \underset{\{\![equiv]\!\}}{||} BEQUIVSERVER) \underset{\{\![result]\!\}}{||} GATHERER(\emptyset)$$

5.3.2 Direct access to the transition function

Since all access to the transition function is read-only, it is not necessary to restrict access to it through a process. In the second process-based decomposition this process ($SERVER_S$ in *MIN1*) is eliminated. This change simplifies significantly the CSP description of $EQUIV_i$ since there is no longer a need to model communication with a server process. Figure 5.2 shows the simplified structure. The definitions of *SENDER* and *GATHERER* remain the same as in Section 5.3.1.

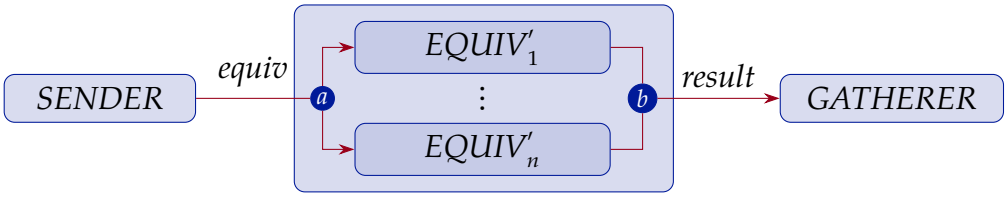


Figure 5.2: Process network for the second variant in which the server process is eliminated. The $EQUIV_i$ processes directly access the transition function data.

$EQUIV'_i$ now reads from its input channel $in.i$ a pair of states (p, q) and emits the result of executing $equiv(p, q, \emptyset, mrd)$ on $out.i$.

$$EQUIV'_i = in.i?(p,q) \rightarrow out.i!\langle p, q, equiv(p, q, \emptyset, mrd)\rangle \rightarrow EQUIV'_i$$

As before, a predetermined number of such processes may execute independently, each with its own input channel.

$$EQUIVS'(n) = |||_{i=1..n}\, EQUIV'_i$$

In order to connect the single channel of *SENDER* to the multiple channels of $EQUIVS'$, an instance of *DBUFF* is used. This ensures that a pair of states that is sent out on *equiv* may be received by any one of the $EQUIV'_i$ processes. Similarly, an instance of *MBUFF* is used to multiplex the results of the $EQUIV'_i$ processes onto channel *result*.

$$\begin{aligned} BEQUIVS' = {} & a.DBUFF(\langle\,\rangle, N_a, n)[\![^{equiv}/_{a.left}]\!] \\ & [a.right \leftrightarrow in] \\ & EQUIVS' \\ & [out \leftrightarrow b.left] \\ & b.MBUFF(\langle\,\rangle, N_b, n)[\![^{result}/_{b.right}]\!] \end{aligned}$$

Putting all these components together, one may define the second variant as follows.

$$MIN2 = (SENDER(Q \times Q) \underset{\{\!|equiv|\!\}}{\|} BEQUIVS') \underset{\{\!|result|\!\}}{\|} GATHERER(\emptyset)$$

5.3.3 Many short-lived processes

A third variant may be created by simply starting, for each pair of states, a process to compute their equivalence. Instead of a *SENDER* process sending out pairs, define a *LAUNCHER* process that starts a process for each pair of states for which equivalence needs to be determined. Figure 5.3 shows the structure, where the *BLAUNCHER* process is the *LAUNCHER* process combined with appropriate buffering, as explained below.

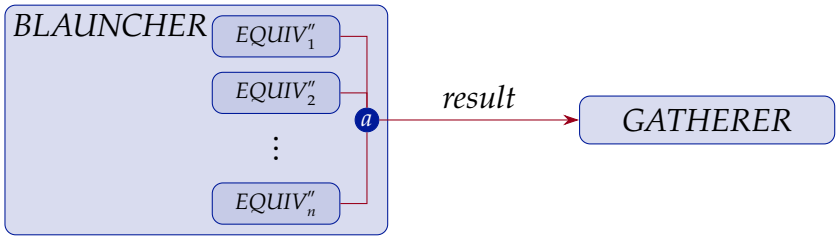


Figure 5.3: An $EQUIV_i$ only computes the equivalence of a single pair of states and then terminates. Many of these processes are created during the execution of the system and n is thus not determined by the implementer, but is equal to the number of pairs that require comparison.

LAUNCHER is parametrised with a counter, i, and a set of pairs of states, T. The counter is used to identify each of the launched processes and T contains the pairs of states from the automaton. When the T is empty *LAUNCHER* terminates. While there are still elements in T, *LAUNCHER* selects an arbitrary element $(p,q) \in T$ and if $p \in F \equiv q \in F$ then it launches a new $EQUIV''_i(p,q)$ process. This new process runs concurrently with a recursive instance of *LAUNCHER* that has appropriately updated parameters.

$$
\begin{aligned}
&LAUNCHER(i,T) = \\
&\quad \text{if } |T| = 0 \text{ then} \\
&\qquad SKIP \\
&\quad \text{else} \\
&\qquad \bigsqcap_{(p,q)\in T} (\text{if } p \in F \equiv q \in F \text{ then} \\
&\qquad\qquad EQUIV''_i(p,q) \;|||\; LAUNCHER(i+1, T \setminus \{(p,q),(q,p)\}) \\
&\qquad\quad \text{else} \\
&\qquad\qquad LAUNCHER(i, T \setminus \{(p,q),(q,p)\}))
\end{aligned}
$$

The new $EQUIV''_i$ process does not read from an input channel any more, but rather accepts the states as parameters when it is created. It simply communicates the result of the computation on its output channel $out.i$ and then terminates.

$$EQUIV''_i(p,q) = out.i!\langle p, q, equiv(p,q,\emptyset, mrd)\rangle \rightarrow SKIP$$

LAUNCHER is composed with an instance of *MBUFF* in order to combine all the different output channels into one from which *GATHERER* may then collect the results. Hence *LAUNCHER* and *a.MBUFF* are linked by connecting the *out.i* channels to the *a.left.i* channels. *a.MBUFF*'s output channel is renamed to *results* which allows it to synchronize with *GATHERER*. *a.MBUFF* has capacity N_a and n is the number of *a.left.i* channels. In this case, n should at least equal the number of $EQUIV_i$ processes created by *LAUNCHER*.

$$
\begin{aligned}
&BLAUNCHER = \\
&\quad LAUNCHER(1, Q \times Q) \\
&\qquad [out \leftrightarrow a.left] \\
&\quad a.MBUFF(\langle\,\rangle, N_a, n)[\![^{results}/_{a.right}]\!]
\end{aligned}
$$

The third variant *MIN3* is then defined as the parallel composition of *BLAUNCHER* and *GATHERER*.

$$MIN3 = BLAUNCHER \underset{\{\!|results|\!\}}{\|} GATHERER(\emptyset)$$

As in the previous case studies, termination is not completely addressed in the above CSP descriptions, but left as an implementation matter. A similar approach to the "poisoning" alluded to in Section 3.2.5 could be followed to address termination fully. One could extend the models to allow for the process network to be "poisoned" when *SENDER*—or *LAUNCHER* in the *MIN3* case—has sent out all the pairs requiring comparison.

5.4 Performance comparison

Using the three process-based variants discussed above, a number of Go implementations were developed. This section compares the run-time performance of these implementations against a sequential implementation.

5.4.1 Implementation

The following Go implementations were developed.

MinS — A sequential implementation of Algorithm 5.2. The implementation deviates slightly from the description in that the set of *distinguished* states G is not computed and the transitive closure of the equivalence relation H^+ is not computed. These simplifications are carried through to all the concurrent implementations.

Min1a — An implementation based on the CSP specification *MIN1* from Section 5.3.1. Recall that a predetermined number of $EQUIV_i$ processes interact with a sequential $SERVER_S$ process to access the transition function.

Min1b — This implementation is a slight modification of *MIN1*. Instead of using a sequential $SERVER_S$ process, a concurrent server $SERVER_C$ is used. That is

$$SERVER_C = delta?id \rightarrow (REPLY(id) \;|||\; SERVER_C)$$

Each request is serviced in a new goroutine that runs concurrently with the server process.

Min2 — An implementation of *MIN2*. Recall that the server process is eliminated and that each $EQUIV'_i$ process accesses the transition function structure directly.

Min3 — This is an implementation of *MIN3* where a new goroutine is created for each pair of states for which equivalence needs to be determined.

5.4.2 Experimental setup

Complete, randomly generated automata of various sizes were used as input to the implementations. The automata were generated using a method by Bassino, David and Nicaud (2008). The method was implemented in Go based on version 1.08.0929 of REGAL (Bassino, David and Nicaud 2007). In order to have a range of alphabet- and automaton sizes, three alphabet sizes were used $\{4, 95, 256\}$ to generate automata with states $\{10, 50, 100, 500\}$. For each combination a hundred automata were generated and for each automaton the minimum time over three runs was recorded for each of the implementations listed above.

REGAL is a library to randomly and exhaustively generate automata.

Go 1.5.3 was used to compile the implementations and they were executed on the same machine as the earlier case studies. It ran Linux 3.10.17 with a six-core hyper-threaded CPU with 16 GB of RAM. The Go runtime was configured to run twelve processes concurrently and the number of $EQUIV_i$ goroutines that were created in Min1a, Min1b and Min2 was $n = 24$.

In cases where the total run time for the three runs exceeded 3 min, the minimum run time of the completed runs was taken.

Intel® Xeon® CPU E5-2630 v2 @ 2.60 GHz. See Section 2.7.3 for more details.

The buffers sizes used in the runs are shown in Table 5.1. Note that in the Min3 case the same buffer size was used as in the other cases even though the number of $EQUIV_i$ processes is greater than n. This was done to keep the scenarios similar in terms of channel configurations.

Table 5.1: Buffer sizes for the channels used in the Go implementations. Note that $n = 24$ is the number of $EQUIV_i$ processes.

Channel	Min1a	Min1b	Min2	Min3
equiv	$2n$	$2n$	$2n$	—
results	$2n$	$2n$	$2n$	$2n$
qry	n	n	—	—
resp.i	1	1	—	—

5.4.3 Observations

The results of the experiments are described in this section. The mean speedup obtained by the different implementations are shown Table 5.2. Also shown is the mean duration of the sequential implementation MinS. The table is grouped by $|\Sigma|$ and the number of states $|Q|$ in the automata. A designator 'All' indicates the mean has been taken over all the groups for that variable. Hence the first row shows the mean speedup over all alphabet sizes and all automaton sizes.

Table 5.2: Average speedup of the implementations grouped by alphabet size and automaton size. The mean run time of the sequential implementation is shown in column T_S. The columns show mean speedup as well as the standard deviation.

$\|\Sigma\|$	$\|Q\|$	T_S(ms)	Min1a	Min1b	Min2	Min3
All	All	440 243.00	0.93 ± 0.16	5.48 ± 1.69	5.65 ± 1.40	5.02 ± 1.70
4	All	16 085.55	0.81 ± 0.21	4.20 ± 1.98	4.68 ± 1.72	4.78 ± 1.82
	10	0.34	0.45 ± 0.08	0.83 ± 0.17	1.72 ± 0.86	1.72 ± 0.76
	50	14.33	0.81 ± 0.03	4.13 ± 0.16	4.95 ± 0.29	5.86 ± 0.23
	100	108.53	0.88 ± 0.02	4.62 ± 0.10	5.14 ± 0.12	5.02 ± 0.36
	500	14 669.33	1.00 ± 0.01	6.09 ± 0.06	6.02 ± 0.06	5.64 ± 1.09
	1000	115 184.93	0.99 ± 0.00	6.42 ± 0.05	6.51 ± 0.05	6.54 ± 0.06
95	All	318 154.55	0.99 ± 0.07	6.05 ± 1.10	6.10 ± 0.90	4.89 ± 1.81
	10	3.07	0.95 ± 0.13	4.09 ± 0.68	4.55 ± 0.70	4.55 ± 0.92
	50	317.92	0.99 ± 0.03	6.40 ± 0.18	6.38 ± 0.18	6.62 ± 0.22
	100	2514.33	1.01 ± 0.01	6.63 ± 0.10	6.53 ± 0.09	6.72 ± 0.13
	500	286 768.80	1.02 ± 0.00	6.71 ± 0.03	6.59 ± 0.03	2.39 ± 0.24
	1000	2 284 182.73	1.00 ± 0.00	6.73 ± 0.01	6.77 ± 0.02	3.41 ± 0.20
256	All	986 488.90	1.00 ± 0.07	6.20 ± 0.96	6.19 ± 0.84	5.39 ± 1.37
	10	7.25	0.97 ± 0.14	4.56 ± 0.74	4.82 ± 0.81	5.03 ± 0.87
	50	777.99	1.01 ± 0.03	6.54 ± 0.21	6.49 ± 0.19	6.69 ± 0.18
	100	6109.03	1.01 ± 0.01	6.58 ± 0.10	6.50 ± 0.10	6.68 ± 0.10
	500	889 141.64	1.02 ± 0.00	6.81 ± 0.02	6.60 ± 0.02	3.40 ± 0.35
	1000	7 086 328.31	1.00 ± 0.00	6.80 ± 0.01	6.83 ± 0.01	4.89 ± 0.08

From the table it can be seen that Min1a performs the poorest. For all groups the observed speedup is close to one or lower. The other three implementations, however, do achieve mean speedup greater than one in most cases. In fact, the difference between Min1a and the others is dramatic.

In order to visually assess in which combinations of alphabet and automaton size the different implementation performed well, the plot in Figure 5.4 was created. It visualises in a grid the mean speedup achieved by the four implementations. Rows represent alphabet sizes, columns represent automaton sizes, and the shade of a grid cell represents the speedup achieved. Darker shades indicate lower speedup and lighter shades indicate higher speedup.

