Parallel Aspects of Numerical Weather Prediction on a Grid of Transputers

Hans Middelkoop
Dick Streefland

RUU-CS-88-19
April 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Parallel Aspects of Numerical Weather Prediction on a Grid of Transputers

Hans Middelkoop
Dick Streefland

Technical Report RUU-CS-88-19
April 1988

Departement of Computer Science
University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht
The Netherlands

Parallel Aspects of Numerical Weather Prediction on a Grid of Transputers

Hans Middelkoop Dick Streefland

April 1988
INF/SCR-88-2

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

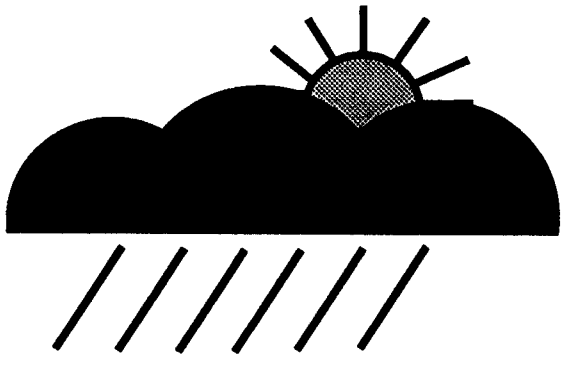
Abstract

This document is a report of a feasibility study, initiated by the Dutch weather forecast service (KNMI) and the department of computer science of the university of Utrecht, of the possibilities of parallelizing numerical weather prediction programs for execution on a grid of transputers.

Our investigations concern programs, in which a set of partial differential equations, describing the behavior of the atmosphere, are numerically solved.

Two different implementation models are presented, together with a criterion for choosing the fastest model. Speedup formulas are derived by which speedup for both models can be calculated.

Applied to HIRLAM, a specific weather prediction program, these results show that both internal and external communication time are negligible with respect to calculation time. As a consequence, for this program, speedup is almost linear in the number of transputers used.



Contents

1 Introduction 1
- 1.1 Computer Architectures . . . 1
- 1.2 Objectives . . . 2
- 1.3 Results . . . 3
- 1.4 Report Overview . . . 5

2 Discrete Fluid Models 7
- 2.1 From Continuous Fluid Models to Discrete Fluid Models . . . 7
- 2.2 Finite Difference Method . . . 7
- 2.3 The Spectral Method . . . 8

3 Finite Difference Method 9
- 3.1 Finite Differences . . . 9
- 3.2 Finite Difference Schemes . . . 10
  - 3.2.1 Space Difference Schemes . . . 10
  - 3.2.2 Time Difference Schemes . . . 11
- 3.3 Staggering . . . 15
  - 3.3.1 One-Dimensional Staggering . . . 16
  - 3.3.2 Two-Dimensional Staggering . . . 16
  - 3.3.3 Implementation of Staggering on Vector Computers . . . 18
  - 3.3.4 One-Dimensional Time Staggering . . . 18
  - 3.3.5 Two-Dimensional Time Staggering . . . 19
- 3.4 Computational Economy of Explicit Schemes . . . 20
  - 3.4.1 Explicit Splitting . . . 21

4 The Hirlam Model 23
- 4.1 The Continuous Hirlam Model . . . 23
  - 4.1.1 The Continuous Equations . . . 24
- 4.2 The Discrete Hirlam Model . . . 26
- 4.3 The Sequential Hirlam Program . . . 28
  - 4.3.1 Program structure . . . 28
  - 4.3.2 Statistics . . . 29
  - 4.3.3 Computer code organization . . . 31

5 The Transputer **33**
5.1 Why Transputers? 33
5.2 What is a Transputer? 34
5.3 Properties of Transputers 34

6 Grid-Based Computations on a Network of Transputers **37**
6.1 Choosing a Network Topology 37
6.2 Implementation of Staggering on a Grid of Transputers 39
6.3 Two Computational Models 39
6.3.1 The Parallel Model 39
6.3.2 The Sequential Model 41
6.4 Comparing the two Models 42
6.4.1 The Optimal Parallel Model 43
6.4.2 The Optimal Sequential Model 43
6.4.3 Comparison of the two Optimal Models 45
6.5 Using More Transputers 46
6.6 Speedup Calculations 49
6.7 Variables for Speedup Calculation 50
6.8 Conclusion 51

7 Parallelization of the Hirlam Program **53**
7.1 Communication in the Hirlam Program 53
7.2 Choosing the Implementation Model 54
7.2.1 Calculation of Q for Subroutine DYN 54
7.2.2 Estimation of Q for the Hirlam Program 57
7.3 Performance Expectations 58
7.3.1 Determining T_{calc} for subroutine DYN 58
7.3.2 Estimating T_{calc} for the Hirlam Program 59
7.4 Performance with T800 transputers 60
7.5 Speedup Calculations for the Hirlam Program 62
7.6 Conclusion 64

8 Conclusions and Future Work **65**
8.1 Conclusions 65
8.2 Future Work 66

A Mathematical Principles used for Finite Differencing **67**
A.1 Finite Differences 67
A.2 Computational Errors 67
A.3 Consistency, Convergence and Stability 68

B Implicit and Semi-Implicit Schemes **71**
B.1 Semi-Implicit Scheme 71
B.2 Implicit Scheme 71
B.2.1 Relaxation Method 72

C Modeling the Dynamics of the Atmosphere **75**
C.1 The Primitive Equations 75
C.2 Coordinate System 77
C.2.1 Vertical Differencing of the Primitive Equations 77
C.3 Time Filtering 80
C.4 Limited Area Modeling 80
C.4.1 Boundary Relaxation 80

D Test Programs **83**
D.1 Program `par.c` 83
D.2 Program `seq.c` 89
D.3 Program `optim.c` 91
D.4 Program `bench.c` 95
D.5 Program `link.c` 97

E Implementation of DYN in C **101**
E.1 Program `DYN.c` 101

F Test and working environment **115**
F.1 Hardware 115
F.2 Software 116
F.2.1 Support 116

Bibliography **117**

Chapter 1

Introduction

The future state of the atmosphere can in principle be computed from the current state by solving a system of partial differential equations which models the behavior of the atmosphere. Such a solution process, using numerical methods, is called a numerical weather prediction. When such a solution process is applied to a fictitious current state, it is called numerical simulation.

