Code generation for large scale applications

Mark, Paul Johannes van der

Code Generation for Large Scale Applications

Paul van der Mark

Code Generation for Large Scale Applications

Proefschrift
ter verkrijging van de graad van Doctor
aan de Universiteit te Leiden,
op gezag van de Rector Magnificus Dr. D. D. Breimer,
hoogleraar in de faculteit der Wiskunde en Natuurwetenschappen en
die der Geneeskunde,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 31 oktober 2006
klokke 13.45 uur

door

Paul Johannes van der Mark

Geboren te Gouda in 1970

Promotiecommissie

Promotor:	prof. dr. H. A. G. Wijshoff
Co-promotor:	dr. A. A. Wolters
Referent:	prof. dr. K. A. Gallivan (Florida State University, USA)
Overige leden:	prof. dr. ir. E. F. A. Deprettere
	prof. dr. F. J. Peters
	prof. dr. S. M. Verduyn Lunel

Code Generation for Large Scale Applications
Paul Johannes van der Mark.
Thesis Universiteit Leiden.
ISBN 90-9021120-9
Subject headings: code generation / parallel and distributed computing

Preface

This dissertation reports about my research that was performed at Leiden University between 2000 and 2004 and at Florida State University in the summer of 2001. This research was partly funded by the dutch foundation of science, NWO (project number 612-053-001) and the FSU Cornerstone Program for Centers of Excellence.

The work described in this dissertation is a continuation on previous work performed by Robert van Engelen at Leiden University. One of the initial goals of the project was to extend CTADEL with a number of advanced numerical techniques. In chapter 7 we describe how CTADEL is able to automatically generate code for semi-Lagrangian formulations. Lagrangian-type formulations have a number of interesting properties in comparison with Eulerian-type formulations, for example a possible increase in the time step size. Use of Lagrangian formulations increases complexity and thus poses a new challenge to CTADEL.

The CTADEL tool was designed with automatic code generation for numerical weather prediction models in mind. In the chapter 4, CTADEL showed its strength with conditional expressions; a number of trigger functions to determine entrainment and detrainment that take place in a cloud. In chapter 5, we describe how CTADEL can generate code for an ocean model. Another extension we made to CTADEL was to let the model specification call external library functions, which we show in chapter 6. As a test case, we specified a turbulence model. In this model we have also implemented an experimental implementation for a certain implicit differential equation.

Contents

1 Introduction **11**

1.1 Problem Solving Environments 12
1.2 Numerical Weather Forecasting 13
1.3 The CTADEL Code Generator 14
1.4 Thesis Outline 15

2 Overview **17**

2.1 Ctadel 17
2.2 A Brief Overview of the System 18
2.2.1 ATMOL: A Domain-Specific Language for Atmospheric Modeling 21
2.2.2 The GPAS Reduction System 22
2.2.3 Common Subexpression Elimination 22
2.2.4 Multi-Platform Code Generation 24
2.3 Related work 25
2.3.1 Numerical Libraries 25
2.3.2 Problem Solving Environments 26
2.3.3 Restructuring, Parallelizing and Symbolic Compilers 26

3 Experimental Setup **29**

3.1 Scalar Architecture 29
3.2 Shared Memory Parallel Architecture 30
3.3 Distributed Memory Parallel Architecture 30

4 The Kain–Fritsch Convection Scheme **33**

4.1 Background 33

4.1.1 The KF convection scheme . . . 34
4.2 Templates in Ctadel . . . 37
4.2.1 Abstraction . . . 38
4.2.2 Implementation . . . 39
4.2.3 Toy Example of a Template . . . 39
4.3 Experiments and Results . . . 42
4.3.1 Conclusion . . . 46

5 A Coupled Atmosphere–Ocean Model 47
5.1 Quasi-Geostrophic Dynamics . . . 47
5.2 Specification and Implementation . . . 50
5.2.1 Spatial resolutions . . . 50
5.2.2 Implementation . . . 51
5.3 Initialization . . . 52
5.4 Ocean and Atmosphere step . . . 54
5.5 Experimental Results . . . 54
5.6 Conclusion . . . 55

6 A Turbulence Scheme 57
6.1 Turbulence . . . 57
6.2 Turbulence scheme . . . 58
6.2.1 Physics and Dynamics . . . 58
6.2.2 CBR Turbulence Scheme . . . 60
6.3 Implementation . . . 61
6.3.1 Implicit equations . . . 62
6.3.2 Specification . . . 64
6.3.3 A Sample Code Generated by Automatic Transformation . . . 65
6.4 Results . . . 68

7 Semi-Lagrangian Formulations 75
7.1 Theory . . . 76
7.1.1 Introduction . . . 76
7.1.2 Size of the Time Step . . . 76
7.1.3 Numerical Principles of Semi-Lagrangian formulations . . . 77
7.2 Sequential Code . . . 79

7.3 Parallelization 85
7.4 Experimental Results 87
7.4.1 Setup for the Experiments 87
7.4.2 Experiments on a Scalar Architecture 88
7.4.3 Experiments on a Distributed Memory Parallel Architecture . . 89
7.5 Halo on Demand 91
7.5.1 Halo on Demand strategy 92
7.5.2 Related Work 92
7.5.3 Experiments 95
7.5.4 Experiments with current input data 97
7.5.5 Experiments with future input data 98
7.5.6 Speedup 104
7.5.7 Discussion on communication time 111
7.6 Conclusion 111

8 Conclusions **113**

Samenvatting (in Dutch) **125**

Curriculum Vitæ (in Dutch) **129**

Acknowledgments **131**

Chapter 1

Introduction

Increasing processing power and the development of new high-performance architectures such as large networks of workstations (for example the Beowulf project [73] and the Grid architecture [21]), multi-core processors [28] and simultaneous multi-threading [77] have led to opportunities for developers of numerical models to change the focus on more numerical detail, finer resolution or larger computational domains. Efficient execution of large-scale application codes is usually a primary requirement in many cases. High efficiency can only be achieved by utilizing architecture-independent efficient algorithms and exploiting specific architecture-dependent characteristics of a given computer architecture. For example, a loop optimization like software-pipelining can increase instruction level parallelism on VLIW processor architectures [44], while high-level code rewriting systems might improve performance on parallel architectures [2].

However, platform specific versions of source code must be avoided in order to limit development and maintenance complexity. This requires a significant programming effort to implement the changes in the simulation application since a simple plug-and-play development paradigm with software components for scientific applications does not exist as of yet. This can lead to huge amounts of undocumented code, for which new versions have to be developed with every new emerging computer architecture.

Usually, the problem can be formulated on an abstract level. For example, by a set of mathematical equations, which is independent of any computational resource. Several types of tools exists which deal with systems of equations on an abstract level. For example problem solving environments (PSEs), like the Matlab package, provide the user with an interactive environment for symbolic specification of models. These PSEs

execute the specification by means of an interpreter which is inefficient, however, and additional tools like a Matlab compiler are needed to obtain efficient code. Therefore, a problem-specific code generator, called CTADEL, has been developed in order to exploit architecture-independent and dependent optimizations.

1.1 Problem Solving Environments

> "A PSE is a computer system that provides all the computational facilities needed to solve a target class of problems. These features include advanced solution methods, automatic and semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. Moreover, PSEs use the language of the target class of problems, so users can run them without specialized knowledge of the underlying computer hardware or software. By exploiting modern technologies such as interactive color graphics, powerful processors, and networks of specialized services, PSEs can track extended problem solving tasks and allow users to review them easily. Overall, they create a framework that is all things to all people: they solve simple or complex problems, support rapid prototyping or detailed analysis, and can be used in introductory education or at the frontiers of science" - Gallopoulos, Houstis and Rice

Problem solving environments can provide a programming paradigm for application development by means of software composition; the automatic translation of a problem, defined at a high level of abstraction, into an executable code. Software modeling and development at a high abstract level allow the application developers to react more quickly to new developments. For example, the SCIRun is a problem solving environment that allows scientists and engineers to interactively steer a computation, changing parameters, recompute, and then revisualize all resulting data, within the same programming environment [38, 57].

Furthermore, studies have shown that programmer productivity, measured by lines of code over time, varies little between languages. Languages that automate more of the low-level work allow a programmer to accomplish more in fewer lines of code.

1.2 Numerical Weather Forecasting

Many decisions in today's society are made based on weather forecasts. Because of the major economic impact of such decisions, research and development of numerical weather forecast systems has been ongoing since the beginning of the twentieth century. Today, numerical weather forecasting, as part of climate modeling, is classified as one of the "Grand Challenges" in computational science [90]. Simulating atmospheric processes is computationally intensive: a typical forecast of the next day's weather requires about a trillion arithmetic operations. Even given the immense processing power of today's supercomputers, a 24-hour weather forecast may still take an hour to complete. Generally, this is the maximum amount of time, and no more time than this can safely be allotted in order to meet the time constraints imposed by the timely delivery of the forecast. Despite the allocation of significant computing power, the quality of weather forecasts can sometimes be disappointingly poor. One of the reasons is that atmospheric circulation processes comprise inherently unstable phenomena, and the mathematical equations governing these processes are nonlinear: a small disturbance of the atmosphere in one part of the globe may have a disproportional large effect on atmospheric motion somewhere else. This also limits the validity of long-term predictions to about two weeks ahead, at the most.

Another reason for inaccurate weather forecasts is the limited resolution of numerical forecast models in general. Grid points lie tens of kilometers apart, too coarse for modeling local meteorological effects. Though the problem of sensitivity cannot be overcome, we can still improve short-term forecasts by refining the resolution. However, since this increases the total number of operations to be performed within the time span allocated, computing power must increase accordingly.

Distributed computers can significantly speed up weather forecasting. Basically, the forecast is computed for small patches of the atmosphere in parallel. Because the atmosphere can be tiled into smaller patches, the computing time for a forecast is inversely proportional to the number of patches, assuming that each patch can be handled by one processor of the parallel computer. Although the basic principles of parallel computation are clear, the parallel implementation of weather forecasts is still a major research topic. One reason for this is that the appearance of several different types of parallel computer architectures has made application development considerably more difficult, because a single, general paradigm for parallel programming does not exist yet. For

example, for message passing several systems exist, like MPI-1 and MPI-2 [23, 49], OpenMP [10] and PVM [43]. Although MPI has been declared a standard paradigm, code written with MPI is hard to understand for non-parallel programming experts and consequently hard to maintain and use.

1.3 The CTADEL Code Generator

The original CTADEL code generator was developed at Leiden University by Robert van Engelen [86]. The design objective is the so-called machine-independent "programming-in-the-large environment"[1], which must be able to generate efficient execution codes for different computer architectures that are typically from architecture-independent problem specifications. In this respect, a challenging application for code generation is the HIRLAM limited area numerical weather forecast system [20, 32]. HIRLAM is a cooperative project of Denmark, Finland, France, Iceland, Ireland, the Netherlands, Norway, Spain, and Sweden. It is used in several of these European countries for routine weather forecast productions.

Over the past four years, several parallel implementations of the HIRLAM forecast model have been realized: a data-parallel implementation, a message-passing version, a data-transposition code, and others. All modifications required for these implementations were made by hand starting from the (vectorized) HIRLAM reference code. As a result, several versions of the forecast system are now available. Clearly, this is an undesirable situation from a maintenance point of view. It also hampers the inclusion of new insights into the model by meteorologists, since they are not acquainted with the parallelization techniques. Furthermore, making these implementations efficient on several types of high performance computer architectures results in a formidable task, since each computer system requires computer architecture-dependent optimizations.

A problem solving environment and code generator can assist the application developer in alleviating the task of coding different implementations for different target hardware architectures. Such a software development environment that integrates software solutions for a specific application is called an application driver.

[1] Programming-in-the-large is concerned with the overall architecture of software systems; it deals with the composition of large systems out of modules, the interfaces between the modules and their specification, and the evolution of the resulting architecture over time[75].

Initially the CTADEL project focused on the generation of codes for the so-called dynamics of the HIRLAM weather forecast model. In this continuation project we extended CTADEL with other advanced numerical techniques like (semi) Lagrangian techniques and we investigated the possibility of applying CTADEL to a subset of the physics routines.

Based on the successful experiences of generating codes for the HIRLAM numerical weather forecast system, the CTADEL system has gradually matured into a small-scale computer algebra system that has the potential of manipulating and transforming specifications for a wider range of applications. For example, we programmed an ocean model using the CTADEL tool.

1.4 Thesis Outline

In this section, an outline of this dissertation is given.

Chapter 2 In this chapter an overview of the CTADEL tool is given. Accompanied with an easy to understand example, the important phases of the code generation are explained. We will discuss the specification language, called ATMOL, the GPAS reduction system, the DICE common subexpression eliminator and the final code generation phase.

In addition, some related research efforts in computational science and engineering are discussed.

Parts of this chapter have been published in [81].

Chapter 3 In this chapter we show the setup for all the experiments we have performed with the generated code.

Chapters 4, 5 and 6 With a number of example applications we show how we can extend the application domain of CTADEL. We will show models which deal with a convection scheme (chapter 4), a coupled ocean–atmosphere model (chapter 5) and a turbulence scheme (chapter 6). These example models show some of the application domains for the CTADEL system and the extensions we have included in the system.

Parts of this chapter have been published in [78, 79, 80].

Chapter 7 This chapter is devoted to the incorporation and the generation of code for semi-Lagrangian formulations. The use of semi-Lagrangian formulations can increase performance of NWPs, but specifying semi-Lagrangian formulations poses new challenges on CTADEL. We show how we have implemented semi-Lagrangian formulations and how they can be used with an example model. Furthermore, we show an elaborate method, called Halo On Demand, for optimizing communication costs for distributed architectures.

Parts of this chapter have been published in [82, 83, 84, 85].

Chapter 2

Overview

In this chapter we will give an overview of the CTADEL system and we will present some of the modules of the system. For a more detailed description of the CTADEL system we refer to [86].

2.1 Ctadel

Many attempts have been made to solve scientific or physical models numerically using computers. Today, a large collection of libraries, tools, and Problem Solving Environments (PSEs) have been developed for these problems. The computational kernels of most PSEs consist of a large library containing routines for several numerical solution methods. These routines form the templates for the resulting code. Examples of these are, the Linear Algebra Kernels (LAKe) [54] or the range of libraries based on the LINear algebra PACKage (linpack) system [53], e.g. the Linear Algebra PACKage (lapack) system [59], Parallel Linear Algebra Package (plapack) [56] and the Java version of lapack (jlapack) [9].

A different approach to the so-called library-based PSEs consists of a collection of tools that generate code based on a problem specification without the use of a library. The power of such a system is determined by the expressiveness of the problem specification language and the underlying translation techniques. Examples of these systems are general purpose PSE systems like maple [46] or SciNaps [1] or domain specific PSE programs like SciFinance [71]. CTADEL belongs to this class of PSEs. Furthermore, a hardware description of the target platform is included in CTADEL's problem

specification. This makes it possible to produce efficient codes for different types of architectures.

The CTADEL system provides an automated means of generating specific high performance scientific codes. These codes are optimized for a number of different architectures such as serial, vector, or shared virtual memory and distributed memory parallel computer architectures. One of the key elements of this system is the usage of algebraic transformation techniques and powerful methods for global common subexpression elimination. These techniques ensure the generation of efficient high performance codes.

The problem specification language for CTADEL is called the ATmospheric MOdeling Language (ATMOL) [88]. The primary design objectives were ease of use, concise notation, and the adaptation of common notational conventions. The high-level constructs in ATMOL are "declarative" and "side-effect free" required for the application of transformations to translate and optimize the intermediate stages of the model and its code. ATMOL is strongly typed and requires the typing of objects before they are used. This helps to pinpoint problems with the specification at an early stage before code synthesis takes place. ATMOL supports both high-level and low-level language constructs such as Fortran-like programming statements which are used to implement and optimize the target numerical code.

In figure 2.1 we show a small example of a specification in ATMOL. The variable *xfo* is first declared as a three dimensional grid of floating point values on a surface called *surface_o*. Then the variable is beeing assigned a value with the expression $ra - atmrad - ocnrad - sl$. The `where` construct serves as a kind of macro, CTADEL replaces the term `atmrad` with the expression `xb*asta`. The designer can anotate the specification by inserting comments, preceded by the percent sign.

2.2 A Brief Overview of the System

In this section we will briefly explain some of the components of the CTADEL system. For a more thorough explanation the reader is referred to [86]. Figure 2.2 shows a simplified diagram of the complete system. We discuss the specification language ATMOL, the `GPAS` reduction system, the `DICE` common subexpression eliminator and the code generator in more detail in the next subsections. The system consists of the following components,

```
xfo :: float field(x,y,z) on surface_o.
xfo = (ra - atmrad - ocnrad - sl
   % local radiation itensity
   where ra = rad((j-1)*dyo+(ny1-1)*dya+0.5*dyo)
   % atmospheric radiation
   where atmrad=xb*asto
   % sensible and latent flux
   where sl = lambda*(sst-asto)
   % ocean infrared
   where ocnrad = xc * sst
).
```

Figure 2.1: An example specification in ATMOLfor the variable `xfo`.

Scripts A collection of scripts is provided which contains libraries of PDE-based operators, skeletons of computer codes, predefined procedures for symbolic manipulation. Also the specification of the model can be specified as a script. Loading and compiling of scripts takes place via a terminal-based command interface.

Rule base A collection of rule bases containing various transformation rules and strategies for applying transformations. The rulebase transformation strategy can be interactively applied on expressions.

Parser The parser scans the input, analyses the syntax and parses scripts and user commands.

Symbolic evaluator Expressions are symbolically evaluated which results in the expansion of symbolic functions and procedures and the evaluation of symbolic expressions.

DICE The common subexpression eliminator. The input and output follow the format of the static single assignment (SSA) form [14, 47].

Synthesizer The construction of the (attributed) abstract syntax tree is handled by the synthesizer. The grammar productions and associated semantic rules are dynamically extended, because new user-defined functions and operators must be

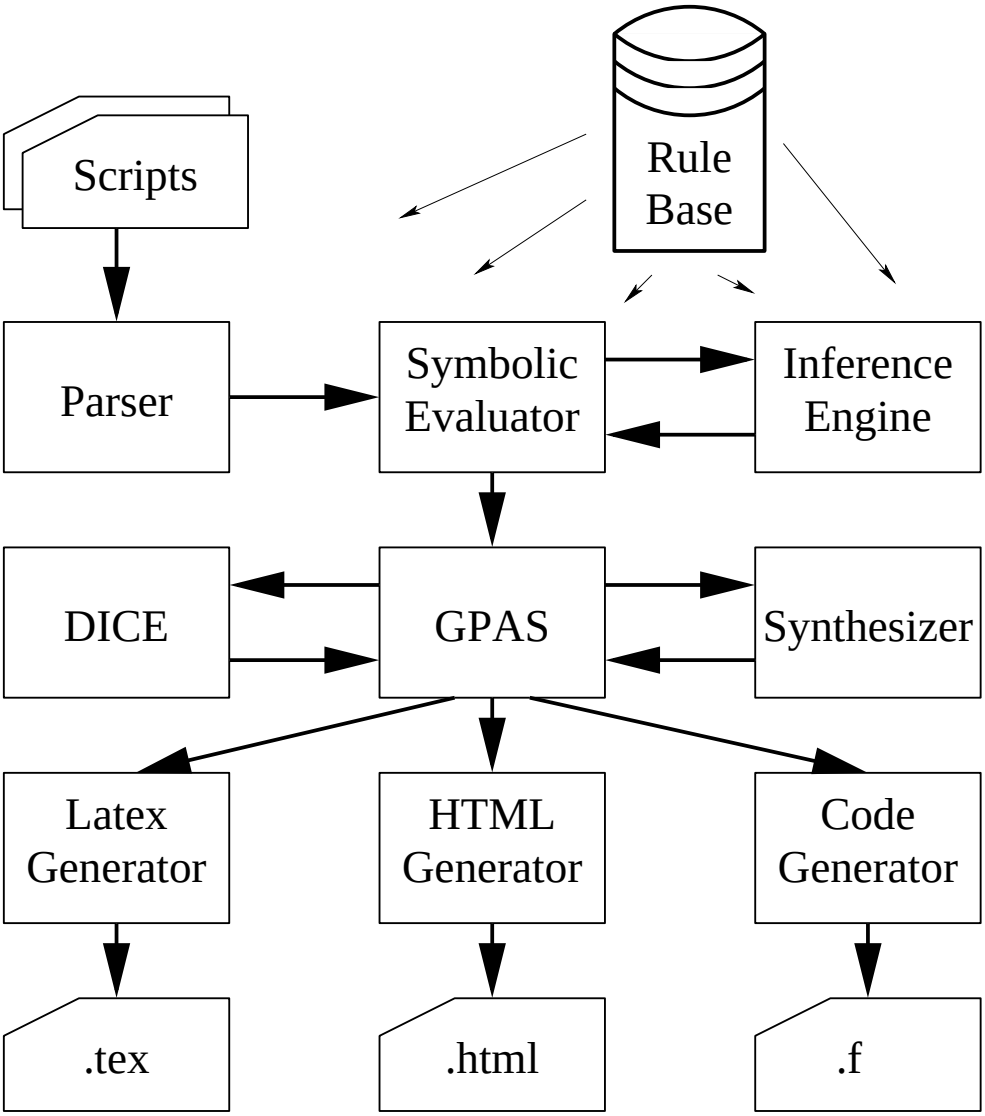


Figure 2.2: A simplified diagram of the CTADEL system.

integrated from their abstract specifications. Global value range and function monotonicity information are propagated through the abstract syntax tree (AST) which results, for example, in the symbolic derivation of array bounds of (multi-dimensional) variables used in the code.

Latex/HTML generator Generation of documentation from the specification is possible in both Latex formatting languages and HTML format. The HTML reports provide the feedback to a user of the system by automatically including inline cross-references from the high-level specifications to the low-level generated codes.

Code generator From the abstract syntax tree optimized Fortran 77 or HPF code is generated.

2.2.1 ATMOL: A Domain-Specific Language for Atmospheric Modeling

When the CTADEL was built, the choice was made to design a completely new programming language instead of using an existing specification language. This has resulted in ATMOL, see eg [88], a high-level language that provides a means for the specification of a PDE-based problem in a natural way. Its power of expressiveness is close to the declarative mathematical formulation of a model in vector notation. Just a few constraints were imposed on the design of the language namely, transparency, self-containment and extensibility.

All elements from the language both predefined in the system or user-defined should be transparent to the user. In this way, a user of the system can inspect the definitions and its specifications at every step of the compilation process and adapt them when necessary. Therefore, the language should be self-contained and extendable, that is, it should be possible to define new language constructs, like matrix/vector operations, in the language itself.

The CTADEL system depends heavily on computer algebra techniques for the symbolic manipulation of expressions. For the incorporation of common mathematical language constructs, such as vector and matrix operations and general arithmetic operations, an attempt was made to combine the ”best and most convenient” language features present in the Maple, Mathematica and Matlab packages.

Parsing of the ATMOL language is based on operator precedence grammar for the syntax of expressions, which is a commonly used technique in symbol computing [41]. The expression syntax based on operator precedence grammars can easily be changed or extended by (re)defining (new) prefix, infix and postfix operators.

2.2.2 The GPAS Reduction System

The ”General-Purpose Symbolic and Algebraic computer System”(GPAS) reduction system is one of the main components of the CTADEL system. Its main purpose is the symbolic manipulation of algebraic expressions. Examples of this are symbolic differential and integration, discretization, factorization, transformations on conditional expressions and the generation and optimization of program codes.

In many mathematical models for scientific problems the operators and functions exhibit properties such as associativity and commutativity. For the symbolic simplification of scientific models and the generation of code for the models, it is crucial that these properties are fully exploited. The associative and commutative laws are captured in modular forms. This enables the underlying term rewriting system of GPAS to be implemented using a strong pattern matching algorithm. The use of modular forms also alleviates some of the phase-ordering problems

2.2.3 Common Subexpression Elimination

An occurrence of an expression in a program is a common subexpression if it meets the following criteria: there is another occurrence of some expression and the operands of this expression remain unchanged between the evaluation [50]. Typically, the numerical schemes of a model based on partial differential equations can be optimized by removing these redundant computations. This operation requires the (partially) computed results of subexpressions to be stored in temporary variables until the results are no longer needed.

Common subexpression elimination (CSE) is generally applied on an intermediate, low-level, representation of the program by regular compilers. By raising this optimization to the higher-level of the numerical schemes, DICE can represent the temporary variables in a symbolic way, for example, as multi-dimensional array variables. It should be clear that using temporary variables this way can dramatically increase mem-

ory usage.

DICE is the Domain-shift Invariant Common-subexpression Eliminator of the CTADEL system. The DICE global optimizer exploits a heuristic cost model of a target computer architecture. In this way common subexpressions are eliminated only if they yield a reduction in hardware cost as a balance between computing time and memory resources. Since DICE is applied on declarative program-language constructs, it is not hindered by control flow which can prevent more optimal common subexpression elimination.

For matching common subexpressions, DICE exploits the associative and commutative properties of operators. In addition, to find array-based common subexpressions, the pattern matching of the indexed array variables in the expressions requires the derivation of linear index transformations. In the CTADEL system, common subexpression elimination is an essential part of the compilation process and a user can experiment with different forms of codes derived with different techniques. To this end, the CTADEL system allow the user to specify the strength of the CSE elimination by setting optional weights for several operators in order to enable or disable the elimination of found subexpressions.

DICE accepts sets of assignments in a semi Static Single Assignment (SSA) representation. Code is in SSA form if every variable being assigned a value occurs as the target of only one assignment and if it occupies its own memory-location [50]. Representing code in SSA-form is a commonly used technique in optimizing and restructuring compilers and allows for a more thorough dependence analysis [47]. Because a variable is assigned a value only once, there are no destructive assignments. Therefore, code in SSA-form contains no write-write or false dependencies. By definition of the SSA-form, aliasing of variables is not possible.

The intermediate form of an SSA-type code produces by DICE may have a large demand on the memory resources of a target machine. Each temporary variable introduced by the common subexpression elimination process is assigned only once, so no reusing of arrays can be applied. By applying advanced loop optimizations like loop fusion, we can apply array contraction. Array contraction enables the decrease of memory usage by stripping indexes of the temporary arrays [25]. We have not built this into Ctadel, because advanced compilers can do this from the generated source code.

2.2.4 Multi-Platform Code Generation

One of the objectives for the CTADEL system was to allow the model designer to write architecture independent model specifications. It is the responsibility of CTADEL to take care of multi-platform code generation and architecture dependent optimizations. Aside from a set of simplified hardware-parameters for the DICE module, such as variable load and store costs, CTADEL generates code through the application of low-level architecture-specific code restructuring transformations. Furthermore, CTADEL has adopted data distribution and domain-splitting methods for distributed parallel computation. Therefore, code generation is possible for serial-, vector-, and distributed/shared memory architectures. The target code produced by the CTADEL backend module is a Fortran dialect, e.g. Fortran 77 for serial architectures or High Performance Fortran for shared memory architecture. Extending CTADEL to produce other high-level programming languages, like C or Java, is straightforward.

Domain Splitting Methods

Domain splitting is a common technique in parallel code: a global domain is decomposed into local subdomains. A processor is assigned a single subdomain on which it calculates a given function. If data is needed from another subdomain, this can be exchanged explicitly, for example by MPI-calls, or implicitly, for example by shared memory. In case no data is needed from other subdomains, for example embarrassingly parallel problems, it is challenging to assign subdomains to processors by means of a load balancing algorithm [13].

A common way to exchange data between neighboring subdomains is the halo method, which entails the creation of a ring around a subdomain overlapping adjacent subdomains. Prior to the generation of code for a specific computer architecture, CTADEL symbolically derives the bounds for the computation of each (temporary) array variable of the intermediate code. CTADEL also determines the size of the halo. Furthermore, the global stencil information from the numerical schemes for the fundamental (input) variables gets calculated. It was an implementation decision to calculate the stencil for the input and not the output variables of the model. More information about the halo method and CTADEL can be found in chapter 7.

Low-level Code Restructuring

In CTADEL, the associative algebraic properties of sequential statement composition and the associative and commutative property of parallel statement composition are used in pattern matching code constructs by the restructuring transformations. These transformations can be (interactively) applied on the code. Several transformations, like loop interchange, loop unrolling and data-parallel conversion of code can be implemented.

2.3 Related work

Little work has been done in the field of automatic generation of program codes that are based on a mathematical specifications. A number of related projects on automatic code generation or automatic parallelization exists and we mention a few below. Here, we give a brief overview of related work in scientific problem solving. We divided this overview into three sub-categories: numerical libraries, symbolic compilers and restructuring compilers.