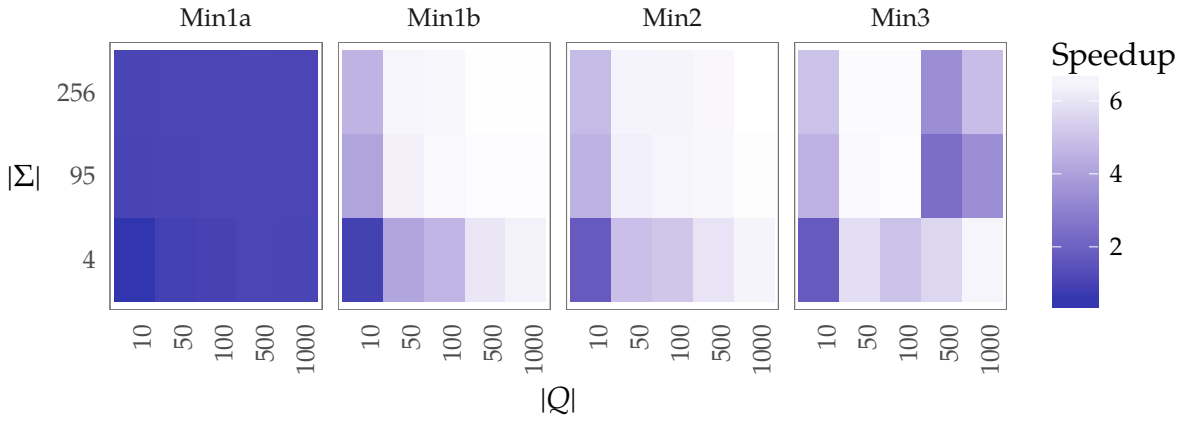


Figure 5.4: Average speedup observed by alphabet size and automaton size. The shading represents the speedup. Darker shades imply lower speedup and lighter shades imply higher speedup. The sequential server implementation (Min1a) clearly does not perform well. The other implementations tend to perform better for greater alphabet sizes and larger automata.

From the figure it is immediately evident that Min1a performs poorly in all cases and that all implementations perform poorly when $|\Sigma| = 10$. The figure also shows that, in the cases of Min1b and Min2, speedup improves both as $|Q|$ increases and as $|\Sigma|$ increases. The performance of Min3 follows a similar performance pattern provided that $|Q| < 500$. Indeed, speedup continues to improve as $|Q|$ increases to 500 or more, provided $|\Sigma| = 4$. However, Min3 does not perform well when the $|Q| = 500$ and $|\Sigma| = 95$ but improves somewhat from this poor performance as $|Q|$ and $|\Sigma|$ increase. This drop in performance is probably caused, in part, by the large number of goroutines that are created during execution, causing runtime scheduler and memory management overhead. However, the fact that the speedup improves when $|Q| = 1000$ suggests that other factors are also involved. The

precise identification of these factors are beyond the scope of this present study.

From Figure 5.4 one cannot easily compare the performance of the different implementations relative to one another. For this reason Figure 5.5 shows the mean speedup of each implementation against the number of states grouped by $|\Sigma|$. The dashed line represents speedup of one. It is again clear that Min1a does not perform well—being on or under the dashed line. This makes sense because the server process in this implementation essentially serialises the execution of the processes computing pairwise equivalence.

The earlier observation that the speedups under Min1b and Min2 increase as automaton sizes increases, is also shown clearly in Figure 5.5.

For automata with up to a hundred states, Min3 usually achieves slightly greater speedup than the other implementations. For automata with more than a hundred states, Min3 performs much worse than the Min1b and Min2 implementations (though not when $|\Sigma| = 4$). As mentioned earlier, this drop in performance is likely due to, *inter alia*, the large number of processes which cause greater overhead compared to the other implementations.

The speedup behaviours of Min1b and Min2 are fairly similar. Typically, for smaller automata Min2 performs better than Min1b. For larger automata, however, Min1a performs better. This observation may be explained by the server process in Min1b that causes overhead that penalises smaller automata. In larger automata, however, the server process may have an advantage if the transition function data may be kept in cache. The cost of communication is then lower than the cost of fetching individually the transition function data. When $|Q| = 1000$ then Min2 again slightly outperforms Min1b, suggesting that the transition function structure does not effectively fit in cache any more and that it is more efficient for each individual $EQUIV_i$ to fetch the transition data.

It was found that the input automata generated in the above experiments were usually minimal. This implies that the processes usually found pairs of states distinguished rather then equivalent. It was deemed prudent to determine whether using automata that are typically not minimal would alter significantly the speedup performance of the implementations. For this reason automata were generated using the Brzozowski construction from Chapter 4. Regular expressions were generated using the method described in Section 4.4.1. A thousand regular expressions of depth 9 and four symbols were generated. These were then used to construct a thousand automata.

Table 5.3 and Figure 5.6 show the mean speedup obtained by the implementations when using these newly created automata as input. Since the

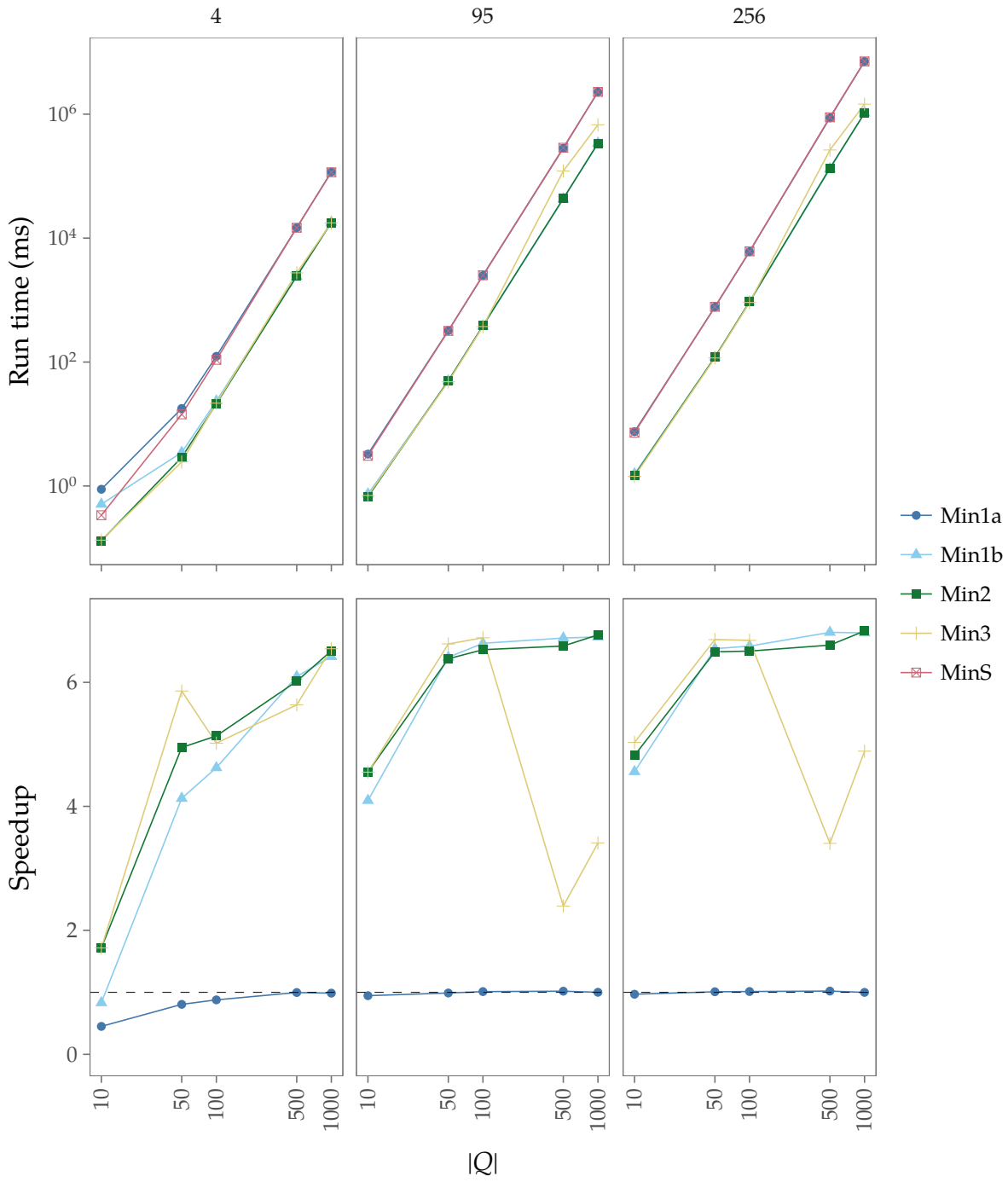


Figure 5.5: Mean run time and mean speedup of each concurrent implementation against automaton size, grouped by alphabet size. Note the logarithmic scale on both the x-axes as well as on the y-axis of the run time plots. Each point is the mean over 100 observations. The plots clearly show that the sequential server implementation performs much poorer than the others.

Table 5.3: Mean speedup achieved when using Brzozowski's DFA construction algorithm using random regular expressions of depth 9 and $|\Sigma| = 4$.

$\|Q\|$	N	T_0(ms)	Min1a	Min1b	Min2	Min3
All	1000	3834	0.53 ± 0.19	1.69 ± 1.34	4.39 ± 2.26	4.56 ± 2.31
[0, 30)	527	4	0.39 ± 0.12	0.75 ± 0.42	2.95 ± 1.98	3.03 ± 2.00
[30, 75)	317	245	0.62 ± 0.10	2.13 ± 0.83	5.80 ± 1.30	6.45 ± 0.75
[75, 300)	154	24 103	0.79 ± 0.08	3.93 ± 1.04	6.40 ± 1.00	5.92 ± 1.61
[300, 750)	2	21 431	0.87 ± 0.00	5.20 ± 0.16	6.15 ± 0.23	3.05 ± 1.72

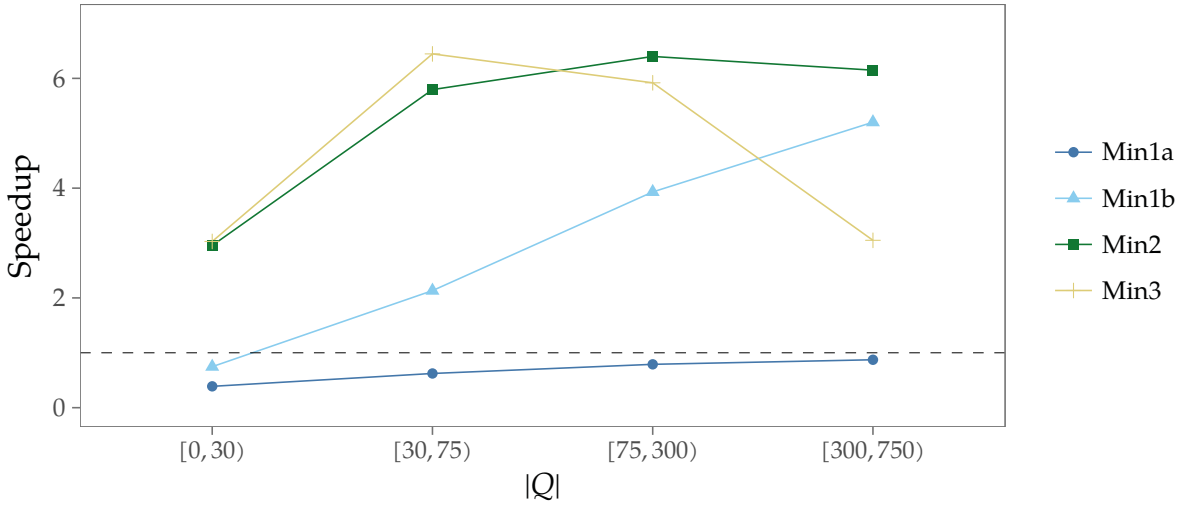


Figure 5.6: Mean speedup against DFA size for a thousand automata generated from random regular expressions with $|\Sigma| = 4$.

automata vary in size, they were grouped into intervals. Only a small number of larger automata were generated. From the table and figure it is clear that the performance of the implementations are fairly similar to the earlier observations. Min1a performs poorly throughout and Min3 achieves the greatest speedup for smaller automata but slows down when larger automata are considered. The speedup of both Min1b and Min2 tend to increase as the automata grow larger. Moreover, Min2 outperforms Min1b, but their performance grow closer as automaton size increases. This is all consistent with the results seen earlier. It is interesting to note, however, that the difference between the performance of Min1b and Min2 is somewhat greater than before.

5.5 Conclusion

In this case study fairly good speedup was obtained using CSP models to guide Go implementations. Using CSP to model the descriptions allows

one to discover alternative approaches. The CSP system structures in this case study was slightly different from this in the Brzozowski case study. In the previous case study the process network formed a loop since feedback was required to continue. In the present case study, all the states that need to be compared are known *ab initio*, and hence the network form a pipeline.

In future work, one should also compute the transitive closure of the equivalence relation. Perhaps one may be able to perform the computation in a concurrent process that is separate from the rest of the system. This will then reduce the number of pairs of states to compare explicitly. Also, instead of endlessly creating processes in Min3, one could dynamically create goroutines up to a set bound and then wait for some of them to terminate before creating more. This should reduce the load on the Go runtime environment. However, a more detailed investigation into the poor performance of Min3 for larger automata is required in order to identify the reasons—apart from the large number of goroutines—for the drop in performance when $|Q| \geq 500$.

6 Single keyword pattern matching

> Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated.
>
> *Edsger W. Dijkstra*

The final case study returns to the problem of string pattern matching. However, in Chapter 3 one had to find all occurrences of *multiple* keywords in a text. In this case, one has to find all occurrences of a *single* keyword. See Cleophas (2003) and Faro and Lecroq (2013) for surveys of pattern matching algorithms.

Since the text is known *ab initio*, one may employ an embarrassingly parallel approach by partitioning the text and searching each partition before combining the results. However, here an attempt is made to employ task-level parallelism by means of a process-based decomposition.

In fact, the implementation DZ5, described below, employs such an approach.

The chapter starts with an overview of the abstract DZ algorithm in recursive form—a particular sequential algorithm in Section 6.1 for single keyword pattern matching, initially proposed by B. W. Watson and R. E. Watson (2003). Two process-based decompositions are then given in Section 6.2. These decompositions were used to guide Go implementations. However, a number of other implementations were also developed. Section 6.3.1 describes these implementations, while Section 6.3.3 shows a performance comparison of these implementations against their sequential counterparts.

6.1 Sequential algorithm

The recursive version of the abstract DZ algorithm derived by B. W. Watson and R. E. Watson (2003) has been reproduced in Algorithm 6.1 in a slightly modified form. The intention in this section is to give a high-level intuitive account of this algorithm. See B. W. Watson and R. E. Watson (2003) for a detailed account of its correctness. Note that there are many concrete ways

in which this abstract algorithm can be instantiated, so that the abstract version really represents a *family* of pattern matching algorithms. See Awid, Cleophas and B. W. Watson (2016), Mauch et al. (2012) and B. W. Watson, Kourie and Strauss (2012) for various studies of concrete versions of this abstract algorithm.

