# CTADEL: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications

*Robert van Engelen*[*†] *& Lex Wolters*[†‡]
High Performance Computing Division
Dept. of Computer Science, Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands
{robert,llexx}@cs.leidenuniv.nl

*Gerard Cats*[‡]
Royal Netherlands Meteorological Institute
P.O. Box 201, 3730 AE De Bilt, The Netherlands
cats@knmi.nl

**Abstract**

The CTADEL system provides an automated means of generating specific high performance scientific codes optimized for serial, vector, or shared virtual memory and distributed memory parallel computer architectures. One of the key elements of this system is the employment of algebraic simplification techniques and powerful methods for global common subexpression elimination to guarantee the generation of efficient high performance codes for various target architectures. In this paper we present the CTADEL Code-generation Tool for Applications based on Differential Equations using high-level Language specifications. A prototype implementation has been developed which is limited to explicit finite difference methods as numerical solution method. Performance results of the codes generated with this prototype implementation will be presented for a limited area numerical weather forecast routine on various hardware platforms. These results show that generation of efficient code is well feasible within the presented approach.

## 1 Introduction

Since the early days of computing, the demand of large scale scientific applications on more powerful hardware platforms has been a driving force for the development of advanced software environments including dedicated, machine optimized libraries for scientific computing. As more and more hardware platforms emerge, the adaption of scientific applications codes to each new computer architecture requires significant programming efforts. Some of these efforts can be alleviated by employing restructuring compilers to restructure existing application codes. However, restructuring compilers need relatively 'clean code' without many programming tricks that are often exploited by human programmers. Otherwise, automatic restructuring of a code for a different hardware architecture can be an unsolved problem for those applications developed with a specific target architecture in mind; in such an implementation, part of the knowledge content of the model is lost. This knowledge may prove to be invaluable with respect to an efficient implementation of the model on a different computer architecture.

While code (re)writing by hand is common practice in the field of application development, it will be evident that the (semi-)automatic generation of code directly from a high-level language description of a problem provides a fast, robust, and, therefore, attractive alternative. In this respect, a problem-specific code generator, also known as *application driver*, should translate the problem into near-optimal code for a given target computer architecture.

In this paper the CTADEL Code-generation Tool for Applications based on Differential Equations using high-level Language specifications will be presented. Currently, a prototype system has been developed. This prototype system adopts explicit finite difference methods as numerical solution method and, for parallel architectures, adopts a domain splitting method to decompose the problem into subproblems that can be solved independently on each processor. Furthermore, CTADEL incorporates a global common-subexpression eliminator that acts in concert with an algebraic simplifier to reduce the computational complexity of the problem on a global scale. These techniques provide the necessary means within the CTADEL system to generate high-performance codes for various computer architectures automatically from a high-level language description of a model. We used the prototype CTADEL system as an application driver for the HIRLAM system, an operational limited area numerical weather forecast system [16]. We will present recent results of this prototype system for generating codes for the HIRLAM system.

**Related work.** Many attempts have been made to solve scientific or physical models numerically or symbolically using computers. Today, a large collection of libraries, tools, and Problem Solving Environments (PSEs) have been developed for solving such problems. The current and future role of these environments is discussed in [12]. Well-known PSEs for solving PDEs, see [11] for an extensive overview, are ELLPACK [19] and DEQSOL [22], and in the field of parallel computing //ELLPACK [15]. The computational kernels of these PSEs consist of a large library containing routines for many numerical solution methods. These routines form the templates for the resulting code. The numerical knowledge of such systems is determined by the power of its library.

*Support was provided by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO) under Project No. 612-17-120.

†Support was provided by the Esprit Agency EC-DGIII under Grant No. APPARC 6634 BRA III.

‡Support was provided through the Human Capital and Mobility Programme of the European Community.


**ICS'96, Philadelphia, PA,USA**
**© 1996 ACM 0-89791-803-7/96/05..$3.50**

A different approach to library-based PSEs consists of a collection of tools that generate code based on a problem specification without the use of a library. The numerical knowledge is determined by the expressiveness of the problem specification language and the underlying translation techniques; the system employs a set of refinement steps to the problem by which the algorithms and data structures are refined from an abstract level into a concrete implementation. Examples are the ALPAL system [7], the SINAPSE system [17], the SUSPENSE transformation system [20], and the DPML Data Parallel scientific Modelling Language [10]. The CTADEL system should also be considered as a system of the last type. However, a novel approach is taken in which the description of the problem is transformed into code by taking target computer architecture characteristics into account. This way, the main objective of the CTADEL system, the generation of efficient codes, can be achieved.

The advantage of this approach is that by exploiting specific hardware characteristics of the target computer architecture there is a complete absence of portability and code-consistency related problems. For each machine, an efficient hardware-specific version of the code can be generated. All versions of the code will be consistent with each other and can incorporate the latest improvements to the model or problem specification. This clearly motivates the use of such a tool with respect to performance critical applications.

This paper is organized as follows. In Section 2 an overview of the CTADEL system will be presented. Section 3 briefly describes finite difference methods, which are employed by CTADEL to solve a model numerically. In Section 4, the high-level specification language is discussed. Section 5 presents the optimization techniques employed for optimizing intermediate code of the numerical solver and Section 6 discusses the generation of high performance codes for various computer architectures from the intermediate code. Section 7 presents results of the prototype implementation for a limited area numerical weather forecast system. To conclude, in Section 8, we summarize our results and conclusions.