In the late 1930's, it became understood, that even a rather simple model, describing the conservation of absolute vorticity following the motion of air particles, suffices for a useful approximate description of large scale motions of the atmosphere. It is generally assumed that various physical processes can be incorporated more easily in the integration of the basic primitive equations than in the integration of modified equations (i.e. integration of the divergence and vorticity equations). Thus, today the primitive equations are mostly used for practical numerical prediction by meteorological services[2, p1].

Most weather predictions are based on numerical weather prediction, which is very time-consuming. It is therefore of particular importance that these calculations are performed in an efficient way.

1.1 Computer Architectures

The first successful weather forecast based on numerical weather prediction in the late 1940's was obtained from the absolute vorticity conservation equation using the first electronic computer ENIAC (Electronic Numerical Integrator And Computer). Much faster computers and improved understanding of computational problems now also enable long term integrations of the basic primitive equations.

Solving partial differential equations requires vast amounts of processing. Therefore, meteorologists were always among the first users of new computer architectures. Nowadays most numerical weather prediction programs are executed on vectorprocessing pipeline computers.

While a wide variety of algorithms and software has been developed for pipeline

computers, only recently much research and experimentation with large scale multiprocessor systems has been started. A multiprocessor system consists of a number of processors, each with its own memory. Processors can execute different instructions. Thus if the multiprocessor system is used for one program, different parts of the program (called tasks) are assigned to different processors. As for memory, each processor has a local memory that can not be accessed directly by other processors. Some multiprocessor systems also have a shared memory that can be accessed by all processors.

When processors are executing tasks of the same program, there might be a need for communication: intermediate results computed by one processor are needed by another processor. In case a multiprocessor has a shared memory, this can be used for the communication. In the other case, each processor is directly connected with a number of other processors by links.

Modern pipeline computers like the CRAY and CYBER-205, are very expensive. This is mainly because the technology used is very sophisticated. For multiprocessor systems, performance can simply be increased by adding extra processors, while for pipeline computers performance can not be changed. In theory, performance for multi processor systems can be increased infinitely. The individual processors used for multiprocessor systems do not need not to be fast, but the price/performance ratio must be as low as possible. Besides that, such a system is more flexible than a pipeline computer.

1.2 Objectives

The questions to be addressed in this report is how and in what sense weather prediction can benefit from the use of a multiprocessor system and how performance predictions can be made. In order to be more specific we will consider the following issues:

Partitioning Is there an obvious way to partition the computations in numerical weather prediction into a number of tasks, each task to be assigned to one processor, so that the load for each processor will be approximately equal?

Communication The need for communication results from dependencies between tasks that are assigned to different processors. For most multiprocessor systems without shared memory, communication via links is rather time consuming compared to computation. Care should be taken that partitioning of the computation process does not lead to excessive communication requirements. It is therefore important to know what dependencies can occur in numerical weather prediction programs.

Distribution In a multiprocessor system it is in general not the case that each processor is directly connected to every other processor. This implies that sending

some data from one processor to another might require the data passing through several other processors, thus increasing overhead. In the multiprocessor system it is therefore important to assign communicating tasks to processors that are close to each other.

Finite number of processors It might be that an obvious partition of the computation leads to a number of tasks that exceeds the number of processors. Thus several tasks must be assigned to the same processor or a new partition must be made. On the other hand, when the number of processors exceeds the number of tasks, a different partition is even necessary, which generally causes extra communication.

Implementation model If several tasks are assigned to the same processor, these tasks have to communicate. There are several ways to implement this "internal" communication.

Adding processors One advantage of a multiprocessor system is the ease with which processors can be added, thus increasing the performance of the system. It is important to be able to predict how the addition of processors affects the overall computation time of a fixed numerical weather prediction program.

1.3 Results

As a vehicle for experiments we used the *transputer* microprocessor. The transputer microprocessor is especially designed for building parallel systems. The transputer is relatively cheap, and among the fastests microprocessors currently available. Because of these properties, a multiprocessor system of transputers could be well suited for performing numerical weather prediction in a cost-effective way.

As an example of a weather prediction model, we had access to a program from the ECMWF (European Centre for Medium Range Weather Forecast), called HIRLAM. HIRLAM is a weather prediction program, containing modern numerical techniques. It is currently in use as a research model. Therefore HIRLAM is an appropriate program for testing the suitability of a parallel transputer system for numerical weather prediction.

The HIRLAM program uses a 3-dimensional grid to model the atmosphere. This 3-dimensional structure is reflected in the computation. Basically, the computation consists of a series of updates of variables associated with gridpoints. Therefore, the most obvious way to partition computations is by taking the calculations associated with one gridpoint as a separate task. When the number of these tasks exceeds the number of available processors, each processor should process a number of tasks.

In numerical weather prediction, many numerical methods can be used for the solution of the partial differential equations. In explicit finite-difference schemes,

variables at gridpoints are dependent of known values at other gridpoints. From numerical viewpoint, it suffices to use values at neighboring gridpoints only (local dependencies). In implicit finite-difference schemes variables at gridpoints are also dependent of unknown values at other gridpoints. Thus there is interdependency of variables at many gridpoints (global dependencies).