2.3.1 Numerical Libraries

Usually programmers tend to "convert" a problem by hand into a program. Several numerical libraries are available to help the programmer by offering a substantial amount of standardized numerical routines. For example, the Basic Linear Algebra Subprograms (BLAS) is available and optimized for a large set of computer architectures [18]. The LAPACK[59] and the parallel PLAPACK[56] provide the user with a collection of direct linear solvers. Several other variants of LAPACK exist, for example, SCALAPACK, which is optimized for shared memory parallel architectures, and JLAPACK, a version for java virtual machines.

However, problem specific information is not used by these libraries, although e.g. the Broadway compiler [26] uses annotations from these libraries for optimizing programs. This approach can yield efficient applications for multiple types of architectures, but programmers tend to be reluctant in the (re)use of "yet another library" and program the numerical problems by hand by tweaking standard examples from "numerical cookbooks" like [61].

2.3.2 Problem Solving Environments

Problem solving environments can provide a programming paradigm for application development by means of software composition, which is the automatic translation of a problem, defined at a high level of abstraction, into an executable code. Software modeling and development at a high abstract level allow the application developers to react more quickly on new developments. Furthermore, studies have shown that programmer productivity, measured by lines of code over time, varies little between languages. Languages that automate more of the low-level work allow a programmer to accomplish more in fewer lines of code. Well known examples of problem solving environments include Matlab and Maple.

Special packages for these PSEs exist which produce program codes. For example, the MathWorks package is a compiler for Matlab specifications which generates `C` and `C++` codes. However, these compilers usually do not exploit target architecture specific optimizations.

2.3.3 Restructuring, Parallelizing and Symbolic Compilers

The use of parallelizing compilers for the restructuring of scientific codes has been an ongoing research topic for many years. The main problem is how to detect the course-grain parallelism from a sequential program in order to effectively use the resources of a parallel architecture without the need for writing a separate parallel version of the program.

Restructuring compilers take a program written in a high-level programming language as input. First, the program is parsed and an abstract syntax tree (AST) is generated, on which standard optimizations can be applied. The restructuring compiler can perform optimizations on the AST by applying a set of transformations. Then, the compiler generates the restructured program in a high-level language, either the same or different as the input language. The set of transformations, which can be provided by the system itself or by the programmer, is usually applied using a pattern-matching mechanism. These compilers use dependence analysis to prove that the semantics of a program is not changed by a transformation.

An example is the Falcon project [16]. It uses an existing high-level array language, Matlab, as source language and performs static, dynamic, and interactive analysis to generate Fortran 90 programs with directives for parallelism. It includes capabilities for

interactive and automatic transformations.

An example of a rewriting compiler is the MT1 compiler [8]. This compiler is a source to source compiler that enable the automatic conversion of programs that operate on so-called dense matrices into equivalent program operating on so-called sparse matrices. The latter are matrices in which the presence of many zeros can be exploited to reduce storage requirements and computational time. Clearly, more powerful transformations than the traditional program transformations are required, because the data structures must be adapted in order to exploit the characteristics of data. Also more general transformations can be specified, for example to perform advanced loop optimizations. This makes the MT1 compiler a perfect intermediate between the generated code from CTADEL and a standard Fortran compiler.

Tolmach and Oliva [76] also take a subset of an existing functional language, ML, as source language and produce C or ADA programming codes. They make use of a so-called type-based macro-extension technique, which they call templates and which resembles the concept of CTADEL-templates. Their work, however, is not targeted toward code generation for mathematical models, but as an aid to speed up application develoment in traditional languages.

Chapter 3

Experimental Setup

In this chapter we will discuss the hardware and software setup we used in the various experiments we performed. Since this dissertation presents work done over a period of four year, several hardware setups were used. When we compare the performance of the code generated by CTADEL with reference code we have run these codes under the same circumstances and with the same compiler settings.

In the following sections, we will give a brief description of every platform and the presented label for this platform will be used in the remainder of this dissertation to identify this setup.

3.1 Scalar Architecture

In this section we list the different scalar processor architectures.

PENT-II A commodity pc with a Pentium II processor, running the GNU operating system, with a Linux kernel version 2.0.36. The used machine was equipped with a 333 MHz Pentium II processor and 64 MB ram.

The compiler used for the platform is the GNU Fortran compiler, version 2.95.2 with the standard optimization turned on (`g77 -O2`).

ATHLON A commodity personal computer, running the Linux/GNU operating system, with a Linux kernel version 2.4.10. The used machine was equipped with a 700 MHz AMD Athlon processor and 384 MB RAM.

On this platform the GNU Fortran compiler, version 3.0.3, was used. We used the standard optimization flags.

PENT-IV An ordinary desktop pc with a Pentium IV processor, running the Linux/GNU operating systems, with Linux kernel 2.4.18. The used machine was equipped with a single 1800 MHz processor and 256 MB RAM.

For the compilation of the programs we used both the GNU and Intel Fortran compiler. We used the Intel Fortran Compiler for Linux, IFC version 7.0, with optimizations for the Pentium IV (-O3 -tpp7 -axW) turned on. For the GNU G77 compiler, version 0.5.26/2.96-113, we used the default optimization flags (-O3 -s -march=i686).

3.2 Shared Memory Parallel Architecture

In this section we list the only shared memory parallel architecture we used, a sun enterprise server.

SUNE450 A Sun E450 enterprise server with four 400 MHz UltraSPARC-II processors, running Solaris 7. The used machine was equipped with 4 GB RAM.

3.3 Distributed Memory Parallel Architecture

In this section we list the different distributed memory parallel architecture.

DAS1 The DAS computer[33]. This is a wide-area distributed computer with 200 processing nodes, spread out over four clusters. All these cluster contain nodes which consists of a Pentium Pro processor, running a Linux 2.2.14-5.0 kernel. The nodes within a cluster are interconnected with fast ethernet for operating system related traffic like NSF and a 1.2 Gb/s Myrinet network [51] for low-latency, high-bandwith user-level data communication. The four clusters are interconnected over an ATM network. However, we only run jobs on a single cluster.

The reader is referred to [33] for more information about the DAS system. The DAS system is also referred to as a “Beowulf” cluster[62].

The compiler used for the platform is the GNU Fortran compiler, version 2.95.2 with the standard optimization turned on (`g77 -O2`). For data communication between the different nodes we made use of the MPICH implementations [55] of the MPI library [23].

DAS2 The DAS-2 computer[31] is the successor of `DAS1`. It is a wide-area distributed computer with 200 processing units, spread out over five clusters. Each unit contains a dual Pentium-III processor and is equipped with at least 512 MB RAM. The units are running a Linux kernel version 2.4.7-10smp. For our experiments we used a limited number of units. For each unit we used both processors, but denote each single processor as a node in the remainder of this dissertation.

The nodes within a local cluster are connected by a Myrinet-2000 network [51], which is a high-speed interconnection network. In addition, Fast Ethernet is used as OS network, for example for file distribution.

Myrinet-2000 is a switched network, capable of full-duplex data rates up to 2+2 Gb/s and has low latencies in the range of a few micro-seconds. The Myrinet is connected with a high-speed level-3 switch. For communication we make use of the MPICH implementation of the MPI message system [55].

For communication between clusters the Globus toolkit can be used. However, we run all our experiments on only one cluster.

On this system we used the GNU compiler, version 3.0.3, with default optimization flags for both the original and generated codes. For communication between the nodes on this computer the MPI library was used.

Chapter 4

Extending the Application Domain: The Kain–Fritsch Convection Scheme

The CTADEL system was developed with the dynamics of a weather forecasting system in mind. In this chapter we show how CTADEL was extended so that the convection scheme could be incorporated.

The convection scheme we use in this chapter is the Kain–Fritsch (KF) convection scheme [40]. A number of trigger functions are calculated to determine if and how much entrainment and detrainment take place in a cloud. Because of these conditions, it is difficult to generate an efficient implementation for vector architectures. For this scheme we developed a new feature of CTADEL: template based code generation. Also do these trigger functions make it harder to generate efficient code for e.g. vector architectures, something that is also not easily solved by CTADEL.

4.1 Background

In this section we go into further detail about the theory and specification of the KF convection scheme and the use of templates in CTADEL. For reference we used the hand-written implementation of the convection scheme in the HIRLAM system, a numerical weather prediction system in operational use at several European meteorological institutes [20, 32].

4.1.1 The KF convection scheme

This scheme is an one-dimensional entraining/detraining plume model in which only a small number of clouds per vertical column (grid box) is resolved. The reason for this is that the mesoscale models for which it was designed, have rather small grid boxes which contain only a small number of different clouds.

In these grid boxes the available buoyant energy (ABE) is consumed by fluctuations in mass in the box. An increase in mass flux can be seen as an increase in the number of clouds present in the grid box, while a decrease in mass flux is indicative of a reduction of the number of clouds. This consummation of mass flux is assumed to take place in time τ_c, where τ_c is the advection time at the Lifting Condensation Level (LCL) which lies between 1800 and 3600 seconds. Because of closing of the continuous functions the parameterized mass flux of the single cloud is usually not enough to remove all ABE in time τ_c; to achieve the full consumption of the ABE the mass flux is adjusted to a level where the remaining ABE is less than 10% of the initial value.

Entrainment and detrainment of mass flux in the cloud takes places because of two processes: up-, and downdraft in the cloud and through environmental air.

One of the important parameters for the updraft calculations is the rate at which environmental air is entrained into the cloud. This rate is assumed to be

$$\delta M_e = M_{u0}(0.03\delta p/R), \tag{4.1}$$

where δM_e is the environmental entrainment rate, δp the depth of the layer in Pa, R and M_{u0} respectively the updraft radius and mass flux at the cloud base. Equation (4.1) prescribes that when there is no detrainment, the updraft mass flux doubles if it travels 500 hPa upwards. The updraft mass, with which this environmental air mixes, must become available at the same rate, which leads to

$$\delta M_t = \delta M_e + \delta M_u. \tag{4.2}$$

Here δM_t is the total rate of mass entrainment into the mixing region and δM_u is the rate of mass entrainment from the updraft into the mixing region. As mentioned previously, the amount of mass that detrains out of the cloud will be dependent on the mixtures of environmental and updraft air. For these sub-parcel mixtures a Gaussian distribution function f is assumed [40] which reads

$$f(x) = A_g \left(e^{-(x-m)^2/2\sigma^2} - k \right), \tag{4.3}$$

where x is the fraction of environmental air in the mixtures, m is the distribution mean (in this case 0.5), σ is the standard deviation (1/6) and k an constant so that $f(0)$ = $f(1)$ = 0, so k = $e^{-4.5}$. A_g is defined such that

$$\int_0^1 f(x)dx = 1, \tag{4.4}$$

which means that A_g = $(0.97\sigma\sqrt{2\pi})^{-1}$. The distribution in equation (4.3) gives a specification of the rates at which various mixtures are generated. Assuming that the subparcel size is independent of the mixing proportion, the total mass distribution can be obtained simply by multiplying the frequency distribution δM_t, so

$$\delta M_e + \delta M_u = \delta M_t \int_0^1 f(x)dx. \tag{4.5}$$

The individual components of this distribution are given by

$$\begin{aligned} \delta M_e &= \delta M_t \textstyle\int_0^1 x f(x)dx, \\ \delta M_u &= \delta M_t \textstyle\int_0^1 (1-x) f(x)dx. \end{aligned} \tag{4.6}$$

From these equations the total entrainment into and detrainment out of the updraft can be calculated. As the negatively buoyant parcels detrain from the updraft, the updraft detrainment rate (M_{ud}) is determined from

$$M_{ud} = \delta M_t \int_{x_c}^1 x f(x)dx. \tag{4.7}$$

Similarly, the environmental entrainment rate M_{ee} is calculated by

$$M_{ee} = \delta M_t \int_0^{x_c} (1-x) f(x)dx, \tag{4.8}$$

where x_c is the fractional amount of environmental mass that just yields a neutrally buoyant mixture. A very small x_c (very dry air) will therefore cause a high detrainment rate while very moist environmental air will enhance the updraft through a large M_{ee}.

A number of functions are used to trigger convection. In the current implementation of the model for the HIRLAM numerical weather prediction (NWP) model, three functions are used. Those functions are based on:

1. The temperature perturbations associated with the vertical wind speed perturbations.

2. The relative humidity.

3. The boundary layer turbulence.

These trigger functions have a certain common feature, namely conditional calculations. Dependent on the outcome of these conditions a number of other functions are calculated. Programming this by hand should be an easy job because the programmer knows certain details about the conditions. This knowledge allows him to make assumptions, which allow full optimization of the call-graph. Automatic code generation tends to be more conservative, and therefore this leads to less efficient programs.

For example, one trigger depends on the relative humidity R_h and can be thought of as variance in the relative distribution ΔT_{rh} which reads

$$\Delta T_{rh} = \begin{cases} 0.25(R_h(LCL) - 0.75)Q_{mix}/(\partial Q_s/\partial T) \\ \qquad \text{if } 0.75 \leq R_h(LCL) \leq 0.95 \\ (1/R_h(LCL) - 1)Q_{mix}/(\partial Q_s/\partial T) \\ \qquad \text{if } R_h(LCL) > 0.95 \\ 0 \quad \text{otherwise,} \end{cases} \tag{4.9}$$

where Q_s and Q_{mix} represent the environmental and the saturation mixing ratios, T the temperature and $R_h(LCL)$ the relative humidity at the lifting condensation level. Programmed by hand, this function is a straight forward piece of code and the original hand-written Fortran code looks like

```
IF(RHLCL.GE.0.75.AND.RHLCL.LE.0.95)THEN
   DTRH = 0.25*(RHLCL-0.75)*QMIX/DQSDT
ELSEIF(RHLCL.GT.0.95)THEN
   DTRH = (1./RHLCL-1.)*QMIX/DQSDT
ELSE
   DTRH = 0.
ENDIF
```

Using a template the resulting code generated by CTADEL looks like

```
IF(0.LE.RHLCL(i)-7.5E-1)THEN
  IF(9.5E-1-RHLCL(i).LT.0)THEN
    DTRH(i)=QMIX(i)*(1.0/RHLCL(i)-1)/DQSDT(i)
```

```
  ELSE
    DTRH(i)=QMIX(i)*(RHLCL(i)*2.5E-1-1.875E-1)
            /DQSDT(i)
  ENDIF
ELSE
   DTRH(i)=0.0
ENDIF
```

If $R_h(LCL)$ is greater than 0.95 the generated template based code also only needs two comparisons while the hand-written code needs three comparisons. The template we used for this case looks like

```
if ( Cond >= Val1 ) then {
   if ( Cond <= Val2 ) then {
           rangedep := Expr2
   } else  {
           rangedep := Expr3
   }
} else {
   rangedep := Expr1;
}.
```

This template compares a certain condition `Cond` for values `Val1` and `Val2` and calculates expressions `Expr1`, `Expr2` or `Expr3` based on the outcome of these comparisons. The assumption is made by the programmer that `Val1` is less than `Val2`, so CTADEL does not have to cover all edges of the call graph.

4.2 Templates in Ctadel

The use of templates is a well known concept to maintain a certain level of abstraction while keeping full control of essential details, e.g. [22, 74]. Templates are used in several high-level programming languages like `C++` [3], however these languages restrict the use of templates to the front-end of the compiler. By moving the usage of templates to the back-end of the compiler, we can implement (near) optimal numerical solutions methods for several types of computer architectures. It also allows the use of library

calls to existing optimized libraries. Ideally, one wants to hide the use and implementation of templates as much as possible for the user of the system. An additional constraint for templates in CTADEL is to keep the implemented templates as generic as possible to prevent a large database with application specific code. However, this is not always possible for efficiency reasons and a trade-off has to be made between abstraction and implementation.

The idea about templates in CTADEL is based on two principles, abstraction and implementation.

4.2.1 Abstraction

The use of templates can hide many low level implementation details. An optimal numerical method for a specific computer architecture might be inefficient for another one. Since a template is only filled in at a late stage of code generation, we can hide these implementation details to the user. By using a template we can do a generalization over similar cases; they are general and reusable. Furthermore, hiding these details in the earlier stages of code generation can improve correctness of the specification and speedup the code generation process. For example if a special numerical solver is needed, it can be programmed from scratch or it can be based on a template without knowledge of the actual implementation details. CTADEL type-checks the arguments and usage of the template, which can be polymorphic, and, if correct, take care of the actual implementation for the target architecture.

A template looks like a polymorphic declaritive function which returns a $n-$dimensional variable (with $n > 1$). It takes a number of parameters, performs a list of operations and returns a value. For every variable one can denote a type, like float, or an abstract type which is dynamically filled in when instantiated by CTADEL. Every variable has a domain, the range for which it is declared, e.g. for a 50 by 50 points array the domain can be *i=1..50 by j=1..50*. One can also give a domain as a parameter to the template, for example to range on which a function can be applied.

A simple example of a polymorphic template is the `reduce` template. It is used in the implementation of the calculation of an integral. The user specifies the integral operator and CTADEL translates this internally into the reduce template, with the appropriate summation operation $_F$. The template looks like

```
reduce(a :: _T, _D :: domain(index), _F
```

```
 :: associative(_T -> _T -> _T)) :: _T function
{  reduce := unit_element(_F);
   for _D do reduce := apply_op(_F, reduce, a)
}.
```

This template performs a (reduce) operation `_F` on all elements of variable a with type `_T` (i.e. a grid of reals) in domain `_D`. The actual implementation of operator `_F` can be another template or (simple) function, predefined by CTADEL or the user.

4.2.2 Implementation

Templates are not target language specific, there is no link between the template and for example Fortran. Rather, they are specified in an abstract representation, which is readily translatable into the target language. In the final stage CTADEL has to generate programming codes for the specified target architecture and target language. By using language-independent templates, only the final's stage depends on the target language. Generating code for different target languages, like `Fortran 77` or high performance Fortran (`HPF`), is therefore postponed until the back-end of CTADEL. This ensures a high degree of portability and maintainability of the implemented methods.

Because a specific function is not implemented or an existing template is too general, the user can choose to implement additional templates. These user templates are parsed and type-checked by CTADEL as normal templates, but the responsibility for correctness lays with the user.

In figure 4.1 we show a schematic overview of the implementation of templates in CTADEL. The user specifies the model in a script, where (s)he can make use of predefined or hand-written templates. These templates are parsed and the actual arguments of their usage in the script are type checked by the front-end of the system. During code generation every statement is evaluated using a `template_eval` call, just before generating the target language. The `template_eval` function type instantiates the correct template, when needed.

4.2.3 Toy Example of a Template

In this section we provide a small toy example of a template called `p_loc` to explain some of the details of templates in CTADEL. The template, which takes as input a

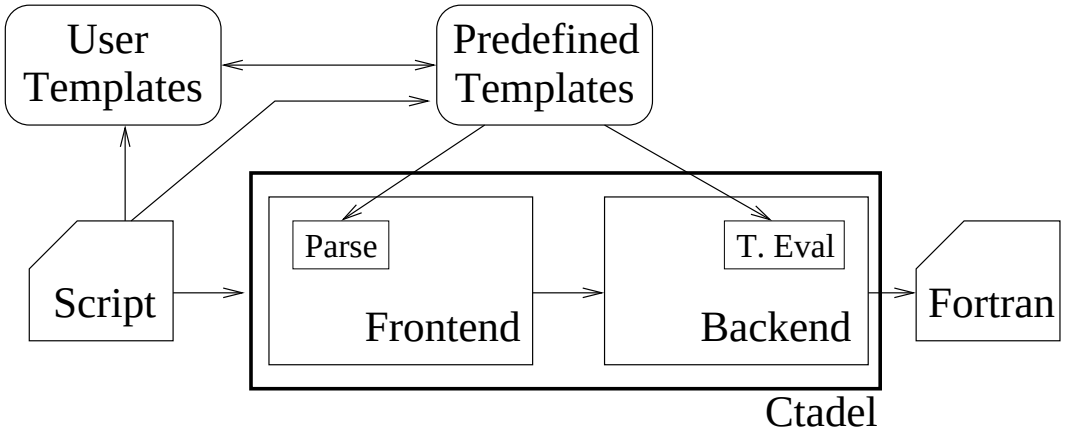


Figure 4.1: A schematic overview of template usage in CTADEL.

variable E, a domain declaration D and a value F, calculates a sum and tries to find the first index n such that

$$\sum_{i}^{n} E_i > F. \tag{4.10}$$

The code of this template reads

```
p_loc(E :: float, D :: domain(index),
      F :: float) :: integer function
{
      tmp :: float;
      tmp := 0;
      p_loc := 1;
      for D do {
          tmp := tmp + E;
          if (tmp > F) then {
              p_loc := index(D);
              jumpout
          }
      };
}.
```

If we use this template in the following specification

```
A :: float field(x,y) on
          i=1..nn by j=1..mm.
B :: integer field(x) on i=1..nn.
B = p_loc(A, j=2..mm,2.0).
```

CTADEL produces the following piece of Fortran code for a scalar architecture

```
   DO 100 i=1,nn
      tv_1=0.0
      ploc0=1
      DO 101 j=2,mm
         tv_1=tv_1+A(i,j)
         IF(tv_1.GT.2.0)THEN
           ploc0=j
           GOTO 1000
         ENDIF
101    CONTINUE
1000   B(i)=ploc0
100  CONTINUE
```

Naive automatic code generation without the use of templates would first calculate every possible sum of the array A and then search for a possible solution. This is because CTADEL only knows about the data-dependences between A and B. For example the next piece of Fortran code was generated by an older version of CTADEL without the use of templates

```
      DO 1000 i = 1,nn
         t(i,mm) = 0
1000  CONTINUE
      DO 1010 j = mm,2,-1
         DO 1020 i = 1,nn
            t(i,j-1)=A(i,j)+t(i,j)
1020     CONTINUE
1010  CONTINUE
      DO 1030 i = 1,nn
         DO 1040 j = 1,mm
```

```
            IF (2.0.LT.t(i,j)) THEN
               B(i) = j
               GO TO 1050
            ENDIF
1040     CONTINUE
         B(i) = 1
1050     CONTINUE
1030  CONTINUE
```

This last sample Fortran code is less efficient with respect to execution time compared to the code generated with the use of templates. This is especially true when a solution can be found with a low index. Because of the polymorphic character of the template, `E` does not have to be a simple array, but can be an expression of which the evaluation is computational expensive. Furthermore, it is obvious that the last code is inefficient with respect to memory usage.

4.3 Experiments and Results

In this section the code generated by CTADEL for the Kain–Fritsch convection scheme is compared with the original hand-written code. We ran the code on two types of computer architectures, the scalar architecture `[ATHLON]` and the distributed architecture `[DAS2]`. For an explanation of the hardware setup, see chapter 3.

We ran both codes with a variable number of horizontal grid points and a constant number of 31 vertical grid points, a typical number for the Hirlam numerical weather prediction (NWP) model. For the input data we took data from a Hirlam run and mapped this to the input grid. Since the horizontal resolution of Hirlam is in the order of 100 grid point, for an input grid of 1000 points we have to copy the Hirlam input ten times. Also we ran the codes on a variable number of nodes on the distributed architecture.

Like we mentioned before, the original code stores its temporary results in scalar variables. Data-dependencies make it possible for the hand-written code to do a substantial amount of calculations in one big loop. CTADEL, on the other hand, keeps all the temporary computed results in arrays and makes use of a lot of small loops. This gives some extra overhead during execution. Optimizations, like loop fusion [4, 11], should make it possible for CTADEL to perform scalar conversion. These optimizations

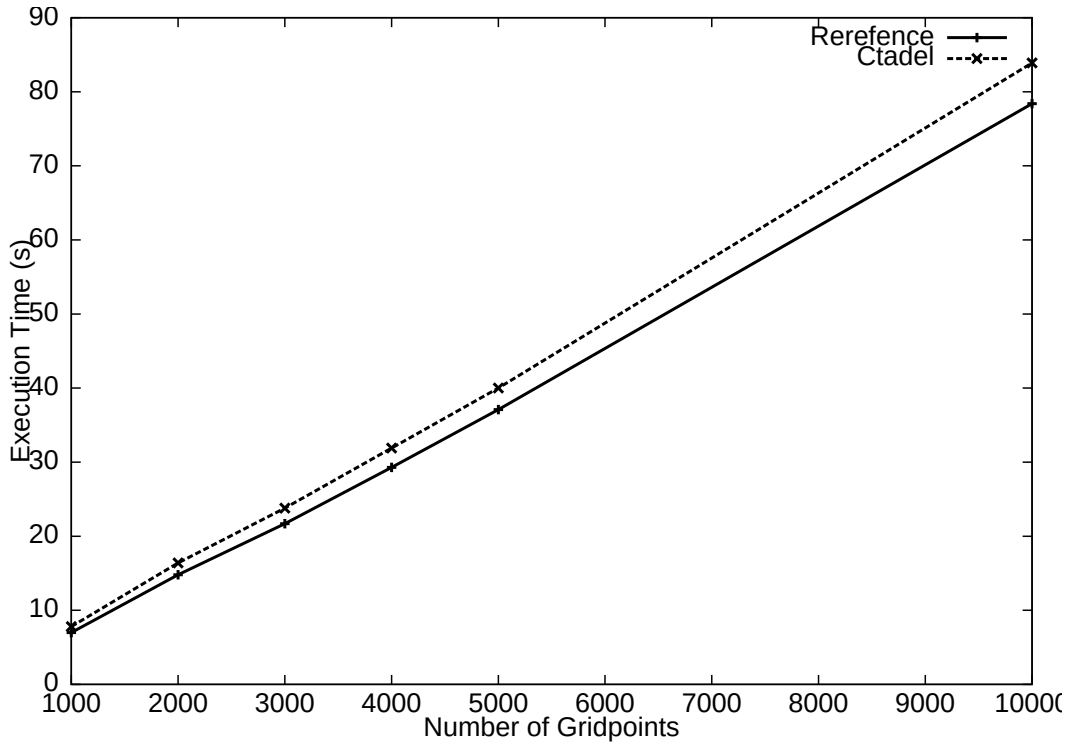


Figure 4.2: Execution times (s) as a function of the input grid size for the generated code and the reference code for the KF scheme.

are not applied yet in CTADEL since they require a powerful data flow dependency and control flow dependency analysis when dealing with conditional statements. A source to source compiler, like MT1 [8], could be used to perform these optimizations.

In figure 4.2 we ran both generated code and hand-written code on a scalar architecture with a variable number of grid points. Since the model is one dimensional and the computations are thus performed 'column-wise', in the vertical direction only, we could expect a linear function of the execution times with respect to the grid size. This is indeed almost the case, for example with a size of 1000 grid points the execution time of the generated code by CTADEL is 7.81 seconds while the execution time is 83.9 seconds with a grid size of 10000. From figure 4.2 we can conclude that the generated code from CTADEL can compete with the hand-written code on a scalar architecture. The difference in execution times is around 10% between both codes.

For the distributed memory architecture we made the generated code parallel by

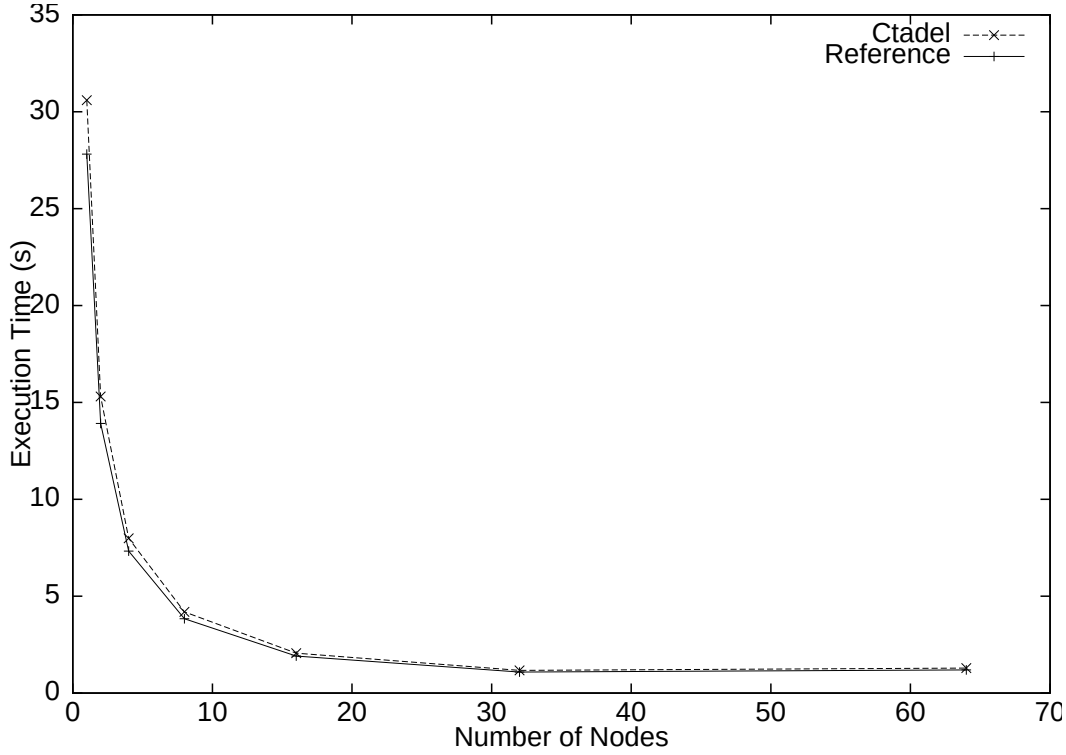


Figure 4.3: Execution times (s) as a function of the number of nodes without MPI communication with a grid size of 25000 by 31 points for the generated and the reference code for the KF scheme.

domain splitting, with the MPI library for communication between the nodes. Since the convection scheme is a one dimensional scheme, all calculations are done on a vertical column and no data is needed from adjacent columns, we only need communication at the start and end of the calculations. This makes the problem embarrassingly parallel, and we could therefore expect an almost linear speedup[1] when running on multiple nodes.