## 2 Overview of the CTADEL System

In this section an outline of the CTADEL system will be presented. More details can be found in [9]. The CTADEL system takes a high-level language description of a scientific model comprising a set of coupled partial differential equations, discretizes these equations using finite difference methods, and generates code optimized for a selected target architecture for efficiently solving the model numerically. An overview of the CTADEL system is depicted in Figure 1. The core of the CTADEL system (hashed box in Figure 1) contains several interacting modules:

**Finite difference discretizer.** The finite difference discretizer discretizes the partial differential equations of the model using predefined and user-defined discrete operators. More details will be given in Section 4.2.

**General Purpose Algebraic Simplifier.** GPAS interacts with the finite difference discretizer to rewrite the discrete equations into a more compact form using axiomatic properties of the applied operators. A more detailed description will be given in Section 5.1.

**Domain-shift Independent Common-subexpression Eliminator.** DICE interacts with GPAS to reduce the computational complexity of the problem as much as possible on a global scale. Some hardware characteristics of the target architecture are taken into account for the reduction in complexity. The result is a simplified intermediate representation of code. More details on DICE can be found in Section 5.2.

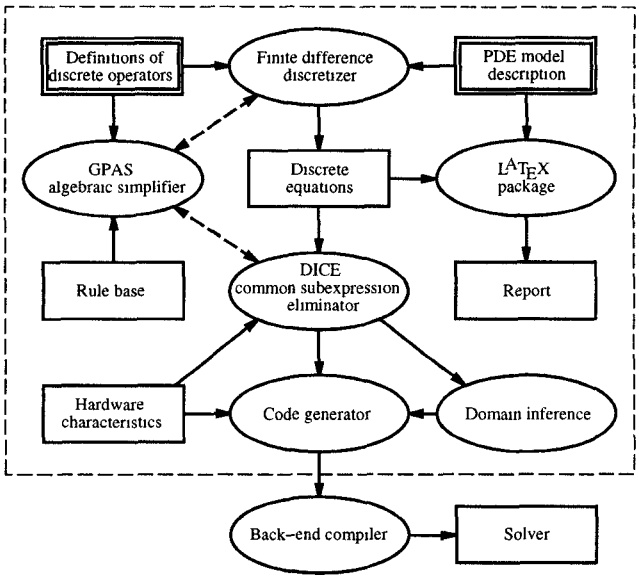


Figure 1: CTADEL system overview (double boxes denote user input, dashed arrows denote module interactions).

**Domain inference.** The bounds of the computational domain of each array variable to be used in the final code is determined from the intermediate code.

**Code generator.** The code generator generates a sequence of statements that is optimal for loop-fusion. In addition, a domain splitting method is employed for parallel architectures to split the global computational domain into (mutually overlapping) sub-domains based on the results of the domain inference. Finally, code is generated for a specific computer architecture. We will give an overview of the types of codes generated in Section 6.

**LaTeX package.** The LaTeX package takes the scientific model and its discretized equations to generate a report. This report serves as a means for the user to verify CTADEL's (semi-) automatic discretization.

Currently a prototype implementation of the CTADEL system has been developed using the public domain SWI-Prolog package [23].

## 3 Finite Difference Methods

In this section a brief introduction to finite difference methods and staggered grids will be given. For a more detailed discussion the reader is referred to a textbook, e.g. [2].

### 3.1 Partial Differential Equations

A scientific model can be written generally as a set of $n$ partial differential equations in the form

$$\frac{\partial}{\partial t}\mathcal{L}_i(u_i) = F_i(u_1, \ldots, u_n), \qquad i = 1, \ldots, n, \tag{1}$$

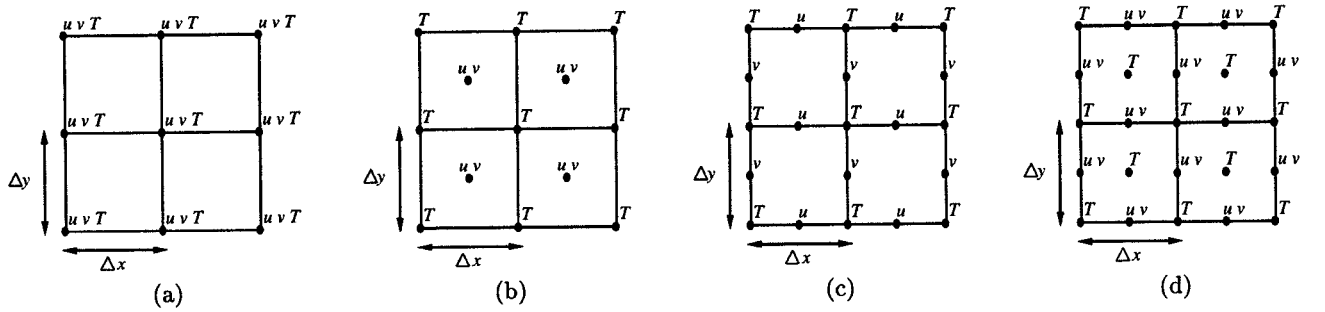


Figure 2: Arakawa A-grid (a), B-grid (b), C-grid (c), and E-grid (d).