The abstract recursive procedure is called *dzmat*. It searches in text S for all occurrences of pattern p. However, it does not search all of S. Instead, its search is limited to a so-called "live" zone—a range of integer indices into S designated as $[low, high)$. Recall the convention that indexing in a string S starts at 0 and ends at position $|S|-1$. Also by convention, integer intervals are generally indicated as closed from below and open from above. Thus, the indices into S are in the interval $[0, |S|)$, while S_i is the symbol in S at index i, and $S_{[i,j)}$ denotes the substring from index i to $j-1$.

String definitions and notation may be found in Section 2.2.

The nomenclature *live* was chosen to indicate that searching in this interval is still a live concern because the algorithm has yet to explore whether some indices in the *live* zone correspond to matches in S.

In earlier versions of the abstract algorithm, a variable *dead* was used to represent a set of "dead" indices—"dead" in the sense that it has already been established whether or not S indexed by an integer in *dead* will lead to a match. Note that some indices in *dead* may be match positions in S that have already been reported. All of those that are indeed match positions are recorded by the algorithm in a set MS, as they are encountered. In B. W. Watson and R. E. Watson (2003), variable *dead* was used to rigorously express the invariants; since the proofs are not shown here, the variable *dead* is also omitted from the current version of *dzmat*. Nevertheless, the phrase *DZ algorithms* is still used to characterise the family of algorithms that are based on growing a dead-zone of indices.

A match position is an index position in S at which an occurrence of the pattern p is found. Thus, if $\langle \forall\, i : i \in [0, |p|) : S_{mp+i} = p_i \rangle$, then mp is a match position.

Note that *dzmat* assumes that S, p and MS are globally available to all its invocations, including recursive invocations. MS is initially the empty set. The algorithm also assumes that $[low, high)$ designates an interval of live indices in S that an invocation *dzmat* is to handle. Finally, it assumes that $mo(0), mo(1), \ldots mo(|p|-1)$ specifies some predefined ordering sequence to be used when matching elements of p against S. This ordering could be, for example, a conventional left to right sequence, or a right to left sequence, or indeed any other permutation of indices in the interval $[0, |p|)$.

The first invocation of *dzmat* is parameterised by the initial boundaries of the *live* zone as follows: $dzmat(0, |S| - |p| + 1)$; that is, the *live* zone encompasses all except the last $|p|-1$ indices of S. The last $|p|-1$ indices are already in the dead-zone as there is no possibility of a match of length $|p|$ occurring there. All those indices are therefore implicitly relegated to the dead-zone.

Algorithm 6.1 (Abstract DZ Matcher):

proc $dzmat(low, high) \rightarrow$
  if $(low \geq high) \rightarrow$ **skip**
  [] $(low < high) \rightarrow$
    $mid := \lfloor (low + high)/2 \rfloor$;
    $i := 0$;
    { **invariant:** $\langle \forall\ k : k \in [0, i) : p_{mo(k)} = S_{mid+mo(k)} \rangle$ }
    do $((i < |p|)$ **cand** $(p_{mo(i)} = S_{mid+mo(i)})) \rightarrow$
      $i := i + 1$
    od;
    { **postcondition:** $\langle \forall\ k : k \in [0, i) : p_{mo(k)} = S_{mid+mo(k)} \rangle$
      $\wedge$ if $i < |p|$ then $p_{mo(i)} \neq S_{mid+mo(i)}$ }
    if $i = |p| \rightarrow MS := MS \cup \{mid\}$
    [] $i < |p| \rightarrow$ **skip**
    fi;
    $left := mid - shleft(i, mid)$;
    $right := mid + shright(i, mid)$;
    $dzmat(low, left)$;
    $dzmat(right, high)$
  fi
corp

□

Turning now to the algorithmic steps within the algorithm, it is immediately clear that $low \geq high$ (i.e. $live = \emptyset$) serves as the recursion base case for terminating the recursion. If, alternatively, $low < high$ (i.e. $live \neq \emptyset$) then *mid* is computed as an index into S from which the next match attempt will take place. Note that although *mid* is computed as the midpoint of *live*, this is not entirely necessary. Other starting positions within the *live* range are also legitimate.

Using i to reference into p and i and *mid* to reference into S, a loop matches symbols of p against those of S. Recall that the order in which this matching takes place is not necessarily left-to-right, i.e. p_0 against S_{mid}, p_1 against S_{mid+1}, etc. Instead, the match order is determined by the bijective function $mo\colon [0,|p|) \longrightarrow [0,|p|)$. The abstract DZ algorithm therefore allows for *any* permuted order to be predetermined by the implementer of the algorithm.

The loop terminates upon the first mismatch, or when a complete match is found. In the latter case, *mid*, the starting position in S of the match, is added to MS.

The next step in the abstract DZ algorithm is the computation of the new portion of *dead* territory that can be inferred as a result of the matchings that have taken place. Two functions, *shright* and *shleft*, are used for this purpose. The returned value of *shright* is added to *mid*, and is considered to be the (open) upper bound of an interval that can be added into the (implicit) dead-zone region. Similarly, the returned value of *shleft* is subtracted from *mid* and is regarded as the (closed) lower bound of this interval—the interval $[left, right)$ is seen as augmenting the dead-zone of S as determined to date. However, no explicit bookkeeping of this dead-zone region is needed. Instead, two recursive calls are made to *dzmat*. The first probes the remaining live zone in the (contiguous) interval $[low, left)$, and the second the rest of the live zone in the contiguous interval $[right, high)$.

Note that, as with *dzmat*, the shift functions are assumed to have global access to S and p. The abstract DZ algorithm leaves it up to the implementer to determine how these shift functions determine their respective shift distances. In the abstract format, the shift functions are shown as having two parameters: *mid*, the start of the window in S where the pattern p is to be matched against the contents of S; and i, indicating how far into the pattern p matching has proceeded before an outcome has been obtained as to whether or not the match attempt has been successful. Not all shift functions require all this information and in some cases, a shift function implementation could infer additional information—such as $S_{mid+p-1}$, the text character aligned with the rightmost symbol of the pattern—from the globally accessible data. Virtually all of the shift functions in the Boyer-Moore (BM)-type of algorithms are adaptable for use in DZ. These include variants and combinations of the functions employed by Boyer and Moore

(1977), Horspool (1980) and Knuth, Morris and Pratt (1977). See Mauch (2016) for an investigation into the performance of various of these shift functions in the context of DZ.

6.2 Process-based decompositions

In this section two process-based decompositions of the abstract DZ algorithm are presented. In the first, a fixed number of processes concurrently both compare the text and the pattern, and determine new live zones based on their input live zones. In the second, a fixed number of processes compare the text and pattern given a starting position in the text. The computation of new live zones, however, is not done by these processes, but rather by a separate process.

6.2.1 Decomposition 1

In the first decomposition, a number of $DZONER_i$ processes is used to model the execution of *dzmat* from Algorithm 6.1. A $DZONER_i$ receives a live zone on its input channel and emits on its output channel the results of its computation as elaborated below.

$$
\begin{array}{l}
DZONER_i = \\
\quad in.i?\langle low, high\rangle \rightarrow \\
\qquad \text{if } low \geq high \text{ then} \\
\qquad\quad DZONER_i \\
\qquad \text{else} \\
\qquad\quad \text{let} \\
\qquad\qquad mid = \lfloor (low + high)/2 \rfloor \\
\qquad\qquad left = mid - shleft() \\
\qquad\qquad right = mid + shright() \\
\qquad\quad \text{within} \\
\qquad\qquad \prod_{p \in \{-1, mid\}} (out.i!\langle\langle low, left\rangle, \langle right, high\rangle, p\rangle \rightarrow DZONER_i)
\end{array}
$$

- $DZONER_i$ receives a pair of numbers $\langle low, high\rangle$ on its input channel $in.i$. This pair represents the interval $[low, high)$ and the process now models an invocation $dzmat(low, high)$.
- In the case where the interval is empty, i.e. $low \geq high$, the process simply reads in a new interval from its input channel.

- If, however, the interval is not empty, the process should model the behaviour of searching for a match at $mid = \lfloor (low + high)/2 \rfloor$ and of computing two new live zones. Instead of modelling explicitly in CSP the comparison of p_j and S_{mid+j} for every $j \in [0, |p|)$, the comparisons are abstracted away using non-deterministic choice. Only one of two outcomes is possible: either a match is found or no match is found. If a match is found, then *mid* is sent out on the output channel and if no match is found, the special value of -1 is sent out. The calculation of the shift distances is similarly abstracted away. The functions *shleft* and *shright* simply return an arbitrary distance in the range $[1, |p|+1)$.
- A $DZONER_i$ communicates output as triples over channel $out.i$. The first two elements of the triple are the two new live zones and the third is a random element from $\{-1, mid\}$ as described above.
- Multiple $DZONER_i$ processes execute independently as shown below. In this case n processes run concurrently, where n is determined by the implementer of the system.

$$DZONERS(n) = |||_{i=1..n} DZONER_i$$

A *MANAGER* process keeps track of all the live zones as well as the set of matches found. *MANAGER* has three parameters. The first, T, is a set of intervals, each interval being part of the live zone that needs to be processed; the second, MS, is the set of index positions where the pattern matches the text; and the third, Cnt, is a counter to aid in termination.

The phrase *live zones* will be used in the narrative to refer to intervals into which probes are still to be made.

$$\begin{array}{l} MANAGER(T, MS, Cnt) = \\ \quad \text{if } |T| > 0 \vee Cnt > 0 \text{ then} \\ \quad\quad SEND(T, MS, Cnt) \,\square\, RECEIVE(T, MS, Cnt) \\ \quad \text{else} \\ \quad\quad SKIP \end{array}$$

- *MANAGER* terminates when there are no more elements in T and also no pending responses ($Cnt = 0$).
- While *MANAGER* is running it may send out an element of T on its output channel or it may receive new live zones and match information on its input channel. Depending on whether a receive or send event occurred, the variables T, MS, and Cnt are updated in the appropriate manner. Sending is described in process *SEND* and receiving is described in process *RECEIVE*.

$$\begin{array}{l} SEND(T, MS, Cnt) = \\ \quad \text{if } T \neq \emptyset \text{ then} \\ \quad\quad \text{let} \\ \quad\quad\quad \langle l, h \rangle = choose(T) \end{array}$$

within
if $l \geq h$ then
$SEND(T \setminus \{\langle l,h\rangle\}, MS, Cnt)$
else
$feeder!\langle l,h\rangle \rightarrow MANAGER(T \setminus \{\langle l,h\rangle\}, MS, Cnt+1)$
else
$SKIP$

$RECEIVE(T, MS, Cnt) =$
$results?\langle new_1, new_2, match\rangle \rightarrow$
if $match = -1$ then
$MANAGER(T \cup \{new_1, new_2\}, MS, Cnt-1)$
else
$MANAGER(T \cup \{new_1, new_2\}, MS \cup \{match\}, Cnt-1)$

- *SEND* sends out a non-empty live zone $\langle l,h\rangle$ on channel *feeder* and then behaves as $MANAGER(T \setminus \{\langle l,h\rangle\}, MS, Cnt+1)$. The element from T to be sent out is selected using function *choose*. If an implementer uses a stack for T, then $choose(T)$ should correspond to the element at the top of the stack. If the interval is empty, another interval is chosen and, if the interval is not empty, it is sent out and Cnt is incremented to indicate that a live zone has been sent out and that a match response is required.
- When T is empty there are no more elements to send and *SEND* simply terminates.
- *RECEIVE* accepts triples on input channel *results*. The first two elements are new live zones to be inserted into T and the third element is a match position. If the match position is -1 it indicates a mismatch and the result is ignored. For any other value, it corresponds with a start position in the text at which the pattern was matched and as such the position is recorded in MS.
- When a result is received, the counter Cnt is decremented. For every live zone that is sent out, a result should be recorded. Cnt represents the number of pending results. *MANAGER* may thus only terminate once T is empty and all pending results have been received.

In order to connect the processes together, instances of *DBUFF* and *MBUFF* are used as shown in *BDZONERS* below.

$BDZONERS(n) =$
$a.DBUFF(\langle\,\rangle, N_a)[\![^{feeder}/_{a.left}]\!]$
$[a.right \leftrightarrow in]$
$DZONERS(n)$
$[out \leftrightarrow b.left]$
$b.MBUFF(\langle\,\rangle, N_b)[\![^{results}/_{b.right}]\!]$

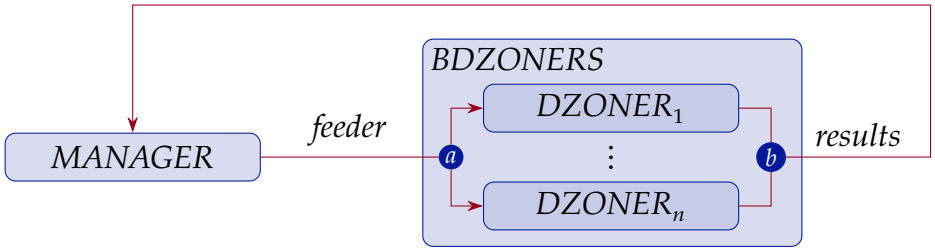


Figure 6.1: Process network for *DZ1*. *MANAGER* sends out live zones to an array of $DZONER_i$ processes. These check for matches and compute new live zone that are communicated back to *MANAGER*.

The input channel of *a.DBUFF*, *a.left*, is renamed *feeder*. The input channels *in.i* of *DZONERS* are linked to the output channels *a.right.i* of *a.DBUFF*. The output channel of each $DZONER_i$ process, *out.i*, is linked to the input channel *b.left.i* of *b.MBUFF*. The output channel of *b.MBUFF*, *b.right*, is renamed *results*. *BDZONERS* thus provides a single channel *feeder* as input interface to the array of $DZONER_i$ processes. Furthermore, the results produced by the various $DZONER_i$ are read from a single output channel, *results*.

The complete system *DZ1* may be composed as shown in Figure 6.1.

$$DZ1(S,p,n) = MANAGER(\{\langle 0, |S| - |p| + 1\rangle\}, \emptyset, 0) \underset{\{\!|results, feeder|\!\}}{\|} BDZONERS(n)$$

MANAGER and *BDZONERS* interact via the channels *feeder* and *results*. The set of live zones initially consists of the single interval $[0, |S| - |p| + 1)$, the set of matchers is empty, and the number of pending results is zero. To simplify the process definitions, it is assumed that the text S and the pattern p are known to all the relevant processes.