The disadvantage of implicit schemes is that it requires communication between many tasks, and therefore between tasks assigned to processors that are not close to each other in the multiprocessor system. Explicit methods do not have this disadvantage. However, a smaller time step in the integration procedure is necessary because of stability requirements. This loss in computation time can be easily compensated for in a multiprocessor environment. We have restricted ourselves to explicit computations.

In the HIRLAM model the number of tasks exceeds the number of transputers and there are more dependencies in vertical direction than in the horizontal directions. Therefore the tasks associated with a vertical column of gridpoints should be executed on the same transputer. Typically, even the number of columns exceeds the number of transputers, so a cluster of columns should be assigned to the same transputer. To lower dependencies between transputers, and therefore communication overhead, such a cluster has to consist of neighboring columns. Inspection of the communication requirements with respect to the distribution of tasks among transputers, reveals how the shape of these clusters should be chosen in order to minimize the amount of communication between transputers.

There are different ways to implement the computations associated with a cluster of neighboring columns. Two different implementation models are presented, as well as a criterion to determine in advance which implementation model will be faster.

For both implementation models expressions for the execution time were derived, and these were used to derive expressions for the speedup, in case the number of transputers would be increased. These formulas contain a number of machine-dependent constants and a number of application-dependent constants.

The machine-dependent constants were measured by running benchmarks on the transputer system. The application-dependent constants of the HIRLAM program were obtained by close inspection of the program text, and by using estimations based on the running time of the HIRLAM program on a Harris HCX-9 computer.

Substitution of these constants in the formulas for execution time and speedup show that the time for communication is negligible with respect to calculation time. As a consequence, for the HIRLAM program, speedup is almost linear in the number of transputers used.

Because of the parallel nature of the computation process associated with solving partial differential equations, numerical weather programs can be easily and efficiently implemented on a system of transputers, provided that a good programming environment is available.

1.4 Report Overview

As a first step we investigated what kind of numerical calculations are used in programs solving a set of partial differential equations, describing fluid models. Two different methods to solve these equations are introduced in chapter 2.

In chapter 3 one of these methods, the finite-difference method, is analized. This is the method used in the HIRLAM program. In particular, the aspects that are relevant with respect to the implementation of numerical weather prediction are investigated. These aspects include explicit, semi-implicit and implicit finite-difference schemes and the distribution of the dependent variables in space and time.

Chapter 4 is a description of the HIRLAM model using the theory of chapter 3.

The transputer microprocessor is introduced in chapter 5. In this chapter the use of transputers for numerical weather prediction is motivated and some properties of the transputer are discussed.

In chapter 6 the implementation of grid-based computations due to using staggering and explicit finite-difference schemes for approximating partial differential equations is investigated. It is argued that the most appropriate network topology for the computations is a 2-dimensional grid of transputers, each one calculating a subgrid of the discrete model. Two different models for implementing the calculations for such a subgrid on one transputer are presented, together with some possible optimalizations. A criterion is determined by which it is possible to choose the fastest implementation model for a given application. Formulas to predict the execution time for both implementation models are derived. These formulas are used to derive a formula for the speedup.

In chapter 7 the results of chapter 6 are applied to the HIRLAM program. Figures with the expected running time for the HIRLAM program running on a grid of T414, respectively T800 transputers, are presented. Also, a figure with the expected speedup for a grid of T800 is included.

Finally, in chapter 8 conclusions are drawn and some suggestions for future work are presented.

Chapter 2

Discrete Fluid Models

2.1 From Continuous Fluid Models to Discrete Fluid Models

We restrict ourselves to continuous fluid models, described by a system of partial differential equations together with boundary and initial conditions, which are recognized as initial boundary value, or propagation problems. A typical physical example of a propagation problem in fluid models is the propagation of pressure waves in the fluid. Propagation problems are mainly described by parabolic and hyperbolic equations[1, p4].

The ultimate aim of discrete methods is the reduction of continuous models to equivalent discrete models, which are suitable for treatment by a high speed computer. The discretization is purely mathematical when the continuous problem formulation is transformed to a discrete formulation. There are two basic approaches to make this transformation: the finite-difference method and the spectral method.

2.2 Finite Difference Method

Applying a finite-difference method, derivatives are replaced by finite difference approximations. Therefore a continuous domain is replaced by a pattern, usually a rectangular mesh of discrete points, and a finite-difference method is also called a gridpoint method. In the next step of the transformation again there are two basic approaches: the Eulerian and the Lagrangian formulation of the differential equations.

The Eulerian formulation In the Eulerian formulation, the independent space variables are related to a fixed spatial coordinate system. The fluid is visualized as moving through this fixed reference frame and is characterized by a time dependent velocity field, which is to be determined by solving an initial value problem[1, p213].

The Lagrangian formulation The Lagrangian formulation is characterized by attaching the coordinate mesh to the moving fluid. Consequently, the independent space variables are related to a reference frame fixed in the fluid and changes with all the distortion and motion of the fluid. The fluid particles are permanently identified by these variables, called Lagrangian variables. Particle positions are among the dependent variable we wish to determine by solving an initial value problem [1, p213].

Euler versus Lagrange Although the Eulerian and Lagrangian formulations are equivalent, this may not be true of the two solutions obtained from the distinct finite-difference approximations. The major disadvantage of the Eulerian method arises when interfaces occur separating fluids of different density. The Lagrangian system does not have the spatial coordinate mesh fixed in advance and can accommodate a required refinement of the mesh as computation advances. Although the Lagrangian approach simplifies the equations of motion, the major disadvantage of the Lagrangian method is the distortion of the mesh as time advances, causing unacceptable inaccuracies[1, p213-221].