In figure 4.3 we show the execution times for the execution of both the generated code and the hand-written code with a constant grid size of 25000 by 31 and a varying number of nodes. The performance from the generated code from CTADEL can compete with the hand-written code, as we can see from this figure.

[1]For speedup we use the definition from Quinn [65] which uses the execution time of a parallel algorithm on a number of processing nodes compared to the execution time of the fastest sequential algorithm.

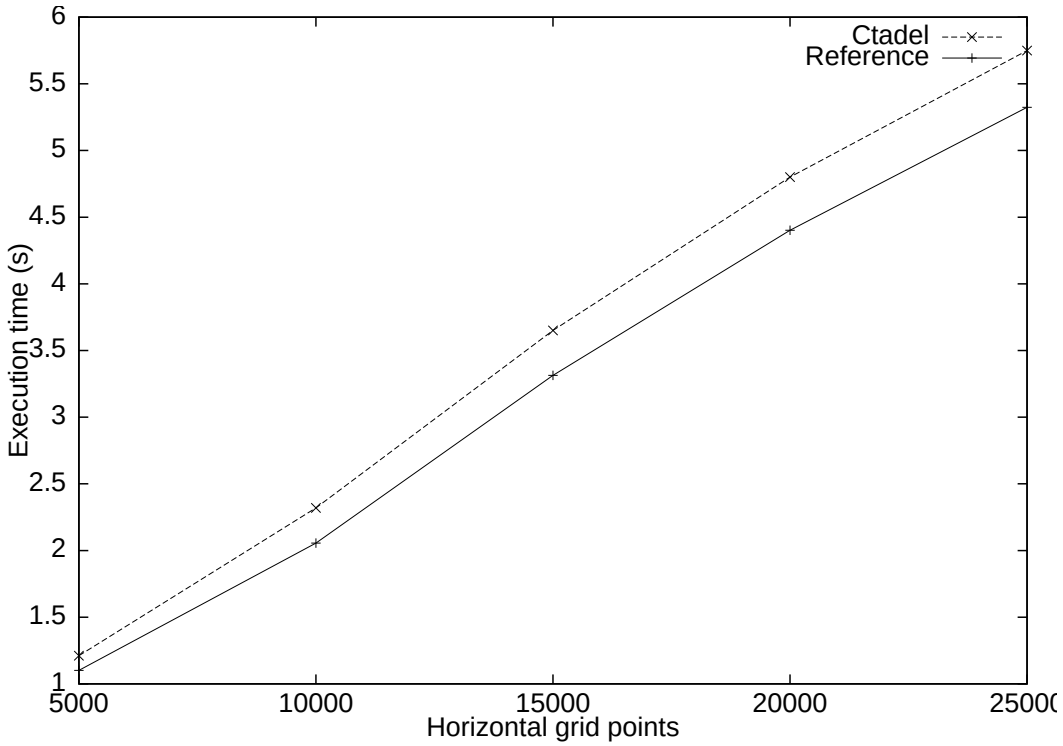


Figure 4.4: Execution times (s) as a function of the grid size on 32 nodes for the generated and the reference code for the KF scheme.

To find out the scalability of the algorithms as a function of the grid size, we executed both codes with a constant number of 32 nodes and a variable grid size, for which we show the execution times in figure 4.4. As expected from previous figures, this is a near linear function.

4.3.1 Conclusion

From the experiments we can conclude two things. First, we see that automatically generated code can compete with hand-written code. Although the generated code has a slightly higher execution time in all cases, no additional loop optimizations were performed. Other experiments, for example in section 6.1 and [87], have shown that this can have some positive performance impact on the generated code. The second thing we can observe from the figures is that both codes achieve a near linear speedup when we do not take communication times into consideration. Since the convection model is one dimensional, this is as expected.

Chapter 5

Extending the Application Domain: A Coupled Atmosphere–Ocean Model

In this chapter we show the possibilities of CTADEL on a coupled ocean–atmosphere model, a quasi-geostrophic climate dynamics model [17]. Coupled models impose certain difficulties on their implementation; often these models use multiple resolutions on the computational grids and separate time steps. This also assesses certain constraints on the interaction between the different parts of the model and the parallelization of the model. By default CTADEL assumes one resolution is used for the whole model, which is not feasable for this coupled model, which contains two submodels. We have extended CTADEL in such a way that resolution is coupled to a variable. For data exchange between the two submodels, we made use of the library calls to external interpolation spline routines.

This chapter is organized as follows, in section 5.1 we discuss some of the theory of the Quasi-Geostrophic model. In sections 5.2, 5.3 and 5.4 we show the specification of the model and the implementation in CTADEL, while we show some experiments we performed with the generated code in section 5.5.

5.1 Quasi-Geostrophic Dynamics

For the sake of understandability and readability we will explain the quasi-geostrophic dynamics model only in some detail. For a more detailed account, the interested reader is referred to [17]. The model describes a mid-latitude coupled climate model, which is

used in an attempt to understand how the ocean climatology is modified by atmospheric coupling. At present, the best comprehensive coupled climate models run at resolutions far coarser than those needed to model the inertial recirculation. The model presented in this chapter is an attempt to explicitly include eddies in general, and inertial recirculation in particular, within the framework of an idealized climate setting. The basic model consists of a quasi-geostrophic channel atmosphere coupled to a simple, rectangular quasi-geostrophic ocean. Heat and momentum exchanges between the ocean and the atmosphere are mediated via mixed layer models and the system is driven by steady, latitudinally dependent incident solar radiation.

Solar radiation is the basic force behind the global climate. About 30% is reflected back to space while the remaining part is absorbed mostly at bottom interface, whether it be land or water. Heat transfers to the atmosphere occur either through sensible or latent fluxes, or long wave radiative surface emissions to which the atmosphere is almost totally opaque.

The model is based on a classical β-plane mid-latitude representation and a two layered version of the quasi-geostrophic (QG) equations. For the most part, classical QG models are adiabatic, i.e. thermodynamics are neglected [30]. If the standard QG scaling is used and the usual Rossby number expansion is employed, the momentum equations yield the non-dimensional vorticity equation

$$(\frac{\partial}{\partial t} + J(p_i))(\nabla^2 p_i + \beta y) = w_{i1z} + HF, \tag{5.1}$$

where J denotes the Jacobian operator, p_i is the layer pressure, y is the meridional coordinate. The quantity w_{ilz} denotes the layer vertical velocity, HF denotes the horizontal frictional effects and

$$\beta = \frac{\partial f}{\partial y} \tag{5.2}$$

with the Coriolis force $f = 2\rho \sin\psi$, with ρ the rotation of the earth and ψ the latitude.

The final dimensional forms of the QG equations for both ocean and atmosphere layers can now be derived, for example the first layers reads,

$$\begin{array}{c} \frac{d}{dt} q_1 = A_{1h}\nabla^6 p_1 + \frac{f_0}{H_1}(w_{ek} - E(-H_1 - h_{1+})) \\ q_1 = \frac{\nabla^2 p_1}{f_0} + \beta y - \frac{f_0}{g' H_1}(p_1 - p_2), \end{array} \tag{5.3}$$

where P_x represents the layer pressure on layer x, f_0 denotes the effects of the upper and lower layer horizontal frictional processes, A is a constant and H_x the distance between

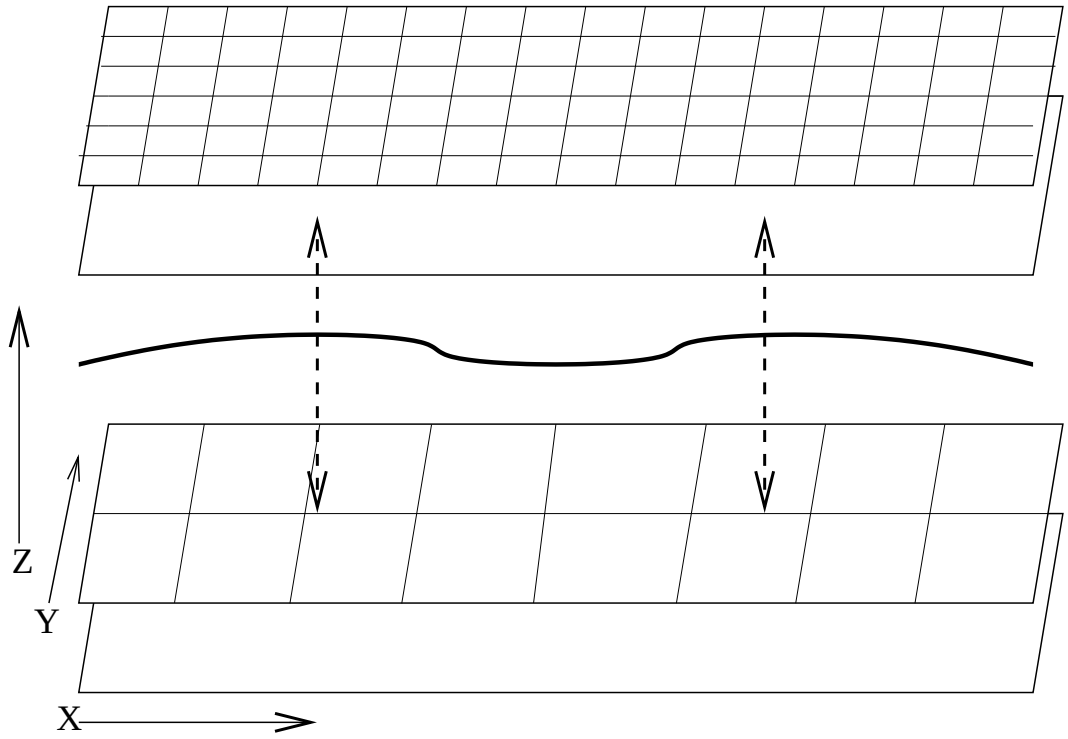


Figure 5.1: A graphical representation of the data grid for the quasi-geostrophic mode. Both ocean (below) and atmosphere model have their own grid size. Data exchange between the models takes place just before execution of an ocean step using spline interpolation.

layers. Here $\nabla^6 p_1$ is used as closure for the latter and super slip boundary conditions (i.e. $\nabla^2 p_i = \nabla^4 p_i = 0$).

5.2 Specification and Implementation

In this section we describe the specification of the quasi-geostrophic model using the ATMOL specification language for CTADEL and one of the main problems with this specification, the separate domain resolutions.

5.2.1 Spatial resolutions

Both the ocean and atmosphere model have their own spatial domain resolution, which leads to several problems when combining both models into a mixed model. For example, when exchanging data between the two models, a spatial interpolation has to take place. Another problem arises with differentiation of a function, since CTADEL cannot assume a default grid size, but has to determine this from the context. For example, one would normally write a differentiation as,

```
coordinates := [dx,dy].
Q = diff(P,y).
```

which would be translated into

```
   DO 10 j = 1,m-1
      DO 20 i = 1,n
         Q(i,j) = (P(i,j-1) - P(i,j)) / dy
20    CONTINUE
10 CONTINUE
```

This example assumes that distances between grid points are the same for every variable in the model. Since we work with two different grid sizes, this assumption is false. One possibility is to hard-code the distance between two grid-points into the specification, which is not desirable. Therefor we couple the specification of a domain to its spatial grid sizes. For example, the definition of a domain could look like,

```
domain_a := i=1..X_a by j=1..Y_a by k=1..Z_a.
domain_a‘coordinates := [dxa,dya].
```

Figure 5.2: Sequential execution of the ocean and atmosphere steps for the Quasi-Geostrophic Dynamics model.

If we define a variable with this domain CTADEL knows which distances to choose from. A problem arising with this implementation is when two variables with different domains are combined in one equation. This has been isolated in the spline interpolation step.

5.2.2 Implementation

For the sake of separation of concern we divided the original model in three clearly distinguishable sub-parts,

1. Initialization. In this initial data for the model is calculated for the ocean and atmosphere models. We discuss this step in section 5.3.

2. Ocean step and atmosphere steps. The ocean step performs one time step for the ocean model, which we will explain in section 5.4. The atmosphere step goes analogue with the ocean part, this sub-part performs one time step for the atmosphere model. This part highly resembles the ocean step and we therefore do not discuss this part.

3. Data coupling, the coupling between the ocean and atmosphere model. Before every step, an interpolation is performed between ocean and atmosphere variables. This is also discussed in section 5.4.

A possible extension to the specification could be a high-grain parallelization of steps two and three. In figure 5.2 we see the current sequential execution of the model, while in figure 5.3 a possible parallel implementation is shown. Unfortunately data dependencies did not allow for a trivial parallelization; for example it has to be taken into account when data has to be exchanged between the models. This would require a redesign of the original model, which is beyond the focus of this work.

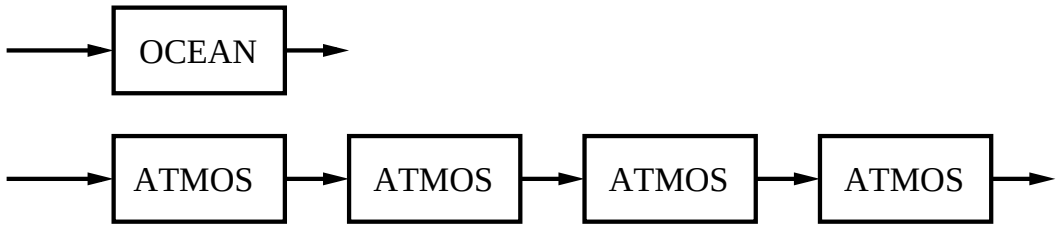


Figure 5.3: Parallel execution of the ocean and atmosphere steps for the Quasi-Geostrophic Dynamics model.

In the implementation of the original model, several numerical library calls were made. For example, for finding the solution to a tridiagonal matrix a tridiag solver was called. The use of calling external library calls from a specification has been incorporated in CTADEL. In section 6.1 we show the implementation of this feature when calling external library calls for implicit equations. Furthermore, we do make use of templates for this specification. The use of templates makes it possible to offer a number of standard numerical solution methods to the designer of the model , but leaves the actual implementation to CTADEL. The user can therefore specify the usage of a method while CTADEL can pick an appropriate and optimal implementation for the target architecture. For example when a Fourier analysis is needed the user can specify this like

```
V = Fourier(Q),
```

without bothering about the actual implementation; CTADEL decides if a library call should be made to an existing optimized library or fill in the code itself. In the current implementation CTADEL makes use of default numerical functions, like `spline2` from [61]. See section 4.2 for an explanation of the implementation of templates in CTADEL.

5.3 Initialization

In the first phase of the model, a number of general input parameters and initial data are read in, after which model specific data are calculated from these parameters. The original implementation of the model also could perform special tasks in this initialization phase, like some sort of crash recovery from an aborted previous run. We did not implement this in our specification.

An example equation belonging to this first step is equation (5.3) which reads in the CTADEL specification,

```
qcomp(P,DX,DY,H,GP,NL) := (C + sign *(f_zero/(GP * H))*(DPO)

          where C = (DX*(lapl P)/f_zero + beta * Y)
          where sign = (1 if k>1 \\ -1 otherwise)
          where (DPO=
                  (P-(P@(k=k+1))) if k<NL \\
                  ((P@(k=k-1))-P) otherwise
                )
          where Y = (j-1)*DY
      ).
CalcO(P,DX,DY, H,GP) :=
       ( zq(P,DY, H,GP,nlo) if borderj_O \\
         mqo(P,DY, H,GP) if borderi_O \\
         qcomp(P, DX, DY, H, GP,nlo) otherwise
       ).
```

This specification shows two generalized functions (including some boundary conditions) which, for example, can be called by,

```
qo = CalcO(po,dxom2, dyo, H_o, g_prime).
```

In this example, `qo` gets assigned the value

$$qo() = \begin{cases} zq() & \text{if condition } borderj\_O \text{ is true,} \\ mqo() & \text{if condition } borderi\_O \text{ is true,} \\ qcomp() & \text{else.} \end{cases} \tag{5.4}$$

In the quasi-geostrophic model for the calculation of the barotropic mode, a `chsolv` routine function is used, which uses a Fourier and a tridiag solver. In our specification we make use of templates to deal with these solvers; in the current specification CTADEL calls an external library function [61] to find a solution. A sample specification for this reads like,

```
tp_ch2 = (ch_solv(boundary_south, aa, ba) where aa=1/dya^2).
tp_ch3 = (ch_solv(boundary_north, aa, ba) where aa=1/dya^2).
```

5.4 Ocean and Atmosphere step

In this part of the specification, one ocean time step is calculated. A typical 'dynamics' equation which can be found is the calculation of the auxiliary currents, e.g. wind tendencies. The specification for the ocean dynamics reads,

```
u_ag = (-C*(P1-P2)*dxam2/f_zero
        where P1=((pa@(j=j+1)) if j<nya \\
                  (pa@(j=nya)) otherwise)@(k=1)
        where P2=((pa@(j=j-1)) if j>1 \\
                  (pa@(j=1)) otherwise)@(k=1)
        where C=(0.5 if noborder(nya) \\ 0 otherwise)
       ).
v_ag = (-1/2*((P@(i=i+1))-(P@(i=i-1)))*dxam2/f_zero
          where P=pa@(k=1)) if noborder(nya) \\
                0 otherwise.
```

In these equations we make heavily use of conditional statements for border conditions.

The atmosphere step goes analogue with the ocean part, this sub-part performs one time for the atmosphere model. This part highly resembles the ocean step and we therefore do not discuss this part.

In the QG model the atmospheric model gets called for every time-step. The ocean model is calculated only after `nstr` steps, with `nstr` a parameter for the model. Before the actual ocean model step is performed, data gets exchanged between the ocean and atmospheric variables. Between atmospheric steps no data get exchanged between the different variables.

The exchange takes place in two steps. First, several boundary conditions are checked. Second, data gets interpolated between the different models using spline interpolations. In the specification of the models, we make use of library calls to these functions.

5.5 Experimental Results

In this section the code generated by CTADEL for the quasi-geostrophic model is compared with the original hand-written code. Since the original code was not hand-optimized,

we can expect a performance difference with the generated code by CTADEL which performs some aggressive optimizations like common subexpression elimination. Standard compiler optimizations like loop optimization [34] where not applied. We ran the code on a scalar type of architecture, `[ATHLON]`. For an explanation of the hardware setup, see chapter 3.

All test-runs used an input grid with variable number of horizontal and vertical points and two layers per model. For some experiments we used a fixed number of thousand time-steps while we also conducted experiments with diverging time-steps. In order to reduce external influences on these times, we ran each experiment a number of times and calculated the average execution time. Because the deviations from these average times are in the order of tenths of percents, we do not include them in the figures.

With each run of the programs we compared the output of the generated code with the reference code. Since numerical solution methods are used over several time-steps, small differences appear in the output of both programs. However, these differences turned out to be relatively small and therefore acceptable.

In figure 5.4 we show the execution times for both the generated code and the hand-written code with a varying input grid size and a constant number of 1000 time-steps. As we see from both graphs, the code generated by CTADEL has a performance gain of 30% over the hand-written code. We also see that neither of the codes has a linear execution time with increasing input sizes, although one would expect this. We attribute this to the limited memory resources on the test machine leading to swapping; with an input size of more than 300 grid points, the model could not run properly anymore.

5.6 Conclusion

In this chapter we have shown how CTADEL can generate code from a specification for a model which uses two different resolutions. By default CTADEL assumes one resolution is used, which is not feasable for this model. We have extended CTADEL in such a way that resolution is coupled to a variable. For data exchange between the two submodels, we made use of the library calls to external interpolation spline routines. With some experiments we show that automatic generated code outperforms the hand-written codes for the quasi-geostrophic model.

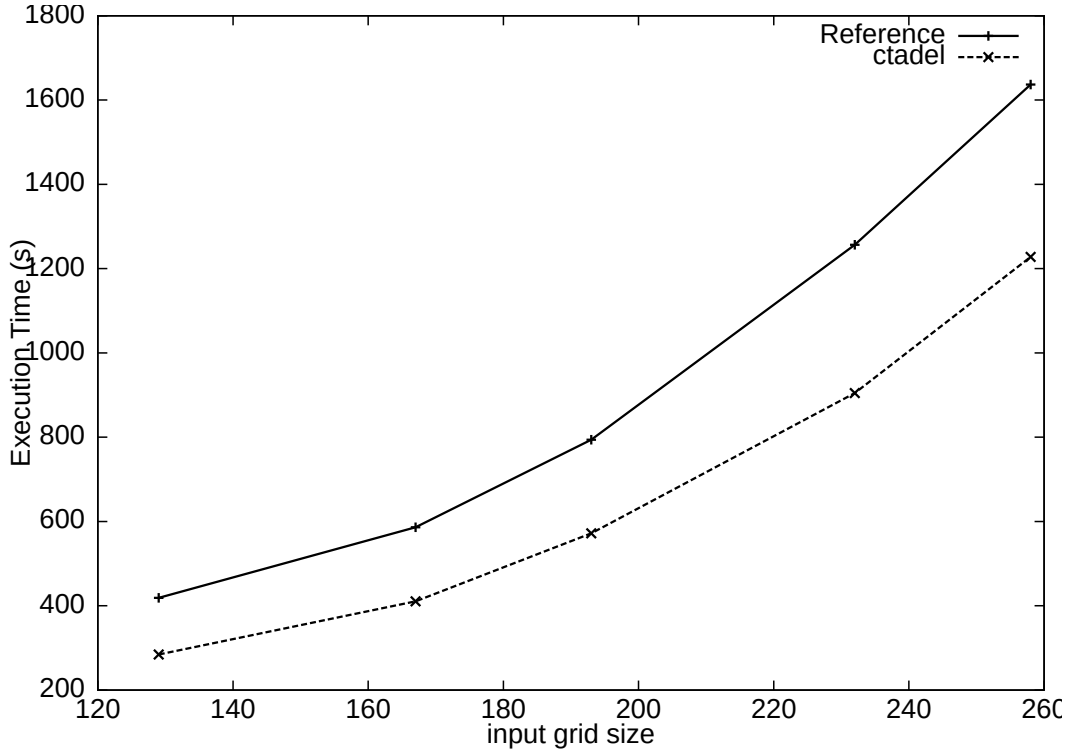


Figure 5.4: Execution times from the generated and reference code for the quasi-geostrophic model with a varying input grid size. Execution times are in seconds.

Chapter 6

Extending the Application Domain: A Turbulence Scheme

In this chapter we describe how to extend CTADEL in order to generate code for a turbulence scheme and specifically we discuss how additional problems related to these schemes, namely the specification of implicit equations, can be dealt with.

6.1 Turbulence

A turbulence scheme is a computationally intensive component of many large scale models, e.g. numerical weather prediction models. These prediction models can be characterized by having two main computational components, the *dynamics* and the *physics*. The *dynamics* are primarily involved with the fluid dynamics of the atmosphere, while the *physics* deal with the computation of the physical parameterizations. Turbulence schemes are part of the physics component.

When specifying turbulence schemes, one of the problems is the presence of implicit equations in the scheme. Implicit equations are harder to solve compared to explicit equations. In the explicit case one variable is defined as a function of other variables. In the implicit case an equation relating the two or more variables is given which cannot usually be solved to directly give one variable as a function of another. This prevents a straight-forward computation and one has to search for a solution. A PDE-problem solver needs to recognize this kind of equations and produce a solution method.

As a test-case for a turbulence scheme we have chosen the CBR scheme, designed

by Cuxart, Bougeault and Redelsperger [12, 66, 70]. The CBR scheme is in use by the HIRLAM system and is part of the physics module. In this scheme the vertical distance, or mixing length, is calculated for which particles can travel up or downward until their kinetic energy becomes zero. The set of equations for these calculations are implicit differential equations. This poses a challenge for CTADEL where equations are specified in an explicit form.

This chapter is organized as follows: in section 6.2 we explain some of the basics of turbulence in general and the CBR scheme in particular. We show the implementation for solving implicit equations in CTADEL in section 6.3, while section 6.4 addresses the produced code and some simulation results.

6.2 Turbulence scheme

In this section we will briefly explain the role of a turbulence scheme in a numerical weather prediction (NWP) model, followed by a more detailed view on the CBR turbulence scheme.

6.2.1 Physics and Dynamics

State of the art NWP models solve the basic equations of motion and conservation of mass. For example those for momentum are the Navier Stokes equations. For the sake of the current discussion, we will reduce these to the advection equations for momentum, which read

$$\frac{\partial u_i}{\partial t} = -u_j \cdot \nabla_j u_i \tag{6.1}$$

with $\nabla_j = \frac{\partial}{\partial x_j}$ and $u_i (i = 1, 2, 3)$ the velocity components of the wind.

It is neither useful nor feasible to solve equation (6.1) for every point in space and time. Instead, the equations are first averaged over suitable space-time domains. Then equation (6.1) becomes

$$\frac{\partial \overline{u_i}}{\partial t} = \overline{-(\overline{u_j} + u_j') \cdot \nabla_j (\overline{u_i} + u_i')}, \tag{6.2}$$

where $\overline{\cdots}$ denotes averaging and $\cdots'$ denotes deviations from the average value. When we work out equation (6.2) and we use the fact that the average over the deviations is

zero, we obtain

$$\frac{\partial \overline{u_i}}{\partial t} = \underbrace{\overline{u_j} \cdot \nabla_j \overline{u_i}}_{(6.3.a)} + \underbrace{\overline{u'_j \cdot \nabla_j u'_i}}_{(6.3.b)}. \tag{6.3}$$

Equation (6.3) consists of two terms,

(6.3.a) This term resembles the original equation (6.1) but now for the averaged quantities. In meteorological research, this term is usually referred to as *dynamics*. To be more precise, *dynamics* would also include similar terms in the full Navier Stokes equations and in the other "primitive" equations.

(6.3.b) The second term in equation (6.3) describes the covariances between the deviations of the wind components from their average values. NWP models deal with these and similar terms in the other equations in a package that is well distinguished from the dynamics; that package is usually referred to as the *physics*. The physics package also treats processes not described by the basic equations of motion, like phase transitions, solar radiation, and surface processes.

In principle, an equation for the time evolution of the physics terms in equation (6.3.b) can be derived by feeding them back into equation (6.1). However, this would result in the appearance of third order covariances, for which, indeed, new equations can be derived involving fourth order covariances, and so on. To get a solvable set of equations, the system is 'closed' by a process called 'parameterization': one tries to express the covariances in terms of the average quantities, symbolically given by

$$\frac{\partial X}{\partial t} = F1_X(u_i, u_j, T, q, \ldots) \tag{6.4}$$

with $X \in (u_i, u_j, T, q)$ where T and q stand for the temperature and the moisture, respectively. Beside the temperature T we also use the potential temperature θ, which reflects the temperature a parcel can get by pressure compression or decompression when moving downwards or upwards. The potential temperature is therefore related to the temperature and the pressure, given by

$$\theta = T(\frac{p_0}{p})^{\kappa} \tag{6.5}$$

with p the presure, p_0 a standard pressure and κ a constant with, for a perfect gas, the value $2/7$ in HIRLAM.

The effect of the physics terms in equation (6.3) is seen in nature as turbulence in the wind field. Hence, the package to parameterize those terms is called the 'turbulence scheme'.

