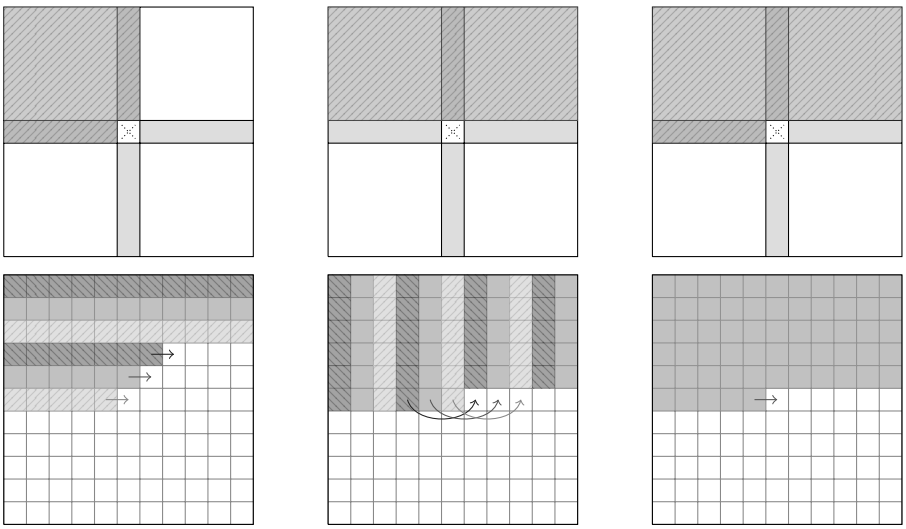


Raphael Reitzig

Automated Parallelisation of Dynamic Programming Recursions

Technische Universität Kaiserslautern

Fachbereich Informatik

Max-Planck-Gesellschaft

Handed in as Master thesis at University of Kaiserslautern, May 2012

Supervisors:

Prof. Dr. Markus E. Nebel	Department of Computer Science University of Kaiserslautern
Umut A. Acar, Ph.D.	Max Planck Institute for Software Systems Kaiserslautern / Saarbrücken

All concepts appearing in this thesis are fictitious. Any resemblance to real objects, animate or silicate, is purely coincidental.

No animals but gummy bears were harmed in the making of this thesis.
All test prints have been recycled as jotting paper.

Version 1.0 (July 25, 2012)

Why program by hand in five days
what you can spend five years
of your life automating?

Terence Parr (2007)

Contents

1 Introduction **4**
- 1.1 Parallel Algorithms . . . 5
- 1.2 Dynamic Programming . . . 11
- 1.3 Related Work . . . 15

2 Theoretical Groundwork **17**
- 2.1 Diagonal Frontier . . . 22
- 2.2 Row Splitting . . . 26
- 2.3 Not Parallelisable Case . . . 30
- 2.4 Relaxing Assumptions . . . 30
- 2.5 Considering Caches . . . 33
- 2.6 Considering Communication . . . 34
- 2.7 Lifting to Higher Dimensions . . . 35

3 Prototype Implementation **37**
- 3.1 Framework . . . 38
- 3.2 Implementing Diagonal Frontier . . . 40
- 3.3 Implementing Row Splitting . . . 44

4 Compiler Integration **50**

5 Conclusion **54**
- 5.1 Future Work . . . 54

A Source Code **57**
- A.1 Prototype Implementations . . . 57
- A.2 Compiler Plugin Samples . . . 64

B Glossary **69**

1

Introduction

Parallel programming has been an active research field for decades, both in theory and practise. Today, machines with multiple but relatively few cores—in 2012, numbers between two and eight are common—are ubiquitous. However, many applications are effectively sequential. This is often unproblematic because normal users utilise multiple cores by running many applications—web browser, instant messenger, office suite, music player, anti-virus software, etc.—simultaneously, each of which does not require parallelism to provide a pleasant usage experience. In use cases that involve huge calculations, on the other hand, it can be frustrating to watch seven cores idle while the eighth takes its time to complete the task.

Parallel programming is still considered hard because of non-deterministic concurrency issues and the challenge of designing and programming algorithms that are either parallel or can be parallelised automatically. There are many approaches to simplifying parallel programming. Pure functional languages, for example, are conceptually easy to parallelise as there are no side effects; their implementations are often not as easy to parallelise, though. Procedural languages, on the other hand, are harder to tackle as side effects make deciding whether and how a given piece of code can be parallelised hard. Techniques such as resource locks and agent models help but introduce their own sets of problems, some of which are hidden behind the shutters of clever libraries. There are also language extensions which allow programmers to annotate their code with respect to potential for parallelisation. As soon as multiple threads read and write the same data in parallel, however, most methods get in trouble. So despite all advances, parallel programming is still a hard task.

Therefore, the goal of this thesis is to develop efficient parallel algorithms for dynamic programming problems, a huge class of problems that are common in scientific computing which is one field that strives to utilise parallelism to analyse huge data sets faster. Programming parallel and even sequential algorithms for dynamic programming problems by hand can be tedious, so we furthermore want to automate as much of the process as possible; ideally, the user would program a dynamic programming recursion

and the compiler would transform it into an efficient, parallel algorithm.

How to read this thesis

The remainder of this chapter introduces notions of parallel algorithms and dynamic programming. We assume the reader is familiar with algorithms and their analysis roughly up to the level of an introductory course on those topics; the introductory first part of Cormen et al. [CLRS09] is a good way to come up to speed. The remainder of the thesis deals with our way towards the goal stated above, divided into three parts:

- theoretical investigation in Chapter 2 starting on page 17,
- prototypical implementation of the derived parallel algorithms and discussion in Chapter 3 beginning on page 37 and
- a proof-of-concept compiler integration presented in Chapter 4 from page 50.

While these three parts can be read independently of each other they are best read in sequence.

We mark new notions and put emphasis using *italic* typeface. Source code is printed in `non-proportional` font. We mark the end of proofs with the customary □, whereas proof ideas or unproven statements are closed with ⬚.

In terms of mathematical notation we try to stick to established standards and define derivations clearly. Note that boldly set variables refer to vectors with dimension greater than one, that is i denotes a number whereas $\mathbf{i}$ may represent a pair of numbers. We also consider 0 a natural number. For reference, we have compiled a list of notations on page 69.

1.1 Parallel Algorithms

In order to talk about quality of parallel algorithms we need appropriate notions. The following definitions have been inspired by Hoßfeld [Hoß83]; we will leave details of the machine model out, though. It suffices to say that we assume a model similar to PRAM proposed by Fortune et al. [FW78], that is a set of $\mathrm{p} \in \mathbb{N}_+$ ordinary RAM machines that operate on shared memory. The machines use the same clock, that is they execute their respective next operation simultaneously. We say that every RAM has its own *thread*; a p-PRAM can execute at most p threads in parallel. For the sake of simplicity, we allow simultaneous memory reads and writes. Using established acronyms, we assume a MIMD[1] machine with CRCW[2] memory

[1] Multiple instruction, multiple data.
[2] Concurrent read, concurrent write.

policy [Par87]. We furthermore assume as usual that all operations, including basic arithmetic operations as well as memory accesses, take constant numbers of clock cycles.

First of all, we have to fix what a parallel algorithm is.

1 Definition: Parallel Algorithm
A *parallel algorithm* A on p processors is an algorithm that can execute up to p statements in parallel. The number of processors p is known to A and its behaviour is uniform in p. We write A_p to denote an instance of A with fixed processor number and $A(x)$ for the result of algorithm A on input x.

The phrase "uniform in p" is hard to define rigorously. We mean that a parallel algorithm should work essentially in the same way regardless of p. What we want to forbid in particular is that a parallel algorithm detects $p = 1$ and executes an efficient sequential algorithm instead of the code used for parallel execution. While this is certainly a valid strategy in practise, it gets in the way of algorithm analysis, in particular comparisons between parallel and sequential algorithms.

We write algorithms in Javaesque pseudo code with some notable extensions following Cormen et al. [CLRS09].

- Global constant `P` contains the number of processors p available in total.
- Keyword `spawn` causes the annotated expression to be evaluated in a new thread, that is on a processor other than the current thread's one. The effect of `spawn` is not specified if there are not enough free processors for all `spawn` commands in a given clock cycle.
- Keyword `sync` causes the current thread to wait until all threads it has spawned have terminated.

As common, we assume that we have exclusive access to the machine at hand, that is we can make full use of all processors.

2 Definition: Runtime
The *runtime* of A_p on input x, denoted by $\mathrm{Time}_p^A(x)$, is defined by the number of clock cycles until all threads of A_p executed on x terminate. Similar to classic definitions,

$$T_p^A(n) := \max\left\{ \mathrm{Time}_p^A(x) : |x| = n \right\}$$

is the *worst-case runtime* of A_p for input size n.

What we are interested in are less concrete runtimes but rather how much faster we can be when parallelising. Therefore, we define the notion of speedup which captures this idea.

3 Definition: Speedup
Let A be a parallel algorithm. The *speedup* of A in the number of processors p is defined as

$$S_p^A(n) := \frac{T_1^A(n)}{T_p^A(n)}.$$

We call

$$S_p^A(\infty) := \liminf_{n\to\infty} S_p^A(n)$$

the *asymptotic speedup* of A on p processors.

It is clear that speedup is bounded by the number of processors if an algorithm is defined uniformly in p[3].

4 Lemma: Maximum Speedup
Let A be a parallel algorithm. Then,

$$S_p^A \leqslant p$$

for all $p \in \mathbb{N}$. ⬚

As usual, we will let input size n go to infinity in order to employ asymptotics machinery. We restrict ourselves to constant numbers of processors p, though, because our goal is to provide theoretic results that have immediate application in practise, that is enable programmers to utilise the amount of parallelism possible on their office machines with few cores. This motivates the following definition of scalable algorithms.

5 Definition: Scalability
Let A be a parallel algorithm and let $g : \mathbb{N} \to \mathbb{R}$ be an arbitrary function. We call

$$A \text{ } g\text{-}\textit{scalable} \quad :\Longleftrightarrow \quad \forall_{p\in\mathbb{N}} S_p^A(\infty) \geqslant g(p).$$

A problem is called g-*parallelisable* if it can be solved by an g-scalable parallel algorithm.

Rephrasing, our goal is to find p-scalable parallelisations for common problems, that is parallel algorithms that are asymptotically optimal with respect to Lemma 4. Note that we are looking for speedups by at most constant factors. This makes our results stronger than usual $\mathcal{O}$-analysis but the investigation may also be more technically involved.

Note that Definition 5 does not require parallel algorithms to be efficient in the usual sense, that is have the same asymptotic runtime as known

[3]Note that this is not necessarily true if memory hierarchies are considered.

sequential algorithms for the same problem. In particular, brute force algorithms for problems that are not known to have polynomial algorithms might have efficient parallelisations. In order to compare parallel algorithm A with a sequential counterpart B in a fair way, we want to allow A to cause only overhead that does not change asymptotic runtime for the worse compared to B. For this purpose, we introduce the following notion.

6 Definition: Foundedness
Let A be a parallel and B a sequential algorithm for the same problem, i.e. $A_p(x) = B(x)$ for all inputs x and all $p \in \mathbb{N}_+$. We say

$$A \text{ is } B\text{-}\textit{founded} \quad :\Longleftrightarrow \quad T_1^A \in \tilde{o}(T^B)$$

where $\tilde{o}(f) := o(f) \cup \{\, g : f \sim g \,\}$. We also say A is *founded in* B.

With this we can formulate an even stronger goal: we want to find p-scalable parallel algorithms that are founded in optimal—or at least efficient[4]—sequential algorithms, that is in essence $p \cdot T_p^A \sim T^B$ should hold for all p. We call the quotient T^B/T_p^A also *real speedup* of A with respect to B.

Let us consider a small example before we continue.

7 Example: Parallel Evaluation of Sums
Assume we want to compute the sum

$$S(n) = \sum_{i=1}^{n} i.$$

See Figure 1.1 on the next page for some algorithms that solve this problem. The first, `sum_seq`, is the naive sequential way to compute $S(n)$; counting only arithmetic operations on elements for the sake of simplicity, it takes $T^{ss}(n) = n - 1$ time.

Now consider its parallel version `sum_par`. The idea is to have thread w sum all i for which $i \mod p = w$. The call of `w` with `i==1` finishes last and performs $\lceil n/p \rceil - 1$ additions inside the loop; then, p additions aggregate the threads' results. So, we have $T_p^{sp}(n) = \lceil n/p \rceil + p - 1$. Computing speedup

$$S_p^{sp}(n) = \frac{T_1^{sp}(n)}{T_p^{sp}(n)} = \frac{n}{\left\lceil \frac{n}{p} \right\rceil + p - 1} \geqslant \frac{n}{\frac{n+1}{p} + p - 1} \xrightarrow[n\to\infty]{} p$$

we get that $S_n^{sp}(\infty) \geqslant p$, that is `sum_par` is p-scalable. Furthermore, with

$$\frac{T^{ss}(n)}{T_1^{sp}(n)} = \frac{n-1}{n} \xrightarrow[n\to\infty]{} 1$$

[4] Whatever that means in your context.

```
int sum_seq(int n) {
  int sum = 0;
  for (int i = 1; i <= n; i++ ) {
    sum += i;
  }
  return sum;
}


int sum_par(int n) {
  return w(n, P);
}

int w(int n, int p) {
  int rest = 0;
  if ( p > 1 ) {
    rest = spawn w(n, p-1)
  }

  int sum = 0;
  for (int i = p; i <= n; i += p ) {
    sum += i;
  }

  sync;
  return rest + sum;
}


int sum_gauss(int n) = {
  return n*(n-1)/2;
}
```

Figure 1.1: Three algorithms that compute $\sum_{i=1}^{n} i$.

we see that the additional effort of result aggregation vanishes for large n; `sum_par` is founded in `sum_seq`. If we compare `sum_par` with the more sophisticated `sum_gauss`, on the other hand, we get with

$$\frac{T^{sg}(n)}{T_1^{sp}(n)} = \frac{3}{n} \xrightarrow[n\to\infty]{} 0$$

that `sum_par` on one processor is asymptotically slower than `sum_gauss`, that is `sum_par` is not founded in `sum_gauss`.

An Aside: Other Criteria For Good Parallel Algorithms

Many approaches to gauge quality of parallel algorithms have been developed over the decades; a complete treatment of the literature is not feasible here. We find some notions worth mentioning, though.

First of all, there is the *work-depth* model [Ble96]. *Work* W is the number of operations that have to be executed by a parallel algorithm A, so essentially $W = T_1^A$. *Depth* D is the amount of work that no degree of parallelisation can do away with; you can say that $D = T_\infty^A$. Brent has proven an upper bound for special circumstances which is together with the trivial lower bound sometimes called *Brent's Law* [Bre74]:

$$\frac{W}{p} \leqslant T_p^A < \frac{W}{p} + D.$$

These bounds inform us that any parallel algorithm that takes more time than $W/p + D$ is certainly not at all optimal. Note that our definition of scalability is orthogonal to that; a p-scalable algorithm can be worse than Brent's upper bound if $D \in o(W)$. Our definitions can be applied to more general settings than Brent's Law, though.

An assumption often made in the context of the work-depth model is the *parallel slackness condition* [FLR98]

$$\frac{W}{D} \gg p \iff W \gg D$$

which essentially states that there is enough parallelisable work to keep all processors busy with. Since we can only expect p-scalable algorithms for $D \in o(W)$[5], parallel slackness is implied in our setting as we let $n \to \infty$. If the condition is not valid even for big n, however, p-scalable algorithms are impossible.

Our notion of foundedness with respect to a sequential algorithm is closely related to *work-efficiency* [Ble96] which can be stated in our terms as

$$p \cdot T_p^A \in \Theta(T^B)$$

[5] If $D \in \Omega(W) \iff T_\infty^A \in \Omega(T_1^A)$ then by definition $S_p^A(\infty) \in o(p)$ as $p \to \infty$.

for parallel algorithm A and sequential algorithm B. Note that overhead by a constant factor is still allowed; this is weaker than B-foundedness together with p-scalability.

In parallel complexity theory, a number of classes have been proposed to characterise the set of efficiently parallelisable problems. The most well known is NC which contains all problems that can be solved by a PRAM with polynomially many processors in polylogarithmic time [Par87]. This definition is not uncontroversial and other notions have been put forth that concentrate on work-efficiency in the above sense [KRS90]. In general, parallel complexity theory asks how many processors you need to solve a given problem up to some instance size very quickly; in contrast, we ask how quickly you can solve a given problem with a given number of processors.