2.3 The Spectral Method

When using finite-difference techniques for evolutionary problems we only consider the values of the dependent variables at the gridpoints. An alternative approach is to expand the dependent variables, such that the spatial dependence of the variables is expressed in terms of a series of orthogonal functions. By substituting the expanded formulation into the system of equations, the equations reduce to a set of ordinary differential equations and the coefficients of the series can be computed as functions of time. Thus, in spectral models not the dependent variables at discrete points are simulated, but the coefficients of the series, called spectral components.

Formerly there was a general agreement that as for efficiency the spectral method could not be competitive with finite-difference methods, but nowadays this situation has changed completely. More information about the spectral method can be found in a review article[11].

Chapter 3

Finite Difference Method

In the finite-difference method the expressions which approximate the derivatives are defined using only values of the dependent variables at discrete (time) intervals. The simplest way of introducing a set of gridpoints in a bounded one-dimensional continuous domain D, having length L, is to require that the gridpoints divide D into an integer number of intervals J of equal length Δx. This length is called the grid interval or grid-length.

For simplicity we start by considering a function u of one independent variable x. We are looking for approximations of $u(x)$ at discretization-points $x = j\Delta x$, j=0,1,...,J and define $u_j \approx u(j\Delta x)$ to be the approximate value of u at discretization-point $j\Delta x$.

3.1 Finite Differences

Now let us consider the differences of values u_j that will be used to construct approximations of derivatives. These differences are called finite-differences and can be calculated over one or more of the intervals Δx. Depending on the relation of the points from which the values are taken to the point where the derivative is required, finite-differences can be centered or uncentered.

An example of an uncentered difference is the forward difference:

$$\Delta u_j = u_{j+1} - u_j$$

More often centered differences are used such as:

$$\delta u_{j+\frac{1}{2}} = u_{j+1} - u_j$$

In a centered difference the difference between values is symmetrical about the point where the difference is being calculated.

Differential equations can be approximated at discrete points by simply replacing the derivatives by appropriate finite-difference quotients. For example, for the first

derivative, one can use the approximation:

$$\left(\frac{du}{dx}\right)_j = \frac{u_{j+1} - u_j}{\Delta x}$$

The finite-difference quotient here is, of course, only one of the many possible approximations to the first derivative at point j.

3.2 Finite Difference Schemes

The algebraic equation obtained when the derivatives in a differential equation are replaced by appropriate finite-difference approximations, is called a finite difference scheme. Because a finite difference scheme is a blueprint for a computer program, a finite-difference scheme could be called a finite difference algorithm as well.

Designing a finite-difference scheme one has to observe several, often conflicting criteria, which can be summarized as follows:

1. Correctness

2. Computational economy

Of course the final decision is always a compromise between these requirements. The first criterion, however, is obviously the most important one and has been studied extensively. Obviously, in our project the second criterion deserves special attention.

The subject of finite-difference solutions of linear advection equations is a vast research area in many fields of fluid dynamics. Therefore we start with this linear advection equation:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{3.1}$$

where u is a function of two independent variables x and t, and c is a positive constant. (3.1) Describes local change by transport of the variable u at a constant velocity c (the phase speed) in the direction of the x-axis.

3.2.1 Space Difference Schemes

Applying a space differencing scheme, the partial differential equation can be rewritten into a differential-difference equation. In figure 3.1 some results are tabulated.

It is now widely understood that the major computational problems in finite difference solutions of (3.1) are those encountered due to space differencing. These problems are the problems of the phase speed (the computed phase speed is less than the true phase speed) and of the computational dispersion (the phase speed of the analytic solution is a constant c, whereas the phase speed of the numerical wave solution is dependent of the wavelength)[3].

space differencing scheme	differential-difference equation
forward	$(\frac{\partial u}{\partial t})_j = -c\frac{u_{j+1}-u_j}{\Delta x}$
backward	$(\frac{\partial u}{\partial t})_j = -c\frac{u_j-u_{j-1}}{\Delta x}$
centered	$(\frac{\partial u}{\partial t})_j = -c\frac{u_{j+1}-u_{j-1}}{2\Delta x}$

Figure 3.1: The advection equation space differenced.

3.2.2 Time Difference Schemes

To define some schemes we consider again equation (3.1). We now assume u to be a function of two independent variables x and t, where t represents time. We define $u_j^n \approx u(j\Delta x, n\Delta t)$ to be the approximate value of u at gridpoint $j\Delta x$ and at instance of time $n\Delta t$.

k-Level Schemes

k-Level time differencing schemes are defined to be schemes that relate values of the dependent variable at two instances of time $n-k+2$ and $n+1$ (and possibly time instances in between). With k-level schemes we can approximate the exact formula:

$$u(j\Delta x,(n+1)\Delta t) - u(j\Delta x,(n-k+2)\Delta t) = \int_{(n-k+2)\Delta t}^{(n+1)\Delta t} \frac{\partial u(j\Delta x,t)}{\partial t} dt \qquad (3.2)$$

When the dependent variable has only one occurrence of time instance $n+1$ in the resulting finite difference scheme, such a scheme is called explicit, else it is called implicit. In numerical integration mostly two- and three level schemes are used. To start the integration, for two level schemes, a single initial condition is needed. Some two level time differencing schemes applied to the differential-difference equation obtained with centered space differencing of figure 3.1 are tabulated in figure 3.2. For three level schemes an additional initial condition is needed, which can be found by applying a two-level scheme as the first step in the integration procedure.

two-level time differencing schemes	
forward	$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -c\frac{u_{j+1}^n-u_{j-1}^n}{2\Delta x}$
backward	$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -c\frac{u_{j+1}^{n+1}-u_{j-1}^{n+1}}{2\Delta x}$
trapezoidal	$\frac{u_j^{n+1}-u_j^n}{\Delta t} = -\frac{1}{2}c(\frac{u_{j+1}^n-u_{j-1}^n}{2\Delta x} + \frac{u_{j+1}^{n+1}-u_{j-1}^{n+1}}{2\Delta x})$

Figure 3.2: The advection equation time- and space differenced.