6.2.2 Decomposition 2

The second decomposition differs from the first in terms of which processes compute new live zones. In the first decomposition above, the $DZONER_i$ processes were responsible for determining a match and also for computing new live zones. In the present decomposition these processes only receive an index position at which to probe for a match—they do not compute new ranges and subsequently are called $MATCHER_i$. This computation of new live zones is done in a new $MANAGER'$ process. As discussed in B. W. Watson, Cleophas and Kourie (2014) this is only feasible when the shift function does not require the index position of the mismatch as input. For example, in Horspool's (1980) algorithm, only the rightmost symbol of the text at the current keyword alignment is required to compute the shift.

Horspool's shifter is used in the implementations in Section 6.3.

Similarly, Quicksearch (Hume and Sunday 1991) uses the symbol to the right of that rightmost symbol to calculate the shift.

The CSP description of the new $MATCHER_i$ processes is simple. A $MATCHER_i$ accepts an index position from its input channel $in.i$ and then arbitrarily emits on its output channel $out.i$ either the index, corresponding to a match, or -1, representing a mismatch.

$$MATCHER_i = in.i?mid \rightarrow \bigsqcap_{p \in \{-1, mid\}} (out.i!p \rightarrow MATCHER_i)$$

As before, a number of matchers may run independently.

$$MATCHERS(n) = |||_{i:1..n} MATCHER_i$$

As in Decomposition 1, instances of *DBUFF* and *MBUFF* are used to add buffers and to simplify the interface to the array of $MATCHER_i$ processes. A single *feeder* channel now distributes indices to the $MATCHER_i$ processes and all the results are multiplexed onto a *results* channel.

$$\begin{array}{l} BMATCHERS(n) = \\ \quad a.DBUFF(\langle\,\rangle, N_a)[\![^{feeder}/_{a.left}]\!] \\ \qquad [a.right \leftrightarrow in] \\ \quad MATCHERS'(n) \\ \qquad [out \leftrightarrow b.left] \\ \quad b.MBUFF(\langle\,\rangle, N_b)[\![^{results}/_{b.right}]\!] \end{array}$$

The new $MANAGER'$ process now both computes and sends out the live zones. A further modification is made: it does not obtain the match results anymore. This is now the responsibility of a new process *GATHERER*. Thus $MANAGER'$ only maintains T and sends out indices for matching.

$$\begin{array}{l} MANAGER'(T) = \\ \quad \text{if } |T| = 0 \text{ then} \\ \qquad SKIP \\ \quad \text{else} \\ \qquad \text{let} \\ \qquad\quad z = choose(T) \\ \qquad \text{within} \\ \qquad\quad SEND'(z, T \setminus \{z\}) \end{array}$$

$MANAGER'$ selects some live zone z from T based on the policy encoded in function *choose*. $MANAGER'$ then behaves as process $SEND'$ parameterised by z and by T from which z has been removed. If z is not empty, $SEND'$ computes the midpoint *mid* and sends that out for matching on channel *feeder*. It also computes two new live zones and recurses to $MANAGER'$ that now has these two new intervals added into T. The behaviour of process $SEND'$ is modelled below.

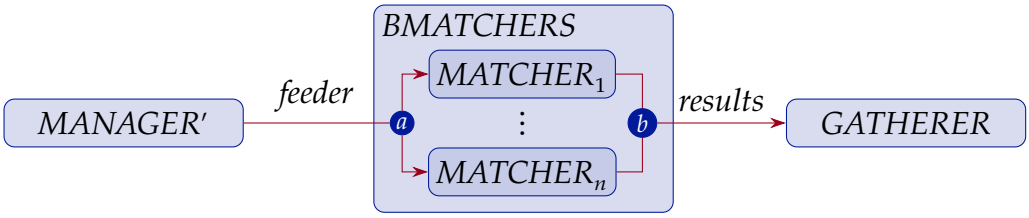


Figure 6.2: Process network diagram for *DZ2*. The *MANAGER* sends out probe points to an array of $MATCHER_i$ processes that send out the match results to *GATHERER*. $MANAGER'$ is responsible for computing new live zones.

$$
\begin{array}{l}
SEND'(\langle low, high\rangle, T) = \\
\quad \text{if } low \geq high \text{ then} \\
\qquad MANAGER'(T) \\
\quad \text{else} \\
\qquad \text{let} \\
\qquad\quad mid = \lfloor (low + high)/2 \rfloor \\
\qquad\quad left = mid - shleft() \\
\qquad\quad right = mid + shright() \\
\qquad \text{within} \\
\qquad\quad feeder!mid \rightarrow MANAGER'(T \cup \{\langle low, left\rangle, \langle right, high\rangle\})
\end{array}
$$

GATHERER is responsible for maintaining the set of indices where the pattern matches the text. It reads results from its input channel *results* and updates *MS* when necessary.

$$
\begin{array}{l}
GATHERER(MS) = \\
\quad results?match \rightarrow \\
\qquad \text{if } match = -1 \text{ then} \\
\qquad\quad GATHERER(MS) \\
\qquad \text{else} \\
\qquad\quad GATHERER(MS \cup \{match\})
\end{array}
$$

The processes can now be combined to form the system shown in Figure 6.2.

$$
DZ2(S, p, n) = (MANAGER'(\{\langle 0, |S| - |p| + 1\rangle\}) \underset{\{\!|feeder|\!\}}{\|} BMATCHERS(n)) \underset{\{\!|results|\!\}}{\|} GATHERER(\emptyset)
$$

As before, it is assumed that both the text S and the pattern p are available to all processes that reference them. The parameter n to *DZ2* is the number of $MATCHER_i$ processes that concurrently search for matches.

These two decompositions may now be used to develop a number of process-based implementations of the abstract DZ algorithm in which pattern match attempts occur concurrently.

6.3 Performance comparison

In this section various DZ implementations are compared based on their run-time performance. The two process-based decompositions of the previous section guided some of the implementations. In addition, other implementations were also developed by executing recursive calls in separate processes and by partitioning the input data.

6.3.1 Implementation

For completion, a very broad description of the sequential version of the DZ algorithm, as implemented in this thesis, now follows. For a fuller description of this version the reader may refer to B. W. Watson, Kourie and Strauss (2012). Recall from Section 6.1 that two aspects of Algorithm 6.1, the abstract sequential DZ algorithm, need to be determined in an actual implementation. These are the order in which characters are compared, represented by the function *mo*, and the choice of the shift functions *shleft* and *shright*.

In all the implementations discussed below, simple left-to-right matching is used, i.e. $mo(i) = i$. This means that, when evaluating the guard of the **do**-loop, p_0 is compared against S_{mid}, then p_1 against S_{mid+1}, etc.

The specific implementations discussed here use a right shift function corresponding to that of Horspool (1980) and a left shift function symmetrically adapted for left shifts. This means that, to determine a shift distance (to the left or right respectively), only the current probe position *mid* is required. The index *i* indicating how far across the pattern the matching had progressed (before a mismatch is encountered or a match is established) is not required to invoke the respective shift functions.

As indicated below, the shifter uses the character of the text that is aligned with the rightmost character of the pattern (i.e. the character $S_{mid+|p|-1}$) to make a right shift. Symmetrically, it uses S_{mid}, the leftmost character to make a left shift. As a result it is possible to compute a new live zone (i.e. to compute new *right* and *left* values) independently of executing the next match attempt (i.e. independently of executing the **do**-loop of the sequential algorithm).

Let $m = |p|$ denote the length of the pattern. When p is compared to the text window $S_{[mid,mid+m)}$ and the character, *ch*, is found at $S_{mid+m-1}$, then the right shift function of Horspool (1980) uses the following formula to

compute the shift distance to the right:

$$shright(ch) = \begin{cases} m & \text{if } ch \notin p_{[0,m-1)} \\ \langle \textbf{MIN } i : [0, m-1) : ch = p_{m-1-i} \rangle & \text{if } ch \in p_{[0,m-1)} \end{cases}$$

Thus, the shift to the right is m if ch is any character that does not occur in the first $m-1$ positions of the pattern. Otherwise, if the character occurs one or more times in the first $m-1$ positions of the pattern, then the shift to the right must correspond to the distance to the end of the pattern of the rightmost occurrence of that character in the pattern. Thus, if that rightmost occurrence is at p_i then the shift will be $m-1-i$.

The computation of the shift to the left is symmetrical. It is based upon the character, ch found at S_{mid} when p is compared to the text window $S_{[mid,mid+m)}$. It is given by the following formula:

$$shleft(ch) = \begin{cases} m & \text{if } ch \notin p_{(0,m-1]} \\ \langle \textbf{MIN } i : (0, m-1] : ch = p_i \rangle & \text{if } ch \in p_{(0,m-1]} \end{cases}$$

Thus, the shift to the left is m for any character ch that does not occur in the last $m-1$ positions of the pattern. Otherwise, if the character occurs one or more times in the last $m-1$ positions of the pattern, then the shift to the left must correspond the distance from the beginning of the pattern to the leftmost occurrence of that character in the pattern. Thus, if that leftmost occurrence is at p_i then the shift will be i.

The following Go code shows how the precomputed shift tables were set up. Strings are implemented as **byte** arrays. The alphabet thus has 256 elements and the length is stored in `SIGMAlen`. Given a pattern, `p`, `NewShifter(p)` creates a structure consisting of two integer arrays. These arrays store the shift distances for each symbol in the alphabet based on pattern `p`.

Strictly, they are not Go arrays, but rather slices. *Array* is used in the generic sense.

Go code 6.1: Computing shift tables.

```
func NewShifter(p []byte) *Shifter {
    sh := new(Shifter)
    sh.left = make([]int, SIGMAlen)
    sh.right = make([]int, SIGMAlen)
    m := len(p)
    for i := 0; i < SIGMAlen; i++ {
        sh.left[i] = m - 1
        sh.right[i] = m
    }
    for i := m - 1; i > 0; i-- {
        sh.left[p[i]] = i - 1
```

```
    }
    for i := 0; i < m - 1; i++ {
        sh.right[p[i]] = m - 1 - i
    }
    return sh
}
```

The left and right tables set up by `NewShifter(p)`, namely `sh.left` and `sh.right`, can then be accessed using the functions `shleft` and `shright` defined below, whereby `sh.shleft(ch)` returns the value of the array element `sh.left[ch]` and `sh.shright(ch)` returns `sh.right[ch]`.

Go code 6.2: Accessors to shift tables.

```
func (this *SK_Shifter) shleft(ch byte) int {
    return this.left[ch]
}
func (this *SK_Shifter) shright(ch byte) int {
    return this.right[ch]
}
```

Note, however, that `sh.shright(ch)` returns $shright(ch)$, whereas `sh.shleft(ch)` returns $shleft(ch) - 1$. The subtraction of 1 at this point removes the need to add 1 to the value of `left` in the recursive call on Line 15 of the code below.

The code below is a recursive implementation of the DZ algorithm. If 1 had not been subtracted in the `sh.shleft(ch)` call, then the call on Line 15 would have to change to `this.match(text, low, left + 1)`. This is in fact seen in the implementation described by B. W. Watson, Kourie and Strauss (2012).

The code corresponds with implementation DZ_{rec}

Go code 6.3: Recursive implementation of DZ.

```
1  func (this *Matcher) match(text []byte, low int,high int) int{
2      numMatches := 0
3      plen := len(this.pattern)
4      if high <= low {
5      } else {
6          mid := (low + high) / 2
7          var i int
8          for i = 0; i < plen && this.pattern[i] ==
           ↪  (text[mid+i]); i++ {
9          }
10         if i == plen {
11             numMatches++
```

```
12          }
13          left := mid - this.sh.shiftleft(text[mid])
14          right := mid + this.sh.shiftright(text[mid+plen-1])
15          numMatches += this.match(text, low, left)
16          numMatches += this.match(text, right, high)
17      }
18      return numMatches
19  }
```

The following implementations are considered in the performance comparison.

DZ$_{rec}$ — The implementation shown above of the recursive algorithm in Algorithm 6.1. Instead of recording the actual match positions *MS*, only the number of matches is counted. This modification is carried through to all the other implementations.

DZ1a — This is an implementation of Decomposition 1 with one *MANAGER* process and twelve $DZONER_i$ processes: $DZ1(S, p, 12)$. The capacity of the *feeder* buffer is 48 elements and the capacity of *results* is double that. The set of live zones T is implemented as a stack using Go slices.

DZ1b — This is essentially the same implementation as DZ1A. The only change is that multiple instances of process *DZ1* execute concurrently and independently. Instead of starting with a single live zone $[0, |S| - |p| - 1)$, partition this live zone into a number of disjoint subsets and start a *DZ1* process for each of these subsets. This is the same as subsetting the string S appropriately and searching for matches in the substrings $S_0, S_1, \ldots$. In the present implementation the number of such *DZ1* processes is twelve, with each running a single *DZONER* process. It may be expressed as the following CSP process $DZ1(S_0, p, 1) \;|||\; DZ1(S_1, p, 1) \;|||\; \cdots \;|||\; DZ1(S_{11}, p, 1)$. Each *feeder* buffer can store 8 elements and each results buffer can store 16 elements.

DZ2 — Decomposition 2 is implemented as DZ2. In this implementation twelve $MATCHER_i$ processes execute, which corresponds with $DZ2(S, p, 12)$. The *MANAGER* process stores the live zones in a stack data structure. Both the *feeder* and *results* buffers has a capacity of 1024 elements.

The reason for using twelve processes is that it equals the number of virtual cores in the machine that is used for the experiments.

DZ3 — Here DZ2 is modified so that the $MATCHER_i$ processes are created dynamically and there is not a predetermined number of such processes. Each dynamically created process receives a probe point upon creation, performs its matching attempt, communicates the result on a shared channel, and then terminates.

This implementation may cause many short-lived goroutines to be created during execution since there is no limit set for the creation of new goroutines. The channel over which the processes communicate their results has a capacity of 1024 elements.

DZ4 — The remaining implementations are not based on either of the two decompositions described in Section 6.2. In the present implementation, the two recursive calls in Algorithm 6.1 are handled by separate goroutines and the results are returned on a single channel, with capacity 1024, shared by all goroutines. This implementation could also cause many goroutines to be created during run time.