together with a set of *initial conditions* and a set of *boundary conditions* which are said to hold on the boundary $\partial\Omega$ of the domain $\Omega \subseteq \mathbb{R}^d$. Here, $u_i = u_i(\mathbf{x}, t)$, $i = 1, \ldots, n$, are scalar functions of the space coordinates given by $\mathbf{x} \in \Omega$, and time $t$, called the *dependent variables* of the problem. The space coordinates $\mathbf{x}$ and time $t$ are called the *independent variables* of the problem. In this paper we will use the phrase variable and field interchangeably. $F_i$ is a function involving the $u_i$ as well as their space and time derivatives and $\mathcal{L}_i$ is a space differential operator which in many cases is the identity operator, i.e. $\mathcal{L}_i(u_i) = u_i$, or the Laplacian operator, i.e. $\mathcal{L}_i = \Delta = \nabla^2$.

Specific to finite difference methods is the approximation of partial derivatives by finite difference schemes. These schemes are based on difference quotients. For example, the partial derivative of a variable $u$ with respect to $x$ can be approximated using a central difference:

$$\frac{\partial u}{\partial x} \approx \frac{u(x_{i+1}, y_j) - u(x_{i-1}, y_j)}{x_{i+1} - x_{i-1}}, \tag{2}$$

where an $n \times m$ grid of points $(x_i, y_j) \in \Omega$, $i = 1, \ldots, n$, $j = 1, \ldots, m$, is used to discretize the domain.

### 3.2 Staggered Grids

An important consideration in the choice of finite difference schemes is the arrangement of variables on a grid. *Staggered grids* can be used to avoid producing separate solutions on alternating grid-points. This problem occurs when partial derivatives are approximated by centered difference quotients. Consider for example the approximation given by Equation (2). The approximation of the derivative of variable $u$ with respect to $x$ is determined by the values $u(x_{i+1}, y_j)$ and $u(x_{i-1}, y_j)$ at grid-points $(x_{i+1}, y_j)$ and $(x_{i-1}, y_j)$ respectively, and is thereby completely decoupled from the value $u(x_i, y_j)$ at grid-point $(x_i, y_j)$. This decoupling results in two separate numerical solutions for the problem which is clearly undesirable. This problem can be solved by rearranging the variables on a grid. Various arrangements of variables in the horizontal domain were classified by Arakawa [3]. Some typical Arakawa grids are depicted in Figure 2. Values that are required in the interval between grid-points are obtained using interpolation methods.

## 4 Problem Specification

In this section the declarative high-level language of CTADEL is described. The objective of this language is to specify a one-, two-, or three-dimensional PDE-problem in a natural way, close to the mathematical formulation of the problem in vector notation.

### 4.1 Language Features

The high-level language components are illustrated by means of a simple example. This example describes the advection of temperature in an incompressible medium in two-dimensional geometry discretized on a uniform C-grid (see Figure 2c):

```
grid i = 1:n, j = 1:m.      % declare an n by m grid
def hx = 1/(n-1).           % grid-point x distance
def hy = 1/(m-1).           % grid-point y distance
field u(x,y): stag(1,0).    % u is staggered in x
field v(x,y): stag(0,1).    % v is staggered in y
vector w = [u,v].           % fluid flow vector
field t(x,y).               % temperature
field dTdt(x,y).            % temperature derivative
div([A,B]) := dx A + dy B. % define 2D divergence
dTdt = -div(w*t).           % eq. for advection
latex t       = "T".
latex dTdt    = "\frac{\partial T}{\partial t}".
latex div(Vec) = ["\nabla\cdot", Vec].
```

A logically rectangular grid is defined using a 'grid' declaration which defines the iteration indices and bounds for the computational domain of the *derived* (output) variable 'dTdt' as opposed to the *fundamental* (input) variables 'u', 'v', and 't' whose domains are to be determined from their use. The distances between the grid-points are defined by the 'def' statements. Each field of the problem is declared using a 'field' declaration stating the dependency of the field on the spatial coordinates and its type of staggering. This implicitly means that a staggered grid is used through the use of the 'stag' type with the parameters '0' denoting non-staggering and '1' denoting staggering with respect to $x$, $y$, or $z$. Vectors are written using brackets. By using this notation the divergence of a vector with two coefficients can be defined. The only equation in this example states the problem for one time step. An 'if' function (not used in the example above) provides control of execution *within* equations, so boundary conditions can be easily incorporated by e.g. testing on the value of indices. Furthermore, several reduction and scan functions are provided. Finally, the 'latex' declarations are not essential but useful to override the default LaTeX output in order to produce a more readable report. In this example, $\frac{\partial T}{\partial t} = \nabla \cdot (\mathbf{w}T)$ will be printed instead of $dTdt = \mathrm{div}\,(\mathbf{w}\,t)$.