1.2 Dynamic Programming

The term *dynamic programming* has been coined—and the corresponding theory been founded—by Richard Bellman [Bel57]. He used his theory to describe and analyse *multi-stage decision processes* in order to overcome the *curse of dimensionality*, that is state space explosion in high-dimensional problems, the prime adversary in scientific computing at the time. Recurrences he dealt with would look like this:

$$f(x) = \max_{0 \leqslant y \leqslant x} \left[g(y) + h(x - y) + f(ay + b(x - y)) \right]$$

for g, h some known functions and $a, b \in [0, 1)$, and many more complex forms. The core of Bellman's investigation was the *principle of optimality*, later named after him, which he phrased as follows:

> “ An optimal policy has the property that whatever the initial state and initial decision are, the remaining decision must constitute an optimal policy with regard to the state resulting from the first decision. ”

Since Bellman considered continuous domains, his algorithmic investigations were restricted to numerical algorithms.

Nowadays in computer science, the term dynamic programming is usually associated with discrete problems that can be solved by first computing optimal solutions to partial problems recursively and then combining them to an optimal solution of the original problem; see for instance Cormen et al. [CLRS09] for a host of such problems. This sounds similar to *Divide & Conquer*, another ubiquitous algorithmic, recursive strategy, but there is a crucial difference: in dynamic programming, solutions of smaller subproblems are needed for many larger subproblems. Therefore, dividing does not imply conquering for dynamic programming problems at all: computing a

given dynamic programming recursion directly typically takes exponential effort, whereas divide & conquer recursions often lead to efficient algorithms.

Not surprisingly we will stick closer to the modern computer science point of view on dynamic programming because we also want to deal with kinds of problems that Bellman did not consider, in particular decision procedures. Another problem with Bellman's original treatment is that—as anecdotal evidence [Sni78, Mor82] indicates—the precise meaning of Bellman's principle of optimality is mostly unclear and several conflicting interpretations have been put forth. The core properties concerning actual computation seem to be that

- problem instances can be partitioned into independently solvable subproblems,
- combining solutions of subproblems yields solutions of the original problem,
- if such a combination merges optimal solutions for the respective subproblems the resulting solution is optimal for the original problem and
- subproblems have the same property.

For the purpose of this work we do not care too much about all that as we do not attempt to solve some optimisation problem. In fact, we do not even care what a given recursion computes at all. All we want to do is find scalable parallel evaluation schemes for recursions of this type. We are therefore mainly interested in which subproblem depends on which others. To this end we propose a slightly different working definition of dynamic programming recursions; the task of finding a function that fits both a given problem and our definition is outside of this work's scope.

8 **Definition: Computation Dependency**
For any set X we call a relation $\gamma \subseteq X^2$ whose transitive closure γ^+ is irreflexive, i.e.

$$(x, y) \in \gamma^+ \implies x \neq y,$$

a *computation dependency relation* in X. A function f on X is *compatible* with γ if

$$(x, y) \in \gamma \iff \text{computation of } f(x) \text{ directly requires}^6 \ f(y).$$

In this case, we also write γ_f for γ.

[6] Think of f as recursive program; $(x, y) \in \gamma_f$ then means that computation of $f(x)$ causes a recursive call $f(y)$.

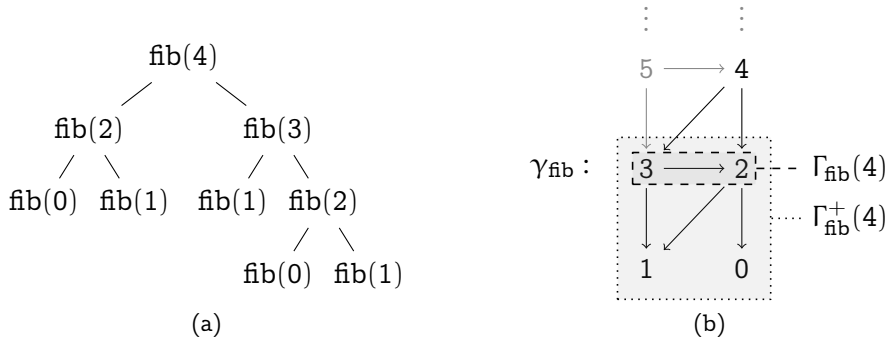


Figure 1.2: Recursion tree of fib(4) (a) and the corresponding dependency relation (b).

Now we define the set of *direct computation dependencies* of x as

$$\Gamma(x) \;:=\; \{\, y : (x,y) \in \gamma \,\}$$

and the set of *computation dependencies* of x as

$$\Gamma^+(x) \;:=\; \left\{\, y : (x,y) \in \gamma^+ \,\right\}.$$

If f is compatible with γ we also write Γ_f and Γ_f^+ and consider zipped versions

$$\widetilde{\Gamma}_f(x) \;:=\; \{\, (y, f(y)) : y \in \Gamma_f(x) \,\} \text{ and}$$
$$\widetilde{\Gamma}_f^+(x) \;:=\; \left\{\, (y, f(y)) : y \in \Gamma_f^+(x) \,\right\}$$

for notational convenience.

Let us look at a small example to clarify what we do here.

9 Example: Computation Dependencies of Fibonacci Numbers
Consider the recursive definition of the infamous Fibonacci numbers:

$$\text{fib}(n) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ \text{fib}(n-2) + \text{fib}(n-1) & , n \geqslant 2 \end{cases}$$

If implemented like this, the computation of $\text{fib}(n)$ can be viewed as a tree; see Figure 1.2a for an example. From that, it is easy to read off γ_{fib} as shown in Figure 1.2b. Note also that

$$\widetilde{\Gamma}_{\text{fib}}(4) = \{\, (3,2), (2,1) \,\}$$

and

$$\widetilde{\Gamma}^+_{\text{fib}}(4) = \{\, (3,2), (2,1), (1,1), (0,0) \,\}.$$

Now we have all tools necessary to give a concise definition of dynamic programming recursions.

10 Definition: Dynamic Programming
Let

$$I_{\mathbf{n}} := \left\{ \mathbf{i} \in \mathbb{N}^k : 0 \leqslant i_j \leqslant n_j - 1 \text{ for all } j = 1, \ldots, k \right\}$$

with $\mathbf{n} \in \mathbb{N}^k$ and $k \in \mathbb{N}_+$ a k-dimensional, bounded index domain and $\gamma \in I_{\mathbf{n}} \times I_{\mathbf{n}}$ a computation dependency relation. Let furthermore D be an arbitrary value domain.

A *dynamic programming recursion* is a recursion $d : I_{\mathbf{n}} \to D$ of the form

$$d(\mathbf{i}) := f(\mathbf{i}, \widetilde{\Gamma}_d(\mathbf{i}))$$

where $f : I_{\mathbf{n}} \times 2^{I_{\mathbf{n}} \times D} \to D$ and d is compatible with γ, that is $f(\mathbf{i}, \widetilde{\Gamma}_d(\mathbf{i}))$ does not directly depend on values of d other than the ones provided by $\widetilde{\Gamma}_d(\mathbf{i})$. We call f the *cell function*[7] of d. Note that d might depend on problem parameters other than index $\mathbf{i}$; these are assumed to be given as global values for the sake of clarity.

Many dynamic programming representations for typically encountered decision or optimisation problems match the proposed definition.

11 Example: Edit Distance
The *edit-distance* [Gus97] between two words $v \in \Sigma^n$ and $w \in \Sigma^m$ is given by $e(n, m)$ where e is defined by

$$e(i_1, i_2) := \begin{cases} 0 & , i_1 = i_2 = 0 \\ i_2 & , i_1 = 0 \wedge i_2 > 0 \\ i_1 & , i_1 > 0 \wedge i_2 = 0 \\ \min \begin{cases} e(i_1 - 1, i_2) + 1 \\ e(i_1, i_2 - 1) + 1 \\ e(i_1 - 1, i_2 - 1) + [v_{i_1 - 1} \neq w_{i_2 - 1}] \end{cases} & , \text{else} \end{cases}$$

for all $(i_1, i_2) \in I = [0..n] \times [0..m]$. Note that f from Definition 10 is essentially the right-hand side of the definition of e and the computation dependencies are given by the parameters of the recursive calls, that is

$$\Gamma_e(i_1, i_2) = \{ (i_1 - 1, i_2), (i_1, i_2 - 1), (i_1 - 1, i_2 - 1) \} \cap I$$

for all $(i_1, i_2) \in I$ with $i_1, i_2 > 0$ and $\Gamma_e(i_1, i_2) = \emptyset$ otherwise. As a concrete example, consider the distance of the strings $v = \texttt{bald}$ and $w = \texttt{boulder}$;

[7] The image of d on $I_{\mathbf{n}}$, that is $(d(\mathbf{i}))_{\mathbf{i} \in I_{\mathbf{n}}}$, is a k-dimensional matrix.

computing $e(4,7)$ requires the following matrix of values:

		b	o	u	l	d	e	r
	0	1	2	3	4	5	6	7
b	1	0	1	2	3	4	5	6
a	2	1	1	2	3	4	5	6
l	3	2	2	2	2	3	4	5
d	4	3	3	3	3	2	3	4

Note the highlighted path which leads to the final result $e(4,7) = 4$ and corresponds to the *alignment*

b	o	u	l	d	e	r
b	a	–	l	d	–	–

.

Dynamic programming recursions do not imply efficient algorithms per se: if computed in a straight-forward way we might have to evaluate f exponentially often; consider for example the recursive definition of the famous Fibonacci numbers. A way of computing such functions efficiently is the very algorithmic idea that has become one of the most fundamental algorithmic design patterns and is inseparable from the term *dynamic programming* today: filling a matrix with solutions of all subproblems, also called bottom-up strategy. This leads to clear polynomial time algorithms for many important problems such as string edit distance, all-pair shortest paths, longest common subsequence, forward-backward algorithm and many more. This work aims at offering insight into how parallel versions of the canonical algorithm for dynamic programming in an almost general and uniform way can work.

For a more thorough introduction to dynamic programming in the modern computer science sense and a number of concrete examples see Cormen et al. [CLRS09].

1.3 Related Work

Considering the popularity of both parallel algorithms and dynamic programming, there is a vast amount of work on both; we therefore give only examples of the available literature.

Solving recursions in parallel is obviously a matter of excitement and has been studied both in complexity theory [IT94] and in practise [NCTT09]. There are also approaches to achieve parallelisation using robust programming language primitives [MAFC11, FLR98]. Divide & Conquer and Branch & Bound as natural candidates for parallelisation have also been studied thus [RR99, TP96].

There is general work about parallel dynamic programming in complexity theory [Bra94, Ryt88]. Furthermore, there are many instances of solving

specific problems in parallel, both in complexity theory [EI96] and algorithms [ARQ93, ACDS03], some of which are close to our own ideas.

On the other hand, there are approaches to automatically generating efficient sequential code for solving dynamic programming problems given in some abstract representation [SJG11, PBS11].

2

Theoretical Groundwork

In this chapter we investigate how well dynamic programming recursions in our sense can be computed in parallel with respect to the notions defined in the previous chapter, that is in theory. For the sake of simplicity, we restrict ourselves to two-dimensional domains in the form of matrices, i.e. $I = [0..n_1 - 1] \times [0..n_2 - 1]$. We will state further assumptions that render the huge space of possible recursions more amenable and introduce some structure we can exploit. Keep in mind that we are looking for parallel algorithms that are as general as possible for we want to select and apply them algorithmically, given dynamic programming recursions.

We first introduce some notions to classify dynamic programming recursions by the structure of their dependency relations.

12 **Definition: Dependency Areas**
In the spirit of Definition 10, let

$$I_{\mathbf{n}} := \left\{ \mathbf{i} \in \mathbb{N}^2 : i_j \leqslant n_j - 1 \text{ for } j = 1, 2 \right\}$$

with $\mathbf{n} \in \mathbb{N}^2$ two-dimensional, bounded set of indices. We define the following *dependency areas* for $\mathbf{i} \in I$ that partition the matrix around $\mathbf{i}$; see also Figure 2.1 on the following page.

$$\begin{aligned}
L(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 = i_1, j_2 < i_2\} \\
UL(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 < i_1, j_2 < i_2\} \\
U(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 < i_1, j_2 = i_2\} \\
UR(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 < i_1, j_2 > i_2\} \\
R(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 = i_1, j_2 > i_2\} \\
DR(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 > i_1, j_2 > i_2\} \\
D(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 > i_1, j_2 = i_2\} \\
DL(\mathbf{i}) &:= \{\mathbf{j} \in I : j_1 > i_1, j_2 < i_2\}
\end{aligned}$$

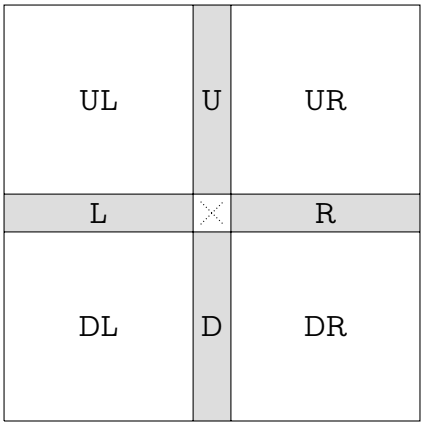


Figure 2.1: Dependency areas of the marked cell.

The set of all areas is denoted by

$$\text{Areas} := \{\mathrm{L}, \mathrm{UL}, \mathrm{U}, \mathrm{UR}, \mathrm{R}, \mathrm{DR}, \mathrm{D}, \mathrm{DL}\}.$$

We now consider several kinds of dynamic programming problems and investigate in which way they can be parallelised. In order to keep things appropriately clear, we restrict ourselves to such recursions that can be characterised nicely in terms of the areas we just defined; this is useful for the upcoming theoretic treatment as well as automated classification of dynamic programming recursions given as source code[1]. More specifically, for d a dynamic programming recursion over I with cell function f and computation dependencies Γ_d, we want $\Gamma_d(\mathbf{i})$ to have the same form for all indices $\mathbf{i} \in I$ (A2) and its transitive closure $\Gamma_d^+(\mathbf{i})$ should cover whole areas (A1). We also want to assume for technical reasons that I is completely used (A3) and that the runtime of cell function f is independent of its parameters (A4). In formal terms, we make the following four assumptions:

- Γ_d is *area-complete*, i.e.

$$\Gamma_d^+(\mathbf{i}) \cap A(\mathbf{i}) \neq \emptyset \implies A(\mathbf{i}) \subseteq \Gamma_d^+(\mathbf{i}) \tag{A1}$$

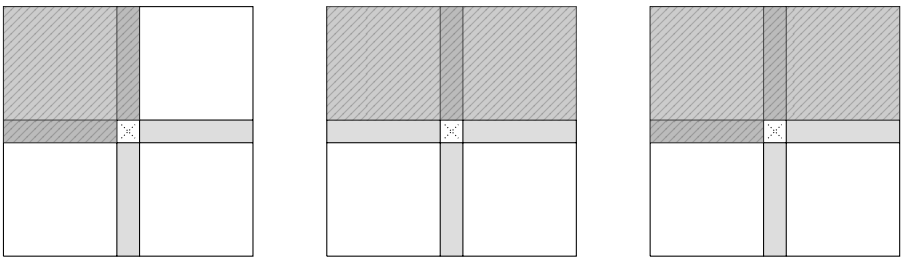
  for all $\mathbf{i} \in I$ and $A \in \text{Areas}$.

- Γ_d is *uniform*, i.e.

$$\exists_{\mathbf{i} \in I}\ \Gamma_d^+(\mathbf{i}) \cap A(\mathbf{i}) \neq \emptyset \implies \forall_{\mathbf{i} \in I}\ A(\mathbf{i}) = \emptyset \vee \Gamma_d^+(\mathbf{i}) \cap A(\mathbf{i}) \neq \emptyset \tag{A2}$$

  for all $A \in \text{Areas}$.

[1] We will need to analyse Γ_d^+ algorithmically, so it can not be too complicated.



(a) Case (C1) (b) Case (C2) (c) Case (C3)

Figure 2.2: Cases of dynamic programming recursions. The dependencies Γ_d^+ of the marked cell are highlighted.

- All partial results of d are needed, i.e.

$$\Gamma_d^+((n_1, n_2)) = I \setminus \{(n_1, n_2)\}. \tag{A3}$$

- Runtime of cell function f is independent of its parameters, i.e.

$$\mathrm{Time}^f\left(\mathbf{i}, \widetilde{\Gamma}_d(\mathbf{i})\right) = \mathrm{Time}^f\left(\mathbf{j}, \widetilde{\Gamma}_d(\mathbf{j})\right) \tag{A4}$$

  for all $\mathbf{i}, \mathbf{j} \in I$. We write T^f in short.

These assumptions seem to be very strong. They still allow for a rich class of problems, though, and can be weakened considerably without invalidating the upcoming results. For the sake of keeping our proofs nice and tidy we stick to the assumptions as proposed; refer to Section 2.4 starting on page 30 for additional remarks.