Although it is possible to construct a finite-difference scheme for (3.1) by an

independent choice of finite-difference approximations to space and time, not all combinations are useful.

As an example forward integration together with centered space differencing gives an unstable scheme for (3.1). This can be understood by substitution of a tentative solution:

$$u_j = Re\left[U(t)e^{ikj\Delta x}\right] \tag{3.3}$$

into the forward difference scheme of figure 3.2:

$$U^{n+1} = U^n + i(-c\frac{\Delta t}{\Delta x}sin(k\Delta x))U^n \tag{3.4}$$

where $U(t)$ is the amplitude of a wave with wavenumber k. According to the von Neumann method for stability analysis the amplification factor is:

$$\lambda = 1 - i(c\frac{\Delta t}{\Delta x}sin(k\Delta x)) \tag{3.5}$$

which has to suffice the von Neumann necessary condition for stability:

$$|\lambda| \leqq 1$$

which reflects the amplitude requirement of the true solution (no growth of amplitude).

Therefore the scheme is only stable for $k\Delta x = n \bmod \pi$. Because there are usually more wavenumbers contained in the solution the finite-difference scheme is called unstable.

Computational mode According to (3.4), applying a three-level scheme we have:

$$U^n = \lambda U^{n-1} \tag{3.6}$$

$$U^{n+1} = \lambda^2 U^{n-1} \tag{3.7}$$

Solution of (3.7) gives two solutions λ_1 and λ_2. In general, a k-level scheme will give $k-1$ solutions of the form (3.6). A solution of this type corresponding to a single value of λ is called a mode. If a solution of the form (3.6) is to represent an approximation of the true solution then λ tends to 1 as Δt tends to 0. Solutions associated with λ_i, where λ_i tends to 1 are usually called physical modes because we are always solving equations describing physical processes. The other solutions are not approximations to the true solution, and are called computational modes.

Leap frog scheme By applying the leap-frog scheme to:

$$\left(\frac{du}{dt}\right)_j = -c\frac{u_{j+1} - u_{j-1}}{2\Delta x} \tag{3.8}$$

we get the explicit finite-difference scheme

$$\frac{u_j^{n+1} - u_j^{n-1}}{2\Delta t} = -c\frac{u_{j+1}^n - u_{j-1}^n}{2\Delta x} \tag{3.9}$$

Because the leap-frog scheme is a three level scheme with centered space differencing it has a computational mode, which is spurious and source of error. The CFL-condition for a computational stability is $\left|\frac{c\Delta t}{\Delta x}\right| \leqslant 1$.[2, p3]

Although the leap-frog scheme is probably at present the most widely used scheme in atmospheric models it is not the most economical scheme for any system of equations.

Forward-backward scheme The forward-backward scheme is a two level scheme that can be applied to a system of equations by first integrating one variable forward and then the others backward.

A convenient example for illustrating this scheme is the system of one dimensional linearized shallow water equations, describing the simplest case of gravity waves in the atmosphere:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} = 0 \tag{3.10}$$

$$\frac{\partial h}{\partial t} + H\frac{\partial u}{\partial x} = 0 \tag{3.11}$$

where u, h, g, H are the velocity, height, gravitation and the water depth respectively. Equations (3.10) and (3.11) represent the second law of Newton and the equation of continuity (conservation of mass) respectively.

A forward-backward approximation to (3.10) and (3.11) using centered space differencing is:

$$\frac{u_j^{n+1} - u_j^n}{\Delta t} + g\frac{h_{j+1}^n - h_{j-1}^n}{2\Delta x} = 0 \tag{3.12}$$

$$\frac{h_j^{n+1} - h_j^n}{\Delta t} + H\frac{u_{j+1}^{n+1} - u_{j-1}^{n+1}}{2\Delta x} = 0 \tag{3.13}$$

where the variables u and h are integrated forward and backward respectively.

Because the time-step in the forward-backward scheme required for stability is twice the time-step imposed by the CFL criterion for the leap-frog scheme, the forward-backward scheme needs half the computation time needed for the leap-frog scheme[2, p54].

Lax-Wendroff scheme The Lax-Wendroff scheme is a very special scheme in that it can not be constructed by an independent choice of finite-difference approximations to the space and time derivatives. It is second order in both space and time, explicit and since it is a two level scheme there is no computational mode.

To obtain a Lax-Wendroff scheme for equation (3.1) first two provisional values are calculated at the middle of the two rectangular meshes denoted by an circle in figure 3.3. The calculation of the provisional values is done using centered space