In the discrete model the distances between the horizontal points are much larger than the distances between the vertical points. In equation (6.3.b) the gradients of the horizontal components ($j = \{1, 2\}$) can be neglected compared to the vertical component ($j = 3$). Therefore, we can ignore all components in equation (6.3.b), except those that involve the vertical components. This immediately points out an essential difference between dynamics and physics: dynamics involve horizontal differencing, whereas physics work 'column-wise': all operations act on quantities in a vertical column of the atmosphere.

6.2.2 CBR Turbulence Scheme

In HIRLAM, the turbulence scheme is the CBR scheme [12]. The basis of this scheme is the complete equation system for the second-order turbulent fluxes, variances and covariances. In the CBR scheme, it is recognized that equation (6.4) is not sufficient to accurately describe turbulence. To describe the time development of the second order covariances, some additional equations are needed, like

$$\frac{\partial \overline{u_i' u_j'}}{\partial t} = F2_X(u_i, u_j, T, q, \overline{u_i' u_j'}, \ldots). \tag{6.6}$$

In order to accurately describe atmospheric turbulence, the CBR scheme assumes one additional equation. This is the equation for the time evolution of the turbulent kinetic energy, given by

$$\varepsilon = \frac{1}{2} \sum \overline{u_i' u_j'}. \tag{6.7}$$

This yields a prognostic equation for ε. Equation (6.4) is modified to make use of the now known turbulent kinetic energy, which results in

$$\frac{\partial X}{\partial t} = F3_X(u_i, u_j, T, q, \varepsilon, \ldots), \tag{6.8}$$

where X is allowed to include ε, so $X \in (u_i, u_j, T, q, \varepsilon)$. The prognostic equation for the turbulent kinetic energy (TKE), symbolically defined in equation (6.8) for $X = \epsilon$,

reads

$$\begin{array}{rcl} \frac{\partial \varepsilon}{\partial t} = & \frac{1}{\rho_{ref}} \frac{\partial}{\partial x_k}(\rho_{ref} \overline{u_k} \varepsilon) - \overline{u_i' u_k'} \frac{\partial \overline{u_i}}{\partial x_k} \\ + & \frac{g}{\theta_{vref}} \delta_{i,3} \overline{u_i' \theta_v'} \\ - & \frac{1}{\rho_{ref}} \frac{\partial}{\partial x_j} \left(-C_\varepsilon \rho_{ref} L \varepsilon^{1/2} \frac{\partial \varepsilon}{\partial x_j} \right) \\ - & C_\epsilon \frac{\varepsilon^{3/2}}{L}. \end{array} \tag{6.9}$$

The virtual potential temperature is represented by θ_v, L is the mixing length, g the gravity acceleration and $\delta_{i,j}$ is the Kronecker delta tensor. The mixing length L at a given level z is determined as a function of the stability profile $\frac{\partial \theta}{\partial z}$. The algorithm for L relies on the computation of the maximum vertical displacement of a parcel of air.

In a relatively stable situation, the potential temperature at the levels above z is higher than that at z and therefore a parcel cannot easily move to a higher level. The only way that the parcel can go up, is if it has a vertical velocity, which is sufficient to overcome the temperature gradient. This implies that it must have sufficient turbulent kinetic energy. The initial kinetic energy of the parcel is assumed to be the mean kinetic energy $\varepsilon(z)$ at the level z. A parcel will have the possibility to go upwards until $\varepsilon(z)$ equals the energy absorbed by the temperature profile. Similarly a parcel may go downwards, leading to an equation for the downwards mixing length. The implicit equations for the upwards component l_{up} and downwards component l_{down} read

$$\begin{array}{rcl} \int_z^{z+l_{\text{up}}} \frac{g}{\theta_{vref}} \left(\theta_v(z') - \theta_v(z)\right) \mathrm{d}z' & = & \varepsilon(z), \\ \int_{z-l_{\text{down}}}^{z} \frac{g}{\theta_{vref}} \left(\theta_v(z) - \theta_v(z')\right) \mathrm{d}z' & = & \varepsilon(z). \end{array} \tag{6.10}$$

In CBR, the total mixing length L is assumed to be the geometric mean of both components

$$L = \sqrt{l_{\text{up}} * l_{\text{down}}}. \tag{6.11}$$

Equation 6.10 constitutes implicit equations for l_{up} and l_{down}.

6.3 Implementation

In this section the specification in CTADEL of the CBR turbulence scheme is described. The focus will be one of the problems presented by the scheme: implicit equations. In this section we focus on the implicit integral equations given by equation (6.10) for l_{up} and l_{down}.

The CBR scheme is defined as a set of continuous equations, which will be solved numerically. For this purpose the continuous domain is transformed into a grid, and the equations are discretized. Next, a discrete solution according to these discrete equations is obtained on the grid-points. The real solution, however, does not have to be on a discrete grid-point. Therefore the CBR scheme performs a linear interpolation to find the real solution in the continuous domain.

Equation (6.10) consists of two components, an upwards and downwards mixing length. Because calculation of both components use the same principle we will only discuss the downward component.

6.3.1 Implicit equations

The equation (6.10) for the mixing length L components can be symbolically rewritten as

$$f(\lambda) = 0. \tag{6.12}$$

This equation is an implicit equation which needs a solution method. Several of these solution methods exists, e.g.,

1. Elaborate methods, like Newton-Raphson or Runge-Kutta. These are popular methods and applied in many models.

2. Naive methods like a step method. This algorithm calculates f with small variable steps ($\Delta\lambda$) until a change of sign is detected, which means a solution of equation (6.12) has been found.

Newton-Raphson like methods are the most popular because these methods can be faster in finding a solution, compared to step methods [37]. However, in the hand-coded CBR scheme a step method is applied for two reasons. First, if a solution does exists, the step method guarantees it will find it. Second, if multiple solutions exists, the first found solution will be the one which is the closest to the origin of the search.

With the step method a number of monotonically increasing values l_n is chosen, so this implies that $l_{n+1} > l_n$. With these values $f(l_n)$ is calculated. A solution is assumed to be between l_n and l_{n+1} if the outcome of f changes sign between these two values. For example, if $f(l_n) < 0$ and $f(l_{n+1}) > 0$, λ is assumed to lie in between l_n and l_{n+1}. We can find λ by linear interpolation between $f(l_n)$ and $f(l_{n+1})$. An example is given in figure 6.1.

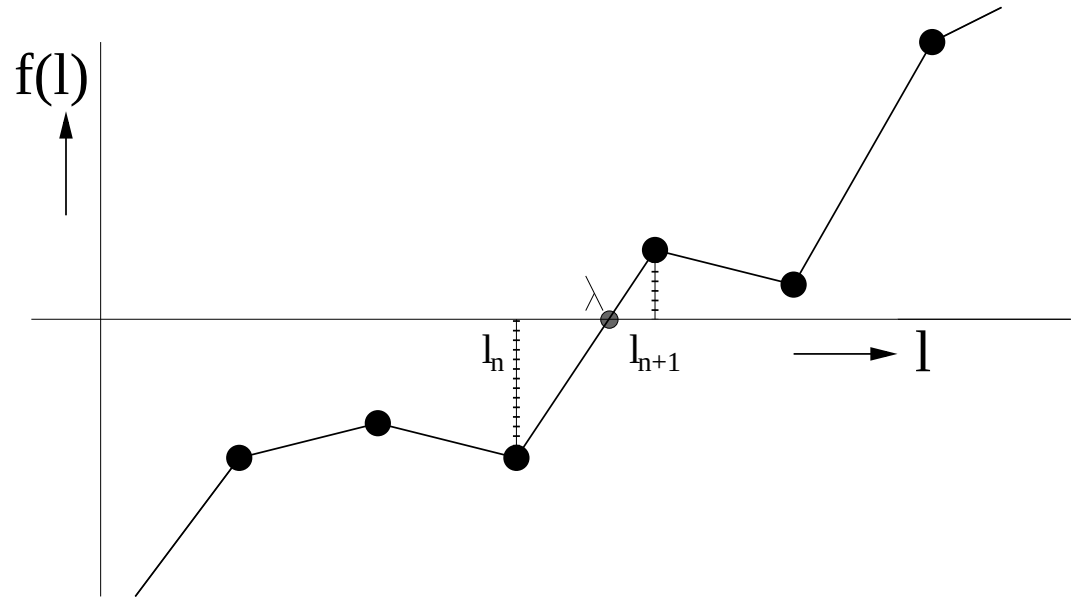


Figure 6.1: Example plot of $f(l)$. With the step method a number of monotonically increasing values l_n is chosen, so this implies that $l_{n+1} > l_n$. With these values $f(l_n)$ is calculated. A solution is assumed to be between l_n and l_{n+1} if the outcome of f changes sign between these two values. For example, if $f(l_n) < 0$ and $f(l_{n+1}) > 0$, λ is assumed to lie in between l_n and l_{n+1}.

It is possible to find several solutions for λ. In the CBR scheme it is assumed that the solution for λ is the first solution that is found. Additionally, it is also possible that the CBR scheme does not find a solution for λ in the available range. In this case CBR chooses the largest possible l_n allowed by the model as a solution for λ.

6.3.2 Specification

Automatic code generation for an implicit equation consists of three issues:

1. recognition of the implicitness of the equation,
2. determination of an efficient solution method,
3. generating efficient codes for the chosen solution method.

The first issue is basically a specification problem. In CTADEL the specification

```
A = integrate(F,eta=l..z)
```

means

$$A = \int_l^z B \; d\eta. \tag{6.13}$$

Depending on whether the values of l and A are defined in an other specification rule, CTADEL should be able to distinguish if these variables are input or output variables and if this specification is an explicit or implicit equation. Since CTADEL does not enforce a strict type specification of variables or order of the specification rules, it is not always possible to recognize the type of equation. Therefore, the code generator has to keep track of the kind of usage of variables. In the above example, for example, if the variable `A` was defined by a previous specification and the variable `l` is an output variable, CTADEL can indicate this equation as implicit. As an alternative, the user can specify the implicitness of the equation through the use of the `?=` operator, e.g.

```
l ?= A = integrate(F,z=z..l)
```

This operator tells CTADEL that the expression on the right-hand side of the operator should be solved for the variable on the left-hand side. In this example, for a given variable *A* and *F*, we should solve the equation for *l*.

The second issue deals with the fact that an implicit equation can be solved in many ways. The code generator has to choose an optimal solution method. One way of doing

this is automatically. CTADEL has to recognize the type of equation and pick a solution method suitable for this class of implicit equation. A drawback of this method is that an "optimal" solution method not only depends on the kind of equation, but also on the kind of input data. Another way is that the user specifies the complete solution method. Both methods have their advantages. CTADEL can try multiple solution methods, but it needs to have knowledge of various types of equations. It can be assumed that the user has this knowledge, and therefore the second method seems to be preferable. However, the selected solution method by the user might not be optimal. CTADEL can therefore search for a viable solution method and ask the user, as a last attempt. However, in the current implementation it is assumed that the user does have this knowledge and the user has to explicitly specify the implicitness of the equation and the give the solution method CTADEL should use.

Third, after the determination of an solution method, the generator has to generate efficient code for this method. We have tried to do this using a transformation based method. In the next two sections this method is discussed. Some parts of the specification and fragments of the generated code are also shown. Another possible method is library based, where the specification calls an external library function for the solution of the problem. Both methods have their advantages. Calling an external routine will provide an efficient solution method if it is assumed that an optimal library routine is available. This method has also the disadvantage of an extra overhead. When CTADEL supplies its own specification, the generated codes are inlined and can be further optimized by common subexpression elimination.

6.3.3 A Sample Code Generated by Automatic Transformation

After picking a solution method, the generator has to produce code for this method. We extended the knowledge base with a number of rewriting rules to accomplish this.

CTADEL makes use of a knowledge base in order to transform specifications and to produce efficient codes. This knowledge base, which uses a pattern-matching mechanism, makes use of rewriting rules. These rules are divided into groups, called strategies. A user can specify a number of strategies to solve a problem. Rules look like

```
strategy; pattern
          => action
          [<- condition].
```

Examples of rewriting rules are

```
diff;   diff(Const,_)
        => {0}
        <- is_constant(Const).
opdiff; df(exp(X),   1)
        => exp(X).
```

The first rule means

$$\frac{\partial Const}{\partial \xi} = 0, \quad \text{if } Const \text{ is a constant.} \tag{6.14}$$

The second rule is a rewriting rule for differentiating an exponential function. A rewriting rule for solving an implicit equation could look like

```
implicit; Var ?= F(Var) == 0
          => solution method
          specification for Var.
```

In our specification we use the following algorithm

1. Find the first occurrence where F changes sign.

2. Perform an interpolation to find the continuous solution if a solution is found.

The specification for the first step looks like

```
D_T1 ?= firstloc(
          integrate(F,k=1..nlev)
          -E_tk==0,
          j=1..nlev).
```

Internally CTADEL transforms this into

```
D_T1 = Conditional_integrate((zbeta *
         ((D_T0 @ (k=j)) - t_v_p) *
         (wdzh @ (k=j))) - E_tk == 0,
       k=1..nlev,D_T12).
```

The operation `Conditional_integrate` performs two things. First it performs a stepwise summation of the function and second it checks with each step if a change of sign has occurred. When the operation finishes the variable `D_T1` satisfies the conditions

$$\sum_{D\_T1(i,k)-1}^{k} F - E\_tk \leq 0, \tag{6.15}$$

$$\sum_{D\_T1(i,k)}^{k} F - E\_tk > 0. \tag{6.16}$$

if a solution was found. If no solution was found, the variable `D_T1` holds the value of 0 in which case no interpolation is needed.

Generating code from the specification of `D_T1` yields the following Fortran code

```
      DO 1430 k = 1,nlev
       DO 1440 i = 1,nhor
        DO 1450 j = k,1,-1
           dt12(i,k) = dt12(i,k)-
     .  9.80665*wdzh(i,j)*(1-
     .  dt0(i,j)/TPVIR(i,k))
           IF (dt12(i,k).GT.TKE(i,k)) THEN
               dt1(i,k) = j
               GO TO 1460
           ENDIF
1450    CONTINUE
        dt1(i,k) = 0
1460    CONTINUE
1440   CONTINUE
1430 CONTINUE
```

After these loops, a discrete index is found and the continuous downwards part of the mixing length is calculated, with respect to the continuous level height Δz.
Finally the mixing length can be specified as Equation (6.11)

```
L = sqrt( (D_T3 + ((D_T2) @ (j=k))) *
          (U_T3 + ((U_T2) @ (j=k)))).
```

In this specification, both the downwards and upwards component consist of a discrete part and a continuous part. When no solution is found, this continuous part is equal to zero.

6.4 Results

In this section the code generated by CTADEL for the CBR scheme is compared with the reference turbulence code from HIRLAM. For the distributed architecture we used a suitable wrapper to distribute the initial data. We also applied some standard optimizations like loop-fusion. To see the influence of the underlying computer hardware, we ran the code on a variety of architectures, like `[PENT-II]`, `[SUNE450]` and the `[DAS1]`. For an explanation of the hardware setup, see chapter 3.

Parallelization of the codes for the multiprocessor architectures was done manually. This was done by domain-splitting; each node only computes an appropriate part of the input grid. Because of the 'column-wise' behavior of the CBR scheme, data distribution is only needed before and after the calculations. On the DAS computer we used the MPI package for data distribution.

For the compilation of the programs we used the GNU Fortran compiler, version 2.95.2, for all architectures, with standard optimization turned on (`g77 -O2`). With each run of the programs we compared the output of the generated code with the reference code. Since the turbulence scheme is a numerical model, small differences appear in the output of both programs. However, these differences are relatively small and therefore acceptable.

All test-runs used an input grid with 31 vertical points and variable number of horizontal points. This number of vertical points is typical for the HIRLAM system. The results in table 6.1 reflect the absolute execution times. In order to reduce external influences on these times, we run each experiment a number of times and calculated the average execution time. Because the deviations from these average times are in the order of milliseconds, we do not include them in the tables or figures.

For the execution times, we only measured the calculation of the turbulence routines and not the extra overhead of reading and writing the input and output to and from disk. On the multi-processor architectures we included also process-synchronization in the execution times.

	Generated	Reference
PENTII	2.290	2.322
SUNE450	0.6021	0.6141
DAS1 20 nodes	0.1748	0.1897

Table 6.1: Execution times (s) on different architectures for the generated code and the reference code for the CBR-scheme using a 5000x31 input gird.

From table 6.1 we see that the generated code by CTADEL is comparable to the hand-written reference code on several different architectures.

To compare the scalability of the different codes we ran them on a multiple number of nodes on the DAS. We carried out two different experiments for which we compared the speedup,

1. A constant number of nodes (20) and a variable horizontal number of grid points.

2. A constant number of grid points (15000) and a variable number of processor nodes.

In Fig. 6.2 we ran both the generated code and the reference code with an input-grid of 15000 x 31 on a different number of nodes of the DAS system and present the execution times. If one looks in more detail, the generated code from CTADEL has a small advantage over the hand-written reference code. This advantage decreases slightly with an increasing number of nodes. The resulting speedup of both codes is shown in Fig. 6.3. For speedup we took the definition from [65], which defines speedup as the ratio between the time needed for a sequential algorithm to perform a computation and the time needed to perform the same computation on a machine incorporating parallelism. For our experiments we used the execution times of the parallel implementation on a single node as the sequential algorithm.

We see two interesting issues in Fig. 6.3:

1. Both codes show a good scalability.

2. The speedup for the reference code and the generated code seems to be super-linear.

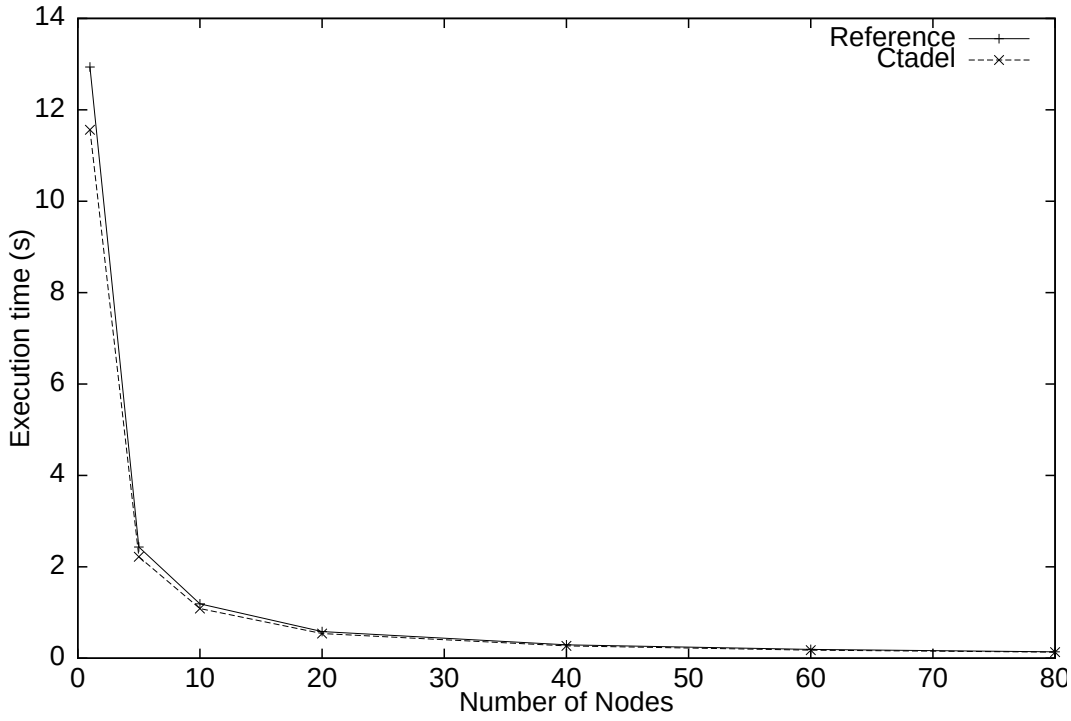


Figure 6.2: Execution time (s) as function of the number of nodes for the generated code and reference code for the CBR-scheme. Both codes were ran with an input-grid of 15000x31 on a different number of nodes of the DAS computer.

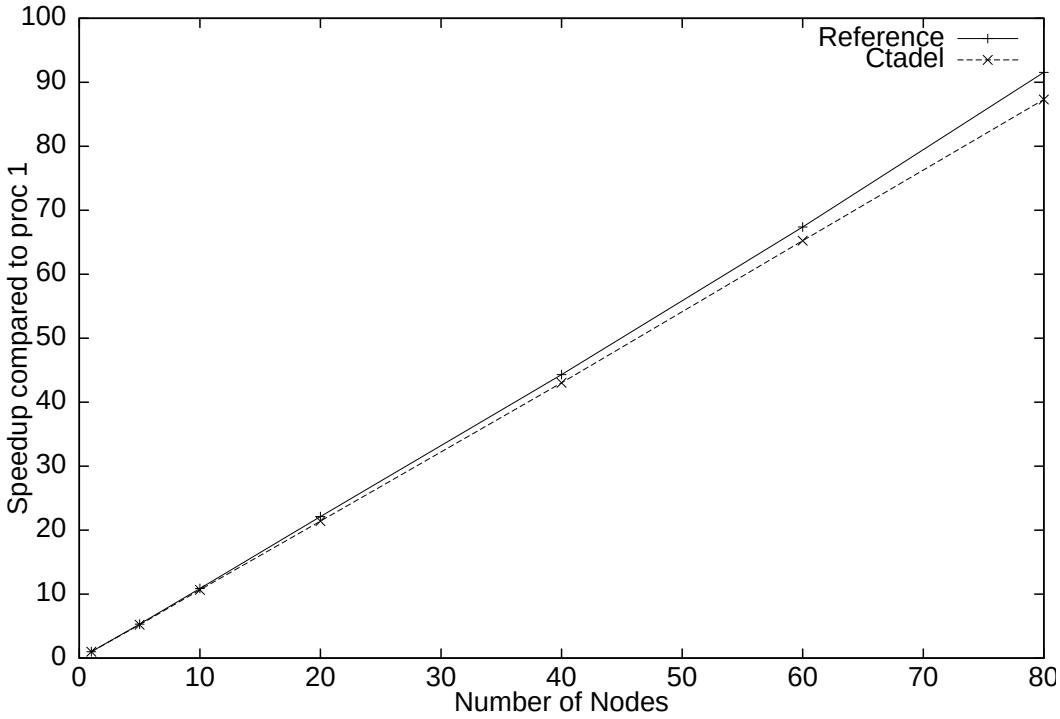


Figure 6.3: Speedup on the DAS computer for the generated code and reference code for the CBR-scheme. Both codes were ran with an input-grid of 15000x31 on a different number of nodes of the DAS computer. The horizontal axis gives the number of nodes while the vertical axis gives the relative speedup.

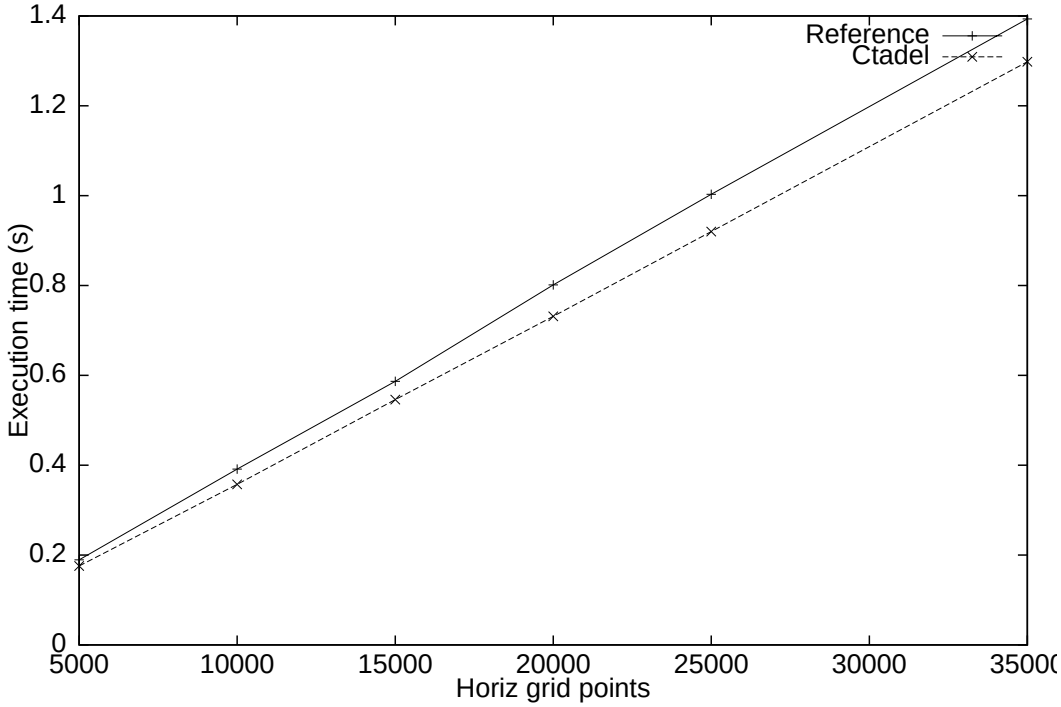


Figure 6.4: Execution time (s) on 20 nodes of the DAS computer as function of the number of horizontal gridpoints for both the generated code and the reference code for the CBR-scheme.

The second issue is related to the L2 cache of the Pentium Pro which has a size of 512 KByte. With an input grid of 10000x31, a single processor has an inefficient cache utilization. Since multiple processors only get a part of the input grid, the cache efficiency increases with the number of processors. This continues until the cache efficiency is optimal. For example, the relative speedup between one and five nodes is 5.1 (still super-linear), while this number between twenty and forty nodes drops to 1.9 (near linear, which would be 2.0). Therefore, we can conclude that speedup is not an appropriate measure for these experiments. This can be further demonstrated by increasing the input grid to 35000 x 31, where we obtain a speedup of 300 for 100 nodes. The reason for this fact is that the memory usage on one processor is so huge, that the node has to swap data to disk.

For Fig. 6.4 we ran both codes on 20 nodes of the DAS computer and varied the

number of horizontal gridpoints. We see that the generated code for the turbulence scheme is slightly faster than the reference code.

From the experiments in this section we draw three conclusions. First, in all experiments the generated code by CTADEL is equal in performance to slightly faster than the hand-written reference code. Second, on the DAS system we obtain an artificial super-linear speedup due to a limited cache/memory capacity if one uses a small number of processors. Third, taken into account this artifact of super-linearity, the speedup is near linear as could be expected from the fact that no communication is necessary. Finally, it should be mentioned that the generated code by CTADEL has a larger memory usage (1.2 times) than the reference code. It is expected that the inclusion of better loop-optimizations in CTADEL could reduce this usage.

Chapter 7

Semi-Lagrangian Formulations

Several formulations for solving advection equations exist, for example Eulerian and (semi-)Lagrangian formulations. Because of its simplicity, CTADEL used Eulerian schemes for solving these equations. However, Lagrangian-type formulations have a number of interesting properties in comparison with Eulerian-type formulations, for example a possible increase in the time step size. Use of Lagrangian formulations increases complexity and thus poses a new challenge to CTADEL.

In section 7.1 we will first describe some theory behind semi-Lagrangian and Eulerian formulations. We will only briefly touch the physical theory behind these formulations and the interested reader is referred to [72]. In section 7.2 we describe the sequential implementation of the semi-Lagrangian method and in section 7.3 we show how CTADEL is able to automatically generate code for semi-Lagrangian formulations. Automatic code generation for semi-Lagrangian formulations is a new challenge for the CTADEL system and we discuss new features like dimension-independent interpolation methods and an iterative approach for calculating expressions. In section 7.4 we show some results from experimental runs from the generated code. In these experiments we see that communication cost becomes the limiting factor on execution time on massive parallel machines. In section 7.5 we show a method, called “Halo On Demand”, to optimize communication between processors.

7.1 Theory

In this subsection we will discuss some of the numerical theory behind semi-Lagrangian formulations.

7.1.1 Introduction

State of the art NWP models solve the basic equations of motion and mass. It is neither useful nor feasible to solve these equations for every point in space and time. Instead, the equations are first averaged over suitable space-time domains. The distances in the time domain, the size of a time step, are often determined by the numerical methods applied. The HIRLAM system contains the following options: fully explicit versus semi-implicit schemes in combination with Eulerian versus Semi-Lagrangian advection formulations.