Now we have to utilise our setup towards creating a manageable set of different kinds of recursions we have to investigate. First of all, we can disregard many combinations of dependency areas because we get from (A1) and (A2) that

$$L(\mathbf{i}) \subseteq \Gamma_d^+(\mathbf{i}) \quad \Longrightarrow \quad R(\mathbf{i}) \cap \Gamma_d^+(\mathbf{i}) = \emptyset;$$

otherwise we had cyclic dependencies, contradicting the definition of dependency relations. Similar arguments work for all other combinations of opposite areas. In fact, it is up to symmetry—rotation and reflection—sufficient to distinguish the following cases of dynamic programming recursions, visualised in Figure 2.2:

$$\forall_{\mathbf{i}\in I}\ \Gamma_d^+(\mathbf{i}) \subseteq L(\mathbf{i}) \cup UL(\mathbf{i}) \cup U(\mathbf{i}), \tag{C1}$$

$$\forall_{\mathbf{i}\in I}\ \Gamma_d^+(\mathbf{i}) \subseteq UL(\mathbf{i}) \cup U(\mathbf{i}) \cup UR(\mathbf{i}), \tag{C2}$$

$$\forall_{\mathbf{i}\in I}\ \Gamma_d^+(\mathbf{i}) = L(\mathbf{i}) \cup UL(\mathbf{i}) \cup U(\mathbf{i}) \cup UR(\mathbf{i}), \tag{C3}$$

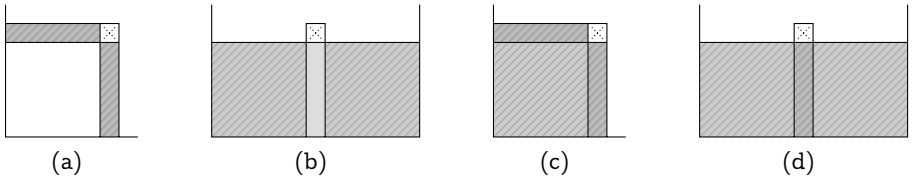


Figure 2.3: Dependency area patterns that can not (a, b) respectively can (c, d) occur; see proof of Theorem 13 on the current page.

Note that the three cases are not disjoint; in particular, a recursion might well belong to both case (C1) and (C2), for example if $\Gamma_{\mathrm{d}}^{+}(\mathbf{i}) = \mathrm{L}(\mathbf{i})$. The case distinction is complete in our setting, though.

13 Theorem: Complete Case Distinction
Let d be a dynamic programming recursion according to Definition 10 on page 14. Under (A1) and (A2), at least one of (C1), (C2) and (C3) applies to d up to symmetry.

Proof
The proof rests on three observations.

1. Because of (A1) and (A2), no two opposing areas can overlap with Γ_{d}^{+}; otherwise γ_{d}^{+} would have circles. Therefore, Γ_{d}^{+} can overlap at most two small (i.e. two out of L, U, R and D) and two big (i.e. two out of UL, UR, DR and DL) areas, respectively.

2. A pattern as depicted in Figure 2.3a can not occur; because of (A2), the highlighted cells require those not highlighted.

3. Similarly, a pattern as shown in Figure 2.3b can not occur.

Therefore, all possible patterns of three areas match Figure 2.3c or 2.3d; up to rotating the matrix, these correspond to (C1) and (C2), respectively. Four areas can only be placed using either of those plus an area of the missing kind; either case leads to (C3) after rotating. More than four areas can not occur and all possibilities of placing less than three are already subsumed in the derived cases. □

We can therefore restrict our investigations to those three cases.

Before we discuss parallel computation of two-dimensional dynamic programming recursions, let us recall how to compute them sequentially. It is typically done by filling the matrix corresponding to index set I row by row, that is as shown in Figure 2.4 on the next page; see Figure 2.5 for details. We call this algorithm `R` in the sequel. Note that it corresponds to the bottom-up algorithm in Cormen et al. [CLRS09].

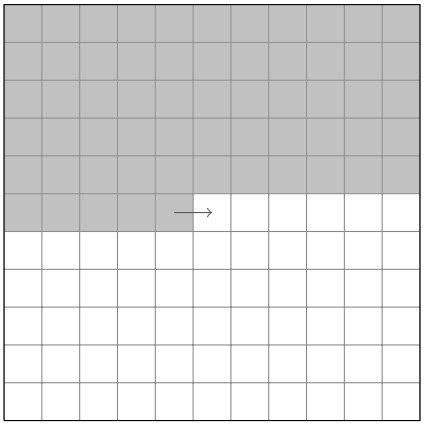


Figure 2.4: Row by row filling algorithm R after some steps.

```
S rowfill(T inA, T inB) {
  S[][] table = new S[inA.size][inB.size];

  for ( int i=0; i<inA.size; i++ ) {
    for ( int j=0; j<inB.size; j++ ) {
      table[i][j] = f(i,j,table);
    }
  }

  return table[inA.size - 1][inB.size - 1];
}
```

Figure 2.5: Row by row filling algorithm R of dynamic programming matrix with f the cell function.

14 Lemma: Sequential Row Filling
Any two-dimensional dynamic programming recursion with cell function f over $I = [0 : n_1 - 1] \times [0 : n_2 - 1]$ that conforms to either of cases (C1) to (C3) can be computed with algorithm `R` in time $\Theta(n_1 n_2 T^f)$.

Proof
Correctness of `R` is immediately clear; all three cases are top-left oriented and the matrix can therefore be filled from left to right and top to bottom. For future reference, we give the precise runtime. Let

- c_a be the constant runtime of creating a new array,
- c_{fi} be the constant runtime of initialising a `for` loop,
- c_{fc} be the constant runtime of managing the loop counter in a `for` loop iteration and
- c_m be the constant runtime of one array access.

Then, the runtime of `R` is given by

$$\begin{aligned} T^R(n_1, n_2) &= c_a + c_{fi} + n_1(c_{fi} + n_2(T^f + c_m + c_{fc}) + c_{fc}) + c_m \\ &= n_1 n_2 \underbrace{(T^f + c_m + c_{fc})}_{C_0} + n_1 \underbrace{(c_{fi} + c_{fc})}_{C_1} + \underbrace{(c_a + c_{fi} + c_m)}_{C_2} \\ &\in \Theta(n_1 n_2 T^f)\,. \end{aligned}$$

□

Under (A3), no algorithm can be significantly faster as f has to be computed for all cells. The parallel schemes we investigate now follow the idea of `R` closely; we will therefore use `R` as gold standard to compare our parallel algorithms with.

2.1 Case (C1): Diagonal Frontier Approach

We now consider dynamic programming recursions according to case (C1), i.e.

$$\Gamma_d^+(\mathbf{i}) \subseteq L(\mathbf{i}) \cup UL(\mathbf{i}) \cup U(\mathbf{i})$$

for all $\mathbf{i} \in I$.

15 Example: Edit Distance (continued)
Recall the edit distance problem as given in Example 11 on page 14. Note that

$$\Gamma_e(i_1, i_2) \subseteq \{(i_1 - 1, i_2), (i_1, i_2 - 1), (i_1 - 1, i_2 - 1)\}$$

holds for all $(i_1, i_2) \in I$. Consequently we have $\Gamma_e^+(i_1, i_2) = L(i_1, i_2) \cup UL(i_1, i_2) \cup U(i_1, i_2)$; e does indeed conform to case (C1).

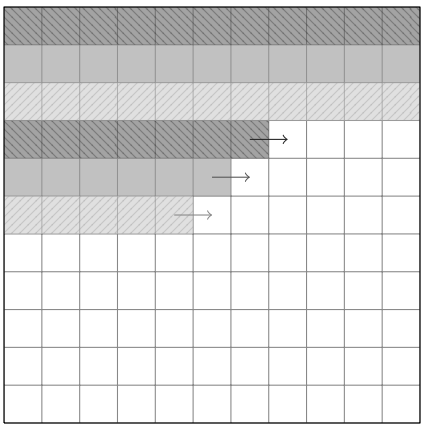


Figure 2.6: Diagonal frontier scheme DF after some steps.

We propose the *diagonal frontier* scheme (DF) to achieve p-scalable and R-founded parallelisation of case (C1) recursions. The key observation is that in this case, matrix cell (i, j) does not depend on cells in $\mathrm{UR}(i, j)$. Algorithm R, however, *does* compute all those cells before (i, j). We now exploit that $f(i, j, \_)$ can be computed as soon as $(i-1, j)$ and $(i, j-1)$ are done, assuming a row by row filling point of view. This can be done by filling rows independently, each row lagging behind its predecessor by one cell. See Figure 2.6 for a visualisation.

Of course we can only fill as many rows in parallel as we have processors. See Figure 2.7 on the following page for a detailed implementation. Algorithm R is still vaguely visible; workers fill their partial matrix following the same principle. Their respective first row has to be dealt with separately as they have to spawn their succeeding colleague. Note that because of assumption (A4) workers can never overtake each other; the algorithm is therefore correct.

16 **Theorem: (C1) Parallelised by Diagnoal Frontier Scheme**
The diagonal frontier scheme DF is

1. p-scalable and
2. R-founded

for dynamic programming recursions of type (C1) under assumption (A4) on page 19.

Proof
In addition to the names designated in the proof of Lemma 14 on the preceding page let

- $c_{\mathtt{if}}$ the constant cost of evaluating an `if` with an integer comparison,
- c_{sp} the constant cost of spawning a new thread and

```
S diagfront(T inA, T inB) {
  S[][] table = new S[inA.size][inB.size];

  worker(table, 0);

  return table[inA.size - 1][inB.size - 1];
}

void worker(S[][] table, int p) {
  if ( p < table.length ) {
    table[p][0] = f(p, 0, table);

    if ( p < P-1 ) {
      spawn worker(table, p+1);
    }

    for ( int j=1; j<inB.size; j++ ) {
      table[p][j] = f(p, j, table);
    }

    for ( int i=p+P; i<inA.size; i+=P ) {
      for ( int j=0; j<inB.size; j++ ) {
        table[i][j] = f(i, j, table);
      }
    }

    sync;
  }
}
```

Figure 2.7: Diagonal-frontier parallel filling scheme DF of dynamic programming matrix with f the cell function.

- c_{sy} the constant cost of syncing, disregarding the time actually spent on waiting.

Clearly, DF's runtime consists of initialisation, result propagation and the time the longest running worker needs, i.e.

$$T_p^{DF}(n_1, n_2) = c_a + \max\left\{ [n_1 \geqslant i] \cdot T_p^{w(i)}(n_1, n_2) : 0 \leqslant i < p \right\} + c_m.$$

Assuming $n_1 \geqslant p$, the finishing time of worker i is given by the time before it is spawned plus the time it runs, that is

$$\begin{aligned} T_p^{w(i)}(n_1, n_2) = \quad & i \cdot (c_{if} + T^f + c_m + c_{if} + c_{sp}) \\ & + c_{if} + T^f + c_m + c_{if} + [i < p-1]\, c_{sp} \\ & + c_{fi} + (n_1 - 1) \cdot (c_{fc} + T^f + c_m) \\ & + c_{fi} + \frac{n_1 - i - 1}{p} \cdot (c_{fc} + c_{fi} + n_2 \cdot (c_{fc} + T^f + c_m)) \\ & + c_{sy} \\ = \quad & i \cdot (2c_{if} + T^f + c_m + c_{sp}) \\ & + 2(c_{if} + c_{fi}) + T^f + c_m + [i < p-1]\, c_{sp} + c_{sy} \\ & + (n_2 - 1) \cdot C_0 + \frac{n_1 - i - 1}{p} \cdot C_1 + \frac{n_1 - i - 1}{p} \cdot n_2 \cdot C_0. \end{aligned}$$

The runtime of DF on one processor follows immediately; we get

$$\begin{aligned} T_1^{DF}(n_1, n_2) = \quad & c_a + T_1^{w(0)}(n_1, n_2) + c_m \\ = \quad & \overbrace{c_a + 2(c_{if} + c_{fi}) + T^f + c_m + c_{sy}}^{C_3} \\ & + (n_2 - 1) \cdot C_0 + (n_1 - 1)C_1 + (n_1 - 1)n_2 C_0. \end{aligned}$$

ad 1. We consider the quotient of T_1^{DF} and T_p^{DF} for $n_1, n_2 \to \infty$. First, note that the maximum in DF's runtime can be overestimated by choosing $i = 0$, considering waiting time as if $i = p - 1$ and assuming $p > 1$. In total, we get

$$\begin{aligned} T_p^{DF}(n_1, n_2) &\leqslant C_4 + (n_2 - 1)C_0 + \frac{n_1 - 1}{p} \cdot C_1 + \frac{n_1 - 1}{p} \cdot n_2 \cdot C_0 \\ &\in \Theta(n_1 n_2 T^f) \end{aligned}$$

where

$$C_4 \;:=\; p \cdot (c_{if} + T^f + c_m + c_{if} + c_{sp}) + 2c_{fi} + c_{sy}.$$

Now we consider the quotient:

$$\frac{T_1^{DF}(n_1, n_2)}{T_p^{DF}(n_1, n_2)} \geqslant \frac{C_3 + (n_2 - 1) \cdot C_0 + (n_1 - 1)C_1 + (n_1 - 1)n_2 C_0}{C_4 + (n_2 - 1)C_0 + \frac{n_1 - 1}{p} \cdot C_1 + \frac{n_1 - 1}{p} \cdot n_2 \cdot C_0}.$$

We can underestimate the quotient even more by dropping the numerator's first three summands as they are non-negative. This yields:

$$\begin{aligned}\frac{T_1^{DF}(n_1,n_2)}{T_p^{DF}(n_1,n_2)} &\geqslant \frac{(n_1-1)n_2C_0}{C_4+(n_2-1)C_0+\frac{n_1-1}{p}\cdot C_1+\frac{n_1-1}{p}\cdot n_2\cdot C_0}\\ &= \frac{p\cdot C_0}{\frac{p}{(n_1-1)n_2}\cdot C_4+\frac{p(n_2-1)}{(n_1-1)n_2}C_0+\frac{1}{n_2}\cdot C_1+C_0}\\ &\xrightarrow[n_1,n_2\to\infty]{} p.\end{aligned}$$

So we have shown that $\mathcal{S}_p^{DF}(\infty)\geqslant p$ for all $p\in\mathbb{N}_+$. $\square_{1.}$

ad 2. Recall from the proof of Lemma 14 on page 22 that

$$T^R(n_1,n_2)=n_1n_2C_0+n_1C_1+C_2.$$

We consider the quotient of T_1^{DF} and T^R for n,m to ∞. We get

$$\frac{T_1^{DF}(n_1,n_2)}{T^R(n_1,n_2)}=\frac{C_3}{T^R(n_1,n_2)}+\frac{(n_2-1)C_0}{T^R(n_1,n_2)}+\frac{(n_1-1)C_1}{T^R(n_1,n_2)}+\frac{(n_1-1)n_2C_0}{T^R(n_1,n_2)}.$$

Clearly, the first three summands converge to 0 as $T^R\in\Theta(n_1n_2T^f)$. For the last summand, we get

$$\begin{aligned}\frac{(n_1-1)n_2C_0}{n_1n_2C_0+n_1C_1+C_2} &= \frac{\left(1-\frac{1}{n_1}\right)C_0}{C_0+\frac{1}{n_2}C_1+\frac{1}{n_1n_2}C_2}\\ &\xrightarrow[n_1,n_2\to\infty]{} 1\end{aligned}$$

and therefore $T_1^{DF}\in\tilde{o}(T^R)$. $\square_{2.}$

As both statements have been shown the theorem is proven. $\square$

We can conclude that we found a—in our terms—optimal parallelisation of case (C1) dynamic programming recursions.

2.2 Case (C2): Row Splitting Approach

We now consider dynamic programming recursions according to case (C2), i.e.

$$\Gamma_d^+(\mathbf{i})\subseteq UL(\mathbf{i})\cup U(\mathbf{i})\cup UR(\mathbf{i})$$

for all $\mathbf{i}\in I$.

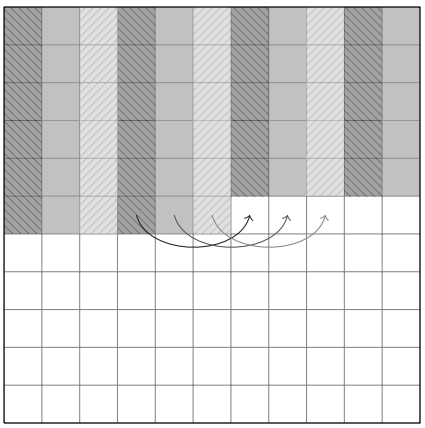


Figure 2.8: Row splitting scheme RS after some steps.

17 Example: Single-Source Shortest Paths
Given a directed weighted graph $G = (V, E, c)$, the *single-source shortest path* problem is to find shortest paths from a given source $s \in V$ to all other vertices. The *Bellman-Ford algorithm* is a way of solving it even if c has negative values [CLRS09]. It can be expressed as a recursion over $I = [0..n] \times [1..n]$, assuming that $V = [1..n]$ and $c(i, i) = 0$ for all $i \in V$:

$$\mathrm{bf}(i,j) = \begin{cases} 0 & , i = 0 \wedge j = s \\ \infty & , i = 0 \wedge j \neq s \\ \min\limits_{k \in V} \big(\mathrm{bf}(i-1,k) + c(k,j) \big) & , \text{else} \end{cases}$$

Note that $\mathrm{bf}(n, i)$ is the length of the shortest path from s to i; the whole last matrix row is the result. As for cell dependencies, we note that

$$\Gamma_{\mathrm{bf}}(i,j) = \begin{cases} \emptyset & , i = 0 \\ \{ i-1 \} \times [1..n] & , i > 0 \end{cases}$$

and therefore bf is a (C2) dynamic programming recursion.