### 4.2 Finite Difference Discretization

The method of *operator overloading* provides a means for defining discrete operators that operate differently on expressions with different type of staggerings. This notion is closely related to the *object oriented* programming style. For example, let $\mathcal{G}_x$ be a type descriptor for expressions defined on whole grid points with respect to the $x$-direction and let $\mathcal{S}_x$ be a type descriptor for expressions defined on staggered grid points with respect to the $x$-direction. Now, the definition of the central difference of a variable depends on the type of staggering of this variable and the type of staggering of the result of this operation:

$$\delta_x(u) = \begin{cases} \delta_x^-(u) = \frac{u_i - u_{i-1}}{\Delta x} & u : \mathcal{S}_x, \delta_x(u) : \mathcal{G}_x \\ \delta_x^+(u) = \frac{u_{i+1} - u_i}{\Delta x} & u : \mathcal{G}_x, \delta_x(u) : \mathcal{S}_x \\ \delta_{2x}(u) = \frac{u_{i+1} - u_{i-1}}{2\,\Delta x} & u, \delta_x(u) : \mathcal{G}_x, \end{cases} \quad (3)$$

where $\Delta x$ denotes the grid-point distance in the $x$-direction, $u_i$ denotes the discretized version of a field $u$, that is, if $u$ is non-staggered with respect to $x$, we have that $u_i = u(x_i)$ on the grid-points $x_i$, $i = 1, \ldots, n$ and if $u$ is staggered with respect to $x$, we have that $u_i = u(x_{i+1/2})$ on the half grid $x_{i+1/2}$, $i = 1, \ldots, n$. The type of staggering of a variable depends on the type of grid (see also Figure 2).

In CTADEL we introduce the concept of a *union* of similar but differently typed discrete operators. Using this concept, the built-in definitions of the central differences corresponding to Equation (3) are:

```
def dx0(Expr: stag(1,Y,Z)): stag(0,Y,Z)
                = (Expr - Expr @ (i-1,j,k))/hx.
def dx1(Expr: stag(0,Y,Z)): stag(1,Y,Z)
                = (Expr @ (i+1,j,k) - Expr)/hx.
def dx2(Expr: stag(0,Y,Z)): stag(0,Y,Z)
    = (Expr @ (i+1,j,k) - Expr @ (i-1,j,k))/(2*hx).
union dx(Expr) = [dx0(Expr), dx1(Expr), dx2(Expr)].
```

Note the use of the parameterized 'stag' type: '0' and '1' denote non-staggering or staggering, respectively. That is, we have stag(0,Y,Z)=$\mathcal{G}_x$ and stag(1,Y,Z)=$\mathcal{S}_x$ for all values of 'Y' and 'Z'. Here, the type parameters 'Y' and 'Z' are variable type parameters which are used in the definitions above to copy the type of staggering with respect to $y$ and $z$ from the type of the argument 'Expr' to the result type of the operator. The grid-point distance $\Delta x$ is denoted by 'hx'.

The definition of the 'union' of operators in the fragment above provides a means for overloading the 'dx' central difference operator. Operator identification then amounts to resolving which of the operators 'dx$_i$', $i = 0, 1, 2$, is to be selected from the union-list for 'dx' to replace the 'dx' operator. If none of the operators listed can replace the 'dx' operator because one or more types of the arguments and result types do not match, an operator is selected from the union list which requires a minimal total number of type conversions. Type converters are applied to the arguments and result. In general, the use of overloaded operators in an expression results in sets of feasible types. Therefore, type inference follows a forward/backward scheme [1]. However, in contrast to most programming languages, we do not restrict the type of an expression to be unique: there are cases in which more than one discrete form of an expression exists. With the approach presented, we select the form which requires the least number of type (i.e. (de)stagger) conversions.

The concept of a union provides a level of abstraction in such a way that e.g. the 'dx' operator can be viewed as taking the partial derivative of an expression with respect to $x$. Exploiting this concept provides the first pass for the discretization of the model equations. The second pass is provided by calling GPAS as will be discussed in the next section. Note, however, that the user is free to use discrete operators e.g. one of the 'dx$_i$', $i = 0, 1, 2$, directly to by-pass the operator selection mechanism if the automatic discretization is not according to his/her preference.

## 5 Optimization

In this section optimization methods employed by CTADEL will be discussed. More details can be found in [9].

### 5.1 GPAS

While a problem can be conveniently specified in a compact form using CTADEL's high-level language, the underlying numerical complexity of the problem can be huge. Much of this complexity can be reduced by exploiting algebraic simplifiers of computer algebra packages such as REDUCE [14], Maple [6], and Mathematica [24]. However, most algebraic simplifiers of these existing computer algebra packages produce expressions in canonical forms which are easy to read but not very economical with respect to arithmetic complexity. Furthermore, a large collection of rewriting rules should be added considering axiomatic laws that describe e.g. the linearity of the implemented discrete operators. Therefore, the decision was made to build a general purpose algebraic simplifier, GPAS, with an extensible rule base.

The role of GPAS is twofold. Firstly, to rewrite the discretization of a model into a compact form, and secondly, to perform simplifications on the intermediate code in collaboration with the global common subexpression eliminator DICE. Considering the latter case, the aim of GPAS is to rewrite expressions into a more efficient form. For example,

$$\sum_{k=1}^{n} \delta_x(\begin{cases} a\,u_{i,j,k} & k < m \\ b & \text{otherwise} \end{cases})$$

is rewritten into

$$\begin{cases} a\,\delta_x(\sum_{k=1}^{n} u_{i,j,k}) & n < m \\ a\,\delta_x(\sum_{k=1}^{m} u_{i,j,k}) & n \geq m > 1 \\ 0 & \text{otherwise.} \end{cases}$$

However, in some cases even the first form may be more desirable depending on the target computer architecture.