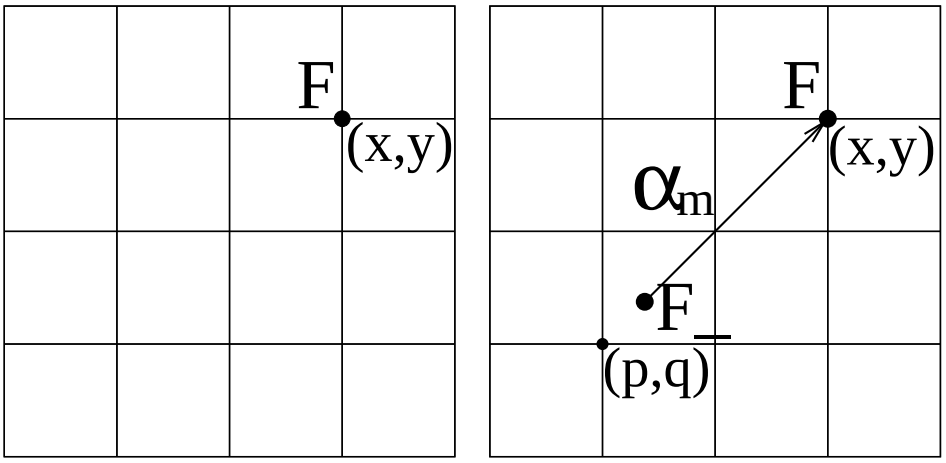
In an Eulerian scheme, the variables are determined on fixed points in a discretized space (see figure 7.1a), at every time step. In general in a Lagrangian formulation, on the other hand, one follows the particles along the path they follow in time (this path is called the "trajectory"). In a Lagrangian formulation, after some time, the chosen set of particles will be distributed irregularly.

7.1.2 Size of the Time Step

A number of numerical integration techniques require, for stability, that the Courant or Courant-Friedricks-Lewy (CFL) criterion should be met. This criterion reads,

$$C \equiv \frac{v\Delta t}{\Delta x} \leq 1, \tag{7.1}$$

with C the Courant number, Δt and Δx, respectively the time step and spatial steps and v is the maximum speed of an event. For explicit schemes in NWPs this event is the speed of sound, while with semi-implicit methods and a semi-implicit scheme this event is equal to the much lower velocity values of meteorological waves. Therefore, semi-implicit schemes already allow a significant increase of Δt with constant Δx over explicit schemes. Lagrangian formulations give a complete decoupling of v which can, in principle, yield infinitely large time steps, in as far as stability of the advection scheme is the limiting criterion. However, the time step is bounded by other parts of the NWP, like the physics and accuracy considerations. In practice the time step can be increased by a factor of at least 3 [64, 72].



(a) (b)

Figure 7.1: Different formulations of physical problems: (a) Eulerian formulation; (b) Semi-Lagrangian formulation. F is the required variable and F_{-} the value from the particle that moves via displacement vector α_m to coordinates (x, y). The coordinates (p, q) are used as base to calculate F_{-} using interpolation.

7.1.3 Numerical Principles of Semi-Lagrangian formulations

Physical problems, like NWPs, can be formulated in different ways. A common way is to describe the model in an Eulerian formulation: at each time step the variables are determined on fixed points in space (see figure 7.1a). A different way is the Lagrangian formulation. With this formulation, a set of particles is chosen, which is followed as they evolve in time. However, after some time the chosen set of particles will be distributed irregularly. As a result the required variables are known at the places where the particles are concentrated and almost no information is known at other places. A method that combines both formulations is known as semi-Lagrangian. Each time step the trajectories of only the set of particles, which during that time step ends exactly in one of the fixed grid points are computed (see figure 7.1b).

In figures 7.1a and 7.1b the basic numerical principles of Eulerian and Semi-Lagrangian formulations are presented in a two-dimensional grid.

Eulerian For every time step a quantity F is calculated based on other variables at point (x, y) and its direct neighbors in the input grid.

Semi-Lagrangian For every time step, the trajectory of each particle is determined. For

each grid point (the arrival point of the particle) the displacements in all dimensions to the departure point of the particle are determined. This departure point reflects the spatial position of the particle at the beginning of the time step. These displacements are calculated with an iterative process using the wind speeds at the arrival and departure points in time and space.

After finding the departure point of the trajectory, the value F_- from the beginning of the time step is determined by interpolation. Using this value and the found displacement vector α_m, the new value for F is calculated. This can be seen as following a particle from time t_n to $t_n + \Delta t$, where we know that the particle arrives at (x, y) at time $t + \Delta t$ following path α_m. Here Δt is the applied time step. This can be done with a two or three time step scheme. With the three step scheme, besides the departure point, also the midpoint of the the trajectory at time $t_n + 0.5 * \Delta t$ is calculated.

The algorithm for numerically calculating the quantity F can therefore be summarized in three steps:

1. Iteratively find the vector α_m with end point (x, y). In a finite number of steps the displacements to the departure point of the particle is determined using inter- or extrapolation. In each step we determine a discrete base-point (p, q) and use this point, among others depending on the interpolation method, to determine the start point of the vector.

2. Since the start point of the vector will not be on a grid point, we have to determine F_- by using an interpolation over values of grid points surrounding the departure point.

3. Finally, we update the value of the variable at the arrival point.

The complexity of the semi-Lagrangian method originates from the fact that the resulting values not only depend on neighboring points, but on points that are possibly far away. The location of the departure points must be determined iteratively, because it depends on an interpolation of the wind fields at departure and arrival points. Both are only known after interpolation in space (at the departure point) and extrapolation in time (at the arrival point). This can be illustrated by a simple example (taken from [72]),

namely the one-dimensional advection equation for F,

$$\frac{dF}{dt} = \frac{\delta F}{\delta t} + U(x,t)\frac{\delta F}{\delta x} = 0, \tag{7.2}$$

where $U(x,t)$ is a given function. We assume that the values F are known at all grid-points x_m at the departure time t_n. In that case the semi-Lagrangian formulation determines the values on the grid points at time $t_n + \Delta t$ by

$$\frac{F(x_m, t_n + \Delta t) - F(x_m - \alpha_m, t_n)}{\Delta t} = 0, \tag{7.3}$$

where the displacement α_m is the distance a particle travels in the x-direction in time Δt. If α_m is known, the value of F at arrival point x_m and time $t_n + \Delta t$ can be calculated with equation (7.3). The value α_m can be obtained by approximating $U(x,t)$ by

$$\alpha_m = \Delta t U(x_m - \alpha_m, t_n). \tag{7.4}$$

This equation can be solved by an iteration process

$$\alpha_m^{(k+1)} = \Delta t U(x_m - \alpha_m^{(k)}, t_n), \tag{7.5}$$

with an initial guess for $\alpha_m^{(0)}$. The values of U, possibly between grid points, are determined by an interpolation formula. An interpolation method should also be used to obtain the values $F(x_m - \alpha_m, t_n)$ in equation (7.3). It is possible to apply a linear, quadratic, cubic or linear/cubic interpolation procedure. This method can straightforwardly be extended to multi-dimensional problems, see [72].

7.2 Sequential Code

In this section we discuss the specification of an advection model using a semi-Lagrangian formulation. We first discuss the specification and code generation of sequential scalar Fortran code. When specifying a semi-Lagrangian method, two important issues arise: firstly, determination of the displacement vector α_m using an iterative process; secondly, interpolation between grid points of variables.

Displacement vector

Finding the displacement vector for the semi-Lagrangian method is done in a number of steps, applied at each grid point. First, an initial displacement vector is chosen, and

the departure point is calculated. This initial vector is estimated from the wind velocity from the previous time-step and the derivative of the grid distance and time step (e.g., dx/dt); this yields a rough estimate of the distance the particle could have traveled in the previous time step. A new displacement vector is calculated from the wind fields at the departure and the arrival points. This new displacement vector is fed back to recalculate the departure point. This can be repeated until a fix-point is found, in numerical sense. However, because of computational reasons the number of iterations is fixed. In the current implementation of the HIRLAM NWP a number of 2 is used. Summarizing, the determination of the displacement vector is as follows:

1. Estimate an initial displacement vector.
2. Calculate coordinates for a new displacement vector[1].
3. Calculate values by interpolation from grid points.
4. Repeat steps 2 and 3 once more.

For this iterative process we added a special `BLOCK` construction to the CTADEL system. A block is a coupled system of equations, for which the outcome of each iteration is used in the next step. A block construction also has a single convergence test attached. For simplicity one can compare it with a `DO-WHILE` construction in a programming language. An example block statement looks like,

```
BLOCK (n<5) (
  A = F(n-1) * B(n-1) iter on n.
  B = B(n-1) + G(n-1) iter on n.
).
```

For some variables `A,B,F,G` this block describes a fixed number of iterations for the equations $A_n = F_{n-1} * B_{n-1}$ and $B_n = B_{n-1} + G_{n-1}$ with n the iteration counter and pre-determined initial values. Because of this construct, CTADEL can perform loop fusion on the 2 statements and keep the dimension of the arrays. In a naive implementation, CTADEL would have to add an extra dimension to the arrays (for n), which can be

[1]Instead of keeping track of the length of the displacement vector, the coordinates for the base-point of the departure point are stored. For example, in figure 7.1b, the coordinates (p, q) are stored instead of the length/direction of vector α_m.

impractical (for large iteration counts) to impossible (for complex convergence criteria, like $A_n < 0.01$). Another practical use arises when generating parallel code, shown in section 7.3.

Interpolation Methods

Interpolation methods are "building blocks" for programmers of numerical models. However, programming them by hand is not as trivial as it may look. Writing interpolation methods creates large expressions consisting of several indirectly indexed array. A programming error is quickly made and debugging large expressions with indexed arrays is cumbersome. Furthermore, once a method is chosen, it is usually difficult to switch to another method. Automatic code generation of these methods relieves the programmer of this task.

We adapted the CTADEL system to allow "pluggable" interpolation methods. In this way, interpolation methods are transparent for the user (the developer of the model) and they allow the user to define his own methods. Specification of interpolation methods is done in a script file, which can be included and extended by the user. Compilation of a specification using interpolation methods is divided into three steps:

1. The include file and the model specification are read by CTADEL and parsed. CTADEL recognizes templates and checks the arguments for type and dimension.

2. A specific interpolation method is picked. This process can be done on criteria like grid-type or user specification. In the sequel we always pick the easiest method, the linear interpolation.

3. The specification for the interpolation method is merged in the model.

To show the concept of automatic code generation of interpolation we give a small example for the linear interpolation of a three-dimensional array. In this example, interpolation takes place over the variable *PARG* using the weight variables *PALFA*, *PBETA* and *PGAMA*; the integer arrays *KP*, *KQ* and *KR* contain the coordinates of the displacement vector. For each dimension, a variable is used for the weight and displacement vectors; for example *PGAMA* and *KR* for the vertical dimension. The result of the calculation is stored in the variable *PRES*.

An interpolation in CTADEL can be specified as `interpol(PARG, PALFA, KP)`. Although this specification is only one-dimensional, it can be easily extended

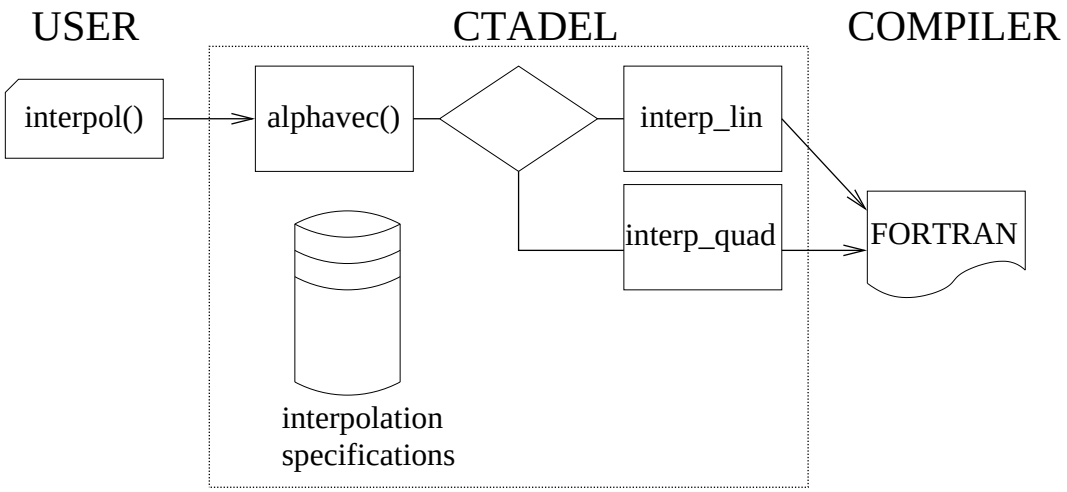


Figure 7.2: Simplified diagram of the process of code generation for interpolation methods by CTADEL.

to a multi-dimensional case. This can be done, because in our implementation, the result of one interpolation can be fed as an argument to another interpolation. So, the three-dimensional specification of the previous example looks like

```
PRES = interpol(PARG, PALFA, PBETA, PGAMA, KP, KR, KQ),
```

which is semantically equivalent with

```
PRES = interpol(
          interpol(
             interpol(PARG,PALFA,KP),
             PBETA,KR
          ),
          PGAMA,KQ
       ).
```

When CTADEL parses the specification of the model, the `interpol` operation is translated into an internal function called `alphavec` for every dimension. In this way, we can easily specify and pick several interpolation methods. In theory, we can even use different interpolation methods for different dimensions. For example, a quadratic method for the horizontal domain and a linear method for the vertical domain. The

specification for the alphavec function, in this example overloaded with two different interpolation methods, is given by

```
alphavec(_F, _A, _V, _X)‘interp_overloaded :=
     [ interp_lin, interp_quad ]
```

with interp_lin and interp_quad respectively a linear and a quadratic method. The parameters of alphavec, prefixed by an underscore, denote the variable or expression to interpolate (F), the displacement vector (A), the coordinates of the vector (V) and the coordinate to interpolate on (X).

In the front-end of CTADEL, the interpolation method is picked from the overloaded interpolation methods from alphavec. For example, the specification for the linear method looks like

```
interp_lin(F :: float dim Unit1, A :: float dim _, V :: coordinate,
           X :: coordinate) :: float dim Unit1 :=
({A} * F @ (X=V+1)) + ((1-{A}) * F @ (X=V)).
```

The generated code looks like

```
DO i,j,k
  t_4(i,j,k)=1-PBETA(i,j,k)

DO i,j,k
  t_6(i,j,k)=1-PALFA(i,j,k)

DO i,j,k
  PRES(i,j,k)=(1-PGAMA(i,j,k))*(PBETA(i,j,k)*(PARG(KP(i,j,k),1+K
.  Q(i,j,k),KR(i,j,k))+(PARG(1+KP(i,j,k),1+KQ(i,j,k),KR(i,j,k))-PA
.  RG(KP(i,j,k),1+KQ(i,j,k),KR(i,j,k)))*PALFA(i,j,k))+t_4(i,j,k)*(
.  PALFA(i,j,k)*PARG(1+KP(i,j,k),KQ(i,j,k),KR(i,j,k))+PARG(KP(i,j,
.  k),KQ(i,j,k),KR(i,j,k))*t_6(i,j,k)))+PGAMA(i,j,k)*(PBETA(i,j,k)
.  *(PARG(KP(i,j,k),1+KQ(i,j,k),1+KR(i,j,k))+(PARG(1+KP(i,j,k),1+K
.  Q(i,j,k),1+KR(i,j,k))-PARG(KP(i,j,k),1+KQ(i,j,k),1+KR(i,j,k)))*
.  PALFA(i,j,k))+t_4(i,j,k)*(PALFA(i,j,k)*PARG(1+KP(i,j,k),KQ(i,j,
.  k),1+KR(i,j,k))+PARG(KP(i,j,k),KQ(i,j,k),1+KR(i,j,k))*t_6(i,j,k
```

```
.  )))
     ENDDO
   ENDDO
ENDDO
```

This piece of code consists of three different three-dimensional loops. The common subexpression eliminator decided to keep outcome of the expressions $1-BETA(i,j,k)$ and $1-PALFA(i,j,k)$ in temporary variables. Loop fusion could be applied to merge all three loops into one.

We should point out one important difference between the handwritten code and the generated code. In the handwritten code, *PALFA*, *PBETA* and *PGAMA* are calculated within a loop over the vertical domain from three-dimensional variables and can therefore be passed as a two-dimensional parameter to an interpolation function. For the handwritten code, these three variables have the following property,

$$\begin{aligned} \textstyle\sum_{i=1..n} V(X,Y,i) = 1 \quad : \quad & \forall X,Y \in \text{ domain and} \\ & V \in \{PALFA, PBETA, PGAMA\}, \end{aligned} \tag{7.6}$$

with n the number of points used in the interpolation. For the linear interpolation, with $n = 2$, this gives $V(X,Y,1) + V(X,Y,2) = 1$. For the specification we take this property into account and use, for example, $V(X,Y)$ and $1 - V(X,Y)$ instead of $V(X,Y,1)$ and $V(X,Y,2)$ for the linear interpolation.

The hand written version of this interpolation, as it appears in the HIRLAM code, with loop indexes `JX` and `JY`, looks like,

```
IDX = KP(JX,JY)
IDY = KQ(JX,JY)
ILEV = KR(JX,JY)
ILM1 = ILEV - 1
PRES(JX,JY) = PGAMA(JX,JY,1)*(
     +   PBETA(JX,JY,1)*( PALFA(JX,JY,1)*PARG(IDX-1,IDY-1,ILM1)
     +                    + PALFA(JX,JY,2)*PARG(IDX  ,IDY-1,ILM1) )
     + + PBETA(JX,JY,2)*( PALFA(JX,JY,1)*PARG(IDX-1,IDY  ,ILM1)
     +                    + PALFA(JX,JY,2)*PARG(IDX  ,IDY  ,ILM1) ) )
     +                + PGAMA(JX,JY,2)*(
     +   PBETA(JX,JY,1)*( PALFA(JX,JY,1)*PARG(IDX-1,IDY-1,ILEV)
```

```
+                    + PALFA(JX,JY,2)*PARG(IDX  ,IDY-1,ILEV) )
+ + PBETA(JX,JY,2)*( PALFA(JX,JY,1)*PARG(IDX-1,IDY  ,ILEV)
+                    + PALFA(JX,JY,2)*PARG(IDX  ,IDY  ,ILEV) ) )
```

7.3 Parallelization

Parallelization of the dynamics of a NWP is usually done by domain-splitting. The computational domain is split-up and mapped onto the underlying architectural hierarchy, for example, a two-dimensional mesh. Because of communication patterns , splitting is only done in the horizontal domain. When data is needed from an adjacent domain this is communicated, for example, with a message system like MPI [55].

With the semi-Lagrangian method, indirect array addressing is used for finding the displacement vector. Theoretically, it is possible that with every iteration step, data is needed from a different processing node. This results in irregular communications. It should be noted that this only applies for interpolations; other calculations for the model can be done in parallel.

Based on a realistic maximum wind speed value, the applied horizontal grid size and the time step, it can be shown that the horizontal distance between the arrival point and the departure point of the vector will never be greater than four grid size values (Δx). This means that the base point for the interpolation will be at most three grid points away from the initial point. Because we need some adjacent points from the end point for the interpolation, the total maximum distance will be five grid points.

In HIRLAM these observations are used and the system offers two methods: extrapolation of the trajectory or the use of an `halo`. Extrapolation is used at the edges of the model domain. The halo contains a view on (border) data from neighbor processing nodes and it is used during iterations. Before the iteration process these halos are exchanged and no data exchange is needed between iteration steps. In the current implementation of HIRLAM a halo of width 5 is used, as explained previously. In case this is value not sufficient, the program aborts. Also, it is assumed that the size of the domains available on the neighboring processors is sufficient to cover the halo, i.e., that the domain on each processor is at least 5 by 5 grid points. This limits the number of processors. In realistic implementations, the resolutions are in the order of 100 grid points or more per processor; so this is not a serious limitation.

For the automatic code generation for code for border data we look at two target

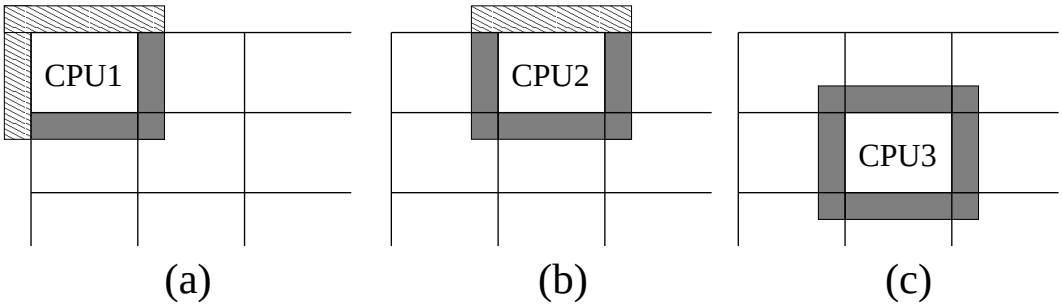


Figure 7.3: Three different processors at different positions in the partitioned data space. Corner processor 1 in (a) receives data from three adjecent processors. Processor 2 in (b) receives data from five adjecent processors. Processor 3 in (c) receives data from eight neighbor processors. In the figure, dotted areas represent data transmitted with the Halo method while striped areas represent data obtained by extrapolation.

Fortran dialects:

HPF Automatic code generation of code for border data is trivial for high performance Fortran (HPF). The `halo` approach resembles the concept of "shadow regions". For each array that shares its border data with adjacent processes, the HPF |SHADOW— and the |DISTRIBUTE— directives need to be generated and HPF takes care of the distribution. Since CTADEL makes use of the `block` structure for the iterative search, we know exactly which data needs to be shared.

Fortran 77 In Fortran 77 we have to make use of a message passing system[2]. In our case, we make use of the widespread MPICH implementation of MPI [55]. Since we known which borders need to be exchanged before the iterative search, we can simply generate MPI calls before this block. In our case, we make use of `MPI_Sendrecv` and we assume that an appropriate cpu mapping has been performed with, e.g., `MPI_Cart_create`, see for example [58].

Not every processor has to send the same amount of data; corner processors only have to send to 3 neighbor processors, border processors to 5 neighbor processors and center processors to 8 neighbor processors, see figure 7.3. When we increase the number of processors, the amount of data transmitted per node decreases. However, the

[2] Although shared memory is an option too, we did not investigate this.

total amount of transmitted data increases. The average amount of data to transmit per processor is in the order of

$$O((N-1)/(N \times N)), \tag{7.7}$$

with a cpu-mesh of $N \times N$ processors. The total amount of transfered data increases with order $O(N)$.

In the Full Halo method each processor receives data from all its neighbor processors, e.g. the processors not laying on the borders receives data from 8 processors. This could be done in 8 separate communications steps, for every neighbor processor data is exchanged. Disadvantage of this method is that we have to send/receive 4 large and 4 small data blocks. An alternative, more efficient way is to have only 4 communications in two steps. In figure 7.4 we have drawn this method in case of 4 processors. In step one, data is exchanged in north-south and south-north direction and and in step two, data is exchanged in west-east/east-west direction. For example, data for the halo of processor 4 from processor 2 is received in the first step. In the second step processor 4 receives data from processor 3, which also contains the corner of processor 1. It is necessary to complete the north-south communication before the east-west communication starts.

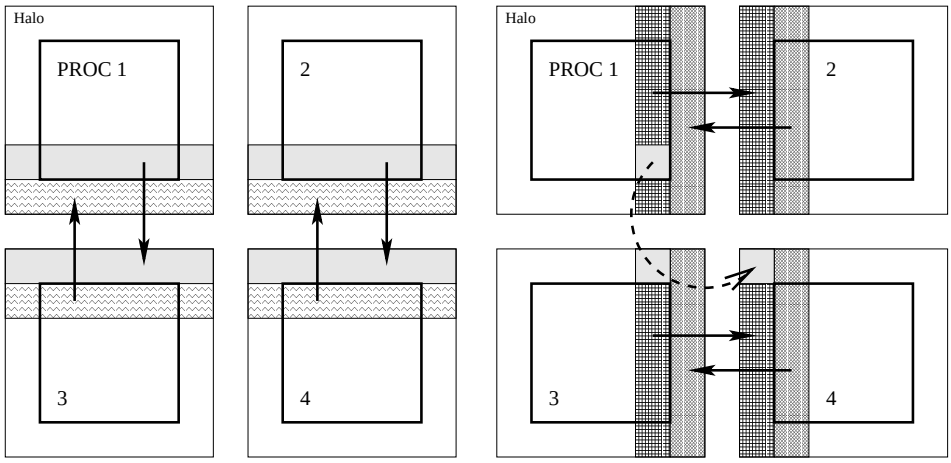
7.4 Experimental Results

In this section the code generated by Ctadel for the semi-Lagrangian scheme is compared with the reference code from HIRLAM. Advanced compilers apply loop fusion and other otimizations. However, the available compiler on our distributed architecture, gcc, was not capable of these optimizations yet. Hence we applied them by hand on the generated code.

7.4.1 Setup for the Experiments

To see the influence of the underlying computer hardware, we ran the code on a variety of architectures, like `[PENT-IV]` and `[DAS2]`. For an explanation of the hardware setup, see chapter 3.

In the automatically generated code, from all calculated expressions the values are stored in memory and no memory reuse is performed. For example, all temporary variables (introduced by the common subexpression elimination), are kept in separate vari-



(a) (b)

Figure 7.4: A four processor grid. Each processor has its own copy of the border data of the neighbor processors, denoted as halo and shown in the picture as square around the processors data. Communication of halo data is first done north-south/south-north (a) and in step two west-east/east-west (b).

ables while memory reuse might be possible. Since the number of iterations is rather low, this is not a serious restriction. However, with very large input grids, this can become a problem since memory is usually not an unlimited resource on most computers. At the moment we trust on techniques like array contraction, either applied by hand or by an optimizing Fortran compiler.

7.4.2 Experiments on a Scalar Architecture

The test-run on the scalar architecture was done with an input grid of 31 vertical points and a horizontal resolution of 114 by 100. These numbers are typical for the HIRLAM system. For our experiments we used real input data from the HIRLAM system. We only ran one time-step for the model. The results in table 7.1 reflect the absolute execution times measured with the `mclock` Fortran timer. In order to reduce external influences on these times, we ran each experiment a number of times and calculated the average execution time. Because the deviations from these averages are in the order of a few percent, we do not include them in the tables and figures. The "(opt.)" annotation denotes code with additional manual applied loop optimizations, like loop fusion.

	Intel IFC compiler	GNU G77 compiler
Generated Code	0.373	0.585
Generated Code (opt.)	0.342	0.530
Reference Handwritten Code	0.335	0.545

Table 7.1: Execution times (s) on a scalar architecture for a 114x100x31 input grid for the semi-Lagrangian formulations on a scalar architecture with the Intel Fortran compiler (IFC) and the Gnu Fortran-77 compiler (G77).

As we can see from table 7.1, the automatic generated code performs slightly better than the handwritten code, when we use the gnu Fortran compiler. This advantage diminishes when we make use of the Intel compiler. This compiler is an aggressive optimizing compiler, which fully exploits the Pentium IV processor architecture. This processor has the capability of executing 4 floating point operations in parallel (called Streaming Single-instruction- multiple-data Extension (SSE) by Intel) [7]. This explains the big differences between the code produced by the Gnu and Intel compiler. In table 7.1, we notice that the reference code gains more benefit from the Intel compiler then the generated code. However, the differences are rather small.

7.4.3 Experiments on a Distributed Memory Parallel Architecture

For all our experiments, we have made use of a distributed memory parallel architecture, the `[DAS2]`. For an explanation of the hardware setup, see chapter 3.

For the experiments on the DAS system, we applied domain-splitting for parallelization and made use of the full halo method as described in section 7.3. The horizontal domain is mapped on the processors in a square way. For example, when we use 9 processors this is seen as a square of 3×3 and the domain is mapped onto this square.

The test-runs for the DAS system were done with an input grid of 31 vertical points and a horizontal resolution of 114 by 100 points. Because of domain-splitting, this means that on, for example, 4 processors (2x2) each processor works on an input grid of 57x50x31.