DZ$_{\text{iter}}$ — An iterative sequential implementation of the DZ algorithm is derived by B. W. Watson, Cleophas and Kourie (2014). Instead of using recursion, live zones are explicitly represented and stored in a stack. Initially the stack contains the live zone $[0, |S| - |p| - 1)$. In each iteration an element is removed from the stack and, if the interval is not empty, a match is attempted and two new live zones are then computed and added to the stack. The iteration continues until the stack is empty.

DZ5 — Multiple instances of DZ$_{iter}$ execute concurrently on subsets of the live zone. The live zone $[0, |S| - |p| - 1)$ is partitioned into twelve disjoint subsets. An instance of DZ$_{iter}$ is initialised with each of these subsets. After all the instances have terminated, their match results are aggregated to form the final result.

Similar to DZ2, twelve partitions is used in DZ5 since that is the same number as virtual cores in the machine used for the experimnets.

Note that, in the cases where the input is partitioned, the schema allows for finding matches that start in one partition and end in another.

6.3.2 Experimental setup

The above-mentioned implementations were compiled with Go 1.5.3 and executed on a Linux 3.10.17 platform with a six-core hyperthreaded CPU and 16 GB of RAM. The Go runtime was configured to run twelve processes concurrently.

Intel® Xeon® CPU E5-2630 v2 @ 2.60 GHz. See Section 2.7.3 for more information.

Several different text types were considered and patterns lengths ranged from 2 to 131 072 symbols, incremented by a factor of four: $\{2, 8, 32, \ldots, 131\,072\}$. The following text and pattern combinations were used for the experiments. All the texts are 50 MB in size.

SSB — This text simply comprises the character ‘a’ repeated 52 428 800 times. The patterns are repetitions of the symbol ‘b’. This will cause the implementations to never find matches of the pattern in the text. The shift distances will

also be maximal since no match is possible in the pattern. The iteration in Algorithm 6.1 where characters in the text and the pattern are compared will be short, since a mismatch will be found immediately.

SSA — Here, the text again comprises only 'a' characters. The patterns, however, are now also repetitions of the character 'a'. In this case the pattern will match at all possible positions and the shift distances will be minimal, since a match will be found when shifting by a single character. The iteration in Algorithm 6.1 will be maximal since the entire pattern needs to be compared in order to detect a match.

English — The remaining texts were obtained from the *Pizza & Chili* text collection (Ferragina and Navarro 2005). The first is a concatenation of English text files selected Gutenberg Project (`http://www.gutenberg.org`). The headers relating to the project has been deleted so as to leave only the real text.

Pitches — A sequence of pitch values (bytes in 0 to 127, plus a few extra special values) obtained from MIDI files freely available on the Internet.

Proteins — A sequence of newline-separated protein sequences (without descriptions, just the bare proteins). Each of the 20 amino acids is coded as one uppercase letter.

In the single symbol text types only one keyword per pattern length was used while 100 keywords per pattern length were used in the other text types. The keywords were generated by selecting an arbitrary portion of the text equal to the pattern length. This approach ensures that there is always at least one occurrence of the pattern in the text.

Each keyword and text combination was executed in all the implementations listed above. Each execution was repeated five times and the minimum run time was recorded as the data point for that keyword and text combination.

6.3.3 Results

The results of the experiments are presented next.

The mean observed speedup is shown in Table 6.1 and visualised in Figure 6.3 which also shows the mean run times against pattern length, grouped by text type. Note the logarithmic scale on the y-axis of the runtime plot.

The performance metric of interest is the speedup of the concurrent implementation over the sequential implementation, that is, the run time of the sequential implementation divided by the run time of the concurrent implementation. In the present case, there are two sequential implementations: DZ_{rec} and DZ_{iter}. All concurrent implementations apart from DZ5, are compared against DZ_{rec}. The speedup of DZ5, however, is calculated against DZ_{iter}, since DZ5 is a concurrent version of DZ_{iter}.

- For all but the SSA case, run times go down as pattern length increases. This is due to the fact that it is possible to make larger shifts and grow the dead zone more rapidly for larger patterns than smaller patterns. This is, however, not the case with the SSA texts. Since the text and the patterns always match, the number of iterations comparing characters just grows, while the shift distance is always only one symbol.
- Typically DZ5 has the lowest run time of all the implementations. It is also the only implementation that usually outperforms the sequential implementations. It is only for large patterns in the SSB case that it is outperformed by the sequential algorithms. The two sequential implementations are then second and third fastest. All the other implementations are typically slower than the sequential implementations. This is clearly seen in the speedup plots where they achieve speedups smaller than one.
- It is only in the SSA case where the sequential algorithms are outperformed by the concurrent implementations when the patterns get large enough. For DZ1B this happens when pattern length reaches 128 symbols. At 512 symbols, DZ2 and DZ3 exceed the sequential implementations and they are followed by DZ1A and DZ4 at 2048 symbols.
- In SSA, DZ5 also is outperformed by other concurrent implementations for very long patterns. Specifically, DZ1B is the best, closely followed by DZ3 when patterns are 131 072 symbols.

Table 6.1: Observed mean speedup for the DZ implementations. The table is grouped by text type and pattern length. The column T_0 contains the mean run time of the recursive sequential implementation. Note that the single symbol cases only have one observation per pattern length, so no standard deviation is shown in those cases. Recall that there are 100 observations per entry for the other cases.

Text	$\|p\|$	T_0(ms)	DZ1A	DZ1B	DZ2	DZ3	DZ4	DZ5
SSB	All	56.37	0.03 ± 0.015	0.14 ± 0.092	0.08 ± 0.026	0.05 ± 0.022	0.25 ± 0.154	2.87 ± 2.200
	2	352.60	0.02	0.11	0.05	0.03	0.17	4.95
	8	99.76	0.02	0.12	0.06	0.04	0.21	4.71
	32	33.68	0.03	0.16	0.08	0.05	0.29	5.32
	128	18.43	0.07	0.35	0.14	0.11	0.61	5.25
	512	2.23	0.04	0.17	0.07	0.06	0.30	2.80
	2048	0.50	0.03	0.13	0.07	0.05	0.26	1.32
	8192	0.13	0.04	0.10	0.08	0.06	0.22	0.62
	32 768	0.02	0.02	0.06	0.07	0.04	0.12	0.62
	131 072	0.01	0.03	0.03	0.08	0.04	0.06	0.25
SSA	All	1 977 914.11	2.18 ± 2.445	3.04 ± 2.398	2.78 ± 2.655	2.66 ± 2.593	2.30 ± 2.562	5.83 ± 0.344
	2	1054.00	0.02	0.15	0.05	0.04	0.02	6.50
	8	1720.77	0.03	0.23	0.08	0.06	0.03	6.27
	32	4798.58	0.09	0.61	0.19	0.17	0.07	5.94
	128	15 272.15	0.26	1.62	0.59	0.53	0.23	5.79
	512	54 073.28	0.78	3.45	2.16	1.91	0.86	5.65
	2048	210 315.09	2.84	4.97	5.09	4.79	3.12	5.61
	8192	836 697.01	4.38	4.84	5.30	4.95	4.95	5.57
	32 768	3 340 051.99	5.59	5.67	5.79	5.69	5.69	5.62
	131 072	13 337 244.10	5.59	5.83	5.78	5.80	5.72	5.54

Table continues on the next page…

Table 6.1…continued from previous page.

Text	$\|p\|$	T_0(ms)	DZ1A	DZ1B	DZ2	DZ3	DZ4	DZ5
English	All	129.89	0.03 ± 0.004	0.22 ± 0.043	0.07 ± 0.008	0.05 ± 0.006	0.27 ± 0.035	7.12 ± 1.041
	2	463.87	0.02 ± 0.002	0.13 ± 0.015	0.06 ± 0.004	0.04 ± 0.003	0.19 ± 0.014	6.25 ± 0.519
	8	179.86	0.03 ± 0.001	0.18 ± 0.016	0.07 ± 0.002	0.05 ± 0.002	0.25 ± 0.009	6.91 ± 0.475
	32	106.95	0.03 ± 0.001	0.25 ± 0.009	0.08 ± 0.002	0.06 ± 0.002	0.29 ± 0.009	8.13 ± 0.101
	128	85.24	0.04 ± 0.001	0.26 ± 0.010	0.08 ± 0.002	0.06 ± 0.002	0.31 ± 0.009	8.36 ± 0.095
	512	75.33	0.04 ± 0.002	0.26 ± 0.016	0.08 ± 0.004	0.06 ± 0.003	0.30 ± 0.019	8.04 ± 0.154
	2048	68.53	0.03 ± 0.002	0.25 ± 0.018	0.08 ± 0.004	0.06 ± 0.003	0.29 ± 0.020	7.41 ± 0.204
	8192	62.76	0.03 ± 0.001	0.23 ± 0.012	0.07 ± 0.003	0.05 ± 0.002	0.27 ± 0.010	6.94 ± 0.351
	32 768	63.59	0.03 ± 0.001	0.23 ± 0.009	0.07 ± 0.002	0.05 ± 0.002	0.27 ± 0.009	6.35 ± 0.811
	131 072	62.85	0.03 ± 0.001	0.23 ± 0.010	0.07 ± 0.003	0.05 ± 0.002	0.27 ± 0.011	5.65 ± 1.160
Pitches	All	71.19	0.05 ± 0.021	0.28 ± 0.124	0.10 ± 0.043	0.07 ± 0.036	0.39 ± 0.152	5.09 ± 1.655
	2	391.67	0.02 ± 0.001	0.11 ± 0.005	0.05 ± 0.001	0.04 ± 0.001	0.18 ± 0.007	5.44 ± 0.167
	8	124.86	0.03 ± 0.001	0.14 ± 0.010	0.06 ± 0.002	0.04 ± 0.001	0.23 ± 0.007	5.62 ± 0.333
	32	54.26	0.04 ± 0.001	0.21 ± 0.018	0.08 ± 0.003	0.06 ± 0.002	0.31 ± 0.010	6.59 ± 0.489
	128	30.63	0.05 ± 0.007	0.32 ± 0.037	0.11 ± 0.012	0.08 ± 0.010	0.45 ± 0.061	6.85 ± 0.580
	512	16.79	0.07 ± 0.038	0.43 ± 0.161	0.14 ± 0.091	0.11 ± 0.079	0.58 ± 0.204	6.21 ± 0.677
	2048	7.44	0.05 ± 0.013	0.35 ± 0.101	0.11 ± 0.021	0.08 ± 0.018	0.45 ± 0.116	5.26 ± 1.415
	8192	5.40	0.05 ± 0.014	0.32 ± 0.081	0.11 ± 0.028	0.08 ± 0.022	0.43 ± 0.112	4.06 ± 1.293
	32 768	4.88	0.05 ± 0.008	0.31 ± 0.063	0.11 ± 0.012	0.08 ± 0.010	0.41 ± 0.075	3.42 ± 0.968
	131 072	4.78	0.05 ± 0.010	0.33 ± 0.068	0.12 ± 0.019	0.09 ± 0.014	0.45 ± 0.083	2.34 ± 0.669
Proteins	All	132.01	0.03 ± 0.004	0.22 ± 0.039	0.07 ± 0.008	0.05 ± 0.006	0.26 ± 0.030	7.40 ± 0.668
	2	453.72	0.02 ± 0.001	0.13 ± 0.008	0.06 ± 0.002	0.04 ± 0.002	0.20 ± 0.006	6.18 ± 0.272
	8	169.62	0.03 ± 0.000	0.18 ± 0.010	0.07 ± 0.001	0.05 ± 0.001	0.25 ± 0.004	6.73 ± 0.231
	32	101.90	0.03 ± 0.001	0.25 ± 0.007	0.08 ± 0.001	0.06 ± 0.001	0.29 ± 0.006	7.97 ± 0.113
	128	88.29	0.03 ± 0.001	0.26 ± 0.010	0.08 ± 0.002	0.06 ± 0.002	0.30 ± 0.010	8.13 ± 0.114
	512	82.84	0.03 ± 0.001	0.25 ± 0.009	0.08 ± 0.002	0.06 ± 0.002	0.29 ± 0.010	7.98 ± 0.109
	2048	78.45	0.03 ± 0.002	0.23 ± 0.015	0.07 ± 0.004	0.05 ± 0.003	0.27 ± 0.017	7.67 ± 0.141
	8192	70.90	0.03 ± 0.001	0.22 ± 0.007	0.07 ± 0.002	0.05 ± 0.002	0.25 ± 0.010	7.61 ± 0.150
	32 768	70.43	0.03 ± 0.001	0.22 ± 0.011	0.07 ± 0.003	0.05 ± 0.002	0.26 ± 0.010	7.35 ± 0.447
	131 072	71.94	0.03 ± 0.001	0.22 ± 0.009	0.07 ± 0.002	0.05 ± 0.001	0.26 ± 0.007	6.99 ± 0.443