### 5.2 DICE

While algebraic simplification can only be performed on expressions locally, the elimination of common subexpressions on a global scale may additionally reduce the computational complexity of the solver for a problem at the cost of introducing temporary variables. Especially the application of finite difference operators inevitably leads to arithmetic expressions involving many sub-terms of the form $u_{i\pm\delta_i, j\pm\delta_j, k\pm\delta_k}$, where $u$ is a three-dimensional discretized field and $\delta_i$, $\delta_j$, $\delta_k \in \mathbb{N}$. The domain-shift independent common-subexpression eliminator, DICE, employs methods to detect subexpressions of common value for which the computational domains are shifted by a constant distance with respect to each other. That is, the subexpressions are basically identical, but the index of the array variables comprising the expression are offset by a (symbolic) constant. In addition, the eliminator will find subexpressions

Table 1: Number of operations per grid-point per time step for the Euler equations compared with different methods for common subexpression elimination (c.s.e.).

| | #unary mins | #add & sub | #mul & div | total arith | #assign |
|---|---|---|---|---|---|
| No c.s.e. | 3 | 49 | 39 | 91 | 5 |
| Pre-c.s.e. | 3 | 43 | 29 | 75 | 31 |
| Post-c.s.e. | 3 | 43 | 34 | 80 | 17 |
| DICE | 3 | 38 | 22 | 63 | 20 |

in common even when defined on rectangular subspaces of the global computational domain. These expressions are characterized by the occurrence of a (symbolic) constant index for some of the dimensions of array variables. For example, consider the partial sum $\sum_{k=k+1}^{n} u_{i,j,k}$ and the sum $\sum_{k=1}^{n} u_{i+1,j,k}$. The latter expression is recognized by DICE as common with the former, but defined on a subspace with $k = 0$ and shifted one grid-point to the right in the $x$-direction. Therefore, DICE creates a new temporary array variable, say $t_{i,j,k} = \sum_{k=k+1}^{n} u_{i,j,k}$, and replaces the latter sum by $t_{i+1,j,0}$.

DICE also interacts with GPAS to minimize function calls to intrinsic and linear operators. For example, for the expression $\log(u_{i,j,k}) - \log(u_{i-1,j,k})$ DICE creates a temporary, say $t_{i,j,k} = \log(u_{i,j,k})$, and replaces the expression with $t_{i,j,k} - t_{i-1,j,k}$. However, if $\log(u)$, and hence $t$, does not appear elsewhere in the model, then DICE may call GPAS to rewrite the expression into the expression $\log(u_{i,j,k}/u_{i-1,j,k})$ since $\log(u_{i,j,k})$ is not needed elsewhere in the code.

In Table 1, several methods for the detection and the elimination of common subexpressions are compared for the time-dependent Euler equations for an inviscid, compressible flow in a two-dimensional geometry discretized on an E-grid with unit grid-point distances. The number of arithmetic operations and the total number of assignments to (temporary) variables per grid-point per time step are shown. In the second method (pre-c.s.e.), common-subexpression elimination is employed on the discretized equations before the discrete operators are expanded. In our experience, this is the strategy often exploited by human programmers. In the third method (post-c.s.e.), common subexpression elimination is employed after expanding the discrete operators of the discretized model equations. The fourth method is the method employed by DICE as described above.

A trade-off between the reduced computational complexity and the incurred additional memory usage plays an important role in the generation of efficient code. On the one hand, the global common subexpression eliminator should be able to detect as many subexpressions as possible. On the other hand, addition of new temporary array variables in the code should be minimized to keep memory usage low. To this end, DICE builds an intermediate representation of the code using a directed acyclic graph (DAG). Each node in the graph represents an aggregate array operation, or *data-parallel* operation. After the DAG is completed, DICE adopts an iterative scheme [1] to compute the fixed-point DAG representation of the code under the removal of nodes by substitution of the array operation in the appropriate expressions. A node is removed based on a heuristic cost criterion specified by a hardware architecture description. Finally, for each remaining node in the DAG a temporary array variable is introduced.

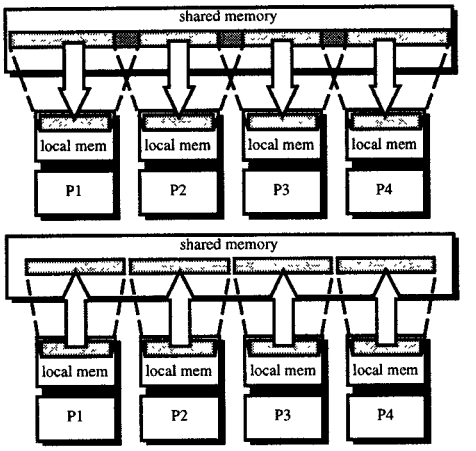


Figure 3: Data traffic for domain splitting methods with shared memory parallel architectures.

## 6 Multi-Platform Code Generation

In this section the generation of high performance codes for various types of hardware platforms will be discussed.