In figure 7.5 we show the execution times on the DAS system on a varying number of processors. We assume that in production system the input is already available on the nodes or gets distributed through a high-speed interconnection network. Therefore, we

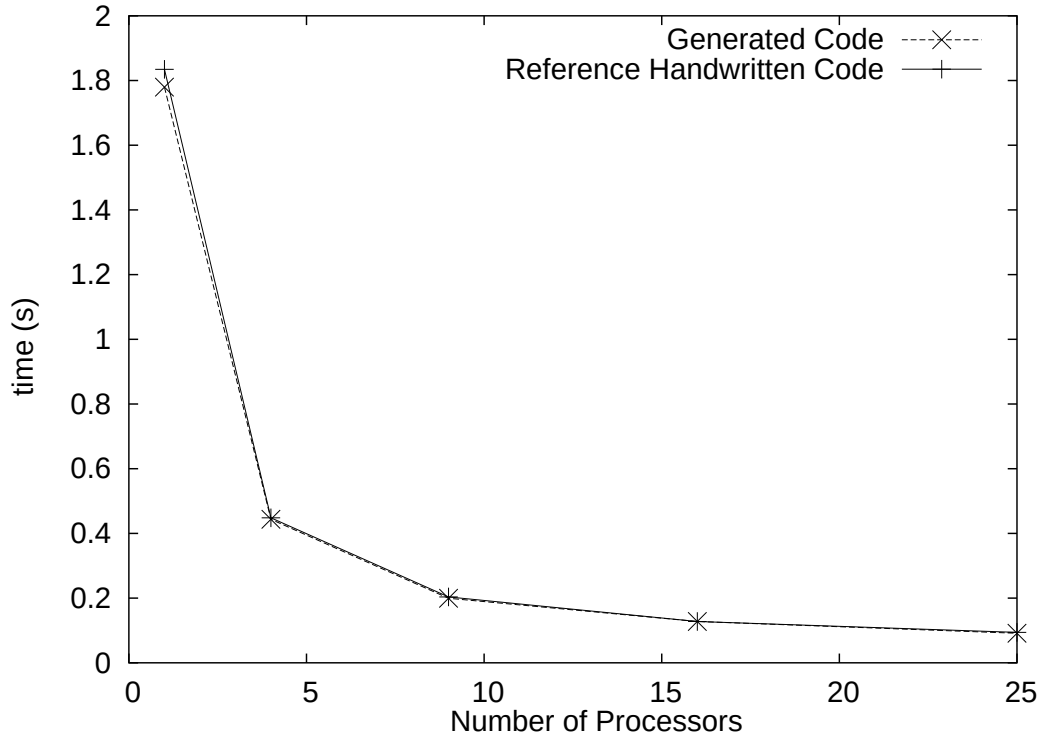


Figure 7.5: Execution times (s) on the DAS system for a 114x100x31 input grid on a varying number of processors for the generated and the reference code for the semi-Lagrangian formulations.

do not include the initial reading from disk and distributing of the input data in these execution times. However, communication between iterations and data collection at the end of the runs are included in the timings.

From figure 7.5 we see that both codes perform the same and both do not show a linear speedup. This can be attributed to communication. One would expect that with an increasing number of processors and a decreasing input size the amount of data to be sent will decrease. From equation (7.7) we see that the amount of data send per processor decreases. However, analysis shows that above 25 processors most of the time is spent in communications.

In figure 7.6 we have split execution time of an experiment into computation and communication parts. As we can see from this figure, the computation time shows a nearly linear speedup with an increasing number of processors. However, since the

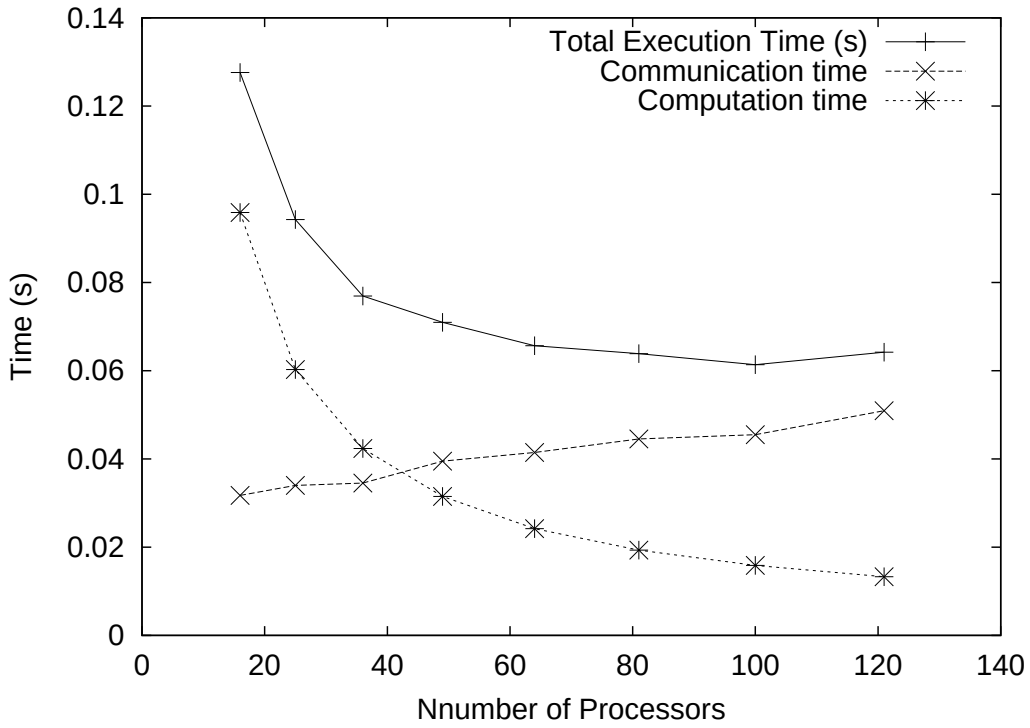


Figure 7.6: Performance analysis of an experiment split up in computation and communication time as a function of the number of processors used when using semi-Lagrangian formulations.

communication time increases almost lineary, the effective speedup of the model will not increase linearly. In the next subsection we show a more elaborate analysis of the execution time and communication time, called Halo on Demand.

7.5 Halo on Demand

In the previous sections we have explained how to automatically generate code for semi-Lagrangian methods from a specification of the model using the CTADEL tool. With a small number of experiments we have shown that these generated codes can compete with hand-written code on both scalar and parallel architectures. However, with both the generated and handwritten code, we observed a decrease in efficiency on the parallel architecture with an increasing number of processors. Analysis of the execution times

revealed that with a large number of processors used for computation, the time spent communicating data between the processors became larger than the computational time.

In this section we describe a solution, called "Halo on Demand", to improve communication. With this method, we examine the input data at runtime and we determine the kind of data to communicate and to which processor we have to communicate this data to. Using this strategy we are able to decrease the amount of data to be sent and thus decrease the communication time and increase the effective speedup of the model.

7.5.1 Halo on Demand strategy

One of the drawbacks of the Full Halo method is the large amount of data transferred between different processors. In figure 7.7 a scenario is drawn where computations are performed on nine processes. With the Full Halo method processor 5 receives halo data from eight other processors. However, if we look at the wind-flow in figure 7.7, only data from three processors is needed. An optimal implementation should therefore only transfer data that is actually needed based on the dynamically determined wind-direction.

Before the iterative search of the displacement vector α_m, an initial guess α_m^0 is made. When we analyze this vector at runtime, we can make a prediction of the wind direction. With this information we decide to which processor we send data. For example, in figure 7.7, sending data to from the processor number 5 to "west" processor number 6 with a west wind is ineffective. In this case only data from processors numbered 1, 2 and 4 have to send data to processor 5. As explained in section 7.3, we do this in two communication steps. An improvement of 50% procent compared to the full halo method which needs 4 communication steps. In theory, the direction of the displacement vector can change during the iterative search; the direction of the initial guess vector α_m^0 was wrong. However, we never observed this happen in our experiments.

We call this strategy where not all halo data is transfered, "Halo on Demand". It is a dynamic application-driven data communication strategy.

7.5.2 Related Work

A lot of effort and research have been devoted to communication problems with parallel programs. Several solutions exist, mainly targeted on data layout or global scheduling,

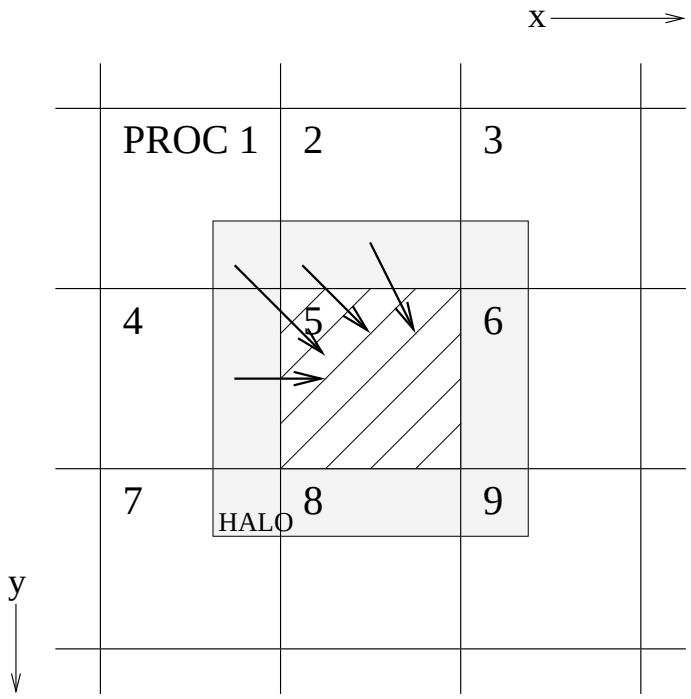


Figure 7.7: Calculation with a horizontal domain mapped on nine processors. With the Full Halo method, processor 5 (hatched area) receives data (gray area) from eight neighbor processors. Based on realistic wind-flow directions (expressed by arrows) data is only needed from processors 1, 2 and 4 in this example.

see e.g. [60, 65]. We look at two approaches, one of which resembles the Full Halo approach: the HPF shadow region, and the other resembles the Halo on Demand strategy: the Inspector-Executor method.

The Full Halo approach resembles the concept of "shadow regions" in High Performance Fortran (HPF), see for example [36] and the VFC compiler [5]. The Fortran programmer instructs the compiler about the shadow regions and shadow region widths by means of the `SHADOW` and `DISTRIBUTE` directives and the hpf-compiler generates the appropriate code for data transfer. An HPF example with shadow regions looks like

```
      REAL, DIMENSION (1000) :: A
!HPF$ DISTRIBUTE(BLOCK), SHADOW(1:2) :: A

      FORALL (i = 2, 998)
         A(i) = 0.25 * (A(i) + A(i-1) + A(i+1) + A(i+2)
      END FORALL
```

This piece of code performs a computation on a one-dimensional array `A`. This array is distributed over a number of processors. Each processor can keep most of the data local and only needs the boundary values (one at the beginning and two at the end). It is the compiler that takes care of the communication.

Basically, the shadow region is equivalent with the Full Halo approach. HPF performs some of the work the programmer has to do with the Full Halo approach. However, programming the halo by hand might give some opportunities for fine-tuning and optimizations. Another draw back is the fact that HPF compilers are not as widely available as communication libraries like `MPI`.

The Inspector-Executor approach was developed by Saltz *et al.* [69]. Its goal is to hide latency problems for parallel problems with irregular data accesses. With this method the compiler compiles a loop into two separate loops, an inspector and an executor. The inspector examines the data access patterns in the loop body and creates a schedule for fetching the remote values. After that, the executor retrieves these remote values according to this schedule and performs the real execution of the loop. As an optimization one can reuse the communication schedule if the inspector finds that the access pattern is the same for multiple iterations. An pseudo code example is given in figure 7.8 and figure 7.9.

```
for iteration = 1 to n
   for i = startindex to endindex
     a[i] = b[i] + c[ia[i]]
   end
   process(a)
end
```

Figure 7.8: Pseudo code example for the Inspector-Executor approach; the original loop.

The Inspector-Executor method resembles the Halo on Demand strategy; the data is first examined before the actual data transfer and execution takes place. However, the Inspector-Executor is based on low-level data access patterns, while the Halo on Demand strategy includes high-level application dependent information. The Halo on Demand examines the content of certain variables to make an assumption about the wind-direction and based on that analysis sends a large amount of data before the iterations. The Inspector-Executor examines the access pattern of the variables and based on that it decides to exchange data. If we would apply the Inspector-Executor method to our model, it would generate a lot of communication with small messages during and within the iteration steps. The total amount of data to be sent would be smaller, but the final communication times would be higher.

7.5.3 Experiments

In this section we present a number of experiments. In subsection 7.5.4 we show experiments with current input data (grid size $114 \times 100 \times 31$), taken from a reference NWP. This data is read in by a dedicated process, divided by a domain-splitting algorithm and distributed over the worker processors. The processors are configured as a two-dimensional mesh of $N \times N$ processors, with $1 \leq N \leq 11$. Only the horizontal plane of the input data grid is split up over the processors. For example, if $N = 2$ we use 4 processors, and for a total input grid of 114×100 horizontal points, this results in an input grid of 57×50 per processor. However, this yields unrealistic small input grids on large cpu-configurations. We, therefore, also run a number of experiments with larger data sets. We took the original data and interpolated this to a sixteen (4×4) times larger grid ($456 \times 400 \times 31$ points). We show the experiments with this enlarged set

```
//inspector phase
for i = startindex to endindex
   a[i]=b[i] + c.inspect
end

//create the communication schedule
c.schedule()

for iteration = 1 to n
   // fetch the remote values
   c.fetch()
   for i = startindex to endindex
     a[i] = b[i] + c.execute(ia[i])
   end
   process(a)
end
```

Figure 7.9: Pseudo code example for the Inspector-Executor approach, the transformed loop.

in subsection 7.5.5. Although this data set is unrealistically large for current weather applications, it might become the default standard for future simulations[3].

For all our experiments we have made use of a distributed memory parallel architecture, the `[DAS2]`. For an explanation of the hardware setup, see chapter 3. All code was compiled with the MPICH mpif77 compiler, which is a wrapper for the GNU G77 compiler, version 3.2.2. All timings were measured with the `MPI_Wtime` call, which yields the wall time clock as a double precision. All experiments were run hundred times to get accurate timings by calculating the average time for every experiment.

To present the results of the experiments we use different graphs:

Communication Time. For each processor, we show the measured communication time. With this data we calculate an average communication time for every mesh-size used, and these lines are called “Average over all processors”. Communication time is

[3]Technology has caught up with us and this enlarged size is already the standard for simulations at the time of writing this dissertation.

not equal for all processors since the border processors have to send fewer data then the center processors (see section 7.3). Therefore, deviations from the average communication time are significant. To express this, we also calculate the average communication time per processor for every mesh-size and denote these points as "Averages per processor". For a mesh-size of $N \times N$ this yields N^2 values.

To show the impact of external events on our measurements, we plotted the maximum and minimum communication time from every experiment. In theory, these values should be the same for experiments with the same set parameters. For example, for a mesh-size of ten by ten processors, the maximum should be the same for every run since the amount of data to be sent is constant and does not change between experiments.

Amount of Data Exchanged. For each processor, we also measure the size (in bytes) of the data that are sent to the neighbor processors. With these numbers we calculate a) the average size for each mesh-size over all processors and b) the average size of transferred data for each mesh-size per (single) processor.

Computation Time. For each processor, we present the measured computation time. With these numbers we also draw two graphs. In contrast to the amount of data exchanged, computation time is almost the same for all processors. Deviations from the average originate from the domain splitting process where in certain cases the domain could not be distributed in equal sizes to all processors. For example, 100 grid-points distributed over 3 processors, means sizes of 33, 33 and 34 points.

Allthough deviations where negligible small, for some experiments we found some unanticipated high values for a tiny amount of experiments. We suspect this to be external to our program, which show up in our experiments since we measure wall clock time. Therefore, we plotted the maximum and minimum computation time for every experiment.

7.5.4 Experiments with current input data

In figure 7.10 we show the communication time for both the Full Halo method (7.10a) and Halo on Demand strategy (7.10b). We observe an interesting paradox in figure 7.10: although the amount of data sent per processor decreases with an increasing number of processors, the time taken to communicate this smaller amount increases. We will explain this in subsection 7.5.7.

We show the minimum and maximum communication time per experiment for the

Full Halo method (7.11a) and Halo on Demand strategy (7.11b) in figure 7.11. In an ideal world, we would see only two values per cpu mesh-size; maximum and minimum communication times would be the same for every experiment with the same parameters. However, the DAS2 cluster is used in the real world, where we have to take a large number of external influences into account. The effect of other users using the same Myrinet switch in a multi-programmed multi-user environment or external hardware influences like slightly different bus frequencies can influence our measurements. In both figures, we see that a number of values have a huge deviation from the average. However, the number of outliers is too small with respect to the total number of experiments that we do not think this is a serious problem.

The amount of data sent per processor is shown in figures 7.12a and 7.12b. The graphs in figure 7.12 match the predicted function $(N-1)/(N \times N)$ for the average amount of data sent (see subsection 7.3).

In figure 7.13 we show the computation time for both methods. We can make the following observations:

1. Computation times scale nearly perfectly linear (even slightly super-linear with a large number of processors, caused by an improved usage of data cache of the processors).

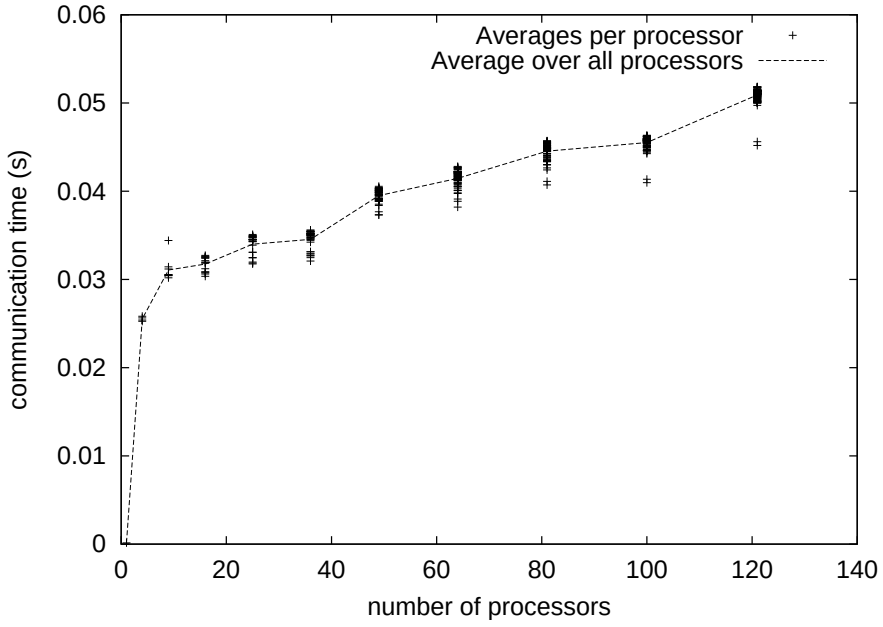
2. The computational overhead caused by the Halo on Demand strategy can be neglected.

In figure 7.14 we show the minimum and maxium computation time for the Full Halo method (7.14a) and Halo on Demand strategy (7.14b). We would expect these values to adhere to the values from figure 7.13, which they do but for a tiny amount of outliers. We presume this to be a hardware glitch.

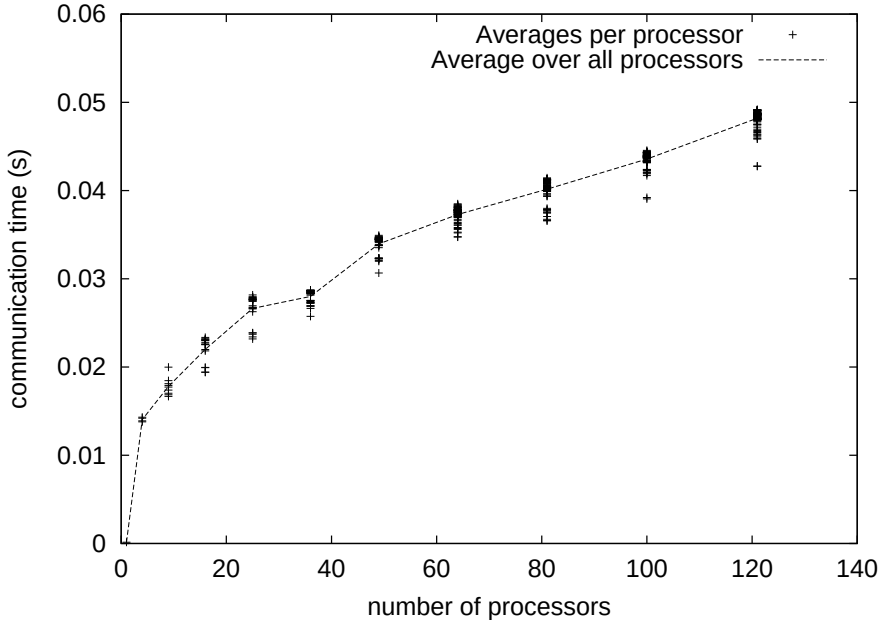
7.5.5 Experiments with future input data

To investigate the impact of the size of the input grid, we run the previous experiments on a 16 times larger input grid. We took the input data from our reference NWP and interpolated these to a 4×4 bigger data field, resulting in a $456 \times 400 \times 31$ input grid, which might become the default grid size for future NWPs.

In figures 7.15a and 7.15b we show the communication times. Compared with the results from the experiments with the current input (see figures 7.10a and 7.10b) we ob-

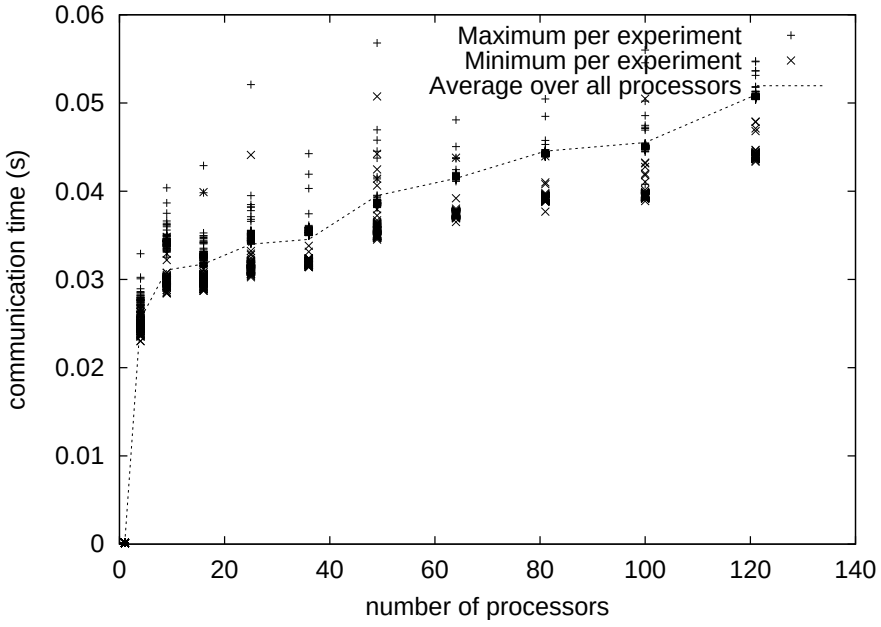


(a)

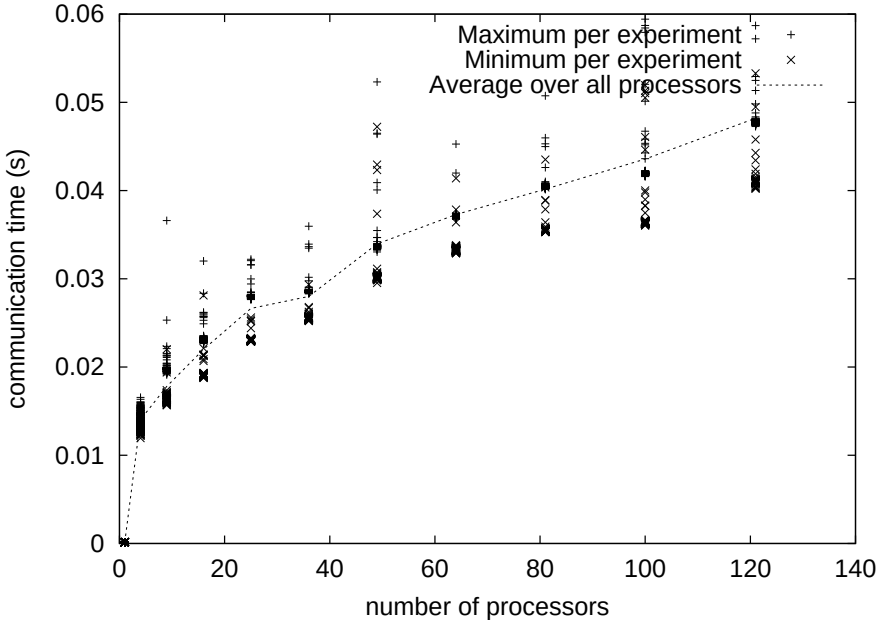


(b)

Figure 7.10: Communication time for a current sized input grid with semi-Lagrangian methods with (a) Full Halo and (b) Halo on Demand. For every individual processor we plot the average from all the runs and we denote this as “Averages per processor”. For every permutation of the number of processors we calculate the average of all processors for that number of processors and denote this as “Average over all processors”.

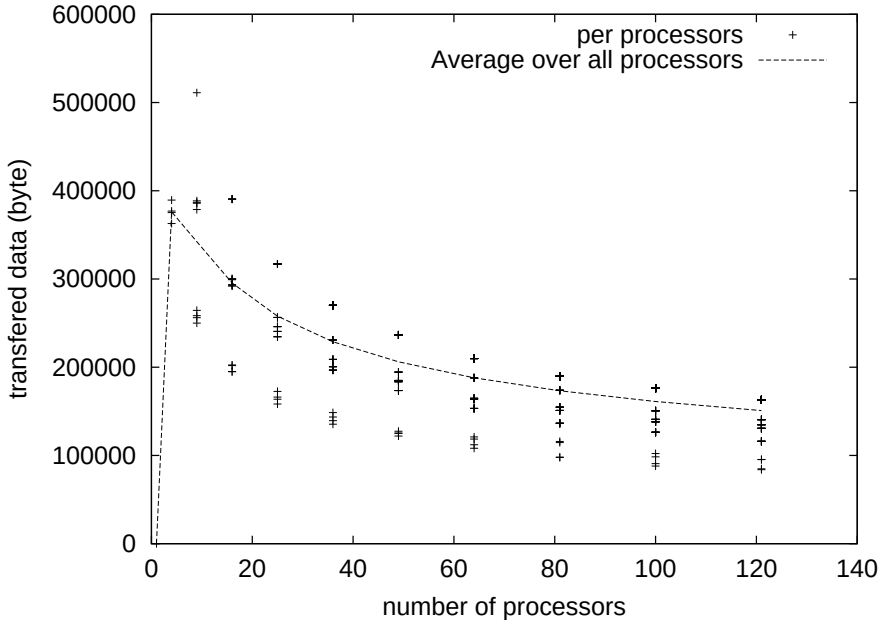


(a)

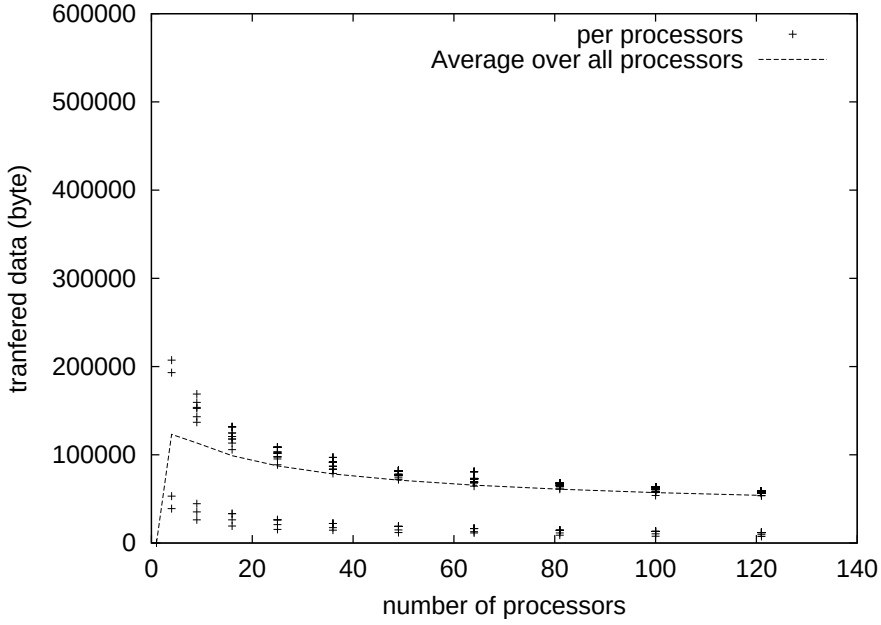


(b)

Figure 7.11: Maximum and minimum communication time per experiment for a current sized input grid with semi-Lagranian formulations with (a) Full Halo and (b) Halo on Demand. For every permutation of the number of processors we calculate the average of all processors for that number of processors and denote this as “Average over all processors”.