We propose the *row splitting* scheme (RS) to compute such dynamic programming recursions in parallel. The idea here is simple: as elements do not depend on other elements in the same row, we can distribute computation of one row over processors in any way we like, for example as shown in Figure 2.8. While leading to a scheme closer to R than DF, this implies that we have spawning and syncing overhead per row, causing more overhead than DF. It is still only linear overhead which vanishes for large inputs, though. See Figure 2.9 on the next page for an implementation.

18 Theorem: (C2) Parallelised by Row Splitting Scheme
The row splitting scheme is

```
S rowsplit(T inA, T inB) {
  S[][] table = new S[inA.size][inB.size];

  for ( int i=0; i<inA.size; i++ ) {
    for ( int p=0; p<P-1; p++ ) {
      spawn worker(table, i, p);
    }
    worker(table, i, P-1);
    sync;
  }

  return table[inA.size - 1][inB.size - 1];
}

void worker(S[][] table, int i, int p) {
  for ( int j=p; j<inB.size; j+=P ) {
    table[i][j] = f(i, j, table);
  }
}
```

Figure 2.9: Filling of dynamic programming matrix by row splitting algorithm RS with f the cell function.

1. p-scalable and
2. R-founded

for dynamic programming recursions of type (C2) under assumption (A4) on page 19.

Proof

We reuse the abbreviations defined in the proofs of Lemma 14 on page 22 and Theorem 16 on page 23. First, we determine the runtime of RS; clearly, initialisation and result propagation takes some constant effort, as does managing each row. A row is finished when the last worker for this row finished. Therefore, we get the following runtime:

$$T_p^{RS}(n_1, n_2) = \quad c_a + c_m + c_{fi} + n_1 \cdot (c_{fc} + c_{sy} + c_{fi}) + n_1 \cdot T_p^{max}(n_2)$$

where

$$T_p^{max}(n_2) = \max \left\{ i \cdot c_{fc} + (i + [\, i < p - 1 \,]) \cdot c_{sp} + T_p^{w(i)}(n_2) : 0 \leqslant i < p \right\}$$

and

$$\begin{aligned} T_p^{w(i)}(n_2) &= c_{fi} + \left(\left\lfloor \frac{n_2}{p} \right\rfloor + [\, n \bmod p > i \,] \right) \cdot (T^f + c_m + c_{fc}) \\ &\leqslant c_{fi} + \frac{n_2 + 1}{p} \cdot C_0. \end{aligned}$$

In the special case of $p = 1$, this yields

$$T_1^{RS}(n_1, n_2) = n_1 n_2 \cdot C_0 + n_1 \cdot (C_1 + c_{sy}) + C_2.$$

ad 1. Generous estimation yields

$$\begin{aligned} T_p^{RS}(n_1, n_2) \leqslant \quad & \frac{n_1(n_2 + 1)}{p} \cdot C_0 \\ & + n_1 \cdot (c_{fc} + c_{sy} + 2c_{fi} + (p - 1)(c_{fc} + c_{sp})) \\ & + C_2 \\ \sim \quad & \frac{n_1 n_2}{p} C_0 \end{aligned}$$

so that $S_p^{RS}(\infty) \geqslant p$. $\square_{1.}$

ad 2. Note that

$$T_1^{RS}(n_1, n_2) = T^R(n_1, n_2) + n_1 c_{sy}$$

and remember that $T^R(n_1, n_2) \in \Theta(n_1 n_2 T^f)$, so clearly $T_1^{RS} \sim T^R$, proving that RS is R-founded. $\square_{2.}$

As both statements have been shown the theorem is proven. $\square$

So we can conclude that we found a—in our terms—optimal parallelisation also in case of (C2) dynamic programming recursions.

2.3 Case (C3): Not Parallelisable

We now consider dynamic programming recursions according to case (C3), i.e.

$$\Gamma_d^+(\mathbf{i}) = \mathrm{L}(\mathbf{i}) \cup \mathrm{UL}(\mathbf{i}) \cup \mathrm{U}(\mathbf{i}) \cup \mathrm{UR}(\mathbf{i})$$

for all $\mathbf{i} \in \mathrm{I}$. We do not know of natural examples for this case; this is probably for the better given the following result.

19 Theorem: (C3) Not Parallelisable
Dynamic programming recursions of type (C3) can not be g-parallelised for any $g > 1$ under assumptions (A1) and (A2) on page 18.

Proof
We show that the sequential algorithm R computes in every iteration of the inner loop a cell that could not have been computed before, that is a cell computed in step i depends on the result in the cell that is computed in step $i-1$. That implies that R already parallelises as much as possible, thus proving the theorem.

Let d be a dynamic programming recursion of type (C3). We use induction over I, following the execution order of R. The base case is given by cell $(0,0)$; as it is the very first cell R computes it could clearly not have been computed earlier. Now assume that for a fixed $\mathbf{i} \in \mathrm{I}$, all $\mathbf{j} \in \mathrm{I}$ with either $j_1 < i_1$ or $j_1 = i_1 \wedge j_2 \leqslant i_2$ have been computed by R as early as possible and consider the computation of $f(i,j)$. We distinguish two cases.

$j = 0$: The last cell computed by R before (i,j) was $(i-1, n_2-1)$. By assumption, $(i-1, n_2-1) \in \Gamma_d^+(i,0)$ and it could not have been computed earlier by induction hypothesis; therefore, R computes (i,j) as early as possible.

$j \neq 0$: Similar to the first case but with $(i,j-1) \in \Gamma_d^+(i,j)$.

This concludes the proof. □

With this negative result, we have dealt with all cases of dynamic programming recursions that remain in our abstract setting. The remainder of this chapter presents some additional thoughts regarding the strong assumptions we made, how memory hierarchies and inter-thread communication factors in and how to apply our results to dynamic programming recursions with more than two dimensions.

2.4 Relaxing Assumptions

There certainly are dynamic programming recursions that do not conform to assumptions (A1) to (A4). While we claim that we cover many—maybe

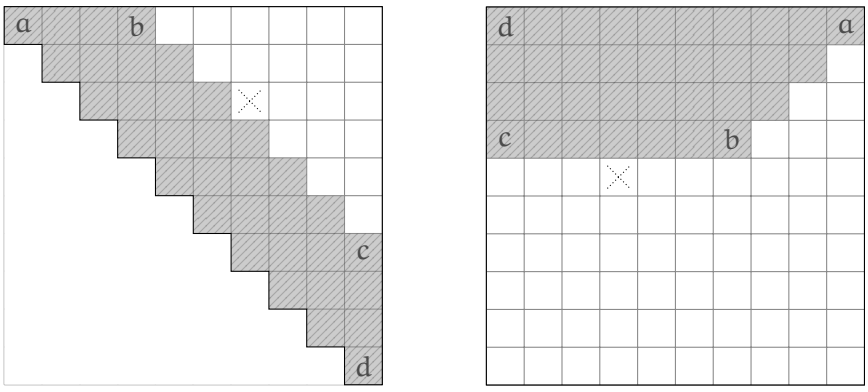


Figure 2.10: Example for how a dynamic programming recursion working from the diagonal towards a corner (a) can transformed to top-left orientation and overapproximated (b).

most—relevant instances it is worthwhile to investigate the boundaries of the approaches we have discussed.

An interesting type of recursion we seem to miss out on are those that work from the diagonal towards a corner, for instance variants of the knapsack problem or finding optimal binary search trees [CLRS09]. However, if we rotate the problem by $\pi/4$ plus appropriate multiples of $\pi/2$ they can be overapproximated to fit one of our three cases; see Figure 2.10 for a sketch.

Assumption (A1) is very strong in that it allows us to abstract many possible shapes of dependency relations and only consider huge areas. For dynamic programming recursions that have strictly smaller dependencies Γ, for instance the CYK algorithm [AU72], we overapproximate. This is not a problem for cases (C1) and (C2); we can parallelise the more intricate cases just in the same way as the general case we investigate. We might have to compete with sequential algorithms other than `R` as it might not be optimal anymore.

There are, however, conceivable dependency relations that are subsumed by (C3) but for which dropping (A1) offers new possibilities; one example is depicted in Figure 2.11 on the next page. It turns out that a recursion of this form can be parallelised by using `DF` with `k` cells head start for every thread. There may be many more recursions that can be parallelised but we do not cover. Remember, though, that our goal is to recognise parallelisability algorithmically; this is hopeless in general, therefore our structurally simplifying assumption.

Assumption (A2) serves a similar purpose as (A1). There may be parallelisable dynamic programming recursions with non-uniform dependencies

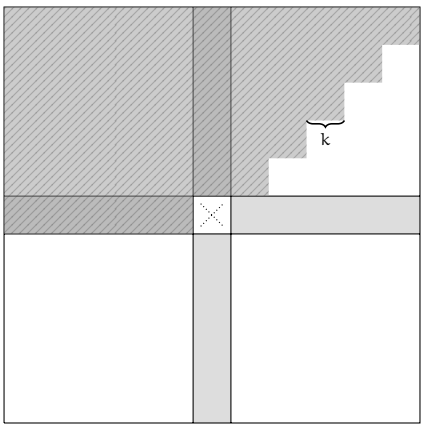


Figure 2.11: A case (C3) dependency structure that can be parallelised: cells may depend on others in UR that lie at most k columns to the right, k a constant.

but we expect them to be very hard[2] to identify in general. One could try to set up direct evaluation of the recursion using memoisation and work-stealing scheduling; if it is parallelisable, this should achieve at least some speedup.

We consider (A3) a more technical assumption than the first two. You can certainly define dynamic programming recursions that do not need the full matrix; we claim that in such a case, an equivalent one that does so can be found. Note that branch and bound tuning of dynamic programming algorithms does not violate the assumption per se as it is not a priori clear which values need not be considered.

Assuming a cell function f whose runtime Time^f is independent of its parameters in (A4) is by far the most blatant of the four assumptions—we do not know of even one example for a dynamic programming recursion that does *not* violate it! More specifically, the matrix' top and left edge are typically easier done than the interior. If this is the only violation we can just adapt the assumption to state that these edge cases are not slower than the others, bound Time^f by its interior case for the analysis and all is fine. There are, however, plenty of examples of f accessing varying numbers of already computed values, for example CYK. Consider alternatively the assumption of monotonicity, i.e.

$$i_1 \leqslant j_1 \wedge i_2 = j_2 \implies \text{Time}^f\left(\mathbf{i}, \widetilde{\Gamma}_d(\mathbf{i})\right) \leqslant \text{Time}^f\left(\mathbf{j}, \widetilde{\Gamma}_d(\mathbf{j})\right) \qquad \text{(A4')}$$

for all $\mathbf{i}, \mathbf{j} \in I \subset \mathbb{N}^2$. This is clearly weaker than (A4) but covers both edge cases and recursions like the one of CYK; in fact, we have yet to meet a dynamic programming recursion that does not conform to it. Besides, it

[2]Read: impossible, as in not computable.

is immediately clear that DF remains correct and Theorem 16 on page 23 still holds[3]. RS reacts in a different way. It does not care about runtime differences of rows as DF does because it synchronises after each row, anyway. Instead, it can become inefficient if and only if runtimes are distributed over rows such that one thread does much more work than the others. Therefore, cell function runtimes have to be spread out evenly if RS is to be fast; we do not bother to formalise this rigorously.

2.5 Considering Caches

It is well known that R breaks down in the presence of a memory hierarchy, that is its performance is dominated by cache misses for big instances. Assume a cache that can hold—besides control variables—the κ cells most recently accessed and consider, for instance, the recursion from Example 11 on page 14. If $\kappa < m + 1$ about every cell computation causes two cache misses as $e(i_1 - 1, i_2 - 1)$ and $e(i_1 - 1, i_2)$ are no longer in cache. For simple cases such as this one the effect can be mitigated significantly by subdividing the matrix into appropriately sized chunks [LTC10].

The problem certainly exists for our parallelisation schemes as well[4] and we even face the additional challenge of ensuring correctness under memory hierarchy. This is apparent in the case of DF; only the first thread encounters cache misses and is thus overtaken by its fellows. Note that correctness of RS is not in danger.

Scheme DF can be adapted to work—as best as possible—assuming memory hierarchy. Instead of filling rows completely, we apply the scheme to blocks of size $\sqrt{\kappa} \times \sqrt{\kappa}$ and schedule those blocks column by column. As whole blocks fit into cache, misses only occur for each block's first column and the complete matrix' first row; in total, that amounts to about $n + {}^{nm}/_{\sqrt{\kappa}}$ many misses. Note that as worker threads work row by row cache misses are nicely distributed among them so that DF remains correct if we handle the first row separately.

There is not much hope to account for caching in general as we might have dynamic programming recursions with $|\Gamma_d(\mathbf{i})| > \kappa$, in which case any scheme has to submit to domination by memory latency. We can therefore only hope to provide schemes that are useful in many cases and delegate the task of choosing suitable recursions to the individual programmer. We take comfort in the knowledge that sequential versions can fare no better so we can still achieve good speedups—in theory.

[3] The proof becomes incredibly messy, though, which is why we stuck to (A4) in the first place.

[4] The attentive reader notices that DF causes less cache misses than R.

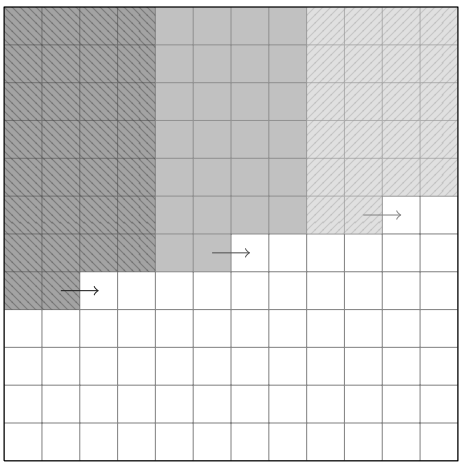


Figure 2.12: The column filling scheme for case (C1) after some steps.

2.6 Considering Communication

Another concern with respect to real machines is communication between threads. In above treatment we have had threads communicate implicitly via shared memory, assuming that a memory access cost is about constant. This is, however, not true in practise. On real multi-core machines, each core has its own caches, some cores might share additional cache levels and only then main memory is accessed. So far, this is no different from the sequential case. However, we have to synchronise caches in parallel settings which adds significantly to the time a memory access takes; if we access the same data from several threads—as we do above—this can cause significant overhead. We will try and quantify the effect of communication to our parallelisation schemes; for the sake of simplicity, we count the number of cell transfers we observe.

Consider, for instance, our algorithm DF. Every cell is written by one thread and read by at least one other; therefore, every cell has to be communicated at least once. In the case of edit distance (see Example 11 on page 14), we obtain the best case of only one transmission per cell. This already implies $\Theta(n_1 n_2)$ overhead; our scheme can not be expected to scale.

Fortunately, the effect can be controlled to some extent. Consider the alternative scheme *column filling* (CF). We divide the matrix into p columns and every thread fills its column row by row, giving its left neighbour the first row as head start; see also Figure 2.12. Even though this causes linear overhead[5] we note without formal proof that CF is R-founded and p-scalable. It is also apparent that only $(p-1)n_1$ cells are communicated in the case of edit distance or similar recursions; we have avoided losing scalability in

[5]Remember head starts in DF caused only constant overhead.

the presence of communication cost. However, if cells rely on many cells to the left, i.e. $|\Gamma(\mathbf{i}) \cap (\mathrm{L}(\mathbf{i}) \cup \mathrm{UL}(\mathbf{i}))| \in \omega(1)$, we have quadratic overhead yet again.

In the case of RS we can not make a general statement as communication overhead depends on the way we split rows and the dependencies at hand. If cells do not depend on cells in far-away columns and we split rows into not too small blocks, prospects are good. In general, dependencies may span all UL or UR which makes a good splitting unlikely.

Avoiding many cell transmissions is hard in general. In the worst case, $|\Gamma(n_1, n_2)| = n_1 n_2 - 1$ so that the last cell alone causes a quadratic amount of communication. However, we think that even in non-pathological settings, a general parallel algorithm can not avoid succumbing to communication overhead.

20 Conjecture: Unavoidable Communication Overhead
No p-scalable parallel algorithm for dynamic programming recursions that is general for case (C1) (resp. (C2)) can achieve $o(n_1 n_2)$ cell transmissions on all corresponding recursions, even if $|\Gamma(\mathbf{i})| \in o(n_1 n_2)$ for all $\mathbf{i} \in I$.