### 6.1 Determination of the Bounds for the Computational Domains

Prior to the generation of code for a specific computer architecture, the bounds for the computational domain of each (temporary) array variable are derived symbolically from the intermediate code by following def/use chains in order to be able to generate declarations and loop bounds for the (temporary) arrays used in the code to solve the problem. Furthermore, the domain splitting method for parallel architectures requires an additional phase for inference of the bounds for the sub-domains.

### 6.2 Target Architectures

**Serial architectures.** Fortran 77 code is generated for serial architectures.

**Vector architectures.** The code generated for vector architectures consists of Fortran 77 code with loops structured in such a way that vector operations can be optimized by the back-end compiler.

**Data parallel systems.** For the data-parallel code, Fortran 90 code is generated including HPF '`FORALL`' statements.

**Shared virtual memory parallel systems.** For shared memory parallel computers where each processor is equipped with local memory, a SPMD Fortran 77 code is generated using a domain splitting method to split the global domain into sub-domains. Domain splitting methods are widely used for the parallelization of scientific models, see e.g. [4]. The input for each solver for a sub-domain is copied from shared to local memory. In general, the sub-domains have overlapping regions. By allowing overlapping regions, each subdivided problem can be solved independently using local memory only, thereby avoiding data traffic between shared and local memory. After each solver is completed, the output is copied back from local to shared memory, see Figure 3.

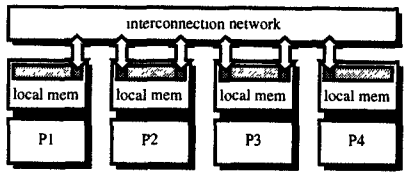


Figure 4: Explicit communication for domain splitting methods with distributed memory parallel architectures.

**Distributed memory parallel systems.** SPMD Fortran 77 code using PVM version 3 is generated for distributed memory parallel computers. A domain splitting method is used to minimize the amount of inter-processor communication. Similar to the previous type of code, the sub-domains have mutually overlapping regions containing duplicated data from logically neighboring processors, see Figure 4. This way, each subdivided problem can be solved independently after a data exchange phase. Message vectorization and combining optimizations are performed to minimize communication overhead. Currently we only allow two-dimensional ($x$ and $y$) data decompositions. This is sufficient for most atmospheric models with two- and three-dimensional grids. The PVM primitives can be easily replaced by MPI primitives.

## 7 Results

In this section performance results of multi-platform generated codes for the HIRLAM weather forecast system are presented and compared with the hand-written production codes.

The development of the CTADEL system received a major impetus by the request of the Royal Netherlands Meteorological Institute for efficient codes for vector and parallel computer architectures to solve the HIRLAM limited area numerical weather forecast model [16]. Over the past few years, several parallel implementations of the forecast model have been realized: a data-parallel implementation [25], a message-passing version [13], a data-transposition code [18], 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, which all differ significantly from the reference code. Clearly, this is an undesirable situation from a maintenance point of view. It also hampers the easy inclusion of new insights into the model by meteorologists, since they are not acquainted with the parallelization techniques.

### 7.1 The HIRLAM Weather Forecast System

The HIRLAM (HIgh Resolution Limited Area Model) system [16] is a numerical weather forecast system developed by the HIRLAM-project group, a cooperative project of Denmark, Finland, Iceland, Ireland, The Netherlands, Norway, and Sweden. HIRLAM is in operational use at several European countries. The HIRLAM weather forecast model contains five atmospheric prognostic variables: two horizontal wind components, temperature, specific humidity, and surface pressure. The computational core of the HIRLAM production code is provided by the *dynamics* and the *physics* routines.

The physics routines compute parameterized processes which are described by the aggregate effect of the physical processes with scales smaller than the model resolution.

In the dynamics routines, a set of three-dimensional coupled nonlinear hyperbolic partial differential equations is solved. Within the dynamics routines, the DYN routine computes the explicit tendencies of the five prognostic variables for each time step. Several solution methods have been implemented. Most of them use explicit Eulerian grid-point finite differencing on a staggered Arakawa C-grid with centered space-differencing and leap-frog time-differencing, resulting in a second order accuracy of the approximations. DYN is a time-consuming routine containing 718 lines of Fortran 77 code optimized for vector architectures. This routine can be reasonably efficiently parallelized for different parallel programming paradigms [8].

### 7.2 Performance Results

The CTADEL system was used to generate five different codes for the the computationally intensive DYN routine almost directly from a mathematical description of the HIRLAM level-1 model: a serial and vector version (same Fortran 77 code, vectorization performed by the back-end compiler), a data-parallel Fortran 90 version, a shared virtual memory parallel version, and a distributed memory parallel version. Three versions of the hand-written DYN code were available which are considered as being 'best by man': the original production Fortran 77 code optimized for vector architectures, a data-parallel Fortran 90 version (see [8, 25]) derived from the original code, and a domain-splitting version (see [8]). We used these three codes as reference codes for the CTADEL experiments.

For the hand-written reference and the generated data-parallel Fortran 90 codes on the MasPar MP-1 and CRAY T3D directives had to be added manually to distribute the two-dimensional horizontal domain using the cyclic (MasPar MP-1) and blocked (CRAY T3D) distributions. Furthermore, all arrays were aligned in memory by setting the lower bounds of the array declarations equal.