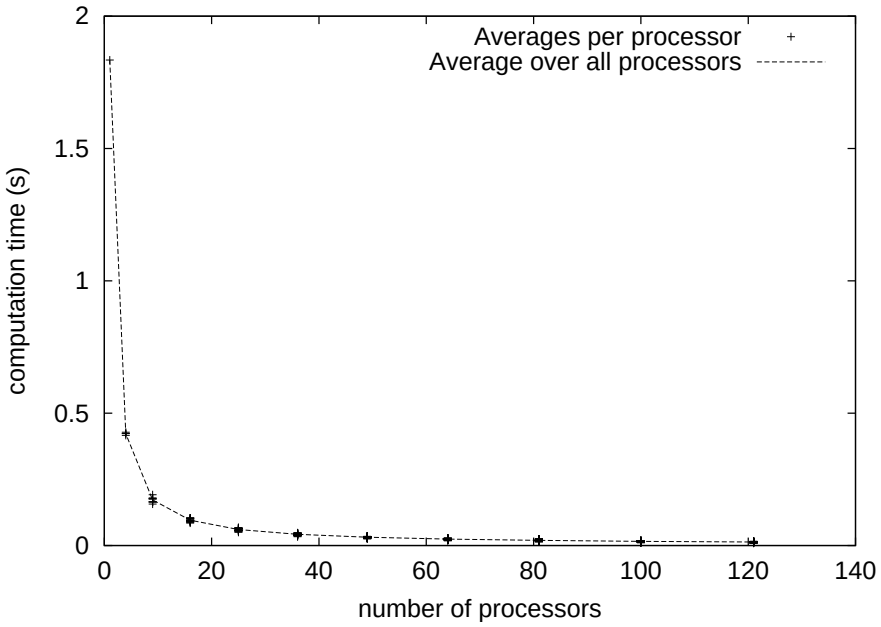


(a)

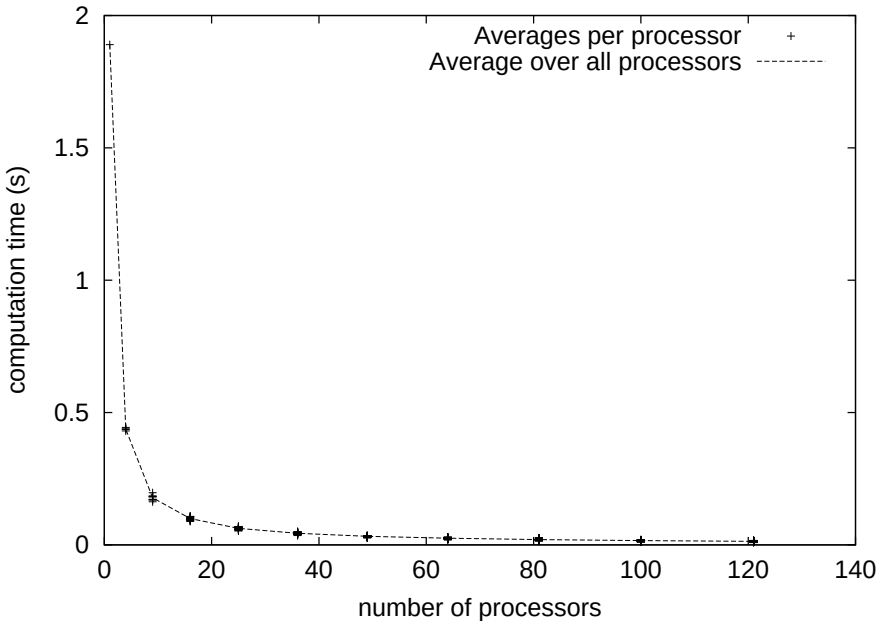


(b)

Figure 7.12: Average number of bytes sent per node and over all processors on a current sized input grid with semi-Lagrangian methods with (a) Full Halo and (b) Halo on Demand.

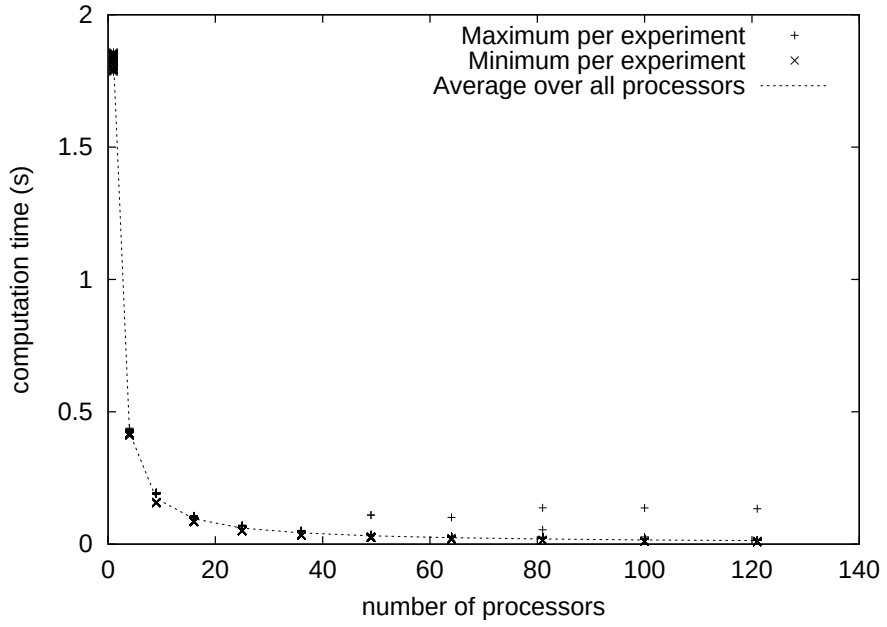


(a)

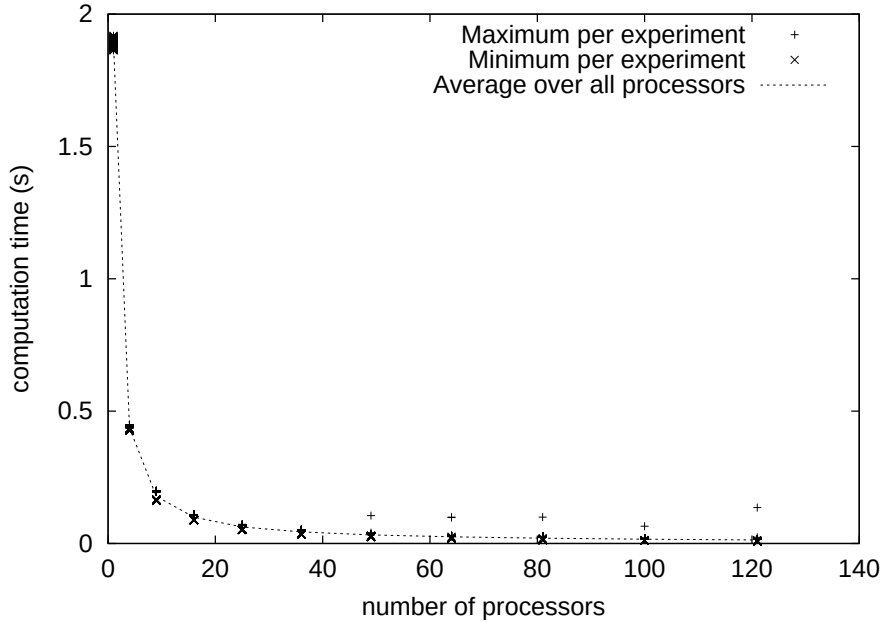


(b)

Figure 7.13: Average computation time per node and all processors on a current sized input grid with semi-Lagrangian methods for (a) Full Halo and (b) Halo on Demand.



(a)



(b)

Figure 7.14: Maximum and minimum computation time per experiment for a current sized input grid for semi-Lagrangian formulations for (a) Full Halo and (b) Halo on Demand. For every permutation of the number of processors we calculate the average of all processors for that number of processors and denote this as “Average over all processors”.

serve a remarkable difference. The communication time is an almost constant function, independent of the number of processors. We explain this in subsection 7.5.7.

We show the minimum and maximum communication time per experiment for the Full Halo method in figure 7.16a and Halo on Demand strategy in figure 7.16b. We observe some extreme outliers in this figure; for example with 9 processors the maximum of a significant number of experiments has twice the communication time as the average. In figure 7.15 we see that these are values from a small number processors. We suspect a hardware glitch or a (DMA) cache problem.

The figures 7.17a and 7.17b present the absolute amount of data sent per processor. Compared to the results with a current input (see figures 7.12a and 7.12b) we observe an increase of data sent with a factor of 4. This is as expected; the area of the input grid increases with an factor of 16, but the circumference only increases with a factor of 4.

This does not hold for the computation time, given in figures 7.18a and 7.18b, which depends on the surface size of the input grid. Compared to figures 7.13a and 7.13b, we see, as expected, an increase of computation time with a factor of 16.

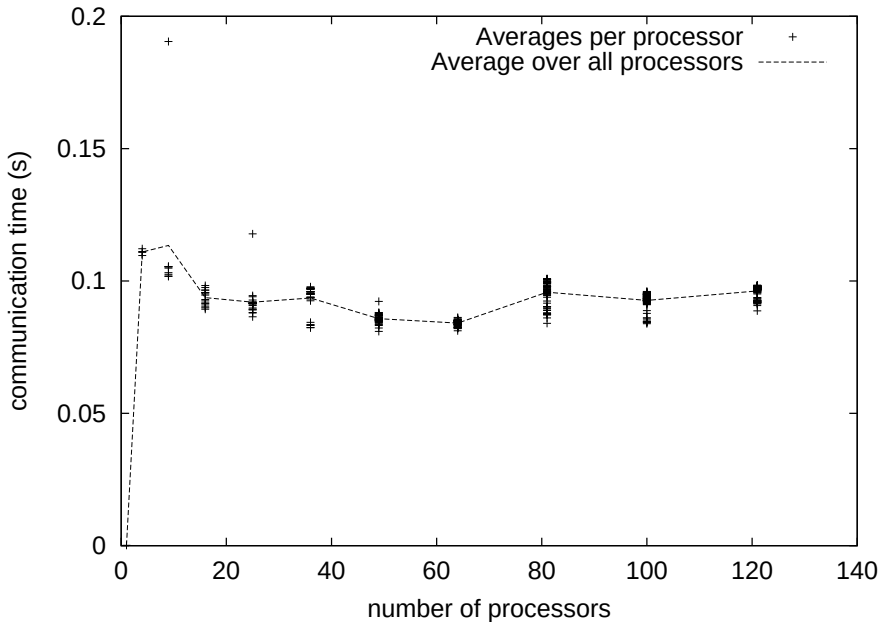
We show the minimum and maxium computation time for the Full Halo method in figure 7.19a and Halo on Demand strategy in figure 7.19b.

7.5.6 Speedup

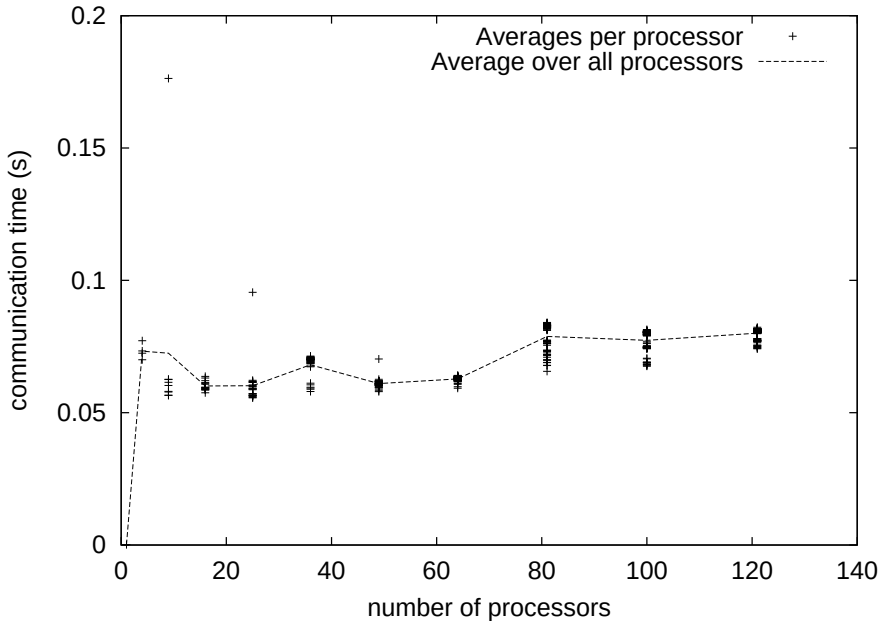
To show the effect of the Halo on Demand strategy on the total execution times we have calculated the speedup of the model with and without the use of the Halo on Demand strategy. In figure 7.20a we show the speedup of the Full Halo method and the Halo on Demand strategy as a function of the number of processors using a current input size. For an enlarged input grid this speedup is shown in figure 7.20b. From these figures we see that

- When increasing the data input size, the speedup increases. Computation time is in the order of $O(N^2)$, with $N \times N$ the input size, while communication time is in the order of $O(N)$. So, with an increasing N we spend more time in computation and have relatively less overhead of communication.

- The Halo on Demand strategy gives an increased speedup compared to Full Halo method.

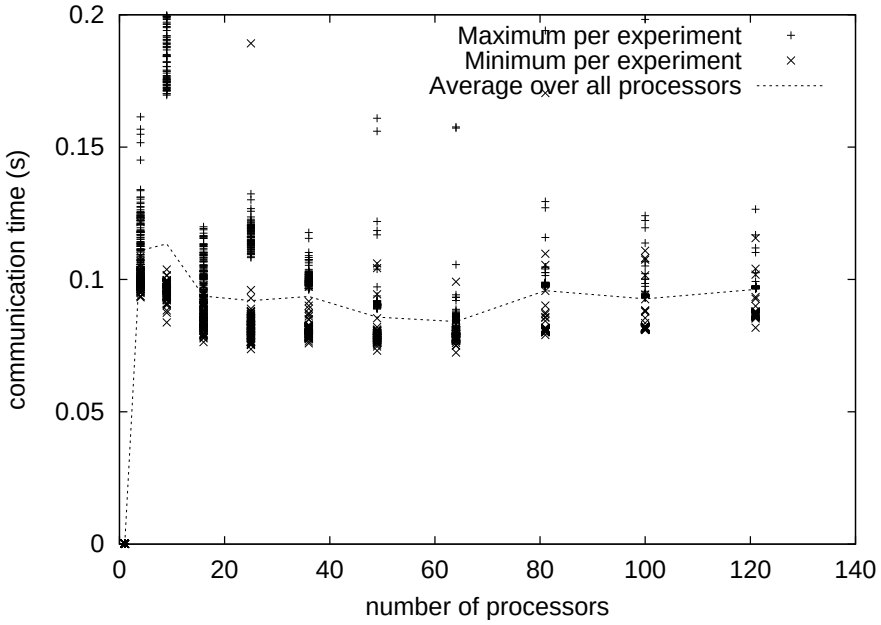


(a)

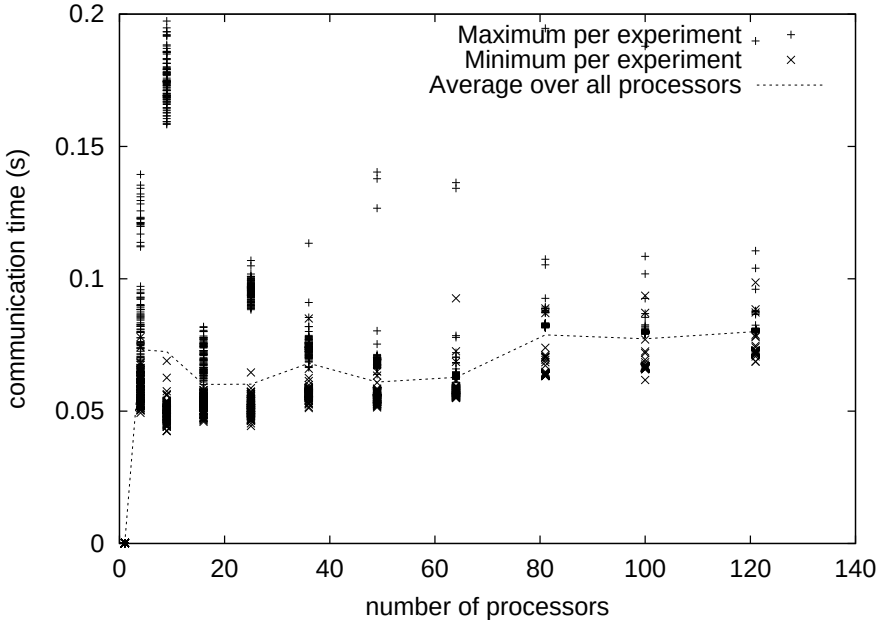


(b)

Figure 7.15: Average communication time per node and over all processors on a future sized input grid for semi-Lagrange forumlations for (a) Full Halo and (b) Halo on Demand.

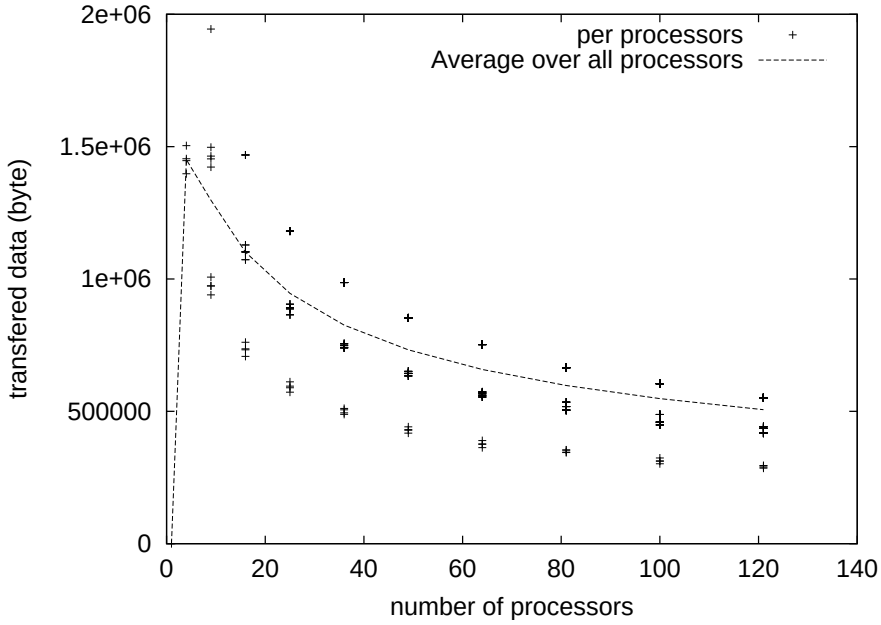


(a)

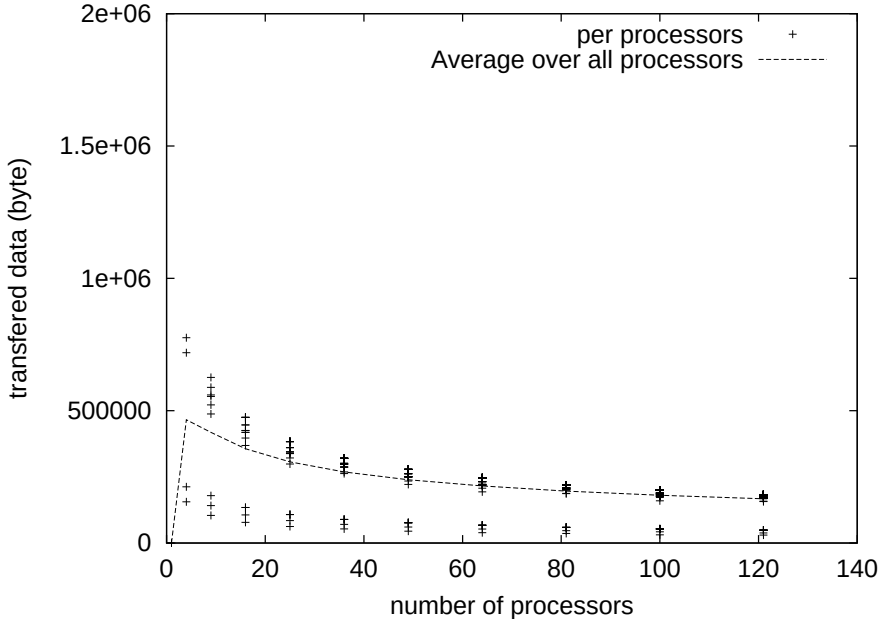


(b)

Figure 7.16: Maximum and minimum communication time per experiment on a future sized input grid with semi-Lagrangian formulations with (a) Full Halo and (b) Halo on Demand. For every permutation of the number of processors we calculate the average of all processors for that number of processors and denote this as “Average over all processors”.

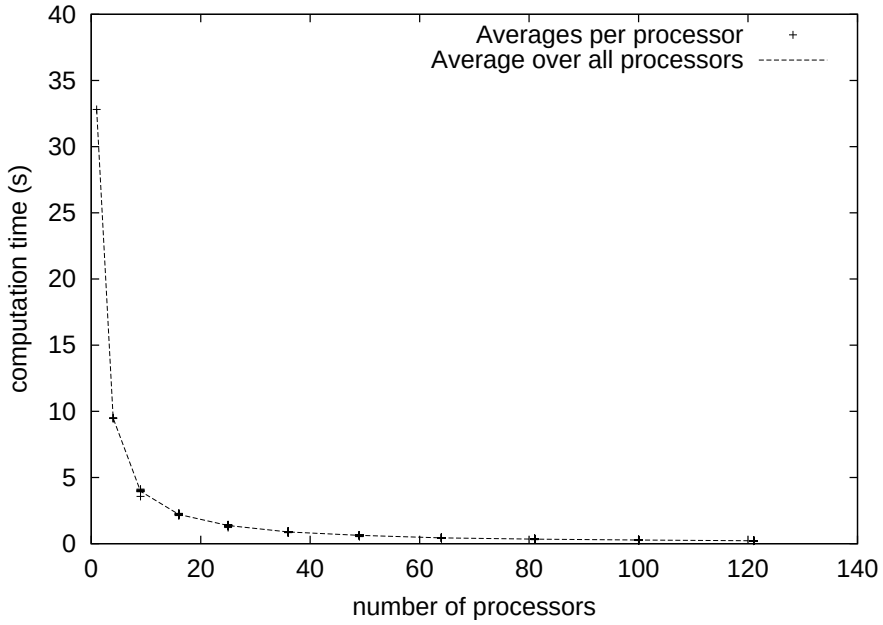


(a)

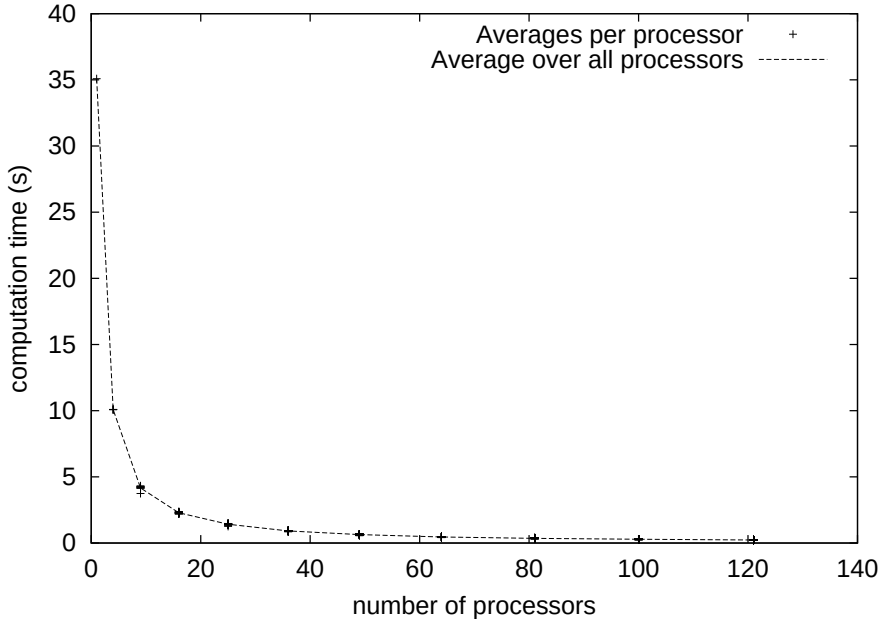


(b)

Figure 7.17: Average number of bytes send per node and over all processors on a future sized input grid for semi-Lagrangian formulatons for (a) Full Halo and (b) Halo on Demand.

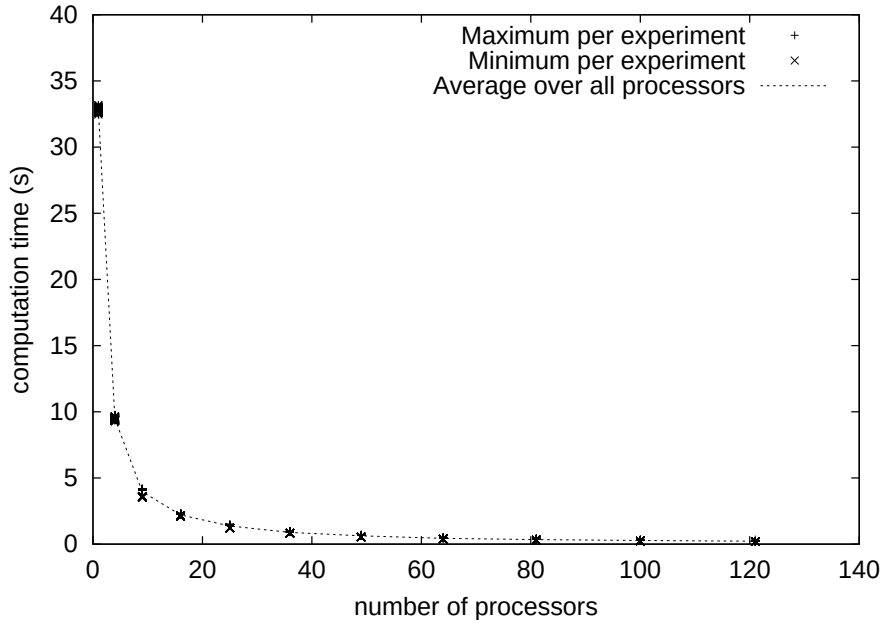


(a)

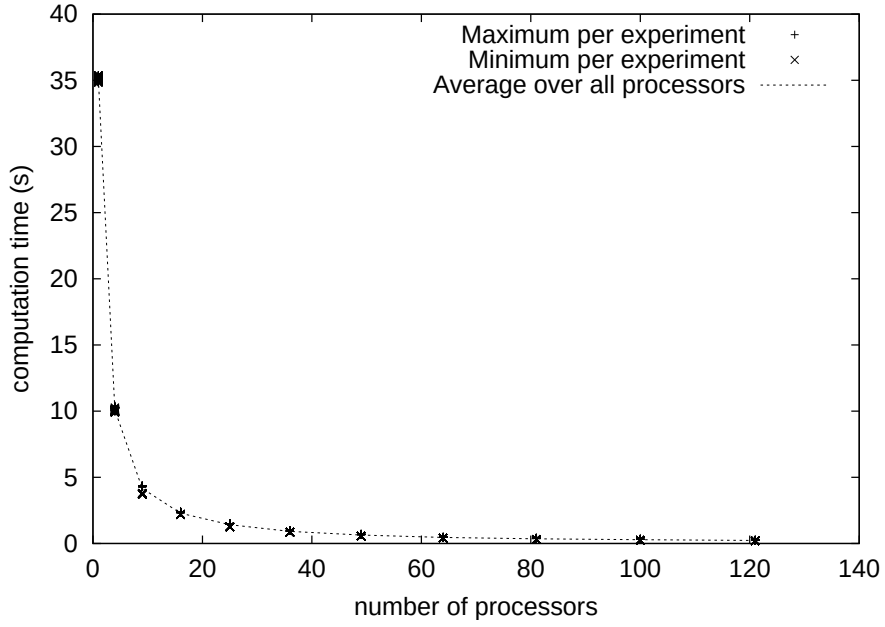


(b)

Figure 7.18: Average computation time per node and over all processors on a future sized input grid for semi-Lagrangian formulations for (a) Full Halo and (b) Halo on Demand.