Proof Idea
Let A be a p-scalable parallel algorithm for either case. Represent I by a graph with $n_1 \cdot n_2$ nodes. Now colour the graph using p colours according to which cells are computed by the same thread using A. It is always possible to insert edges so that the graph has no directed cycles, no $(i_1, i_2) \in I$ has more than $i_1 + i_2$ incoming edges but the number of edges whose incident nodes have different colours is in $\Omega(n_1 n_2)$. ⬚

2.7 Lifting to Higher Dimensions

It is only natural to ask how dynamic programming recursions with more than two parameters can be computed in parallel. We do not investigate this in detail here but offer some thoughts on the matter.

The first remark is rather obvious: if there is a pair of parameters (i_j, i_k) for which—assuming all other parameters are set to arbitrary constants—the local two-dimensional recursion fits either case (C1) or (C2), the whole problem can be parallelised by applying a generalisation of R on all other parameters[6] and parallelising the inner two loops by DF or RS, respectively. By above results, this leads to a p-scalable and R-founded scheme. We conjecture that this condition is necessary, that is to say that a dynamic programming recursion without a suitable pair of parameters can not be parallelised efficiently.

While valid in theory this reasoning is dangerous in practise: we create overhead in the order of $\Theta(n_1 \cdot \ldots \cdot n_{m-1})$ on $I \subseteq \mathbb{N}^m$ which is less than

[6] We can reorder the parameters so that i_j and i_k are the innermost ones.

ideal. It is obvious that we can continue scheme `DF` over all parameter borders for fitting recursions, thus eliminating most of this overhead. Deeper investigation of these issues and characterisations for problem classes are outside the scope of this work and shall remain open for now.

3

Prototype Implementation

In this chapter we will present prototype implementations of the parallel algorithms devised in Chapter 2. We have encountered several challenges in the process which we outline now.

First of all, we can not assume our algorithms run in isolation, that is without contention. The operating system and eventual virtual machines can preempt any thread at any time. This introduces concurrency issues and we therefore have to synchronise more often. In particular, we can not assume a head start is sufficient when implementing `DF`; we will have to make sure a cell's dependencies have been completed before attempting to compute the cell itself.

Another concern is the cost of threads moving between processors. Apparently, some schedulers tend to move threads often even if they are busy all of the time, vainly trying to optimise processor utilisation. This is neither necessary nor advantageous in our case and thus causes unnecessary overhead. We compensate for such behaviour by using a library that ensures threads do not move and, incidentally, can use their assigned processor exclusively [Law12].

Additionally we have to be aware of memory hierarchy and communication effects as discussed in Sections 2.5 and 2.6 starting on page 33. We have tried to account for such effects by implementing the schemes in different ways; see below for the details.

Last but not least, we have decided to implement our prototypes in Java in order to investigate whether our schemes work on a portable and prevalent platform, namely the Java Virtual Machine (JVM). Its wide adoption also ensures that interested readers can reproduce our results. The price we pay is dealing with yet another layer of runtime effects introduced by the virtual machine that are hard to quantify, thus hindering interpretation of measured runtimes.

In the following we present the prototype implementations we devised and compare them. Selected parts of the code are given starting on page 57; full sources and benchmark data are available as supplemental material [Rei12].

3.1 Framework

First of all, we define suitable abstractions for both dynamic programming problems and solvers. Essentially, a problem has to offer methods to check whether cells have already been computed or can be computed, and one to compute a cell. Find the full interfaces in Listing A.1 on page 57.

Our gold standard `R` is implemented in the straightforward way in class `RowFill`. We use it not only to compare runtimes but also to unit test the other implementations; all are run against `RowFill` with several parameter combinations a hundred times each per test run.

In order to actually compare our implementations and not the computed problems, we choose problems for benchmarking that keep cache and communication overheads at a minimum. As representative of (C1) problems we select edit distance as given in Example 11 on page 14; it is implemented in class `EditDistance`. For (C2) we have implemented a most likely not meaningful dummy problem with similar complexity as `EditDistance`, that is each cell depends only on its three neighbours in the previous row; see its implementation in class `RsDummy`.

Benchmarking itself is implemented in class `Benchmark`. A completely built JAR archive can be executed with a couple of command line parameters controlling what is benchmarked. The methodology is as follows:

1. Create one solver instance per combination of implementation, number of used processors and other parameters.
2. For every input size $n \in S$, generate N random (string) inputs.
3. For every thus created input i, benchmark all available solvers in turn.
4. Run each solver r times in succession on every input. Time is measured individually for each run; we later discard both the fastest and the slowest run.

We create either quadratic problems, i.e. $n_1 = n_2 = n$, or flat ones, that is $n_2 = n$ and n_1 is constant. Size n is typically in the thousands for square and hundreds of thousands for flat problems, depending on the benchmarking machine's main memory size. For all schemes but `CF` the matrix' height has no observable impact on speedup; in those cases we are able to restrict matrix height to a constant and thus benchmark for larger row sizes. We choose N and r between five and fifteen, resulting in something between fifty and a hundred net measurements per input size.

Those measurements are written into one file per solver and parameter combination. We then use Ruby script `curate_data.rb` to aggregate average empirical speedups $S_p^A = T_1^A / T_p^A$ and real speedups T^R / T_p^A per input size. Those are written back to text files and then plotted using `gnuplot`, allowing for quick analysis.

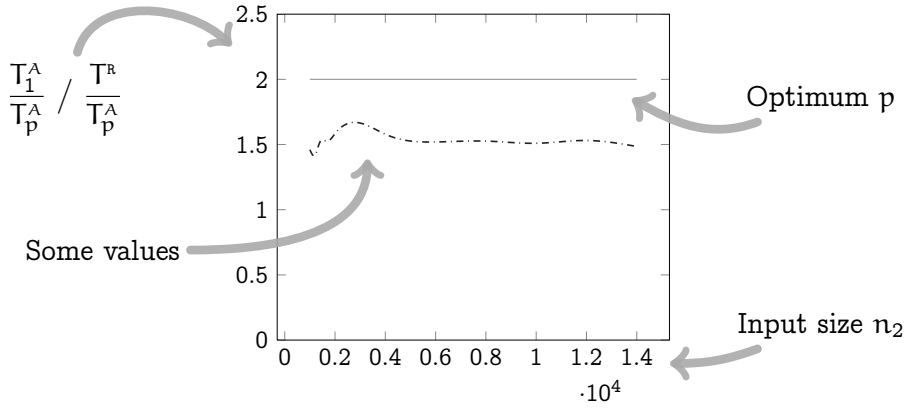


Figure 3.1: How to read benchmarking plots.

We have benchmarked on the following machines:

M_1: Intel® Core™2Duo E4600 2.4GHz (1 socket, 2 cores à 1 thread)
L1 cache: 2×32KiB (synchronous internal write-back)
L2 cache 2MiB (synchronous external write-back)
2×1GiB DIMM DDR2 667MHz 1.5ns
Linux 2.6.38-13-generic #55-Ubuntu i686 i386 GNU/Linux
Sun Java HotSpot™ Server VM 20.1-b02
Java™ SE Runtime Environment 1.6.0_26-b03

M_2: Intel® Core™i7-920 2.67GHz (1 socket (NUMA), 4 cores à 2 threads)
L1 cache: 4×32KiB (synchronous internal write-back)
L2 cache 4×256KiB
L3 cache 8MiB (synchronous internal write-back)
3×2GiB DIMM 400MHz 2.5ns
Linux 2.6.35-32-generic #66-Ubuntu x86_64 GNU/Linux
Oracle Java HotSpot™ 64-bit Server VM 22.1-b02
Java™ SE Runtime Environment 1.7.0_03-b04

The most interesting results are summarised in the respective sections below. A full account including instructions on how to run and evaluate[1] benchmarks is available in the supplements [Rei12].

How to read the plots

The upcoming sections contain lots of plots of benchmarking results. For clarity we have left out some elements, including individual data points and axis labels. See Figure 3.1 for an example plot which has been annotated with explanations.

[1] We have plots! Be warned, they are legion.

Regarding plot interpretation, we should note that the runtime of the benchmarked algorithms does not depend on the concrete inputs but only their size. Therefore, we take averages only to account for nondeterministic runtime artefacts produced by the platform. We expect such effects to be more insignificant as runtimes grow; thus our plots should be more accurate the larger the inputs. The plots do not show sample variances, though; the interested reader may want to inspect the original data [Rei12].

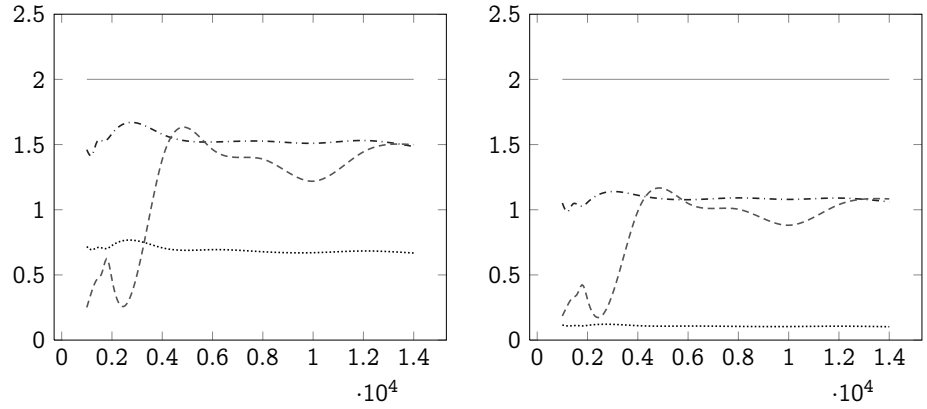
The data used for the printed plots contain between 12 and 34 points each, depending on machine and whether row number was fixed. In order to show trends more clearly, those have been smoothed by `gnuplot`'s function `smooth acsplines` with parameter `1/1000`. We have chosen the parameter such that the original data are represented fairly; nevertheless, some features may be artefacts of the smoothing process.

3.2 Implementing the Diagonal Frontier Scheme

When implementing `DF`, we have to take care that threads do not overtake each other. We do this by checking explicitly whether the next cell can be computed. In the most simple case, we actively wait on every cell we want to compute; we call this implementation `CellCheck` (cf. page 58). Because active waiting may be harmful, we have also tried putting threads to sleep for some time when a cell can not be computed, hoping that the other threads catch up in the mean time, and to wait on some synchronised gate until another thread notifies; we call these implementations `CellCheckSleep` (cf. page 58) and `CellCheckWait` (cf. page 59), respectively. The performance of those three can be seen in Figure 3.2 on the next page; it is clearly not convincing even for two processors.

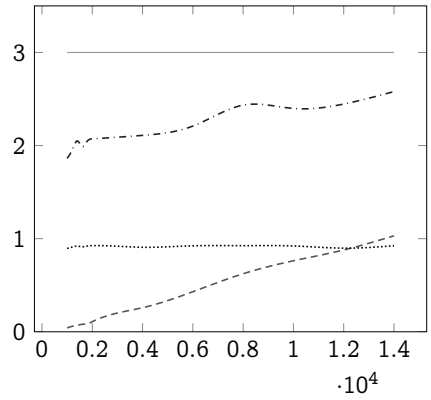
The reason is somewhat obvious: even if threads do not overtake each other—an effect we can not control in any case—we check computability for *every* cell. Given that our example cell function is not very complicated, checking whether the needed cells have already been computed is not a vanishing effort. Therefore, we introduce a hard head-start bound: we divide each row in blocks of size k, with k a fixed parameter of the scheme. When a thread wants to compute the first cell of some block, it checks whether the block's *last* cell can be computed and waits if not. Thus, we do not have to perform any checks for the other cells in the block due to the structure of Γ^+ we can assume in (C1). We call this implementation `BlockCheck` with variants similar to `CellCheck` (cf. pages 59ff). The resulting performance is depicted in Figure 3.3 on page 42; note that the choice of k did not seem to have a big influence if not too small. These variants seem to fare better than those of `CellCheck`.

As discussed in Section 2.6 on page 34 we suspect that data synchronisation between threads is an important factor in parallel performance.

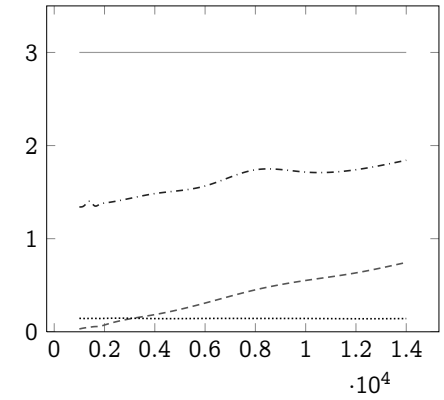


(a) Speedup on two cores

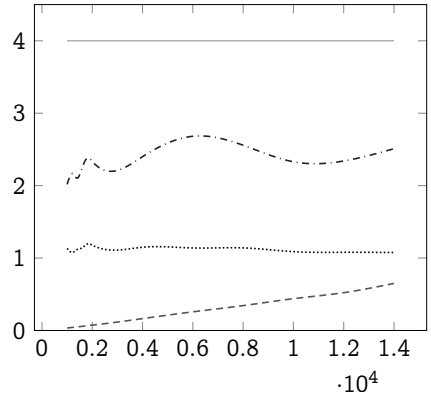
(b) Real speedup on two cores



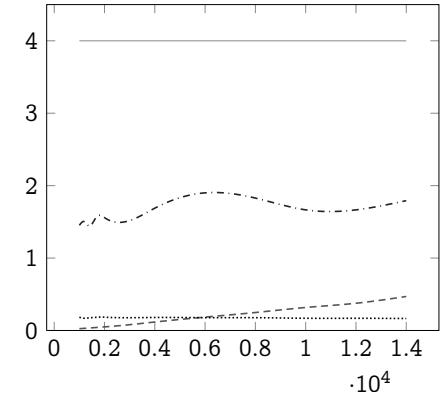
(c) Speedup on three cores



(d) Real speedup on three cores

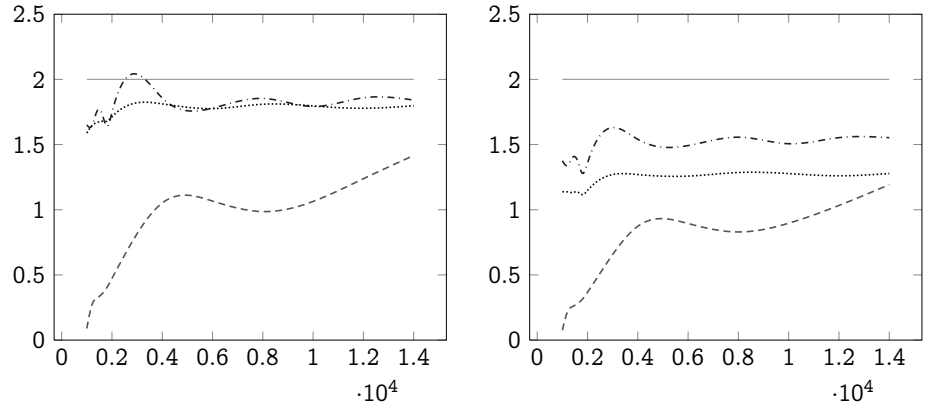


(e) Speedup on four cores



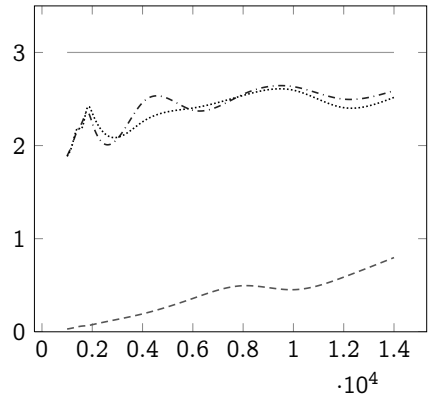
(f) Real speedup on four cores

Figure 3.2: Benchmarking results of `CellCheck` -·-·-, `CellCheckSleep` ---- and `CellCheckWait` ····· on M_2 against input size $n_1 = n_2$. Note the optimum ——.