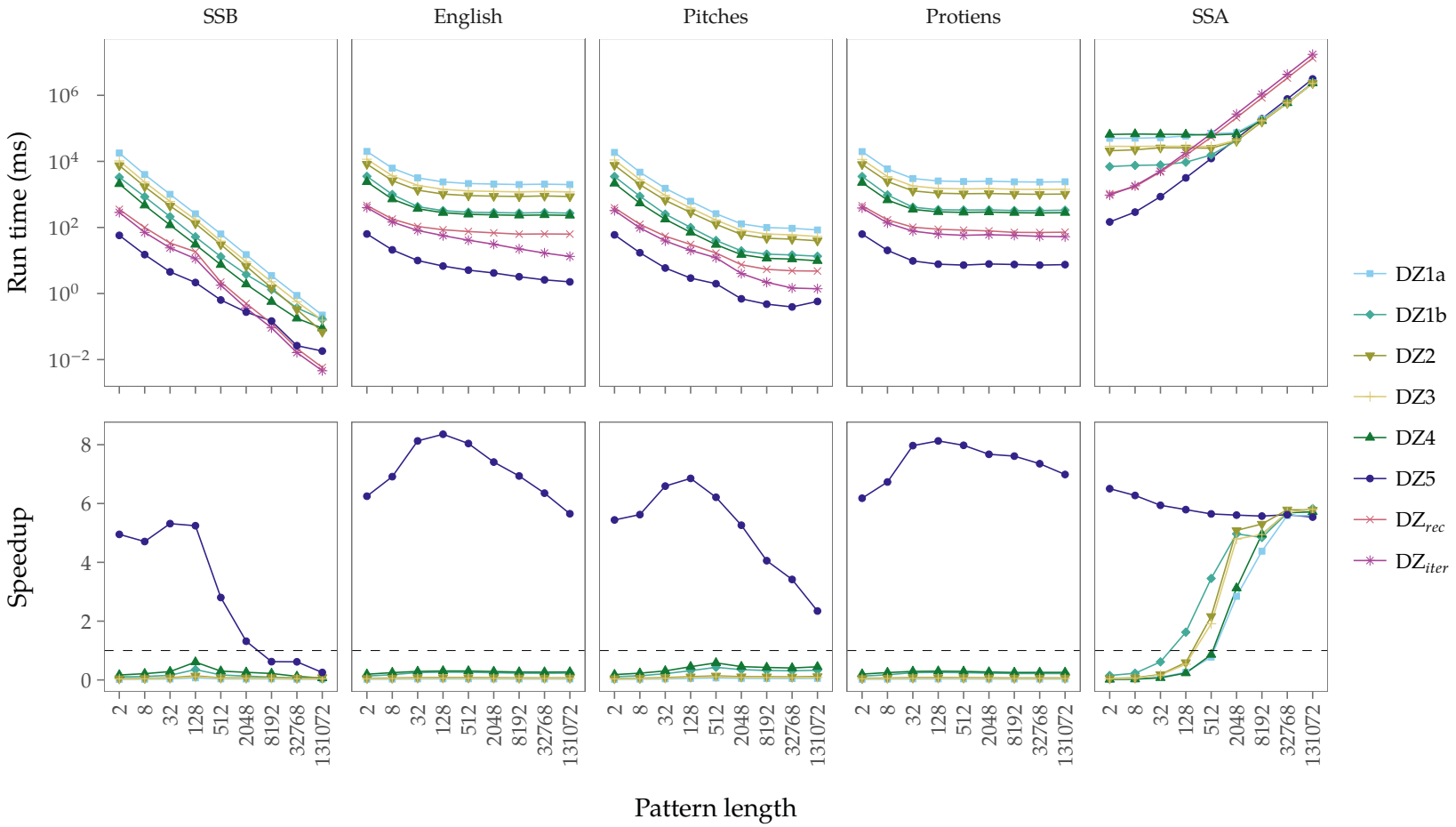


Figure 6.3: Mean run time and speedup of the DZ implementations plot against pattern length, grouped by text type. Note the logarithmic scale on the y-axis of the run-time plot. Each point is the mean over 100 observations.

- DZ5 achieved speedup greater than one on nearly all cases. It is only when $|p| > 2048$ in the SSB case that the sequential implementations beat it. This is probably due to the fact that the run times are very short because of the large shift distances. The cost of consolidating the results then outweighs the benefits of splitting the matching space among processes.
- It is interesting that the runtime of the sequential algorithms grow exponentially in the SSA case. However, the concurrent implementations' curves are flat up to pattern lengths of 512 symbols.

6.4 Conclusion

In the present case study, the process-based implementations did not perform well. Only DZ5 produced speedup in most cases. It seems as if the costs of setting up the process-based systems outweigh the benefits of matching concurrently. The tasks are rather small. The only real "work" that happens concurrently is the matching of characters. This is not computationally intensive. For this reason, following an embarrassingly parallel approach of splitting the input into smaller parts, worked better.

7 Conclusion

> Ring the bells that still can ring,
> Forget your perfect offering.
> There is a crack in everything—
> That's how the light gets in.
>
> *Leonard Cohen*

The thesis draws to a close with this, the final chapter. Section 7.1 summarises the work that has been done. It then briefly reflects on the process-based method that has been explored, commenting on strengths, on lessons that have been learned and on limitations experienced. This is followed by suggestions for future work in Section 7.2.

7.1 Reflection

The primary aim of this work was to explore whether a process-based approach to designing and implementing algorithmic solutions on multicore hardware platforms improves the effectiveness of the resulting software. Would the resulting programs effectively use the available cores to deliver performance improvements? To explore the approach's effectiveness, a case study was conducted on several algorithms from the Stringology domain. The case study affirms that such a process-based approach to implementing algorithms is indeed a useful software development technique and can be effectively used for exploring a wide-range of concurrent solutions. The empirical results reported in Chapters 3–6 provide *prima facie* evidence that runtime performance is frequently improved, sometimes very significantly, when employing this software development technique.

In summary, the project entailed the following.

1. Each of four problems with known sequential algorithmic solutions were studied. Each was solved by decomposing it into a number of collaborating concurrent processes that produce the same results as its sequential counterpart. These decompositions were encoded in the process algebra CSP. In most cases several variants of the initial concurrent solution were also developed.

2. These decompositions, as well as the sequential algorithms, were implemented in the Go programming language. The implementations will be made available online at `http://fastar.org`.

3. The run-time performance of each implementation was investigated and presented. It was found that the concurrent implementations can outperform the sequential implementations. However, some algorithms require more fine-grained concurrency which resulted in overheads that erase the benefits gained by using multiple processors.

4. The empirical results together with the experience of developing concurrent versions of the sequential algorithms provide *prima facie* evidence that such a process-based approach is feasible and generally effective for delivering improved performance on multicore computers.

In retrospect, the following broad approach for developing the concurrent implementations emerged. The first step was to inspect the sequential algorithm for opportunities to exploit concurrency. In general, task-level parallelism was favoured over data-level parallelism. However, when data-level parallelism was deemed appropriate, it was exploited (e.g. 'Split' in Chapter 3 and DZ5 in Chapter 6).

Tasks that can execute independently were identified and modelled as CSP processes. These processes also encapsulate data and state. Data elements that needed to be shared with other processes were identified and CSP channels were defined to facilitate the sharing. Only the "owner" of a data structure may modify it. This exercise gave an idea of the required processes and their interactions. These structures can be represented as process diagrams that were exemplified in this thesis. Such diagrams were found to be pedagogically useful for strengthening insight and intuition. When multiple design options presented themselves, a limited number of additional variants were defined.

These individual processes were then refined into a high-level CSP specification. Specific computations were abstracted away—the only concern was how the processes interact and modify their states to achieve the results of of the sequential algorithm.

An aspect that proved challenging was how to decide that the computation is complete both at the CSP specification level and at the implementation level. Typically, the algorithms run until all the elements of some collection have been processed. Some algorithms initially populate this structure and then the elements monotonically decrease as the system executes. Others, on the other hand, dynamically add and remove elements to the structure, complicating termination.

In the CSP models, the termination question was not addressed completely. Only the main "driver" process terminates, based on a terminating condition. Other "supporting" processes do not terminate. It was decided to address termination during implementation. This results in simpler, more comprehensible CSP processes.

If needed for example, if one wanted to run the CSP model in a model checker such as FDR3, it would be fairly simple, though perhaps tedious, to terminate all the CSP processes using the following approach. Once the "driver" process reaches its termination condition, it sends out a termination signal on all its output channels and terminates. Whenever a process receives this signal, it forwards the signal and terminates. In such a way the entire process network may be "poisoned" to terminate.

This is essentially how the Go implementations terminate, though an explicit signal is not required. A downstream process executes until its input channel is closed by an upstream process.

Once a CSP model reached a certain level of maturity, it was used to guide a Go implementation. The following heuristics were used when translating from CSP to Go. Processes were programmed as functions that could then be started using the **`go`** keyword. Since Go supports channels, these were used to trivially implement CSP channels. In addition, Go supports buffered channels and this feature allows one to implement CSP buffers without additional Go buffer processes. Go's **`select`** statement was used to implement CSP's external choice.

At times, the implementation effort suggested changes to the CSP models and at other times it suggested additional design variants. These changes were used to update the CSP descriptions and/or create additional variants of the existing CSP models which, in turn, necessitated further implementation work. Such iterative incremental development is, of course, common in most software development paradigms.

Finally, when presenting the CSP models in the chapters, it was at times convenient to adapt some of the CSP descriptions for presentation purposes. For example, variable or channel names may have changed to fit the narrative. Also large process descriptions may have been refined into a smaller subprocesses to enhance comprehensibility.

CSP was found to be useful for broadly describing the process decomposition. By abstracting away from details such as process termination and the details of data structures, one is able to keep process descriptions reasonably uncluttered and to focus on the composition of- and interaction between processes. What is more, it seems to have an effect on one's thinking. Initially it is hard to stay in a mode of thinking in terms of processes and interaction together with states and state transitions. One attempts to express processes in an imperative style due to years of imperative programming conditioning. As the project progressed, the thinking in a process-oriented manner became more natural.

Go served well as implementation language, despite the fact that it had to be learnt as the project progressed. Since Go features channels, it is fairly easy to map the CSP descriptions onto Go code, albeit in an ad-hoc manner. Developing more structured heuristics for the translation from CSP into Go is left as future work. It seems as if channels are more effectively used for synchronisation than for high-intensity data exchange. Furthermore, goroutines appear to be very lightweight, as witnessed in the relatively good performance of, for example, Variant 1a in Chapter 3 in which each element in a level was processed by its own goroutine. However, there also appears to be a point at which too many goroutines are detrimental to performance as demonstrated by the performance of Min3 in Chapter 5.

In a project with limited time and resources, certain compromises had to be made. The following limitations can be pointed out.

- As discussed above, process termination was not completely addressed in the CSP models.
- Verification using formal- or semi-formal methods were not done. Using a process algebra invites the use of model checking tools. In the present project the output of a process-based implementation was compared against the that of the corresponding sequential implementation. If the output agrees, the result is accepted. Nevertheless, there is a small chance of deadlock and livelock, although never encountered.
- The Go implementation was not done by a Go expert. Rather, the language was learnt as the project progressed. This means that idioms, conventions, and potential optimisations of Go experts might not have been implemented. However, the fact that decent performance was achieved could be seen as a commendation of Go for its simplicity and elegance.
- The performance analysis only considers the run times of the implementations. Other measures such as memory consumption and cache effects were excluded from the scope of the present project.
- The computer used for the experiments only had six cores. A larger number of cores could perhaps yield other interesting behaviours.
- Since the sequential algorithms were used as the starting point, one is inclined to remain trapped in a sequential mode of thinking. Would one obtain different solutions if one went back to the problem and attempt to find a concurrent algorithm from scratch?

These limitations naturally lead to potential extensions of the current project.

7.2 Future work

The project suggests a number of dimensions in which the work may be extended. In the present project, the domain from which algorithms were selected is Stringology. An obvious step is to find process-based implementations of more algorithms in this domain, thereby covering more of this particular domain. A logical progression is then to extend the effort into other problem domains where process-based implementations might be beneficial.

Instead of (or in addition to) going wider, one could go deeper into the present case studies. More detailed performance analyses may be carried out, focussing, *inter alia*, on the extent to which memory consumption and cache utilisation characteristics affect performance. Particular performance bottlenecks may then be identified and rectified. Atachiants, Doherty and Gregg's (2016) taxonomy of performance problems could be used to guide such an effort.

The present Go implementations are intended as proof-of-concept. Different designs or data structures could be used to improve performance at the implementation level. Process-oriented patterns (Sampson 2008) may prove useful to improve implementations.

In terms of implementation, one might also attempt to develop tools to support a more mechanical or automated translation from CSP specification to Go implementation.

CSP was used only as a notation for describing process models. An advantage of using such a formalism that was not exploited in the present work, is the ability to verify correctness properties of the system before implementation. In future this verification could be added to the design process. One could then check, for example, termination, liveness, and other correctness properties using model checking tools such as FDR3.

Even though it was sought to view the domain afresh from a process-based perspective, obviously the resulting process-based architecture and code were heavily influenced by the existing sequential algorithms. Ignoring the existing sequential algorithm and going back to the problem could yield different process-based solutions. In addition may it be also to use other formalisms to arrive at and reason about concurrent solutions to the problem. For instance, it might be possible to use the correctness-by-construction approach to programming (Kourie and B. W. Watson 2012) together with concurrent separation logic (Brookes 2007; O'Hearn 2007) to derive correct concurrent programs from problem specifications.

In summary, the work described in this thesis affirms that it is often possible to improve algorithmic efficiency on multicore processors, sometimes

radically, by following a process-based development approach to implementing the software. CSP was found to be a handy notation for expressing a process-based design as it emerges from considering known sequential implementations.

Other than using CSP, no specific attempt has been made to develop a set of heuristics, and/or a checklist of "things to do" or "steps to follow" in order to arrive at an effective, efficient, maintainable, understandable (i.e. high-quality) process-based implementation. It will take significantly more person-power and experience to come up with a credible software process (whether of an agile variety such as extreme programming (XP), or a more structured variety such as RUP) to be followed for process-based software construction.

One of the contributions of this work is to provide solid evidence that developing such a well thought-out software process, advocating it into the software engineering community and ensuring that future developers follow it, could well contribute to significantly more efficient code in many domains, without requiring more expenditure on improved hardware.

Bibliography

Adve, S. V. and K. Gharachorloo (1996). 'Shared memory consistency models: a tutorial'. In: *Computer* 29.12, pp. 66–76.

Agha, G. A. (1985). 'Actors: a model of concurrent computation in distributed systems'. PhD thesis. University of Michigan, Ann Arbor, MI.

Aho, A. V. and M. J. Corasick (1975). 'Efficient String Matching: An Aid to Bibliographic Search'. In: *Communications of the ACM* 18.6, pp. 333–340. URL: http://doi.acm.org/10.1145/360825.360855.

Aho, A. V., J. E. Hopcroft and J. D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Alexandrescu, A. (2010). *The D Programming Language*. Pearson Education.

Almeida, M., N. Moreira and R. Reis (2012). 'Finite Automata Minimization'. In: *Handbook of Finite State Based Models and Applications.* Ed. by J. Wang. Chapman and Hall/CRC, pp. 145–169. URL: http://dx.doi.org/10.1201/b13055-8.

Almeida, M., N. Moreira and R. Reis (2014). 'Incremental DFA minimisation'. In: *RAIRO - Theoretical Informatics and Applications* 48.2, pp. 173–186. URL: http://dx.doi.org/10.1051/ita/2013045.

Ambler, S. W. (2016). *The Object-Relational Impedance Mismatch*. URL: http://www.agiledata.org/essays/impedanceMismatch.html (visited on 29/08/2016).

Armstrong, J. (2007). 'A History of Erlang'. In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: ACM, pp. 6-1–6-26. URL: http://doi.acm.org/10.1145/1238844.1238850.

Arnold, J. B. (2016). *ggthemes: Extra Themes, Scales and Geoms for 'ggplot2'*. R package version 3.3.0. URL: https://CRAN.R-project.org/package=ggthemes.