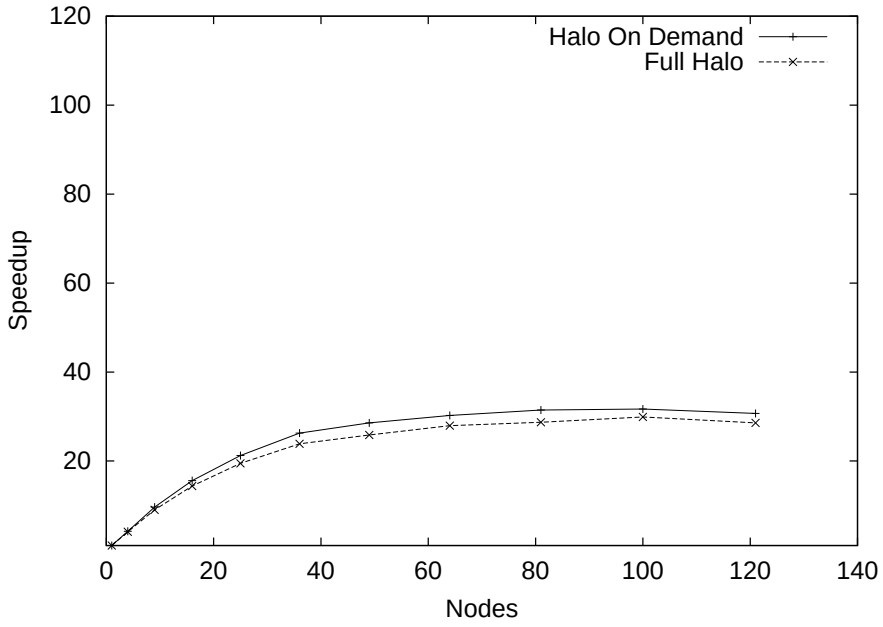


(a)

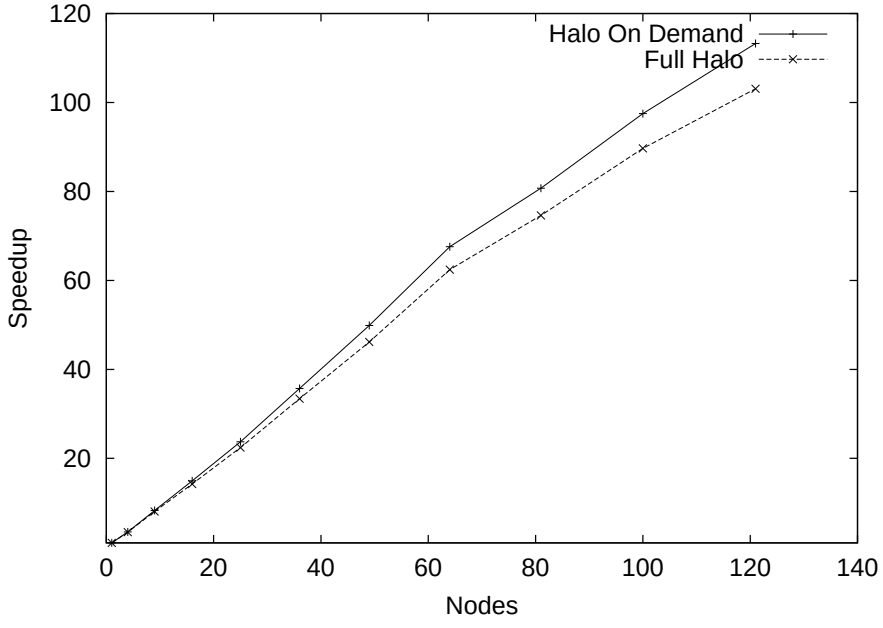


(b)

Figure 7.19: Maximum and minimum computation time per experiment for a future sized input grid for semi-Lagrangian formulations for (a) Full Halo and (b) Halo on Demand. For every permutation of the number of processors we calculate the average of all processors for that number of processors and denote this as “Average over all processors”.



(a)



(b)

Figure 7.20: Speedup for the Halo on Demand strategy and the Full Halo method for semi-Lagrangian formulations using (a) a current sized and (b) a future sized input grid.

7.5.7 Discussion on communication time

When analyzing the communication time in figures 7.10 and 7.15, one can observe an interesting paradox: although the data sent per processor decreases, the communication time stays constant (figures 7.15a and 7.15b) or increases (figures 7.10a and 7.10b) with an increasing number of processors. There are a number of reasons for this behavior:

Synchronization Although all communication is done asynchronously and non-blocking, some kind of synchronization is required. A process can only continue computation after it has received all its data. There will always be one processor the slowest and all his neighbors have to wait for it. Since we have to send several blocks of data, like a domino effect, this synchronization will affect all processors after a number of data exchanges. Because of this, communication time will not decrease linearly if we send less data per processor using an increasing number of processors.

Bandwidth Several performance studies on Myrinet have shown that with a decreasing message size, the effective bandwidth of the Myrinet network decreases. For example Myricom performed extensive benchmarking [52] on the Myrinet-2000 network and found that bandwidth can become near zero for very small messages and almost 400 MB/s for large messages. Similar results were found by Bhoudjang *et al.* [6] on a predecessor of `[DAS2]`, the `[DAS1]`,which used an older version of the Myrinet network.

7.6 Conclusion

The use of semi-Lagrangian formulations can increase performance of NWPs. However, implementing these formulations by hand can be an error prone job. For finding the displacement vector an iterative process is used and with each step several interpolations have to be performed. In this chapter we have described how we adapted the CTADEL system in order to automaticly generate code for semi-Lagrangian formulations. We introduced a “plugable” system for interpolations, where the user can either provide his (dimensionless) interpolation methods or use a number of predefined methods, available in the system. Furthermore, we introduced a `block` operator in order to perform iterative operations. With this operator we can define a set of equations which

are continuously carried out until a convergence condition holds. The generated code from CTADEL can compete with the hand-written reference code in execution time.

On parallel architectures communication can become a huge bottleneck limiting speedup. Since the communication pattern is dependent on the input data and can change during execution of the model, we cannot optimize this at compile time. In this chapter we have analyzed timings of the execution of the iterative search for a displacement vector for the semi-Lagrangian formulation for a convection model on a distributed memory parallel cluster. We have shown that with regular data sets communication time can become the dominant limiting factor for the total execution time on a large number of processors.

In order to optimize execution time, optimizing communication cost will be a necessity. This can be achieved in two ways. First one can try to reduce the influence of communication time by increasing the input data size and thus the computation time. We have shown that, as expected, increasing the input data size with a factor of 16, relatively less time is spent in communication and the speedup becomes nearly linear. However, also with a bigger input data set there will be an upper bound on speedup due to communication time becoming a dominantly limiting factor when using a large number of processors.

Therefore, we have developed a strategy, called Halo on Demand, to reduce communication. With Halo on Demand, a dynamic data driven strategy, we are looking at the displacement vector to predict the wind-direction. Using this run-time analysis we are able to significantly cut back the amount of transfered data. This results in a decrease in communication time and an increase in speedup. At this moment, the Halo on Demand is programmed by hand. Ctadel should be adapted to generate this automatically, to avoid the substantial manual programming effort now needed.

Chapter 8

Conclusions

Many attempts have been made to solve scientific or physical models numerically using computers. In this thesis we have shown a brief overview of the CTADEL system. Furthermore, we showed a number of example applications for CTADEL and the extensions we implemented in the system.

The CTADEL system provides an automated means of generating specific high performance scientific codes. These codes are optimized for a number of different architectures, like serial, vector, or shared virtual memory and distributed memory parallel computer architectures. One of the key elements of this system is the usage of algebraic transformation techniques and powerful methods for global common subexpression elimination. These techniques ensure the generation of efficient high performance codes.

The system consists of several modules of which we described four in more detail. First, the ATMOL specification language, a high-level language that provides a means for the specification of a PDE-based problem in a natural way. Second, the GPAS reduction system, one of the main components used for the symbolic manipulation of algebraic expressions. Third, DICE, the common subexpression elimination. And fourth, the back-end code generator.

In chapters 4, 5 and 6 we showed a number of example applications. We specified the models, let CTADEL generate code for it and compared the performance of these codes with the hand-written versions of the applications. Each of these applications were picked to show a particular strength of CTADEL or to show a novel implementation for methods in CTADEL. For example, in the turbulence model we let the specification

call external functions from a library. In this model we have also implemented an experimental implementation for a certain implicit differential equation. In another example, the coupled ocean–atmosphere model, we show how CTADEL can deal with models with different time-steps for the submodels. In the third example, CTADEL showed its strength with conditional expressions; a number of trigger functions to determine entrainment and detrainment that take place in a cloud.

Although CTADEL is mainly targeted towards weather forecasting, we found no significant problem applying automatic code generation for an ocean model. We, therefore, strongly believe that CTADEL can also be applied in other important application domains, for example the domain of fluid dynamics .

Several formulations for solving semi-implicit equations exist, for example Eulerian and (semi-)Lagrangian formulations. Because of its simplicity, CTADEL used Eulerian schemes for solving these equations. However, Lagrangian-type formulations have a number of interesting properties in comparison with Eulerian-type formulations, for example a possible increase in the time step size. Use of Lagrangian formulations increases complexity and thus poses a new challenge to CTADEL. In chapter 7 we showed some of the theory behind semi-Lagrangian formulations. We showed the implementation of these formulations in CTADEL and how a specification of a model can use these formulations. We compared the performance of the automatically generated code and hand-written code and we observed an interesting aspect with execution on a distributed memory parallel cluster. With an increasing number of computing nodes, communication time between nodes became larger than computation time. We analyzed the communication patterns and designed a novel communication method, called Halo On Demand. With this method we could decrease communication time and, thus, increase overal performance.

Bibliography

[1] Robert L. Akers, Elaine Kant, Curtis J. Randall, Stanley Steinberg, and Robert L. Young. SciNaps: A problem-solving environment for partial differential equations. *IEEE Computation Science and Engineering*, 4(3):32–42, July-September 1997.

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] American National Standards Institute. *Programming Languages C++, ISO 14882*, 1998.

[4] D.F. Bacon, S.L. Graham, and O.L. Sharp. Compiler transformations for high-performance computing. *ACM Computings Surveys*, 26(4):345–420, December 1994.

[5] Siegfried Benkner, Kamran Sanjari, Viera Sipkova, and Bob Velkov. Parallelizing irregular applications with the vienna hpf+ compiler vfc. In *Proceedings HPCN'98 (HPF+ workshop)*, LNCS 1401, pages 797–808, Amsterdam, The Netherlands, April 1998. Springer Verlag.

[6] Raoul A.F. Bhoudjang, Kees Verstoep, Tim Rühl, Henri E. Bal, and Rutger F.H. Hofman. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 71–81, Cambridge, MA, USA, November 2000. ACM Press.

[7] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, 2002.

[8] A.J.C. Bik and H.A.G. Wijshoff. Mt1: A prototype restructuring compiler. Technical Report 93-32, Department of Computer Science, Leiden University, 1993.

[9] Brian Blount and Siddhartha Chatterjee. An evaluation of java for numerical computing. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 35–46. Springer-Verlag, 1998.

[10] OpenMP Architecture Review Board. Openmp: Simple, portable, scalable smp programming. http://www.openmp.org/.

[11] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, 1994.

[12] J. Cuxart, P. Bougeault, and J.L. Redelsperger. A turbulence scheme allowing for mesoscale and large-eddy simulations. *Querterly Journal of the Royal Meteorological Society*, Vol. 126(No. 562):1–30, January 2000.

[13] G. Cybenko. Dynamic load balancing in distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[15] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *The Journal of Supercomputing*, 23(1):23–37, 2002.

[16] L. DeRose, K. Gallivan, E. Gallopoulos, B. Marsolf, and D. Padua. Falcon: A matlab interactive restructuring compiler. In *8th International Workshop, LCPC'95*, Languages and Compilers for Parallel Computing, pages 269–288, Columbus OH, USA, Aug. 1995. Springer Verlag.

[17] William K. Dewar. Quasigeostrophic climate dynamics. *Journal of Marine Research*, Submitted.

[18] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[19] Paul F. Dubois. Designing scientific components. *IEEE Computing in Science and Engineering*, 4(5):84–50, September/October 2002.

[20] E. Kållén (editor). *Hirlam Documentation Manual System 2.5*, June 1996.

[21] Ian Foster (editor) and Carl Kesselman (editor). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers inc., 1999.

[22] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'c: A language for high-level, efficient, and machine-independent dynamic code generation. In *Symposium on Principles of Programming Languages*, pages 131–144, 1996.

[23] Message Passing Interface Forum. MPI: A message-passing interface standard. Technical Report UT-CS-94-230, University of Tennessee, 1994.

[24] Geoffry Fox. Software development around a millisecond. *IEEE Computing in Science and Engineering*, 6(2):93–96, March/April 2004.

[25] Guang R. Gao, Russ Olsen, Vivek Sarkar, and Radhika Thekkath. Collective loop fusion for array contraction. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, pages 281–295, New Haven, Conn., 1992. Springer Verlag.

[26] Samual Z. Guyer and Calvin Lin. Broadway: A software architecture for scientific computing. In R.F. Boisvert and P.T.P. Tang, editors, *The Architecture of Scientific Software*. Kluwer Academic Press, 2000.

[27] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis: A basis for parallelization, optimization and scheduling of programs. In *Workshop on Languages and Compilers for Parallel Computing*, pages 567–585, Portland, OR, USA, 1993. Springer Verlag.

[28] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[29] William L. Hase, Kihyong Song, and Mark S. Gordon. Direct dynamics simulations. *IEEE Computing in Science and Engineering*, 5(4):36–44, July/August 2003.

[30] W. Holland. The role of mesoscale eddies in the general circulation of the ocean: numerical experiments using a wind-driven quasi-geostrophic model. *Journal of Physical Oceanography*, 8:363–392, 1978.

[31] DAS-2: The Distributed Asci Supercomputer home page. http://www.cs.vu.nl/das2/.

[32] Hirlam home page. http://www.knmi.nl/hirlam.

[33] The Distributed Asci Supercomputer home page. DAS-1: . http://www.asci.tudelft.nl/das/das.shtml.

[34] E. Houstis, T. Papatheodorou, and C. Polychronopoulos. Advanced loop optimizations for parallel computers. In *Proceedings of the First International Conference on Supercomputing*, pages 255–277, New York, USA, 1987. Springer-Verlag.

[35] Elias Houstis, Efstratios Gallopoulos, Randall Bramley, and John Rice. Problem-solving environments for computational science. *IEEE Computational Science and Engineering*, 4(3):18–21, July-September 1997.

[36] High performance fortran forum (hpff). http://www.crpc.rice.edu/HPFF/.

[37] E. Isaacson and H.B. Keller. *Analysis of Numerical Methods*. Dover Publications, 1966.

[38] C.R. Johnson, R.S. MacLeod, S.G. Parker, and D.M. Weinstein. Biomedical computing and visualization software environments. *Communications of the ACM*, 47(11):64–71, 2004.

[39] Leo P. Kadanoff. Excellence in computer simulation. *IEEE Computing in Science and Engineering*, 6(2):57–67, March/April 2004.

[40] John S. Kain and J. Michael Fritsh. A one-dimensional entraining/detraining plume model and its application in convective parameterization. *Journal of the Atmospheric Sciences*, 47(23):2784–2802, 1990.

[41] Norbert Kajler and Neil Soiffer. A survey of user interfaces for computer algebra systems. *Journal of Symbolic Computation*, 25(2):127–159, 1998.

[42] Ken Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *Proceedings of International Parallel and Distributed Processing Symposium 2000 (IPDPS 2000)*, pages 297–304, Cancun, Mexico, May 2000.

[43] Oak Ridge National Laboratory. Pvm: Parallel virtual machine. http://www.csm.ornl.gov/pvm/pvm_home.html.

[44] M. Lam. Software pipelining: an effective scheduling technique for vliw machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 318–328, Atlanta, Georgia, United States, 1998. ACM Press.

[45] Rubin H. Landau. Future scientific digital documents with mathml, xml, and svg. *IEEE Computing in Science and Engineering*, 4(2):77–85, March/April 2002.

[46] Maplesoft. Maplesoft webpage. http://www.maplesoft.com/.

[47] Carl McConnell and Ralph E. Johnson. Using static single assignment form in a code optimizer. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(2):152–160, 1992.

[48] John Mellor-Crummey and Vikram Adve. Simplifying control flow in compiler-generated parallel code. In *Proceedings of the Tenth International Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, 1997. Springer-Verlag.

[49] Message Passing Interface Forum MPIF. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, 1996.

[50] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers inc., 1997.

[51] Myricom. http://www.myricom.com.

[52] Performance of MPICH-GM 1.2.4..8a on myrinet-2000. http://www.myricom.com/myrinet/performance/MPICH-GM/index.html.

[53] NetLib. Linpack website. http://www.netlib.org/linpack/.

[54] Eric Noulard and Nahid Emad. A key for reusable parallel linear algebra software. *Parallel Computing*, 27(10):1299–1319, 2001.

[55] MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[56] The University of Texas. Plapack: Parallel linear algebra package. http://www.cs.utexas.edu/users/plapack/.

[57] University of Utah. Scientic computing and imaging institute. http://software.sci.utah.edu/scirun.html.

[58] Peter Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.

[59] LAPACK Linear Algebra PACKage. http://www.netlib.org/lapack/.

[60] C. D. Polychronopolous and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[61] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in Fortran 77*. Cambridge University Press, 1992.

[62] The Beowulf project. http://www.beowulf.org.

[63] Problem solving environments home page. http://www.cs.purdue.edu/research/cse/pses.

[64] J. Pudykiewicz and A. Staniforth. Some properties and comparative performance of the semi-langrangian method of robert in the solution of the advection-diffusion equation. *Atmosphere-Ocean*, 22(3):282–308, 1984.

[65] M. J. Quinn. *Parallel Computing, theory and practice*. McGraw-Hill, 1994.

[66] J.L. Redelsperger and G. Sommeria. D'echelle inferieure a la maille pour un modele tri-dimensionnel de convection nuageuse. *Boundary Layer Meteorology*, Vol. 21:509–530, 1981.

[67] John Reid. The future of fortran. *IEEE Computing in Science and Engineering*, 5(4):59–67, July/August 2003.

[68] John R. Rice and Ronald F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Comptational Science and Engineering*, 3(3):44–53, Fall 1996.

[69] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[70] H. Schmidt and U. Schumann. Coherent Structure of the Convective Boundary Layer Derived from Large-Eddy Simulations. *Journal of Fluid Mech.*, Vol. 200:511–562, 1989.

[71] SciSoft. Pricing derivatives without programming. http://www.scicomp.com/solution/scifinance.

[72] Andrew Staniforth and Jean Côté. Semi-langrangian integration schemes for atmospheric models – a review. *Monthly Weather Review*, 119:2206–2223, September 1991.

[73] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[74] R. Barrett *et al. Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[75] Walter F. Tichy. Programming-in-the-large: past, present, and future. In *Proceedings of the 14th international conference on Software engineering*, pages 362–367. ACM Press, 1992.

[76] Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal Of Functional Programming*, 8(4):367–412, 1998.

[77] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, S. Margherita Ligure, Italy, 1995. ACM Press.

[78] Paul van der Mark, Gerard Cats, and Lex Wolters. Automatic code generation for a turbulence scheme. In *Proceedings of the 15^{th} International Conference of Supercomputing*, pages 252–259, Sorrento, Italy, June 2001. ACM.

[79] Paul van der Mark, Robert van Engelen, Kyle Gallivan, and W. Dewar. A case study for automatic code generation on a coupled ocean–atmosphere model. In *Proceedings of the Second International Conference on Computational Science*, volume LNCS 2329 of *Lecture notes in computer science*, pages 419–428, Amsterdam, The Netherlands, April 2002. Springer-Verlag.

[80] Paul van der Mark, Lex Wolters, and Gerard Cats. Automatic code generation for a convection scheme. In *Proceedings of the 18^{th} ACM Symposium on Applied Computing*, pages 1003–1008, Melbourne, Florida, USA, March 2003. ACM press.

[81] Paul van der Mark, Lex Wolters, and Gerard Cats. Automatic code-generation for large scale applications. In *Proceedings of the Tenth International Workshop on Compilers for Parallel Computers (CPC03)*, pages 157–164, Amsterdam, Netherlands, January 2003.

[82] Paul van der Mark, Lex Wolters, and Gerard Cats. Code generation for semi-lagrangian formulation. In *Proceedings of the Second International Symposium on Parallel and Distributed Computing (ISPDC'03)*, pages 266–273, Ljubljana, Slovenia, October 2003. IEEE Computer Society Press.

[83] Paul van der Mark, Lex Wolters, and Gerard Cats. A dynamic application-driven data communication strategy. In *Proceedings of the 18th ACM International Conference on Supercomputing*, pages 146–153, Saint-Malo, France, June 2004. ACM press.

[84] Paul van der Mark, Lex Wolters, and Gerard Cats. Optimizing data communication for semi-lagrangian formulations on a distributed memory cluster. In *Proceedings of the tenth ASCI conference*, pages 381–388, Ouddorp, The Netherlands, June 2004.

[85] Paul van der Mark, Lex Wolters, and Gerard Cats. Semi-lagrangian formulations with automatic code generation for environmental modeling. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC'04)*, pages 229–234, Nicosia, Cyprus, March 2004. ACM Press.

[86] R.A. van Engelen. *Ctadel: A Generator of Efficient Numerical Codes*. PhD thesis, Universiteit Leiden, 1998.

[87] Robert van Engelen, Lex Wolters, and Gerard Cats. Tomorrow's weather forecast: Productive program generation in atmospheric modeling. *IEEE Computational Science and Engineering*, 4(3):22–31, 1997.

[88] Robert A. van Engelen. Atmol: A domain-specific language for atmospheric modeling. *Journal of Computing and Information Technology*, 9(4):289–303, 2001.

[89] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing (CD-ROM Proceedings)*, San Antonio, Texas, USA, March 1998.

[90] K.G. Wilson. Grand challenges to computational science. *Future Generation Computer Systems*, 5(2-3):171–189, 1989.

Samenvatting (in Dutch)

Bij het ontwerpen en programmeren van software voor grootschalige rekenmodellen, stuit de ontwikkelaar vaak op een aantal praktische problemen. Bijvoorbeeld, bij het omzetten van het model naar een daadwerkelijk computerprogramma, zal rekening moeten worden gehouden met het type architectuur waar de simulatie op zal worden gedraaid. Zo zal een programma geschreven voor een vector architectuur in het algemeen niet optimaal werken op een cluster van conventionele seriële computers en vice versa.

Het optimaliseren van een programma voor een bepaald type architectuur is een arbeidsintensieve taak. Bovendien zijn computer architecturen alweer verouderd binnen een jaar dat ze op de markt verschijnen en nieuwe architecturen kunnen nieuwe eisen aan het programma opleggen waardoor het optimaliseren weer van voren kan beginnen.

Een ander probleem treedt op wanneer men het onderliggende rekenmodel wil aanpassen. Vaak is de software die het model simuleert, niet goed gedocumenteerd of is door het optimaliseren de relatie met het model niet goed meer te herkennen en is aanpassen van de software geen triviale taak geworden. Hierdoor kan het makkelijk gebeuren dat er fouten in de software sluipen.

Een mogelijke oplossing voor deze problemen is om het model te simuleren door middel van het direct uitvoeren van een specificatie van het model. Een voorbeeld hiervan is het programma Matlab. De gebruiker specificeert de wiskundige vergelijkingen van het model in een Matlab-specifieke hoger-niveau specificatietaal waarna het programma deze uitrekent. De gebruiker hoeft geen rekening te houden met het type computer architectuur en een verandering van het model kan snel doorgevoerd worden in de specificatie. Nadelen van deze methode zijn dat de vergelijkingen niet geoptimaliseerd worden of dat er niet gebruik wordt gemaakt van architectuur-specifieke hardware en het model dus sub-optimaal wordt berekend.

In dit proefschrift wordt een andere oplossing besproken, de CTADEL compiler. Net

als bij Matlab specificeert de gebruiker zijn model in een hoger-niveau specificatietaal waarna deze wordt vertaald naar een efficiënte hogere programmeertaal. De gebruiker kan het type computer architectuur aangeven waarvoor code moet worden gegenereerd. De kracht van dit systeem hangt onder andere af van zijn kennis van numerieke methodes en in dit proefschrift worden een aantal uitbreidingen van geavanceerde numerieke methodes in het CTADEL systeem besproken en gedemonstreerd.

De door CTADEL gegenereerde taal voor ieder model hebben we op diverse soorten computersystemen, zoals beschreven in hoofdstuk 3, getest op correctheid en prestatie. We vergelijken deze programma's met een handgeschreven versie van het model. In bijna alle gevallen blijkt dat de code gegenereerd door CTADEL een vergelijkbare of zelfs betere prestatie (sneller) geeft.

In hoofdstuk 2 geven we een overzicht van het CTADEL systeem en de belangrijkste stappen van het vertaal proces van specificatietaal, genaamd ATMOL naar efficiënte hogere programmeertaal. De kracht van het systeem wordt grotendeels bepaald door het herschrijven van wiskundige specificaties. Hierbij wordt het stelsel van vergelijkingen vereenvoudigd, waarbij wordt gekeken of bepaalde berekeningen op een efficiëntere manier kunnen worden uitgevoerd en of een uitdrukking niet meerdere keren wordt berekend.

In de hoofdstukken 4, 5 en 6 laten we een aantal uitbreidingen van CTADEL systeem zien. Voor iedere uitbreiding bekijken we de praktische toepassing hiervan met een voorbeeld model. In hoofdstuk 4 beschrijven we een sjabloon systeem en hoe dit kan worden toegepast bij het modelleren van een convectie model en hoe CTADEL hiervoor automatisch code voor genereert. In hoofdstukken 5 en 6 laten we zien hoe we CTADEL hebben aangepast, zodat er gebruik kan worden gemaakt van (numerieke) functies in externe bibliotheken van functies. In hoofdstuk 5 beschrijven we dit aan de hand van een gemengd oceaan/atmosfeer model, terwijl we in hoofdstuk 6 de specificatie en het genereren van een turbulentie model laten zien. Dit model maakt gebruik van een bepaald type impliciete vergelijkingen en we laten zien hoe CTADEL code kan genereren voor dit soort vergelijkingen.

Veel gebruikte methodes bij het oplossen van vergelijkingen binnen numerieke weersverwachtingsmodellen zijn de Euleriaanse en de (semi-)Langrangiaanse methodes. Vanwege de eenvoud van specificatie, was de Euleriaanse methode de standaard oplosmethode in CTADEL. In hoofdstuk 7 laten we zien hoe we CTADEL hebben aangepast om de semi-Langrangiaanse oplosmethode te gebruiken. We laten zien hoe een gebruiker zijn model

moet specificeren om deze methode te gebruiken. Verder gaan we in op een nieuwe door ons ontwikkelde methode om de communicatie tussen processoren op een parallel systeem te optimaliseren. Dit is een dynamische methode die gestuurd wordt door de toepassing.

Curriculum Vitæ (in Dutch)

Paul van der Mark werd geboren op 6 november 1970 te Gouda. Hij volgde de Atheneum opleiding aan de Christelijk Scholen Gemeenschap te Emmeloord. Hier haalde hij in 1990 met goed gevolg zijn VWO diploma. Hierna werkte hij onder andere als programmeur bij een ingenieursbureau te Enschede en als richter op een stuk bij de elfde afdeling rijdende artillerie te Arnhem. In 1995 begon hij zijn studie informatica aan de Universiteit Leiden, waar zijn interesse werd gewekt voor compiler technologieën. Tijdens zijn afstudeerstage bij het onderzoeksinstituut IRISA te Rennes, Frankrijk, hield hij zich bezig met een nieuwe compilatietechniek, *iterative compilation*. Hier werd ook zijn interesse voor onderzoek geprikkeld en na het slagen voor zijn doctoraal examen in 2000 trad hij in dienst van de Universiteit Leiden als tweede-geldstroom AIO. Gedurende vier jaar verrichtte hij promotieonderzoek onder begeleiding van dr. A. A. Wolters en promotor prof. dr. H. A. G. Wijshoff naar het automatisch genereren van code voor programma's voor het oplossen van grootschalige numerieke problemen. Dit onderzoek is in nauwe samenwerking met drs. G. J. Cats van het KNMI gedaan. Verdere samenwerking vond plaats met dr. R. A. van Engelen van de Florida State University, Tallahassee, Florida, USA. Deze samenwerking leidde in 2001 tot een bezoek aan de Florida State University.

Paul van der Mark is sinds january 2005 in dienst als een "post-doctoral researcher" aan de Florida State University bij de computional evolutionary biology group van de school of computational science.

Acknowledgments

This dissertation would not be possible without the help, advice and support of the following people: Gerard Cats, José-Antonio Garcia-Moya, Robert van Engelen, Kyle Gallivan, Bill Dewar, Elizabeth Kelsey, Jolanda Bos and my parents.