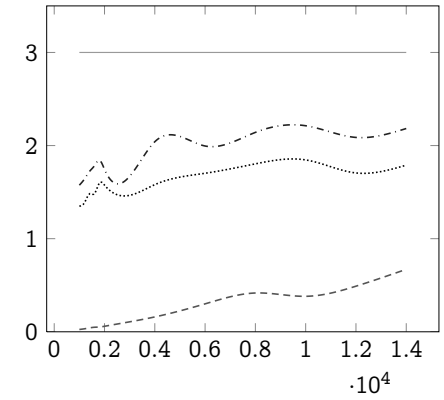


(a) Speedup on two cores

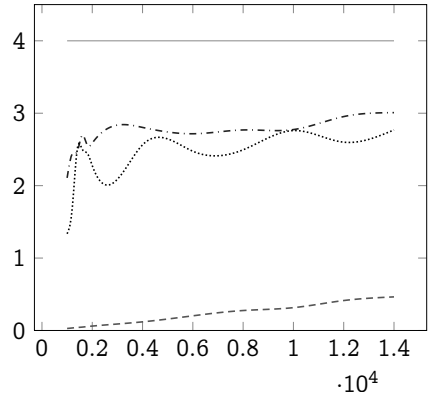
(b) Real speedup on two cores



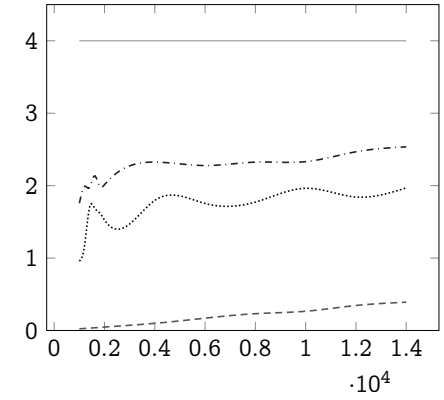
(c) Speedup on three cores



(d) Real speedup on three cores

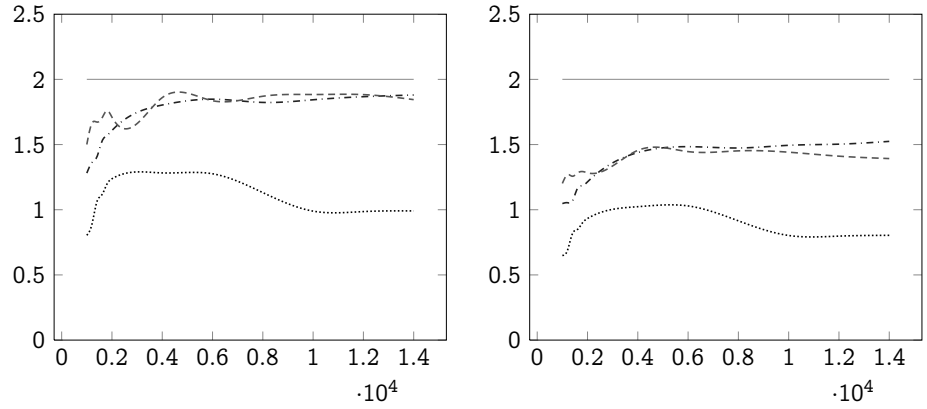


(e) Speedup on four cores



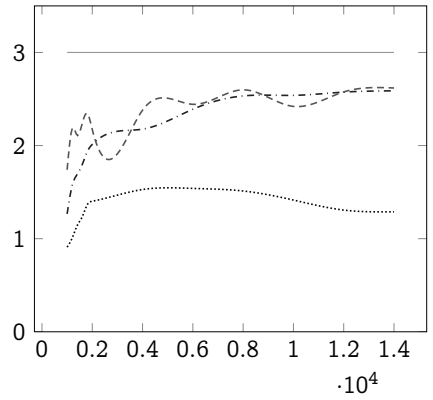
(f) Real speedup on four cores

Figure 3.3: Benchmarking results of `BlockCheck` -·-·-, `BlockCheckSleep` ---- and `BlockCheckWait` ····· with $k = 100$ on M_2 against input size $n_1 = n_2$. Note the optimum ——.

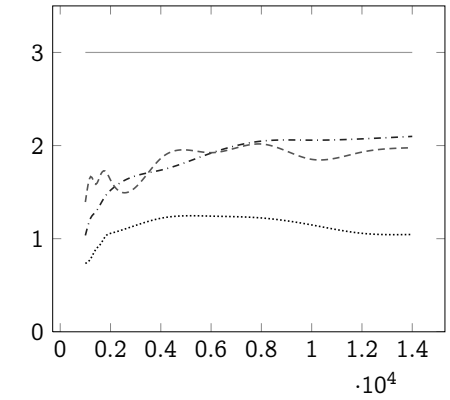


(a) Speedup on two cores

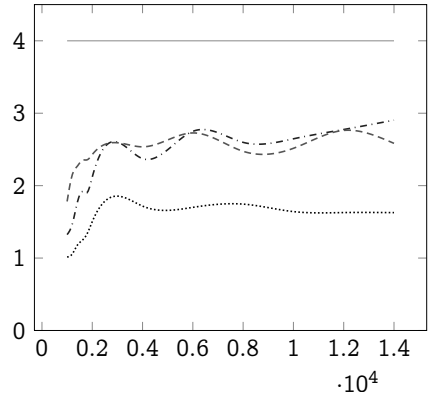
(b) Real speedup on two cores



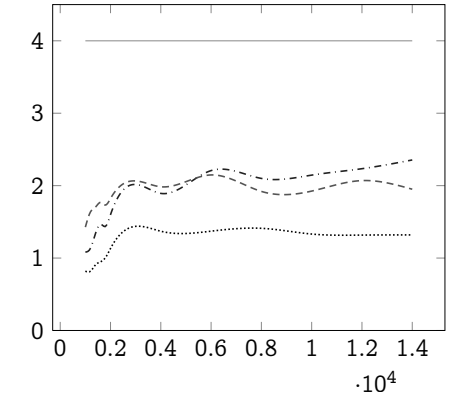
(c) Speedup on three cores



(d) Real speedup on three cores



(e) Speedup on four cores



(f) Real speedup on four cores

Figure 3.4: Benchmarking results of `ColumnBlock` with parameters (k_1, k_2) set to $(100, 100)$ ----, $(100, n_2/p + 1)$ -·-·- and $(n_1/p + 1, n_2/p + 1)$ ····· on M_2 against input size $n_1 = n_2$. Note the optimum ——.

We have therefore implemented CF as ColumnBlock (cf. page 62) with parameters k_1 the number of rows threads look ahead for checks similar to BlockCheck and k_2 the column width; we have tried both constant column width and p columns. See the obtained performance for several choices of k_1, k_2 in Figure 3.4 on the preceding page. ColumnBlock can not outperform BlockCheck by much; we suspect it pays off more for recursions with larger dependencies Γ.

In order to investigate how the implementations scale for larger input sizes, we note that for sufficiently large table height n_1 speedup of DF implementations depends mainly on n_2. We have therefore ran additional benchmarks with such flat tables; see Figure 3.5 on the next page for some results. It is interesting to note that BlockCheckSleep surpasses BlockCheckWait for big n_2. We furthermore see clearly that our schemes do not fare well on M_2 when four cores are used. Comparing 3.5a, 3.5c and 3.5e we see that speedup development becomes more erratic for growing processor number, hinting at hardware limitations. Figure 3.6 on page 46 reveals that four cores do not provide much speedup compared to three cores on M_2.

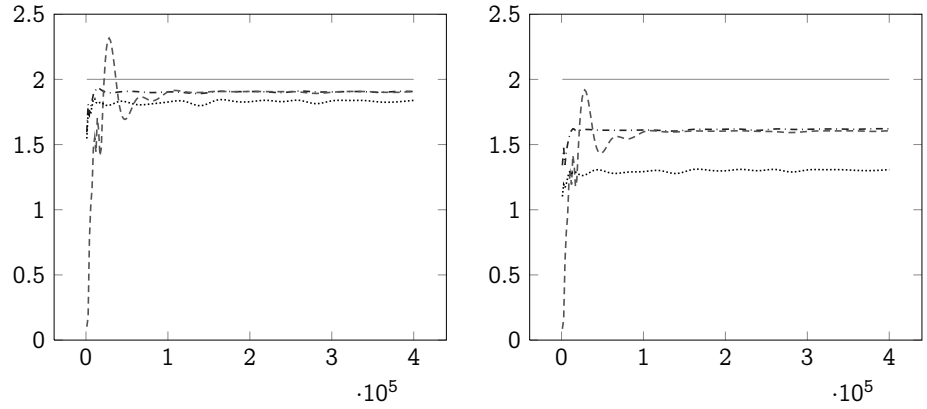
Another interesting observation is that the schemes scale better on M_1 than on M_2 with two cores; compare Figure 3.7 on page 46 with 3.5a and 3.5b. Apparently, processors with more cores can be less efficient at managing parallelism even if only few cores are used.

3.3 Implementing the Row Splitting Scheme

The row splitting scheme RS is simpler than DF and leaves little room for creativity. We provide a straight-forward implementation RowSplit (cf. page 63) which has a parameter k which controls the size of blocks it splits rows into. Again, we can not observe significant differences in performance as long as k is not chosen too small; see Figure 3.8 on page 47.

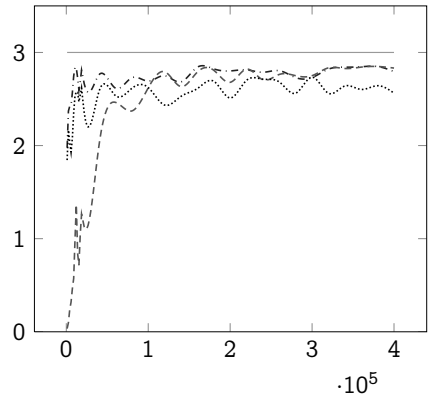
With similar reasoning as above, we investigate how RowSplit scales on flat tables and on different machines. See Figure 3.9 on page 48 and compare with Figure 3.10 on page 48. We see again that M_1 seems to be superior to M_2 on two cores, and using four cores on M_2 yields only a slight advantage compared to three cores. Interestingly, in Figure 3.9a speedup even appears to degrade slowly on M_2 with two cores as instances grow; this does not happen for other core numbers or on M_1.

We observe a curious behaviour: RowSplit surpasses the optimal real speedup; in fact, it is faster than RowFill even on one processor. We are confident that our benchmark methodology is sound and are at a loss to explain the effect.

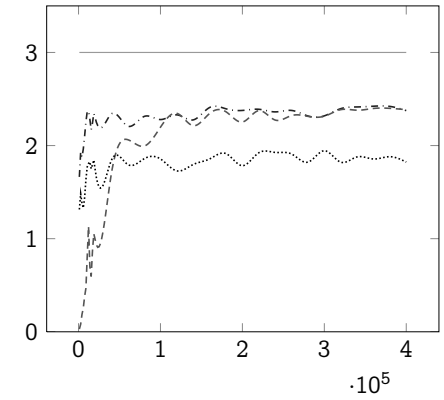


(a) Speedup on two cores

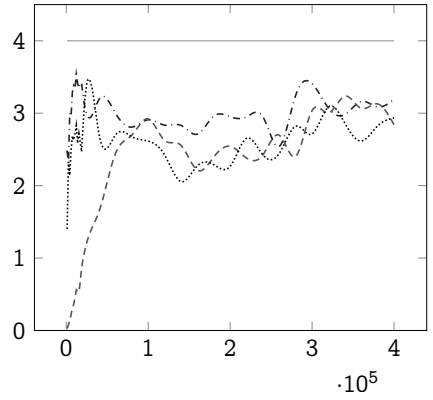
(b) Real speedup on two cores



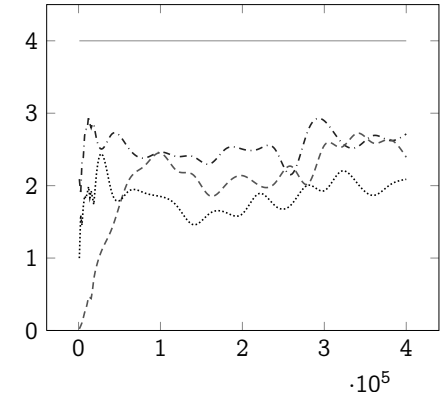
(c) Speedup on three cores



(d) Real speedup on three cores

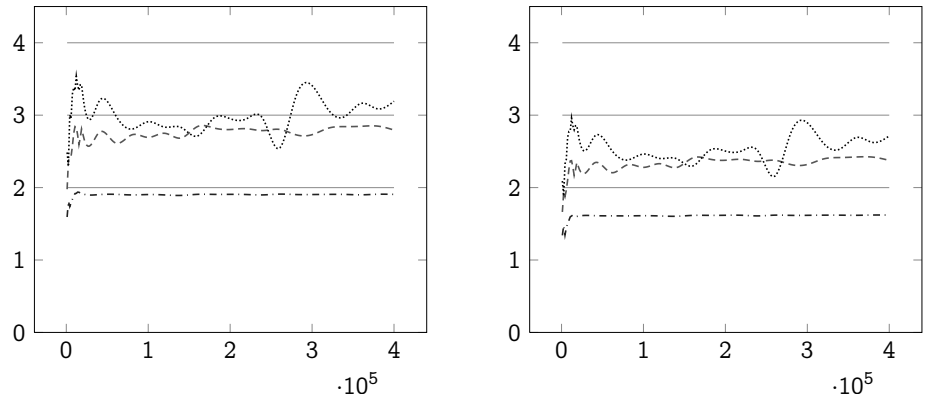


(e) Speedup on four cores



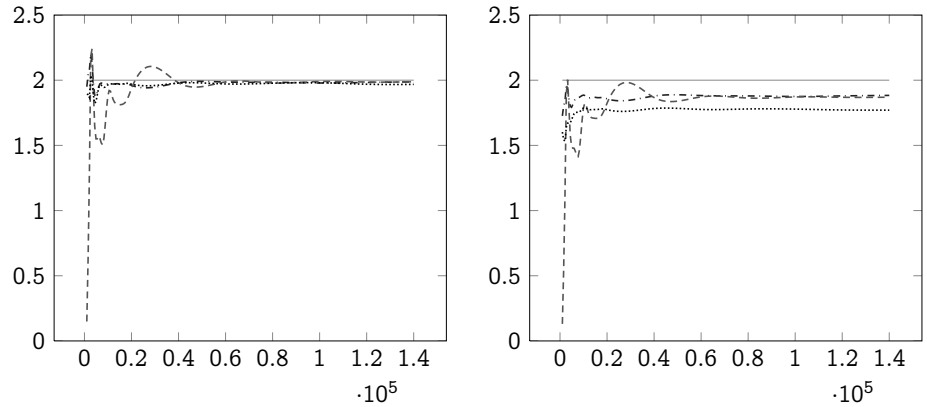
(f) Real speedup on four cores

Figure 3.5: Benchmarking results of `BlockCheck` -·-·- with $k = n_2/p+1 + 1$, `BlockCheckSleep` ---- with $k = 100$ and `BlockCheckWait` ····· with $k = 100$ on M_2 against input size n_2 for fixed $n_1 = 1000$. Note the optimum ——.



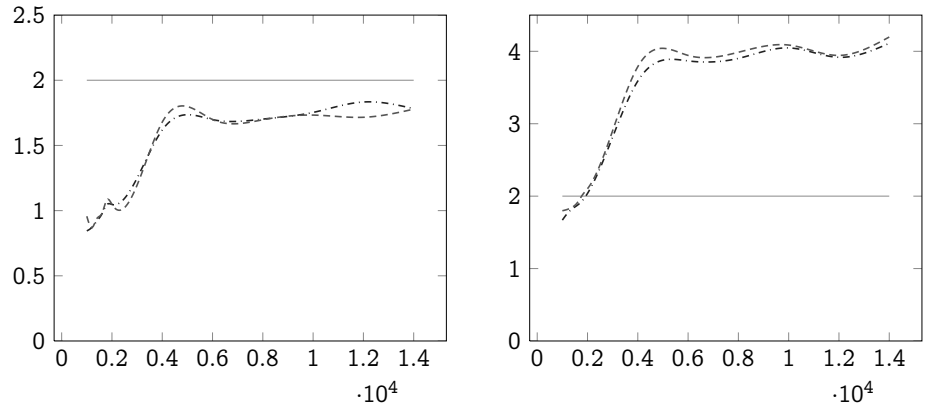
(a) Speedup (b) Real speedup

Figure 3.6: Benchmarking results of `BlockCheck` with $k = n_2/p+1 + 1$ for $p = 2$ -·-·-, $p = 3$ ---- and $p = 4$ ····· on M_2 against input size n_2 for fixed $n_1 = 1000$. Note the respective optima ——.



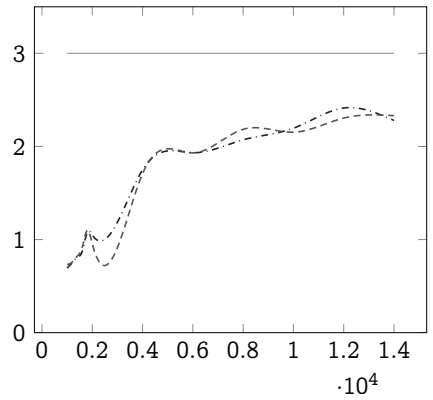
(a) Speedup on two cores (b) Real speedup on two cores

Figure 3.7: Benchmarking results of `BlockCheck` -·-·- with $k = n_2/p+1 + 1$, `BlockCheckSleep` ---- with $k = 100$ and `BlockCheckWait` ····· with $k = 100$ on M_1 against input size n_2 for fixed $n_1 = 1000$. Note the optimum ——.