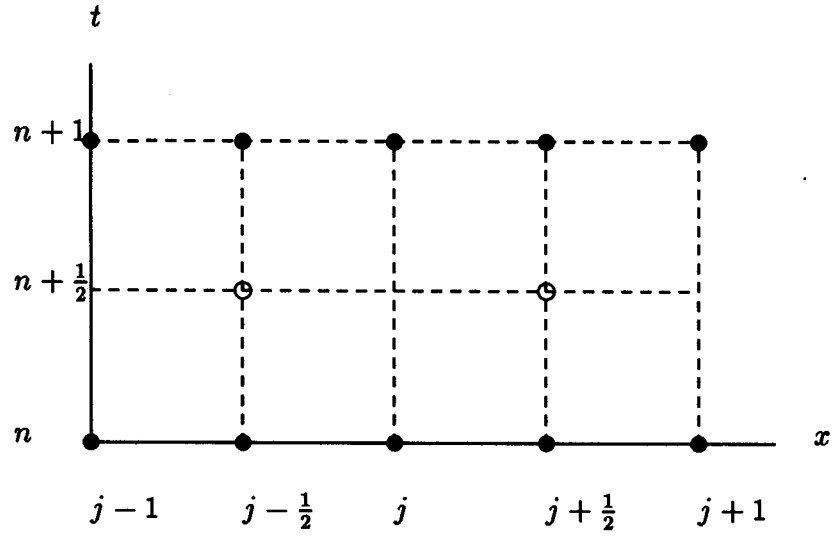


Figure 3.3: The space-time grid used for the construction of the Lax-Wendroff scheme.

and forward time differencing taking for $u^n_{j+\frac{1}{2}}$ and $u^n_{j-\frac{1}{2}}$ arithmetic averages of the values u^n_j at the two nearest gridpoints:

$$\frac{u^{n+\frac{1}{2}}_{j+\frac{1}{2}} - \frac{1}{2}(u^n_{j+1} + u^n_j)}{\frac{1}{2}\Delta t} = -c\frac{u^n_{j+1} - u^n_j}{\Delta x} \tag{3.14}$$

$$\frac{u^{n-\frac{1}{2}}_{j-\frac{1}{2}} - \frac{1}{2}(u^n_j + u^n_{j-1})}{\frac{1}{2}\Delta t} = -c\frac{u^n_j - u^n_{j-1}}{\Delta x} \tag{3.15}$$

Using these provisional values another step is made, centered in both space and time:

$$\frac{u^{n+1}_j - u^n_j}{\Delta t} = -c\frac{u^{n+\frac{1}{2}}_{j+\frac{1}{2}} - u^{n+\frac{1}{2}}_{j-\frac{1}{2}}}{\Delta x} \tag{3.16}$$

Substitution of the provisional values from (3.14) and (3.15) into (3.16) gives the equation:

$$\frac{u^{n+1}_j - u^n_j}{\Delta t} = -c\frac{u^n_{j+1} - u^n_{j-1}}{2\Delta x} + \frac{1}{2}c^2\Delta t\frac{u^n_{j+1} - 2u^n_j + u^n_{j-1}}{(\Delta x)^2} \tag{3.17}$$

The Lax-Wendroff scheme has been fairly widely used in atmospheric models[2, p24-25]. Obviously the principle draw-back is the two step feature, which requires twice as much computation time as the leap-frog scheme, because the stability condition and consequently the imposed time step is the same as for the leap-frog scheme.

Nonlinear Partial Differential Equations

Many of the numerical methods for linear equations can be also applied to nonlinear equations. Questions of stability and convergence are more complicated. For non-linear problems, stability depends not only on the form of finite-difference system, but also generally upon the solution to be obtained. The system may be stable for some values of t and not for others[1, p73-74]. Philips discovered the cause of the instability to be aliasing, a phenomenon where by a wave generated by a non-linear interaction that is too short to be represented on the grid is falsely represented (aliased) as a longer wave-length. Repeated aliasing over many time-steps may give rise to a rapid growth of energy (instability) through feedback into the wave-length band $2\Delta x$ tot $4\Delta x$[12, p170].

One-dimensional non-linear advection equation The advection equation (3.1) generalizes to the non-linear equation of transport:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0 \tag{3.18}$$

where u is a function of two space variables x and t.

It was found by experience that the use of the Lax-Wendroff scheme does suppress non-linear instability and that it is sufficient to use an intermittent Lax-Wendroff step at quite long intervals[2, p37]. Another way of avoiding non-linear instability is to use the Lagrangian formulation for the advection term instead of the Eulerian formulation, because in the Lagrangian formulation $\frac{\partial u}{\partial x} = 0$ and (3.18) becomes linear.

3.3 Staggering

Staggered grids are grids with a spatial distribution of variables, such that not all variables are carried at each gridpoint. Staggering enables centered space differences and forward time differences to be used to approximate derivatives, without developing instability.

3.3.1 One-Dimensional Staggering

Consider the differential-difference equations:

$$\left(\frac{\partial u}{\partial t}\right)_j = -g\frac{h_{j+1} - h_{j-1}}{2\Delta x}, \tag{3.19}$$

$$\left(\frac{\partial h}{\partial t}\right)_j = -H\frac{u_{j+1} - u_{j-1}}{2\Delta x} \tag{3.20}$$

that we obtain when the space derivatives in (3.10) and (3.11) are approximated by centered finite-difference quotients using values at the two nearest gridpoints.

Assuming a grid with two dependent variables that are both carried at every gridpoint, clearly two independent solutions are calculated, which may diverge from each other. On the contrary, one solution is obtained if (3.19) and (3.20) are applied to a grid with the dependent variables u and h carried at alternate points in space as depicted in figure 3.4.

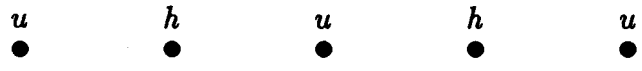


Figure 3.4: A staggered grid

Therefore the computation time needed to solve (3.10) and (3.11) on a staggered grid is reduced by a factor of two compared to computation on a non-staggered grid.

At first glance the staggered grid has a serious draw-back in that the aliasing error is twice as big, but the eliminated waves are precisely the waves with large phase speed errors and negative group velocities[2, p44].

3.3.2 Two-Dimensional Staggering

We now consider the two-dimensional linearized shallow water equations:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} = 0 \tag{3.21}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial y} = 0 \tag{3.22}$$

$$\frac{\partial h}{\partial t} + H(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}) = 0 \tag{3.23}$$

Having three dependent variables and only two dimensions, a large number of spatial arrangements of the variables are possible. Various possibilities are depicted in figure 3.5.