Atachiants, R., G. Doherty and D. Gregg (2016). 'Parallel Performance Problems on Shared-Memory Multicore Systems: Taxonomy and Observation'. In: *IEEE Transactions on Software Engineering* 42.8, pp. 764–785.

Awid, K., L. Cleophas and B. W. Watson (2016). 'Using Human Computation in Dead-zone based 2D Pattern Matching'. In: *Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2015*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 22–32. URL: http://www.stringology.org/event/2016/p03.html.

Baeten, J. C. M. (2005). 'A brief history of process algebra'. In: *Theoretical Computer Science* 335.2, pp. 131–146.

Baeten, J. C. M., T. Basten and M. A. Reniers (2010). *Process algebra: equational theories of communicating processes*. Vol. 50. Cambridge university press.

Barnes, F. R. M. and P. H. Welch (2016). *occam-pi: blending the best of CSP and the pi-calculus*. URL: http://www.cs.kent.ac.uk/projects/ofa/kroc/ (visited on 16/09/2016).

Bassino, F., J. David and C. Nicaud (2007). 'REGAL: A Library to Randomly and Exhaustively Generate Automata'. In: *Implementation and Application of Automata: 12th International Conference, CIAA 2007, Praque, Czech Republic, July 16-18, 2007, Revised Selected Papers*. Ed. by J. Holub and J. Žďárek. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 303–305. URL: http://dx.doi.org/10.1007/978-3-540-76336-9_28.

Bassino, F., J. David and C. Nicaud (2008). 'Random generation of possibly incomplete deterministic automata.' In: *Génération Aléatoire de Structures COMbinatoires*, pp. 31–40.

Bergstra, J. A. and J. W. Klop (1982). *Fixed point semantics in process algebras*. swireport IW 206/82. preprint. Amsterdam: Stichting Mathematisch Centrum.

Bergstra, J. A. and J. W. Klop (1984). 'Process algebra for synchronous communication'. In: *Information and control* 60.1-3, pp. 109–137.

Berstel, J., L. Boasson, O. Carton and I. Fagnot (2010). 'Minimization of Automata'. In: *CoRR* abs/1010.5318. URL: http://arxiv.org/abs/1010.5318.

Boyer, R. S. and J. S. Moore (1977). 'A fast string searching algorithm'. In: *Communications of the ACM* 20.10, pp. 62–72.

Brinch Hansen, P. (2002). 'The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls'. In: ed. by P. Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc. Chap. The Invention of Concurrent Programming, pp. 3–61. URL: http://dl.acm.org/citation.cfm?id=762971.762973.

Brookes, S. D. (1983). 'A Model for Communicating Sequential Processes'. PhD thesis. Oxford University.

Brookes, S. D. (2007). 'A semantics for concurrent separation logic'. In: *Theoretical Computer Science* 375.1, pp. 227–270. URL: http://www.sciencedirect.com/science/article/pii/S0304397506009248.

Brown, N. C. C. (2008). 'Communicating Haskell Processes: Composable Explicit Concurrency Using Monads'. In: *Communicating Process Architectures 2008*. Ed. by P. H. Welch, S. Stepney, F. A. Polack, F. R. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink and A. T. Sampson. Vol. 66. Concurrent Systems Engineering. Amsterdam, The Netherlands: IOS Press, pp. 67–83. URL: http://kar.kent.ac.uk/24103/.

Brown, N. C. C. and P. H. Welch (2003). 'An Introduction to the Kent C++CSP Library'. In: *Communicating Process Architectures 2003*. Ed. by J. F. Broenink and G. H. Hilderink. Vol. 61. Concurrent Systems Engineering

Series. Conference Information: 26th WoTUG Technical Meeting UNIV TWENTE, ENSCHEDE, NETHERLANDS, SEP 07-10, 2003. Amsterdam, The Netherlands: IOS Press, pp. 139–156. URL: http://kar.kent.ac.uk/13921/.

Brzozowski, J. A. (1964). 'Derivatives of Regular Expressions'. In: *Journal of the ACM* 11.4, pp. 481–494.

Buhr, P. A. and A. S. Harji (2005). 'Concurrent urban legends'. In: *Concurrency and Computation: Practice and Experience* 17.9, pp. 1133–1172. URL: http://dx.doi.org/10.1002/cpe.885.

Burgstaller, B., Y.-S. Han, M. Jung and Y. Ko (2011). *On the parallelization of DFA membership tests*. Tech. rep. Technical Report. TR-0003, Department of Computer Science, Yonsei University, Seoul 120–749, Korea. URL: http://elc.yonsei.ac.kr/PDFA.pdf.

Choi, H. and B. Burgstaller (2013). 'Non-blocking parallel subset construction on shared-memory multicore architectures'. In: *Proceedings of the Eleventh Australasian Symposium on Parallel and Distributed Computing-Volume 140*. Australian Computer Society, Inc., pp. 13–20.

Cleophas, L. (2003). 'Towards SPARE Time: A New Taxonomy of Keyword Pattern Matching Algorithms'. MA thesis. Faculty of Computing Science, Eindhoven University of Technology, the Netherlands.

Crochemore, M. A. and W. Rytter (2003). *Jewels of Stringology*. World Scientific Publishing Company.

D (2016). *D programming language*. URL: http://dlang.org/ (visited on 07/11/2016).

Dijkstra, E. W. (1965). 'Solution of a Problem in Concurrent Programming Control'. In: *Communications of the ACM* 8.9, p. 569. URL: http://doi.acm.org/10.1145/365559.365617.

Dijkstra, E. W. (1968). 'Cooperating sequential processes'. URL: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF. Published as: 'Cooperating sequential processes'. In: *Programming Languages: NATO Advanced Study Institute*. Ed. by F. Genuys. Academic Press, 1968, pp. 43–112.

Dijkstra, E. W. (1971). 'Hierarchical Ordering of Sequential Processes'. In: *Acta Informatica* 1, pp. 115–138. URL: http://dx.doi.org/10.1007/BF00289519.

Dijkstra, E. W. (1975). 'Guarded Commands, Nondeterminacy and Formal Derivation of Programs'. In: *Communications of the ACM* 18.8, pp. 453–457. URL: http://doi.acm.org/10.1145/360933.360975.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall.

Drepper, U. (2007). *What Every Programmer Should Know About Memory*. URL: http://lwn.net/Articles/250967/ (visited on 14/11/2016).

Erlang (2016). *Erlang programming language*. URL: http://www.erlang.org/ (visited on 07/11/2016).

Faro, S. and T. Lecroq (2013). 'The Exact Online String Matching Problem: A Review of the Most Recent Results'. In: *ACM Comput. Surv.* 45.2, 13:1–13:42. URL: http://doi.acm.org/10.1145/2431211.2431212.

Farrel, K. (2012). *Intel® Xeon® Processor E5-2600/4600 Product Family Technical Overview*. URL: https://software.intel.com/en-us/articles/intel-xeon-processor-e5-26004600-product-family-technical-overview (visited on 29/09/2016).

Ferragina, P. and G. Navarro (2005). *Pizza&Chili Corpus: Compressed Indexes and their Testbeds*. URL: http://pizzachili.dcc.uchile.cl (visited on 16/08/2016).

Flynn, M. J. (1966). 'Very high-speed computing systems'. In: *Proceedings of the IEEE* 54.12, pp. 1901–1909.

Flynn, M. J. (1972). 'Some Computer Organizations and Their Effectiveness'. In: *IEEE Transactions on Computers* C-21.9, pp. 948–960.

Gerrand, A. (2013). *Concurrency is not parallelism*. URL: https://blog.golang.org/concurrency-is-not-parallelism (visited on 02/10/2016).

Gibson-Robinson, T., P. Armstrong, A. Boulgakov and A. Roscoe (2015). 'FDR3: a parallel refinement checker for CSP'. English. In: *International Journal on Software Tools for Technology Transfer*, pp. 1–19. URL: http://dx.doi.org/10.1007/s10009-015-0377-y.

Go (2016). *The Go Programming Language*. URL: https://golang.org (visited on 29/08/2016).

Go FAQ (2016). *The Go Programming Language: Frequently Asked Questions (FAQ)*. URL: https://golang.org/doc/faq (visited on 29/08/2016).

Go Language Specification (2016). *The Go Programming Language Specification*. URL: https://golang.org/ref/spec (visited on 29/08/2016).

Gries, D. (1980). *The Science of Computer Programming*. second. Springer-Verlag.

Hanneforth, T. and B. W. Watson (2012). 'An Efficient Parallel Determinisation Algorithm for Finite-state Automata'. In: *Stringology*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 42–52.

Harper, R. (2011). *Parallelism Is Not Concurrency*. URL: https://existentialtype.wordpress.com/2011/03/17/parallelism-is-not-concurrency/ (visited on 24/09/2016).

Héam, P. and C. Nicaud (2011). 'Seed: An Easy-to-Use Random Generator of Recursive Data Structures for Testing'. In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*. IEEE Computer Society, pp. 60–69. URL: http://dx.doi.org/10.1109/ICST.2011.31.

Hennessy, J. L. and D. Patterson (2011). *Computer Architecture: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hewitt, C. (1977). 'Viewing control structures as patterns of passing messages'. In: *Artificial Intelligence* 8.3, pp. 323–364.

Hoare, C. A. R. (1972). 'Towards a Theory of Parallel Programming'. In: *Operating System Techniques*. Academic Press, pp. 61–71.

Hoare, C. A. R. (1978). 'Communicating Sequential Processes'. In: *Communications of the ACM* 21.8, pp. 666–677. URL: http://doi.acm.org/10.1145/359576.359585.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. London: Prentice-Hall.

Hoare, C. A. R. (2004). *Communicating Sequential Processes*. Ed. by J. Davis. (Electronic version). URL: http://www.usingcsp.com/cspbook.pdf (visited on 16/09/2016).

Holub, J. and S. Štekr (2009). 'On Parallel Implementations of Deterministic Finite Automata'. In: *Implementation and Application of Automata*. Ed. by S. Maneth. Vol. 5642. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 54–64. URL: http://dx.doi.org/10.1007/978-3-642-02979-0_9.

Holub, J. and J. Zdárek, eds. (2014). *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague.

Hopcroft, J. E., R. Motwani and J. D. Ullman (2007). *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson/Addison Wesley.

Hopcroft, J. E. and J. D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Horspool, R. N. (1980). 'Practical fast searching in strings'. In: *Software — Practice & Experience* 10.6, pp. 501–506.

Huang, M., M. Mehalel, R. Arvapalli and S. He (2013). 'An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® ; Xeon®; Processor E5 Family'. In: *IEEE Journal of Solid-State Circuits* 48.8, pp. 1954–1962.

Hum, H. H. J. and J. R. Goodman (2005). 'Forward state for use in cache coherency in a multiprocessor system'. 6,922,756 (US Patent). URL: https://www.google.com/patents/US6922756 (visited on 19/10/2016).

Hume, A. and D. Sunday (1991). 'Fast string searching'. In: *Software — Practice & Experience* 21.11, pp. 1221–1248.

Intel Corporation (2009). *An Introduction to the Intel® QuickPath Interconnect*. URL: http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html (visited on 11/10/2016).

Intel Corporation (2014a). *Intel® Xeon® Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual*. Reference Number: 329468-002. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/xeon-e5-2600-v2-uncore-manual.pdf (visited on 15/10/2016).

Intel Corporation (2014b). *Intel® Xeon® Processor E5-1600/E5-2600/E5-4600 v2 Product Families: Datasheet Volume One*. Reference Number: 329187-003I. URL: http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-1.pdf (visited on 19/10/2016).

Intel Corporation (2016a). *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order Number: 248966-033. URL: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (visited on 27/09/2016).

Intel Corporation (2016b). *Intel® Xeon® Processor E5-2630 v2 (15M Cache, 2.60 GHz)*. URL: http://ark.intel.com/products/75790/Intel-Xeon-Processor-E5-2630-v2-15M-Cache-2_60-GHz (visited on 29/09/2016).

JáJá, J. and K. W. Ryu (1996). 'An Optimal Randomized Parallel Algorithm for the Single Function Coarsest Partition Problem'. In: *Parallel Processing Letters* 6.2, pp. 187–193.

James, D. (2012). 'Intel Ivy Bridge unveiled—The first commercial tri-gate, high-k, metal-gate CPU'. In: *Proceedings of the IEEE 2012 Custom Integrated Circuits Conference*, pp. 1–4.

Kernighan, B. W. (2005). *A Descent into Limbo*. URL: http://doc.cat-v.org/inferno/4th_edition/limbo_language/descent (visited on 18/09/2016).

Knuth, D. E., J. Morris and V. R. Pratt (1977). 'Fast pattern matching in strings'. In: *SIAM Journal of Computing* 6.2, pp. 323–350.

Ko, Y., M. Jung, Y.-S. Han and B. Burgstaller (2012). 'A speculative parallel DFA membership test for multicore, SIMD and cloud computing environments'. In: *International Journal of Parallel Programming*, pp. 1–34.

Kourie, D. G. and B. W. Watson (2012). *The Correctness-by-Construction Approach to Programming*. Springer Verlag.

Kuehner, M. (2016). *LaTeX Thesis Template*. URL: https://bedienhaptik.de/latex-template/ (visited on 26/10/2016).

Lee, E. A. (2006). 'The problem with threads'. In: *IEEE Computer* 39.5, pp. 33–42.

Lynch, N. A. and M. R. Tuttle (1989). 'An introduction to input/output automata'. In: *CWI Quarterly*, pp. 219–246.

Mauch, M. (2016). 'An Investigation of Dead-Zone Pattern Matching Algorithms'. Thesis (MA). Stellenbosch University. Faculty of Arts and Social Sciences. Dept. of Information Science. URL: http://hdl.handle.net/10019.1/98487.

Mauch, M., D. G. Kourie, B. W. Watson and T. Strauss (2012). 'Performance assessment of dead-zone single keyword pattern matching'. In: *2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012*. Ed. by J. H. Kroeze and R. de Villiers. ACM, pp. 59–68. URL: http://doi.acm.org/10.1145/2389836.2389844.

McDougall, R. (2005). 'Extreme software scaling'. In: *ACM Queue* 3.7, pp. 36–46.

Meade, A., J. Buckley and J. J. Collins (2011). 'Challenges of Evolving Sequential to Parallel Code: An Exploratory Review'. In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*. IWPSE-EVOL '11. Szeged,

Hungary: ACM, pp. 1–5. URL: http://doi.acm.org/10.1145/2024445.2024447.