The performance of the codes was measured on an HP 9000/720 system (`f77 -K +03`), SGI Indy and Indigo systems (`f77 -O3 -ddopt -static`), a Convex C4 system (`fc -noautopar -O2`, one CPU), a CRAY C98 system (`cf77 -Oscalar3 -Ovector3 -Otask0`, one CPU active), a MasPar MP-1 system (1024 PEs, SIMD architecture, `mpfortran -Omax`), and a CRAY T3D (128 PEs, MIMD architecture with a logically shared memory, but physically distributed memory, `cf77 -Oscalar3 -X`).

Table 2 shows the performance of the hand-written reference and CTADEL generated Fortran 77 DYN codes for serial and vector architectures with a 110 × 100 × 16 grid, one of the grid sizes used for the weather predictions of the operational HIRLAM system. Further optimization of the CTADEL generated codes using standard compiler restructuring transformations like loop fusion, loop reversal and array contraction is denoted by '(+opt.)'. These manually applied loop-based and array-contraction optimizations reduced the memory usage of the generated code. These techniques have been used in the hand-written code to keep memory usage low. The memory usage of the generated codes is 3.8 to 4.6 times higher than the hand-written code without these optimizations and only 1.08 to 1.4 times higher with these optimizations applied to the generated codes. These modifications can be obtained automatically using a restructuring compiler, like the MT1 restructuring compiler [5].

Table 2: Elapsed time (ms) of serial and vector DYN codes with a 110 × 100 × 16 grid.

| | Serial architectures | | | Vector architectures | |
|---|---|---|---|---|---|
| | HP 720 | SGI Indy | SGI Indigo | Convex C4 | CRAY C98 |
| Hand-written code | 4870 | 4390 | 2110 | 203 | 99.4 |
| Generated code | 6740 | 9240 | 2658 | 192 | 91.0 |
| Generated code (+opt.) | 4540 | 4111 | 2126 | 180 | 80.7 |

Table 3: Elapsed time (ms) of parallel DYN codes on MasPar and CRAY T3D systems.

| | MasPar MP-1 (1024 PEs) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 29 × 29 × 16 grid | | | 61 × 61 × 16 grid | | | 125 × 125 × 16 grid | | |
| Data parallel hand-written code | 87.4 | | | 508 | | | 1980 | | |
| Data parallel generated code | 86.3 | | | 412 | | | 1603 | | |
| | CRAY T3D | | | | | | | | |
| | 29 × 29 × 16 grid | | | 61 × 61 × 16 grid | | | 125 × 125 × 16 grid | | |
| | 4 PE | 16 PE | 64 PE | 4 PE | 16 PE | 64 PE | 4 PE | 16 PE | 64 PE |
| Shared virt. mem. hand-written code | 124 | 44.2 | 16.0 | 483 | 141 | 44.2 | 1878 | 512 | 141 |
| Shared virt. mem. generated code | 119 | 35.5 | 11.4 | 479 | 124 | 35.5 | 1929 | 486 | 124 |
| Shared virt. mem. generated code (+opt.) | 110 | 35.4 | 11.1 | 472 | 132 | 35.4 | 1912 | 512 | 132 |
| Distributed mem. generated code | 101 | 37.0 | 15.5 | 420 | 112 | 37.2 | 1664 | 421 | 112 |
| Distributed mem. generated code (+opt.) | 100 | 36.2 | 15.3 | 418 | 120 | 36.3 | 1645 | 456 | 120 |

Table 3 shows the performance of the hand-written reference and CTADEL generated parallel DYN codes: data parallel Fortran 90 DYN code, shared virtual memory parallel DYN code, and distributed memory parallel code (only generated code, no hand-written code).

From Tables 2 and 3 it can be concluded that CTADEL is able to generate efficient codes. Most of the codes even outperformed the hand-written codes; we measured a speedup of up to 7.3% on serial architectures (HP 9000/720), 13% and 23% speedup on Convex C4 and CRAY C98 systems, respectively, up to 24% speedup on MasPar (125 × 125 × 16 grid), and a speedup of up to 44% on the CRAY T3D (29 × 29 × 16 grid, 64 PEs). However, for the CRAY T3D, the time to copy shared arrays from/to local (private) arrays is costly: from 13% up to 25% of the total elapsed time. Performance can be improved by using message passing code [8] instead which is currently being included in CTADEL. The codes with manually applied loop-based and array-contraction optimizations are only more efficient in general on serial architectures.

## 8 Conclusions

In this paper we have presented the CTADEL generator of multi-platform high performance scientific codes optimized for serial, vector, or shared and distributed memory parallel computer architectures. The high efficiency of the generated code is a result of applying algebraic simplification and advanced global common-subexpression elimination based on hardware-specific information. The results of the system for the HIRLAM weather forecast model showed that CTADEL is able to generated efficient codes. Most of the codes even outperformed the hand-written codes.

By developing a prototype system, we were able to share preliminary experiences of (potential) users. These experiences where invaluable with respect to the design of the high-level language for the description of models. The high-level language provides a means for the specification of a problem in a natural way; its power of expressiveness is close to the declarative mathematical formulation of the model.


## Acknowledgements

Jan Wielemaker of the Department of Social Science Informatics (SWI) of the University of Amsterdam for providing the public domain SWI-Prolog package [23].