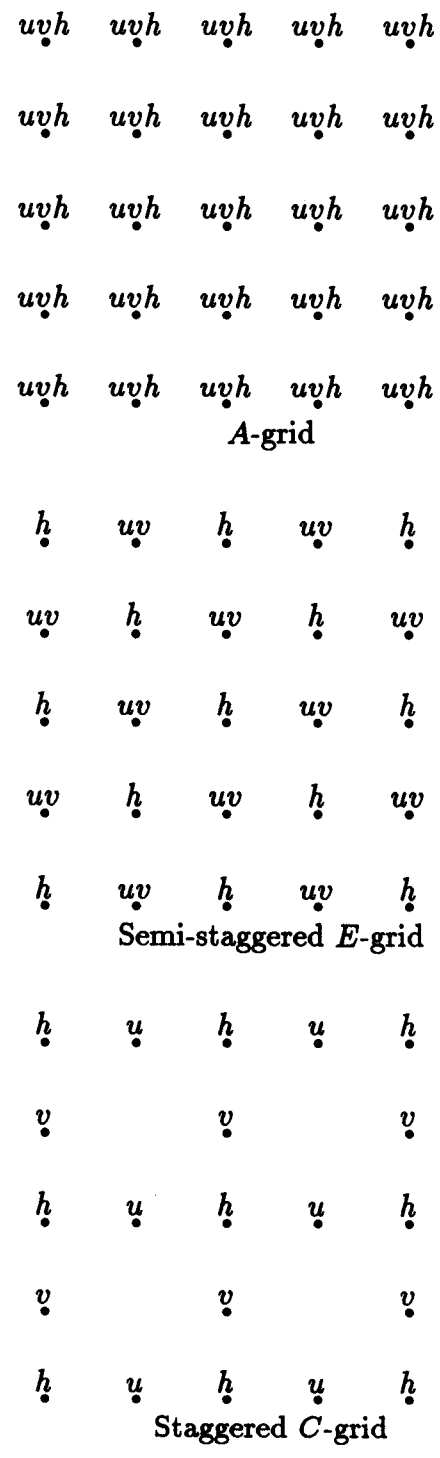


Figure 3.5: Various arrangements of dependent variables in a rectangular grid

Because grid A is a superposition of two E-grids and an E-grid is a superposition of two C-grids, E is a staggered grid of A and C is a staggered grid of E. Arakawa, Winninghof and Lamb investigated the impact of the grid-type on the simulation of important physical processes, e.g. gravity waves and the geostrophic adjustment process, by second-order accurate finite-difference approximations. They concluded the C-grid to be the superior grid both in computation time and in simulation properties[2, p45-50].

3.3.3 Implementation of Staggering on Vector Computers

Concerning programming staggering is rather straight forward. Instead of computing u at odd gridpoints and h at even gridpoints, for the one-dimensional case, the concept of staggering can be made transparent to the program by storing each variable in a separate array. As a result for instance the physical gridpoint, carrying u, as left neighbor of a physical gridpoint carrying h can both be accessed by using the same array subscripts. Using this implementation model two physical neighbor gridpoints can be considered as one logical gridpoint; the logical grid interval will therefore be twice the physical grid interval. This way staggering can easily be implemented on vector-computers. Extra economy is reached by saving half of the constant increments used for subscripting.

The same arguments hold in case of two dimensional staggering with respect to the staggered Arakawa C-grid of figure 3.5. In case of a logical gridpoint consisting of a physical gridpoint carrying h, a left (west) neighbor carrying u and a below (south) neighbor carrying v, for computing h, u, v only southward and westward, eastward, northward communications are needed respectively.

3.3.4 One-Dimensional Time Staggering

The leap-frog scheme with centered space differencing clearly has a computational mode in time if the dependent variable is calculated for all gridpoints at every time-step. Only one solution is obtained if the dependent variable is calculated for half of the gridpoints at one instance of time and for the gridpoints in between at the next instance of time. The concept of time-staggering can be made more clear by reconsidering the linearized shallow water equations (3.10) and (3.11). Elimination of the variable u gives a wave equation:

$$\frac{\partial^2 h}{\partial t^2} - gH\frac{\partial^2 h}{\partial x^2} = 0 \tag{3.24}$$

We can perform the same elimination on the forward-backward finite difference equations (3.12) and (3.13) by execution of the following procedure:

1. Subtract from (3.13) an analogous equation for time instance $n-1$ instead of n

2. Divide the resulting equation by Δt.

3. Substitute (3.12) for u values written for grid-points $j+1$ and $j-1$ instead of j.

We obtain:

$$\frac{h_j^{n+1} - 2h_j^n + h_j^{n-1}}{(\Delta t)^2} - gH\frac{h_{j+2}^n - 2h_j^n + h_{j-2}^n}{(2\Delta x)^2} = 0 \tag{3.25}$$

Note that each of the two equations (3.12) and (3.13) is of first order of accuracy in time, where approximation (3.25) is of second order of accuracy.

If we apply the leap-frog and space centered finite-difference scheme to the system of equations (3.10) and (3.11), and follow an elimination procedure like the one used for deriving (3.25) we obtain:

$$\frac{h_j^{n+1} - 2h_j^{n-1} + h_j^{n-3}}{(2\Delta t)^2} - gH\frac{h_{j+2}^{n-1} - 2h_j^{n-1} + h_{j-2}^{n-1}}{(2\Delta x)^2} = 0 \tag{3.26}$$

Both the finite-difference schemes (3.25) and (3.26) are second order approximations to the wave equation. However, in (3.25) the second time derivative is approximated using values at three consecutive instances of time; in (3.26) it is approximated using values at every second instance of time, that is, at time intervals $2\Delta t$. Thus, while the time-step required for stability with the leap-frog scheme was half that with the forward-backward scheme, (3.26) shows that the variables at every second instant of time can be omitted[2, p54-55]. This temporal distribution of variables is called time staggering. The time staggered leap-frog scheme uses the same amount of computation time as the pure forward-backward scheme.