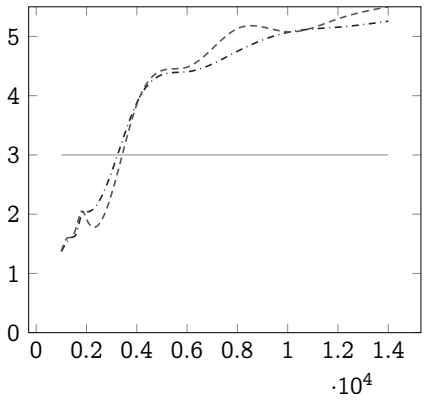


(a) Speedup on two cores

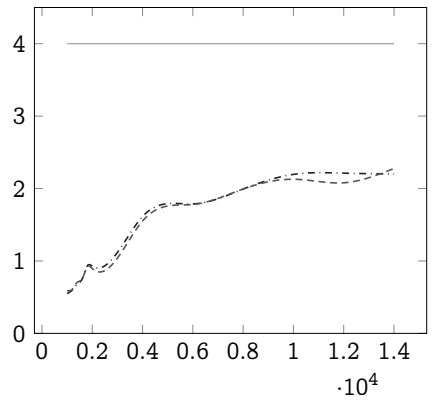
(b) Real speedup on two cores



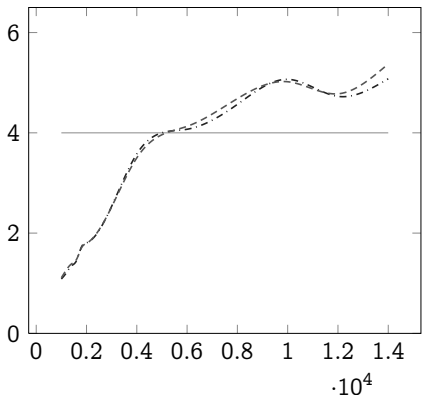
(c) Speedup on three cores



(d) Real speedup on three cores

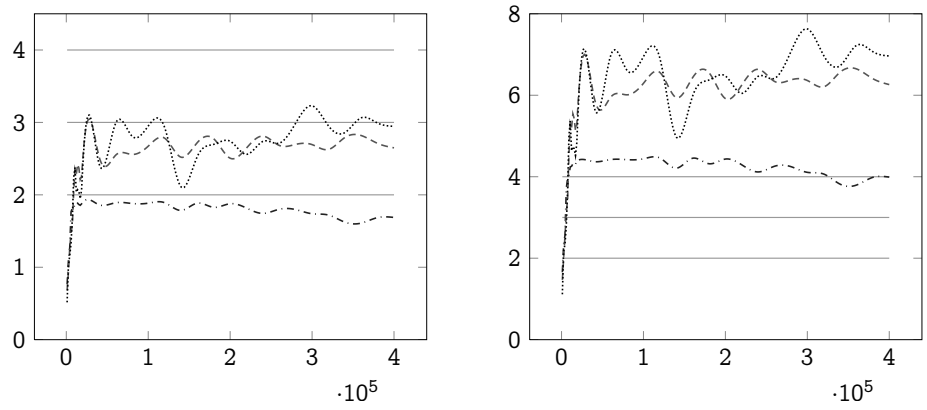


(e) Speedup on four cores



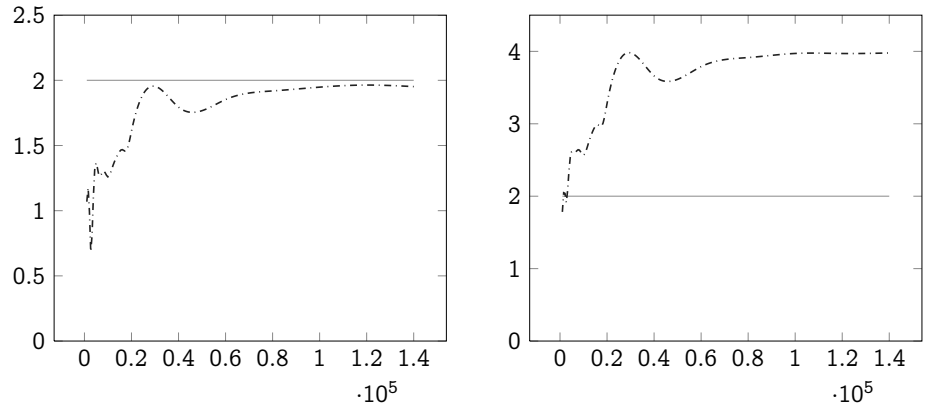
(f) Real speedup on four cores

Figure 3.8: Benchmarking results of `RowSplit` with parameter k set to 100 -·-·- and $n_2/p + 1$ ---- on M_2 against input size $n_1 = n_2$. Note the optimum ——.



(a) Speedup (b) Real speedup

Figure 3.9: Benchmarking results of `RowSplit` with $k = n_2/p+1 + 1$ for $p = 2$ -·-·-, $p = 3$ - - - - and $p = 4$ ····· on M_2 against input size n_2 for fixed $n_1 = 1000$. Note the respective optima ——.



(a) Speedup on two cores (b) Real speedup on two cores

Figure 3.10: Benchmarking results of `RowSplit` with $k = n_2/p+1 + 1$ -·-·- on M_1 against input size n_2 for fixed $n_1 = 1000$. Note the optimum ——.

Summary

We have found decent implementations of the algorithms derived in Chapter 2. While the theoretical results can not be replicated completely, the implementations scale well enough. Performance does not collapse completely for the investigated processor numbers; even if the measured speedups are not nearly optimal for higher numbers, more cores promise better performance. There seem to be limits imposed by hardware and software platform restrictions, though. We have also seen that the choice of scheme parameters can make a difference. It may be possible to choose them depending on the machine at hand by *auto-tuning*, a technique that benchmarks parametric algorithms during compilation [LTC10].

Note that the provided implementations can after some polishing be used as a library.

4

Compiler Integration

In the previous chapter, we have devised prototype implementations of the dynamic programming parallelisation schemes proposed in Chapter 2. While we have not achieved optimal speedup in practise, we have seen *some* speedup. If we can extend an existing compiler to apply our schemes with little effort for the programmer, non-optimal speedups are a lesser concern; we can view our approach as a compiler optimisation that speeds up critical parts of an application by factor three or more[1].

We have decided to extend the Scala[2] compiler as it offers a rich interface for compiler plugins, that is the compiler can be extended without changing its core. Plugins can be injected after any compiler phase and have full access to the decorated abstract syntax tree. Furthermore, Scala resides on the JVM so the scheme implementations we have are a good fit.

We will have to lower our sights with respect to performance for two reasons beyond scheme implementation. First, users might use complex data types as cell content, such as tuples or sets. On the JVM, arrays of such values are always arrays of references which causes our schemes to dereference to the heap a lot. Secondly, we have to assume that any value is also a legal cell content, so we need another way of marking cells that have not yet been computed. We use Scala's `Option` datatype for this, which might cause yet another object to be created and dereferenced for every cell. Note that in Chapter 3, we used integers as domain and otherwise illegal value -1 for marking uncomputed cells, circumventing both issues.

We have identified the following four key steps our extension has to perform:

1. Detect methods that are dynamic programming recursions.

2. Modify detected recursions so they fit our assumptions[3].

[1] In fact, we provide even exponential speedup relative to the recursive user-provided code.

[2] See scala-lang.org for extensive information about Scala.

[3] In particular top-left orientation and compatibility with `R`.

```
def editd(a : String, b : String)
         (i : Int = a.length,
          j : Int = b.length) : Int = {
  (i,j) match {
    case (0,0) => 0
    case (0,j) => j
    case (i,0) => i
    case (i,j) => (editd(a,b)(i-1,j) + 1) min
                  (editd(a,b)(i,j-1) + 1) min
                  (editd(a,b)(i-1,j-1) +
                    (if (a(i-1) != b(j-1)) 1
                     else                  0))
  }
}
```

Figure 4.1: Edit distance as recursive method in idiomatic Scala.

3. Decide which of the three cases applies.

4. Rewrite the methods to use the appropriate scheme.

For the purpose of providing a proof-of-concept implementation we will focus on the last step; we delegate most complexity of the other steps to the user.

Detection of dynamic programming recursions is not possible in general; note that in the context of this work, we do not even have a formal definition of what to look for on a semantic level. Therefore, we assume users know when they have a dynamic programming problem and recursion at hand[4], for instance something similar to the code given in Figure 4.1. If they want to invoke our transformation, they have to annotate such methods with `@DynamicProgramming`.

We omit all attempts at salvaging dynamic programming recursions that do not fit our paradigm, that is can not be solved by standard algorithm `R`. It certainly is impossible in general and we trust our users to be able to express their recursions in a compatible way.

Selecting the correct case comes down to assigning a subset of L, UL, U and UR to every recursive call; the rest is immediate. As parameters can be computed in arbitrarily complex ways, this again is an impossible task in general. We implement some basic heuristics, though; recursive calls with parameters of the form `(i + c)` with `c` an integer literal and similar are easy to deal with. For more complicated situations we provide markers which users can put in front of recursive calls in order to inform our de-

[4]Or, equivalently: they are domain experts which are novice or lazy programmers.

```
@DynamicProgramming
def editd(a : String, b : String)
         (i : Int = a.length,
          j : Int = b.length) : Int = {
  (i,j) match {
    case (0,0) => 0
    case (0,j) => j
    case (i,0) => i
    case (i,j) => ({L;  editd(a,b)(i-1,j)} + 1) min
                  ({U;  editd(a,b)(i,j-1)} + 1) min
                  ({UL; editd(a,b)(i-1,j-1)} +
                     (if (a(i-1) != b(j-1)) 1
                      else                  0))
  }
}
```

(a) Annotated recursion as plugin input.

```
def editd(a : String, b : String)() : Int = {
  val arr = Array.fill[Option[Int]]
              (a.length + 1, b.length + 1)
              (None)
  val f = (i : Int, j : Int) => (i,j) match {
    case (0,0) => 0
    case (0,j) => j
    case (i,0) => i
    case (i,j) => (arr(i-1)(j).get + 1) min
                  (arr(i)(j-1).get + 1) min
                  (arr(i-1)(j-1).get +
                    (if (a(i-1) != b(j-1)) 1
                     else                  0))
  }

  BlockCheck(arr, f);
  arr(a.length)(b.length).get
}
```

(b) Plugin output.

Figure 4.2: Edit distance as annotated recursion in idiomatic Scala (a) and our plugin's output (b). Note how the highlighted parts are used respectively rewritten during the process.

cision. Note that we trust those markers blindly; chaos awaits those who mark incorrectly. If you have to do most of the work, anyway, you can also force the decision to one case by passing it as parameter to the method annotation; for instance `@DynamicProgramming(Case2)` will cause `RS` to be used no matter what. See Figure 4.2a on the preceding page for a fully annotated example and note how minimally invasive our approach is compared to programming an efficient solution yourself.

Finally, we transform the annotated method to use one of our implementations. If we can figure out areas at all, we use at the very least `R`; for the parallelisable cases we have chosen reimplementations of `BlockCheck` and `RowSplit`, both with $k = 100$. See Figure 4.2b on the previous page for the desired final result; note how the biggest part—that is cell function `f`—can be copied with only small changes.

The plugin's implementation itself is mostly tedious and we therefore leave details out here; the curious reader may check pages 64 and 65 for some exemplary code. See also pages 66 to 68 for some examples of code that is analysed correctly. Refer to the supplements [Rei12] for full sources and instructions on how to use them.

5

Conclusion

In Chapter 1 we have proposed a way of evaluating parallel algorithms that focuses on optimising speedup on a given number of processors and is abstract enough to allow rigorous analysis. We have also introduced an abstraction of discrete dynamic programming problems that emphasises features important for parallel computation.

In Chapter 2 we have used both to derive a characterisation of parallelisable dynamic programming problems in a simplified setting. In this setting, we have found asymptotically optimal parallel variants of the classical yet optimal row by row filling algorithm for two cases; these algorithms are general for huge classes of dynamic programming problems. In a third case, we have shown that no scalable parallel algorithm exists.

We have put these theoretic results to the test in Chapter 3 and suggested several prototype implementations. Aside from inevitable disruptive influence of real platforms, we have seen practise agree with theory to a reasonable extent.

Finally, we have integrated a proof of concept, semi-automatic transformation from dynamic programming recursion to parallel iterative algorithm into a real-world compiler in Chapter 4.

In summary, we have shown that semi-automated parallelisation of certain dynamic programming recursions is feasible and—provided the end result's performance can be improved—profitable.

5.1 Future Work

Literally all elements of the above can be improved so there is no want for work opportunities.

First of all, we let both machine model and programming language be implicit and very vague. This has served our purpose well but investigating our results on well defined models is certainly a point of interest. We have also ignored memory hierarchies for the most part—note our forays in Sections 2.5 and 2.6— which may or may not have affected our decision

process negatively; rigorous investigation and improvement of the proposed schemes with respect to advanced cost models is certainly desirable. We suspect that what we tentatively call the *dependency radius*, that is the maximum distance between $\mathbf{i}$ and elements in $\Gamma_d(\mathbf{i})$, may be a quantity of interest.

Furthermore, our theoretical treatment is not as general as one might wish. Higher dimensional problems have to be investigated more closely, as have more intricate structures of Γ_d^+ than we have considered.

On the practical side, we have only provided prototype implementations. There is certainly room for improvement, following up on further theoretic results but also in the way of low-level optimisations that might necessitate switching to a more machine-oriented programming language. In particular, optimising storage of memoisation matrices with respect to locality offers rich potential. On a more humble note, our prototypes have not yet been benchmarked on problems that have larger dependency sets dep than edit distance, and also not on machines with many cores, that is more cores than typical personal computers have; these gaps needs to be closed.

Another useful extension might be enabling result functions more complex than choosing the element in the lower right corner. For instance, edit distance can be modified to ask for the score of *semi-global* alignments which allow a free suffix of deletions; the result is then the minimal value in the last row. Even more interesting is *backtracking* which requires to traverse backwards through the matrix. Neither our prototypes nor our compiler plugin support such concepts so far, again leaving room for further work.

Improving compiler integration is a bottomless pit. So far, we have implemented only rudimentary case detection; there are immediate extensions, such as recognising that `{assert(k > 0); d(i,j-k)}` hits area L and similarly for variations. We have also restricted ourselves to syntactical detection so far; later phases of the compiler might provide useful information, e.g. data flow analysis, that can inform our decision making. Additionally, the generated code's performance is less than exciting and has to be improved.

Lastly, we envision that improved successors of our approach be integrated with one of the tools out there that derive sequential dynamic programming algorithms from abstract problem descriptions.

Acknowledgements

First of all, I have to thank my advisors Markus Nebel and Umut Acar for supporting the adventure of this thesis. Their respective group members Frank Weinberg in one and Arthur Charguéraud and Mike Rainey in the other building deserve thanks for spotting small gaps in my theory early on respectively providing useful pointers to work about parallel algorithms and programming.

Shoutouts go into the virtual realms to `irc.freenode.net/scala` chat resident retronym who helped dissolving a Scala programming roadblock, and to the folks on TEX Stack Exchange for providing the best resource for all LATEX problems and completely necessary embellishments. Also warm greetings to Computer Science Stack Exchange and its denizens for helping with turning procrastination into productive quality time.

Big thanks are due to quality control task force members Ines Raschendorfer, Jan Bormann and Lars Hüttenberger for their invaluable feedback that helped catching many small inaccuracies and not so small presentation issues; you simply can not find those in your own texts.

Sebastian Wild deserves thanks for sharing Anika's office space as well as sweets and coffee with me, and for being up to discussing everything interesting we discovered over time, and sometimes even our work. He and Uli Laube also introduced me to the world of LATEX details normal people need a magnifying glass to see; Uli's package-of-the-day initiative was too infectious.

My friends are to be commended and thanked for respecting my partial free-time hiatus during the final weeks despite me scoffing at them for the very same thing earlier.

And last but certainly not least, heartfelt love and thanks to sambo Janina for bearing with me.