## References


[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley Publishing Company, Reading MA, 1985.

[2] W.F. Ames, *Numerical Methods for Partial Differential Equations, second edition*, Academic Press, New York, 1977.

[3] A. Arakawa and V.R. Lamb, *A Potential Enstrophy and Energy Conserving Scheme for the Shallow Water Equations*, Monthly Weather Review **109**, 1981, pp. 18–36.

[4] M. Ashworth, *Parallel Processing in Environmental Modelling*, proceedings of the Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology, Reading, UK, November 23–27, 1992, pp. 1–25.

[5] 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.

[6] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt, *MAPLE Reference Manual*, Waterloo, 1988.

[7] G.O. Cook, Jr. and J.F. Painter, *ALPAL: A Tool to Generate Simulation Codes from Natural Descriptions*, in Expert Systems for Scientific Computing, E.N. Houstis, J.R. Rice, and R. Vichnevetsky (eds), Elsevier, 1992.

[8] R. van Engelen and L. Wolters, *A Comparison of Parallel Programming Paradigms and Data Distributions for a Limited Area Numerical Weather Forecast Routine*, proceedings of the 9th ACM International Conference on Supercomputing, ACM Press, New York, 1995, pp. 357–364.

[9] R. van Engelen, L. Wolters, and G. Cats, *Ctadel: A Generator of Efficient Code for PDE-based Scientific Applications*, Technical Report 95-26, Department of Computer Science, Leiden University, 1995.

[10] R.S. Francis, I.D. Mathieson, P.G. Whiting, M.R. Dix, H.L. Davies, and L.D. Rotstyan, *A Data Parallel Scientific Modelling Language*, Journal of Parallel and Distributed Computing **21**, 1994, pp. 46–60.

[11] E. Gallopoulos, E. Houstis, and J.R. Rice, *Future Research Directions in Problem Solving Environments for Computational Science: Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science*, Tech. Report 1259, Center for Supercomputing Research and Development, Univ. of Illinois at Urbana-Champaign, 1992.

[12] E. Gallopoulos, E. Houstis, and J.R. Rice, *Problem-Solving Environments for Computational Science*, IEEE Computational Science & Engineering **1**, 1994, pp. 11–23.

[13] N. Gustafsson and D. Salmond, *A Parallel Spectral HIRLAM with the Data Parallel Programming Model and with Message Passing - A Comparison*, in G.-R. Hoffmann and N. Kreitz (eds.), *Coming of Age*, proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, World Scientific Publ., Singapore, 1995, pp. 32–48.

[14] A.C. Hearn, *REDUCE User's Manual, version 3.6*, RAND, Santa Monica CA, July 1995.

[15] E.N. Houstis, J.R. Rice, N.P. Chrisochoides, H.C. Karathanasis, P.N. Papachiou, M.K. Samartzis, E.A. Vavalis, Ko-Yang Wang, and S. Weerawarana, *//ELLPACK: A Numerical Simulation Programming Environment for Parallel MIMD Machines*, proceedings of the 4th ACM International Conference on Supercomputing, ACM Press, New York, 1990, pp. 96–107.

[16] P. Kållberg (editor), *Documentation Manual of the Hirlam Level 1 Analysis-Forecast System*, June 1990. Available from SMHI, S–60176 Norrköping, Sweden.

[17] E. Kant, F. Daube, W. MacGregor, J. Wald, *Scientific Programming by Automated Synthesis*, Chapter 8 in Automating Software Design, M. Lowry and R. McCartney, (eds), AAAI Press/MIT Press, Menlo Park, CA, 1991, pp. 169–205.

[18] T. Kauranne, J. Oinonen, S. Saarinen, O. Serimaa, and J. Hietaniemi, *The Operational HIRLAM 2 Model on Parallel Computers*, in G.-R. Hoffmann and N. Kreitz (eds.), *Coming of Age*, proceedings of the Sixth ECMWF Workshop on the Use of Parallel Processors in Meteorology, World Scientific Publ., Singapore, 1995, pp. 63–74.

[19] J.R. Rice and R.F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, 1985.

[20] Th. Ruppelt and G. Wirtz, *Automatic Transformation of High-level Object-oriented Specifications into Parallel Programs*, Parallel Computing, **10**, 1989, pp. 15–28.

[21] A. Simmons and D.M. Burridge, *An Energy and Angular-Momentum Conserving Vertical Finite-Difference Scheme and Hybrid Vertical Coordinates*, Monthly Weather Review **109**, 1981, pp. 758–766.

[22] Y. Umetani, M. Tsuji, K. Iwasawa, and H. Hirayama, *DEQSOL: A Numerical Simulation Language for Vector/Parallel Processors*, in B. Ford and F. Chatelin (eds.), *Problem Solving Environments for Scientific Computing, Proceedings*, North-Holland, Amsterdam, 1987.

[23] J. Wielemaker, *SWI-Prolog Reference Manual*, University of Amsterdam, 1995. Available by anonymous ftp from `swi.psy.uva.nl`.

[24] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Boston, 1988.

[25] L. Wolters, G. Cats, and N. Gustafsson, *Data-Parallel Numerical Weather Forecasting*, Scientific Programming **4**, 1995, pp. 141–153.