Milner, R. (1989). *Communication and Concurrency*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Milner, R., J. Parrow and D. Walker (1992a). 'A calculus of mobile processes, I'. In: *Information and Computation* 100.1, pp. 1–40. URL: http://www.sciencedirect.com/science/article/pii/0890540192900084.

Milner, R., J. Parrow and D. Walker (1992b). 'A calculus of mobile processes, II'. In: *Information and Computation* 100.1, pp. 41–77. URL: http://www.sciencedirect.com/science/article/pii/0890540192900095.

Moler, C. (2013). *The Intel Hypercube, part 2, reposted*. URL: http://blogs.mathworks.com/cleve/2013/11/12/the-intel-hypercube-part-2-reposted/ (visited on 04/10/2016).

Molka, D., D. Hackenberg, R. Schöne and W. E. Nagel (2015). 'Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture'. In: *Parallel Processing (ICPP), 2015 44th International Conference on*, pp. 739–748.

MPI Forum (2016). URL: http://mpi-forum.org/ (visited on 07/11/2016).

O'Hearn, P. W. (2007). 'Resources, concurrency, and local reasoning'. In: *Theoretical Computer Science* 375.1, pp. 271–307. URL: http://www.sciencedirect.com/science/article/pii/S030439750600925X.

OpenMP (2016). *The OpenMP® API specification for parallel programming*. URL: http://openmp.org (visited on 28/10/2016).

Owens, S., J. Reppy and A. Turon (2009). 'Regular-expression derivatives reexamined'. In: *Journal of Functional Programming* 19.2, pp. 173–190.

Papazian, I. E., S. Kottapalli, J. Baxter, J. Chamberlain, G. Vedaraman and B. Morris (2015). 'Ivy Bridge Server: A Converged Design'. In: *IEEE Micro* 35.2, pp. 16–25.

Patterson, D. (2010). *The Trouble With Multicore: Chipmakers are busy designing microprocessors that most programmers can't handle*. URL: http://spectrum.ieee.org/computing/software/the-trouble-with-multicore (visited on 20/10/2016).

Petri, C. A. and W. Reisig (2008). 'Petri net'. In: *Scholarpedia* 3.4. revision #91646, p. 6477. URL: http://www.scholarpedia.org/article/Petri_net (visited on 01/11/2016).

Pike, R. (1990). 'The implementation of Newsqueak'. In: *Software: Practice and Experience* 20.7, pp. 649–659. URL: http://dx.doi.org/10.1002/spe.4380200703.

Pike, R. (2012a). *Concurrency is not Parallelism*. URL: https://talks.golang.org/2012/waza.slide#1 (visited on 02/10/2016).

Pike, R. (2012b). *Go at Google: Language Design in the Service of Software Engineering*. URL: https://talks.golang.org/2012/splash.article (visited on 25/08/2016).

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: http://www.R-project.org/.

Ravikumar, B. and X. Xiong (1996). 'A Parallel Algorithm for Minimization of Finite Automata'. In: *IPPS*. IEEE Computer Society, pp. 187–191.

Red Hat (2016). *Hibernate. Everything data.* URL: http://hibernate.org/ (visited on 28/10/2016).

Revuz, D. (1992). 'Minimisation of acyclic deterministic automata in linear time'. In: *Theoretical Computer Science* 92, pp. 181–189.

Reynolds, J. C. (2002). 'Separation logic: a logic for shared mutable data structures'. In: *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pp. 55–74.

Ritchie, D. M. (2005). *The Limbo Programming Language*. URL: http://doc.cat-v.org/inferno/4th_edition/limbo_language/limbo (visited on 18/09/2016).

Roscoe, A. W. (1982). 'A mathematical theory of communicating processes'. D.Phil thesis. Oxford University. URL: http://www.cs.ox.ac.uk/people/bill.roscoe/publications/2.pdf.

Roscoe, A. W. (2010). *Understanding Concurrent Systems*. 1st. New York, NY, USA: Springer-Verlag New York, Inc.

Roscoe, A. W. and G. Barrett (1989). 'Unbounded nondeterminism in CSP'. In: *Proceedings of MFPS89*. LNCS 298. Springer. URL: http://www.cs.ox.ac.uk/people/bill.roscoe/publications/29.pdf.

Roscoe, A. W. and S. D. Brookes (1985). 'An improved failures model for communicating processes'. In: *Proceedings of the Pittsburgh seminar on concurrency*. Ed. by A. W. Roscoe, S. D. Brookes and G. Winskel. LNCS 197. Springer, pp. 281–305. URL: http://www.cs.ox.ac.uk/people/bill.roscoe/publications/9.pdf.

Roscoe, A. W., S. D. Brookes and C. A. R. Hoare (1984). 'A theory of communicating sequential processes'. In: *Journal of the ACM*. 31st ser. 3, pp. 560–599. URL: http://www.cs.ox.ac.uk/people/bill.roscoe/publications/4.pdf.

Roscoe, A. W., C. A. R. Hoare and R. Bird (1997). *The Theory and Practice of Concurrency*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Sampson, A. T. (2008). 'Process-oriented Patterns for Concurrent Software Engineering'. D.Phil thesis. University of Kent.

ScrapMaker.com (2015). *Useful lists for geeks, machine learning, and linguists*. URL: http://scrapmaker.com/view/dictionaries/Unabr.dict (visited on 01/06/2015).

Sharpsteen, C. and C. Bracken (2016). *tikzDevice: R Graphics Output in LaTeX Format*. R package version 0.10-1. URL: https://CRAN.R-project.org/package=tikzDevice.

Shavit, N. and D. Touitou (1995). 'Software Transactional Memory'. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distrib-*

uted Computing. PODC '95. Ottowa, Ontario, Canada: ACM, pp. 204–213. URL: http://doi.acm.org/10.1145/224964.224987.

Smith, J. E. and G. S. Sohi (1995). 'The microarchitecture of superscalar processors'. In: *Proceedings of the IEEE* 83.12, pp. 1609–1624.

Smyth, W. F. (2003). *Computing Patterns in Strings*. Addison-Wesley.

Strauss, T., D. G. Kourie and B. W. Watson (2008a). 'A Concurrent Specification of an Incremental DFA Minimisation Algorithm'. In: *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*. Ed. by J. Holub and J. Zdárek. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, pp. 218–226.

Strauss, T., D. G. Kourie and B. W. Watson (2008b). 'A Concurrent Specification of Brzozowski's DFA Construction Algorithm'. In: *International Journal of Foundations of Computer Science* 19.1, pp. 125–135.

Strauss, T., D. G. Kourie, B. W. Watson and L. Cleophas (2014). 'A Process-Oriented Implementation of Brzozowski's DFA Construction Algorithm'. In: *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 17–29.

Strauss, T., D. G. Kourie, B. W. Watson and L. Cleophas (2015). 'Process-Based Aho-Corasick Failure Function Construction'. In: *Communicating Process Architectures 2015. Proceedings of the 37th WoTUG Technical Meeting, 23–26 August 2015, University of Kent, UK*. Ed. by K. Chalmers, J. B. Pedersen, F. R. M. Barnes, J. F. Broenink, R. Ivimey-Cook, A. Sampson and P. H. Welch. Open Channel Publishing Ltd., pp. 183–206. URL: http://wotug.org/cpa2015/programme.shtml.

Strauss, T., D. G. Kourie, B. W. Watson and L. Cleophas (2017). 'CSP for Parallelising Brzozowski's DFA Construction Algorithm'. In: *The Role of Theory in Computer Science: Essays Dedicated to Janusz Brzozowski*. Ed. by S. Konstantinidis, N. Moreira, R. Reis and J. Shallit. To appear, pp. 217–243. URL: http://www.worldscientific.com/worldscibooks/10.1142/10239.

Sutter, H. (2005). *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. Graphs updated in 2009. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 04/11/2016).

Sutter, H. (2012). URL: https://herbsutter.com/welcome-to-the-jungle/ (visited on 04/11/2016).

Syamalakumari, S. (2013). *Intel® Xeon® Processor E5-2600 V2 Product Family Technical Overview*. URL: https://software.intel.com/en-us/articles/intel-xeon-processor-e5-2600-v2-product-family-technical-overview (visited on 11/10/2016).

Tewari, A., U. Srivastava and P. Gupta (2002). 'A Parallel DFA Minimization Algorithm'. In: *HiPC*. Ed. by S. Sahni, V. K. Prasanna and U. Shukla. Vol. 2552. Lecture Notes in Computer Science. Springer, pp. 34–40.

Turon, A. (2013). 'Understanding and Expressing Scalable Concurrency'. AAI3558728. PhD thesis. Boston, MA, USA.

Watson, B. W. (1993). *A taxonomy of finite automata construction algorithms*. Tech. rep. 43. Faculty of Computing Science, Eindhoven University of Technology, the Netherlands.

Watson, B. W. (1994). *The design of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions*. Tech. rep. 22. Faculty of Computing Science, Eindhoven University of Technology, the Netherlands.

Watson, B. W. (1995). 'Taxonomies and Toolkits of Regular Language Algorithms'. PhD thesis. Faculty of Computing Science, Eindhoven University of Technology, the Netherlands.

Watson, B. W. (2001). 'An Incremental DFA Minimization Algorithm'. In: *Proceedings of the Second International Workshop on Finite State Methods in Natural Language Processing*. Ed. by L. Karttunen, K. Koskenniemi and G. van Noord. Helsinki, Finland.

Watson, B. W. (2010). 'Conctructing Minimal Acyclic Deterministic Automata'. PhD thesis. Department of Computer Science, University of Pretoria, South Africa.

Watson, B. W., L. Cleophas and D. G. Kourie (2014). 'Using Correctness-by-Construction to Derive Dead-zone Algorithms'. In: *Proceedings of the Prague Stringology Conference 2014, Prague, Czech Republic, September 1-3, 2014*. Ed. by J. Holub and J. Zdárek. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, pp. 84–95. URL: `http://www.stringology.org/event/2014/p09.html`.

Watson, B. W. and J. Daciuk (2003). 'An Efficient Incremental DFA Minimization Algorithm'. In: *Journal of Natural Language Engineering* 9.1, pp. 49–64.

Watson, B. W., D. G. Kourie and T. Strauss (2012). 'A Sequential Recursive Implementation of Dead-Zone Single Keyword Pattern Matching'. In: *Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers*. Ed. by S. Arumugam and W. F. Smyth. Vol. 7643. Lecture Notes in Computer Science. Springer, pp. 236–248. URL: `http://dx.doi.org/10.1007/978-3-642-35926-2`.

Watson, B. W. and R. E. Watson (2003). 'A New Family of String Pattern Matching Algorithms'. In: *South African Computer Journal* 30, pp. 34–41.

Watson, B. W. and G. Zwaan (1993). 'A taxonomy of keyword pattern matching algorithms'. In: *Proceedings of the Symposium on Computing Science in the Netherlands*. Ed. by H. Wijshoff. Utrecht, The Netherlands, pp. 25–39.

Welch, P. H. (2002). 'Process Oriented Design for Java: Concurrency for All'. In: *Proceedings of the International Conference on Computational Science-Part II*. ICCS '02. London, UK: Springer-Verlag, p. 687.

Welch, P. H. and F. R. M. Barnes (2005). 'Communicating mobile processes: Introducing occam-pi'. In: *Communicating Sequential Processes. The First 25 Years*. Springer, pp. 175–210.

Welch, P. H., N. C. C. Brown, J. Moores, K. Chalmers and B. H. C. Sputh (2007). 'Integrating and Extending JCSP'. In: *Communicating Process Architectures 2007*. Ed. by S. A. Schneider, A. A. McEwan, W. Ifill and P. H. Welch. Vol. 65. Concurrent Systems Engineering. Amsterdam, The Netherlands: IOS, pp. 349–370. URL: http://kar.kent.ac.uk/24001/.

Welch, P. H. and J. B. Pedersen (2010). 'Santa Claus: Formal Analysis of a Process-oriented Solution'. In: *ACM Transactions on Programming Languages and Systems* 32.4, 14:1–14:37. URL: http://doi.acm.org/10.1145/1734206.1734211.

Wickham, H. (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. URL: http://ggplot2.org.

Wickham, H. (2011). 'The Split-Apply-Combine Strategy for Data Analysis'. In: *Journal of Statistical Software* 40.1, pp. 1–29. URL: http://www.jstatsoft.org/v40/i01/.

Ziadi, D. and J.-M. Champarnaud (1999). 'An optimal parallel algorithm to convert a regular expression into its Glushkov automaton'. In: *Theoretical Computer Science* 215, pp. 69–87.

Acronyms and Abbreviations

AC	Aho-Corasick
ACP	Algebra of Communicating Processes
ALU	Arithmetic logic unit
BM	Boyer-Moore
CCS	Calculus of Communicating Systems
CPU	Central processing unit
CSP	Communicating Sequential Processes
DFA	Deterministic finite automaton
DSM	Distributed shared memory
DZ	Dead Zone
FAQ	Frequently asked questions
FDR3	Failures Divergence Refinement 3
FIFO	First in, first out
GCL	Guarded Command Language
GPU	Graphical processing unit
MIDI	Musical Instrument Digital Interface
MIMD	Multiple instruction streams, multiple data streams
MISD	Multiple instruction streams, single data stream
MPI	Message Passing Interface
NUMA	Non-uniform memory access
QPI	Intel® QuickPath Interconnect
RAM	Random access memory
RE	Regular expression
RUP	Rational Unified Process
SIMD	Single instruction stream, multiple data streams

SISD Single instruction stream, single data stream

SMP Symmetric multiprocessor

SMT Simultaneous multi-threading

UMA Uniform memory access

XP Extreme programming

Colophon

This document was produced using LuaLaTeX. The LaTeX source is based on a template by Kuehner (2016) using KOMA-Script. Diagrams were drawn using Ti*k*Z.

The main fonts are TeX Gyre Pagella and TeX Gyre Pagella Math. The Sans-serif font is Libertinus Sans and the monospaced font is DejaVu Sans Mono.

Statistical computations were done using R (R Core Team 2016), often making use of Wickham's (2011) `plyr` package. Statistical plots were produced using `ggplot2` (Wickham 2009) and `ggthemes` (Arnold 2016). These plots were converted into Ti*k*Z code using the `tikzDevice` package by Sharpsteen and Bracken (2016).