A

Source Code

A.1 Prototype Implementations

Listing A.1: Interfaces for dynamic programming problems and solvers.

```
interface DynProgProblem<T> extends Cloneable {
  int[] getDimension();
  boolean isComputed(int[] i);
  boolean isComputable(int[] i);
  void compute(int[] i);
  T getSolution();
  boolean isSolved();
  DynProgProblem<T> clone();
}

interface DynProgSolver {
  void solve(DynProgProblem<?> problem);
}
```

Listing A.2: Core loop of R implementation.

```
final int[] dim = problem.getDimension();
final int[] param = new int[2];

for ( param[0]=0; param[0]<dim[0]; param[0]++ ) {
  for ( param[1]=0; param[1]<dim[1]; param[1]++ ) {
    problem.compute(param);
  }
}
```

Listing A.3: Core loop of DF implementation CellCheck.

```
final int[] param = new int[2];
final int[] dim = problem.getDimension();

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  for ( param[1]=0; param[1]<dim[1]; param[1]+=1 ) {
    while ( !problem.isComputable(param) ) {
      Thread.yield();
    }

    problem.compute(param);
  }
}
```

Listing A.4: Core loop of DF implementation CellCheckSleep.

```
final int[] param = new int[2];
final int[] dim = problem.getDimension();

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  for ( param[1]=0; param[1]<dim[1]; param[1]+=1 ) {
    while ( !problem.isComputable(param) ) {
      try {
        Thread.sleep(1);
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    problem.compute(param);
  }
}
```

Listing A.5: Core loop of DF implementation CellCheckWait.

```
final int[] param = new int[2];
final int[] dim = problem.getDimension();
final int leftNeighbour = ((w - 1) % p) >= 0
                        ? (w - 1) % p
                        : p + ((w - 1) % p);

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  for ( param[1]=0; param[1]<dim[1]; param[1]+=1 ) {
    note.waitWhileNotComputable(leftNeighbour,
                                problem,
                                param);
    problem.compute(param);
    note.notify(w);
  }
}
```

Listing A.6: Core loop of DF implementation BlockCheck.

```
final int[] param = new int[2];
final int[] checker = new int[2];
final int[] dim = problem.getDimension();

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  checker[0] = Math.max(0, param[0] - 1);
  param[1] = 0;

  for ( int offset=0; offset<dim[1]; offset+=k ) {
    checker[1] = Math.min(dim[1] - 1,
                          offset + k - 1);
    while (    param[0] != 0
          && !problem.isComputed(checker) ) {
      Thread.yield();
    }

    for ( ; param[1]<Math.min(dim[1], offset + k);
            param[1]+=1 ) {
      problem.compute(param);
    }
  }
}
```

Listing A.7: Core loop of DF implementation BlockCheckSleep.

```
final int[] param = new int[2];
final int[] checker = new int[2];
final int[] dim = problem.getDimension();

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  checker[0] = Math.max(0, param[0] - 1);
  param[1] = 0;

  for ( int offset=0; offset<dim[1]; offset+=k ) {
    checker[1] = Math.min(dim[1] - 1,
                          offset + k - 1);
    while (   param[0] != 0
           && !problem.isComputed(checker) ) {
      try {
        Thread.sleep(1);
      }
      catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    for ( ; param[1]<Math.min(dim[1], offset + k);
            param[1]+=1 ) {
      problem.compute(param);
    }
  }
}
```

Listing A.8: Core loop of DF implementation BlockCheckWait.

```
final int[] param = new int[2];
final int[] checker = new int[2];
final int[] dim = problem.getDimension();
final int k = blockSize > 0 ? blockSize
                            : dim[1]/(p + 1) + 1;
final int leftNeighbour = ((w - 1) % p) >= 0
                        ? (w - 1) % p
                        : p + ((w - 1) % p);

for ( param[0]=w; param[0]<dim[0]; param[0]+=p ) {
  checker[0] = Math.max(0, param[0] - 1);
  param[1] = 0;

  for ( int offset=0; offset<dim[1]; offset+=k ) {
    checker[1] = Math.min(dim[1] - 1,
                          offset + k - 1);
    if ( param[0] != 0 ) {
      note.waitWhileNotComputed(leftNeighbour,
                                problem,
                                checker);
    }

    for ( ; param[1]<Math.min(dim[1], offset + k);
            param[1]+=1 ) {
      problem.compute(param);
    }

    note.notify(w);
  }
}
```

Listing A.9: Core loop of DF implementation ColumnBlock.

```
final int[] param = new int[2];
final int[] checker = new int[2];
final int[] dim = problem.getDimension();
final int cw = columnWidth > 0 ? columnWidth
                               : dim[1]/p + 1;
final int k = blockSize > 0 ? blockSize
                            : dim[0]/p + 1;
final int leftNeighbour = ((w - 1) % p) >= 0
                        ? (w - 1) % p
                        : p + ((w - 1) % p);

for ( int coffset=w*cw;
      coffset<dim[1];
      coffset = Math.min(dim[1], coffset + cw*p) ) {
  checker[1] = coffset - 1;

  for ( int roffset=0;
        roffset<dim[0];
        roffset = Math.min(dim[0], roffset+k) ) {
    checker[0] = Math.min(dim[0] - 1, roffset + k);

    if ( coffset > 0 ) {
      note.waitWhileNotComputed(leftNeighbour,
                                problem,
                                checker);
    }

    for ( param[0]=roffset;
          param[0]<Math.min(dim[0], roffset+k);
          param[0]++ ) {
      for ( param[1]=coffset;
            param[1]<Math.min(dim[1], coffset+cw);
            param[1]++ ) {
        problem.compute(param);
      }
    }

    note.notify(w);
  }
}
```

Listing A.10: Core loop of RS implementation RowSplit.

```
// param = new int[] { 0, 0 };
// dim =  prob.getDimension();

final int k = blockSize > 0 ? blockSize
                            : dim[1]/p + 1;

for (; param[0] < dim[0]; param[0]++) {
  for (int o = nr * k; o < dim[1]; o += p * k) {
    for ( param[1] = o;
          param[1] < Math.min(o + k, dim[1]);
          param[1]++ ) {
      prob.compute(param);
    }
  }

  // sync
}
```

A.2 Compiler Plugin Samples

Listing A.11: Tree transformer that removes markers from recursive calls.

```
object MarkerRemover extends Transformer {
  override def transform(tree : Tree) : Tree = {
    val tree1 = super.transform(tree)

    tree1 match {
      case Block(statements, result) => {
        val nonLabel = statements.map { c =>
          c match {
            case i @ Ident(n) => {
              if ( Seq("L","UL","U","UR") contains
                   n.toString ) {
                None
              }
              else {
                Some(i)
              }
            }
            case _ => Some(c)
          }
        }.flatten

        if ( nonLabel.size == 0 ) {
          result
        }
        else {
          treeCopy.Block(tree1,
                         nonLabel,
                         result)
        }
      }
      case _ => tree1
    }
  }
}
```

Listing A.12: Tree transformer that rewrites recursive calls to array accesses.

```
class RecursionRewriter(val method_name : String,
                        val array : Tree)
    extends Transformer {
  override def transform(tree : Tree) : Tree = {
    val tree1 = super.transform(tree)

    tree1 match {
      case Apply(Apply(Ident(name), args1),
                 args2) => {
        if ( method_name == name.toString() ) {
          Select(Apply(Apply(array, args2 take 1),
                       args2 drop 1),
                 newTermName("get"))
        }
        else {
          tree1
        }
      }
      case _ => tree1
    }
  }
}
```

Listing A.13: Edit distance in Scala. The plugin detects and translates it correctly.

```
@DynamicProgramming
def editd(a : String, b : String)
         (i : Int = a.length,
          j : Int = b.length) : Int = {
  (i,j) match {
    case (0,0) => 0
    case (0,j) => j
    case (i,0) => i
    case (i,j) => (editd(a,b)(i-1,j) + 1) min
                  (editd(a,b)(i,j-1) + 1) min
                  (editd(a,b)(i-1,j-1) +
                   (if (a(i-1) != b(j-1)) 1 else 0))
  }
}
```

Listing A.14: Longest common subsequence in Scala. The plugin detects and translates it correctly.

```
@DynamicProgramming
def lcs(a : String, b : String)
       (i : Int = a.length,
        j : Int = b.length) : Int = {
  (i,j) match {
    case (0,_) | (_,0)              => 0
    case (i,j) if a(i-1) == b(j-1) =>
      lcs(a,b)(i-1,j-1) + 1
    case (i,j)                      =>
      lcs(a,b)(i,j-1) max lcs(a,b)(i-1,j)
  }
}
```

Listing A.15: Bellman-Ford algorithm in Scala. The plugin detects it correctly, but the translated version causes the compiler to crash.

```
@DynamicProgramming
def shortestPath[T](cost : Array[Array[Option[Int]]],
                    start : Int)
                   (i : Int = cost.length,
                    j : Int = cost.length - 1)
                   : Option[Int] = {
  (i,j) match {
    case (0,s) if s == start => Some(0)
    case (0,s) if s != start => None
    case (i,j) => ((0 until cost.length) map { k =>
        ({UL; U; UR;
          shortestPath(cost, start)(i-1,k)},
         cost(k)(j)
        ) match {
          case (Some(l),Some(c)) => Some(l + c)
          case _                 => None
        }
      }).flatten match {
        case Seq() => None
        case s     => Some(s.min)
      }
  }
}
```

Listing A.16: CYK algorithm in Scala. The plugin detects it correctly, but the translated version causes the compiler to crash.

```
def cyk(input : String, g : CNFGrammar) =
  cykh(input, g)() contains g.start

@DynamicProgramming
private def cykh(input : String, g : CNFGrammar)
                (i : Int = input.length - 1,
                 j : Int = 0) : Set[Grammar.N] = {
  (i,j) match {
    case (0,j) => g.nonterminals filter
                    (g.trules contains (_,input(j)))
    case (i,j) if j + i < input.length => {
      (1 to i) map { k : Int =>
        g.nonterminals filter { n =>
          g.nrules exists { case (n0, (n1,n2)) =>
             (n0 == n) &&
             ({U;  cykh(input, g)(i-k,j)}
                contains n1) &&
             ({UR; cykh(input, g)(k-1,j+i-k+1)}
                contains n2)
          }
        }
      } reduce (_ ++ _)
    }
    case _ => Set() : Set[Grammar.N]
  }
}
```

B

Glossary

Notation

General

i, n	monadic values
$\mathbf{i}, \mathbf{j}$	polyadic values, e.g. $\mathbf{i} = (i_1, i_2)$
$\mathbb{N}$	$\{0, 1, 2, 3, \dots\}$
$\mathbb{N}_+$	$\{1, 2, 3, \dots\}$
S^k	k-ary Cartesian product
$o, \mathcal{O}, \Theta, \Omega, \omega$	Landau symbols
$f \sim g$	asymptotic equality, i.e. $\lim_{x\to\infty} {}^{f(x)}/_{g(x)} = 1$
$[P]$	1 if predicate P is true, 0 else
$[a..b]$	$\{a, a+1, \dots, b-1, b\}$

Thesis-specific

$\tilde{o}(f)$	$o(f) \cup \{g : f \sim g\}$
T_p^A	Parallel runtime, see Definition 2 on page 6
S_p^A	Parallel speedup, see Definition 3 on page 7
$\Gamma_f, \Gamma_f^+, \widetilde{\Gamma}_f, \widetilde{\Gamma}_f^+$	Cell dependencies, see Definition 8 on page 12

`CF`	Column filling scheme, see page 34.
`DF`	Diagonal frontier scheme, see page 24.
`R`	Row by row filling, see page 21.
`RS`	Row splitting scheme, see page 28.

Definitions

1 Parallel Algorithm 6
2 Runtime 6
3 Speedup 7
5 Scalability 7
6 Foundedness 8
8 Computation Dependency 12
10 Dynamic Programming 14
12 Dependency Areas 17

Examples

7 Parallel Evaluation of Sums 8
9 Computation Dependencies of Fibonacci Numbers 13
11 Edit Distance 14
15 Edit Distance (continued) 22
17 Single-Source Shortest Paths 27

Results

13 Complete Case Distinction 20
16 (C1) Parallelised by Diagnoal Frontier Scheme 23
18 (C2) Parallelised by Row Splitting Scheme 27
19 (C3) Not Parallelisable 30
20 Unavoidable Communication Overhead 35

Bibliography

[ACDS03] Carlos E. R. Alves, Edson Cáceres, Frank K. H. A. Dehne, and Siang W. Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Proceedings of the 2003 international conference on Computational science and its applications*, volume 2 of *ICCSA'03*, pages 249–258. Springer-Verlag Berlin, Heidelberg, 2003. URL http://dl.acm.org/citation.cfm?id=1762008.1762040.

[ARQ93] Rumen Andonov, Frédéric Raimbault, and Patrice Quinton. Dynamic programming parallel implementations for the knapsack problem. Rapport de recherche RR-2037, INRIA, 1993. URL http://hal.inria.fr/inria-00074634.

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume Volume 1: Parsing. Prentice-Hall, 1972.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ble96] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39:85–97, March 1996. ISSN 0001-0782. URL http://doi.acm.org/10.1145/227234.227246.

[Bra94] Phillip Gnassi Bradford. *Parallel Dynamic Programming*. PhD thesis, Indiana University, 1994. URL http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR424.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–206, April 1974. ISSN 0004-5411. URL http://doi.acm.org/10.1145/321812.321815.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 978-0-262-53305-8.

[EI96] Ömer Eğecioğlu and Maximilian Ibel. Parallel algorithms for fast computation of normalized edit distances. In *Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 496–503. IEEE, 1996.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223. ACM, New York, NY, USA, 1998. ISBN 0-89791-987-4. URL http://doi.acm.org/10.1145/277650.277725.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118. ACM, New York, NY, USA, 1978. URL http://doi.acm.org/10.1145/800133.804339.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN 978-0521585194.

[Hoß83] Friedel Hoßfeld. *Parallele Algorithmen*, volume 64 of *Informatik-Fachberichte*. Springer, 1983. ISBN 3-540-12283-4.

[IT94] Oscar Ibarra and Nicholas Trân. On the parallel complexity of solving recurrence equations. In Ding-Zhu Du and Xiang-Sun Zhang, editors, *Algorithms and Computation*, volume 834 of *Lecture Notes in Computer Science*, pages 469–477. Springer Berlin / Heidelberg, 1994. ISBN 978-3-540-58325-7. URL http://dx.doi.org/10.1007/3-540-58325-4_213.

[KRS90] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95 – 132, 1990. ISSN 0304-3975. URL http://www.sciencedirect.com/science/article/pii/030439759090192K.

[Law12] Peter Lawrey. Java-thread-affinity. Open source project, 2012. URL https://github.com/peter-lawrey/Java-Thread-Affinity. Accessed April 13th, 2012.

[LTC10] Jiajia Li, Guangming Tan, and Mingyu Chen. Automatically tuned dynamic programming with an algorithm-by-blocks. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE*

16th International Conference on, pages 452–459. IEEE, dec. 2010. ISSN 1521-9097.

[MAFC11] Leandro A. J. Marzulo, Tiago A. O. Alves, Felipe M. G. França, and Vítor Santos Costa. Couillard: Parallel programming via coarse-grained data-flow compilation. Version 1. Published on arXiv, September 2011. URL http://arxiv.org/abs/1109.4925.

[Mor82] Thomas L Morin. Monotonicity and the principle of optimality. *Journal of Mathematical Analysis and Applications*, 88(2):665–674, 1982. ISSN 0022-247X. URL http://www.sciencedirect.com/science/article/pii/0022247X82902232.

[NCTT09] Adrian Nistor, Wei-Ngan Chin, Tiow-Seng Tan, and Nicolae Tapus. Optimizing the parallel computation of linear recurrences using compact matrix representations. *Journal of Parallel and Distributed Computing*, 69(4):373–381, 2009. ISSN 0743-7315. URL http://www.sciencedirect.com/science/article/pii/S0743731509000094.

[Par87] Ian Parberry. *Parallel complexity theory*. Research notes in theoretical computer science. Pitman, 1987. ISBN 978-0-273-08783-0. 1–200 pp. URL http://larc.unt.edu/ian/books/free/.

[PBS11] Yewen Pu, Rastislav Bodik, and Saurabh Srivastava. Synthesis of first-order dynamic programming algorithms. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 83–98. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0940-0. URL http://doi.acm.org/10.1145/2048066.2048076.

[Rei12] Raphael Reitzig. Master thesis companion sources. Personal websites, May 2012. URL http://lmazy.verrech.net/pub/mathesis/.

[RR99] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. *SIGPLAN Not.*, 34(8):72–83, May 1999. ISSN 0362-1340. URL http://doi.acm.org/10.1145/329366.301111.

[Ryt88] Wojciech Rytter. On efficient parallel computations for some dynamic programming problems. *Theoretical Computer Science*, 59(3):297–307, 1988. ISSN 0304-3975. URL

http://www.sciencedirect.com/science/article/pii/0304397588901478.

[SJG11] Georg Sauthoff, Stefan Janssen, and Robert Giegerich. Bellman's GAP - a declarative language for dynamic programming. In *Proceedings of 13th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '11. ACM, 2011. ISBN 978-1-4503-0776-5/11/07. URL http://www.techfak.uni-bielefeld.de/~gsauthof/docs/gapl.ppdp.2011.pdf.

[Sni78] Moshe Sniedovich. Dynamic programming and principles of optimality. *Journal of Mathematical Analysis and Applications*, 65(3):586–606, 1978. ISSN 0022-247X. URL http://www.sciencedirect.com/science/article/pii/0022247X7890166X.

[TP96] Stefan Tschöke and Thomas Polzer. Portable parallel branch-and-bound library: User manual. Technical report, University of Paderborn, 1996. URL http://www2.cs.uni-paderborn.de/cs/ag-monien/SOFTWARE/PPBB/ppbblib.html.