3.3.5 Two-Dimensional Time Staggering

Next we consider an extension of the two-dimensional system of linearized shallow water equations, by adding Coriolis terms to (3.21) and (3.22):

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} - fv = 0 \tag{3.27}$$

$$\frac{\partial v}{\partial t} + g\frac{\partial h}{\partial y} + fu = 0 \tag{3.28}$$

$$\frac{\partial h}{\partial t} + H(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}) = 0 \tag{3.29}$$

If this system of equations is approximated using the E-grid of figure 3.5, the leap-frog scheme and centered space differencing with all the variables calculated at every instance of time, then two independent solutions would be obtained. The solution involving the variables of the Eliassan space-time grid in figure 3.6 would be independent of the solution involving the variables that we left out. Thus, the space-time grid formed by using the E-grid at every instance of time can be considered a

superposition of two elementary subgrids of the figure 3.6 type. However figure 3.6 can also be considered as a superposition of two subgrids, called Richardson grids, where in each of these Richardson grids only the height is kept at one instance of time and the velocity components at the next.

A single Richardson grid is considered a time staggered version of the C-grid of figure 3.5 and suffices for the solution of the pure gravity-wave system (equations (3.21), (3.22) and (3.23)). Thus, using the above difference system, in an Eliassan-grid the pure gravity wave system has two independent solutions, while for the extended gravity wave system these solutions are coupled only by the two Coriolis terms[2, p53].

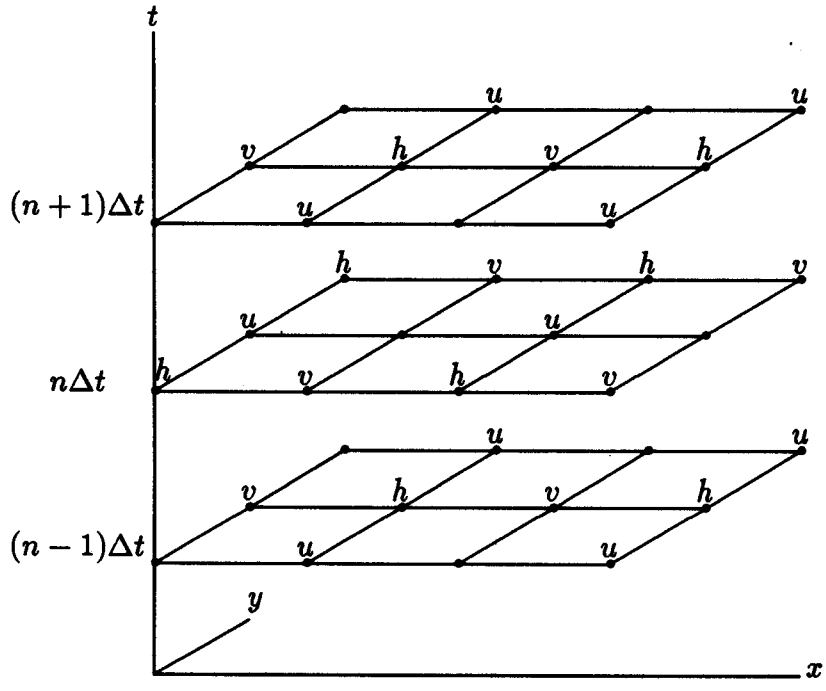


Figure 3.6: A space-time (Eliassan)-grid staggered in both space and time

3.4 Computational Economy of Explicit Schemes

With equal resolution, i.e., with equal wave length of the shortest resolvable wave, all rectangular grids require about the same computational effort per time step. Namely, the total number of tendencies of the dependent variables that have to be calculated does not depend on the grid choice. However, on grids which require more averaging in order to calculate the pressure gradient force and divergence terms, the gravity waves are decelerated, and consequently, longer time steps can

be used with the explicit time differencing schemes. Unfortunately, higher economy is thus achieved at the expense of reduced accuracy.

An inconvenient feature in case of gravity waves is the long computation time required for a solution using explicit schemes for time differencing. The time step imposed by the stability criterion for explicit schemes is generally considered to be much less for an accurate integration of slower quasi geostrophic motion. With these time steps the errors due to space differencing are much greater than those due to time differencing.

A forward-backward scheme is comparable in computation time with the leap-frog time differencing in the Eliassan grid, but with time-steps twice those allowed for the leap-frog scheme. Even these time-steps are considerably shorter than those required for accurate integration of quasi-geostrophic motions and even with these economical schemes the time differencing error is still much less than the space differencing error for typical current atmospheric models.

The computational efficiency can be further improved by a suitable choice of the time integration scheme. Today, there are two widely accepted procedures offering about the same economy. These are semi-implicit and split explicit approaches. A description of the semi-implicit method can be found in appendix B.

3.4.1 Explicit Splitting

The complexity of the system of hydrodynamic equations, that is, the simultaneous presence of a number of physical factors, may cause some difficulties. First, applying an implicit scheme we would obtain a system of equations for variables at instance of time $n+1$ that is practically impossible to solve. Second, if different physical factors are present in this system, we will normally wish to use different schemes for terms associated with them. Thus considering the following linearized system with advection and gravity wave terms:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} + c\frac{\partial u}{\partial x} = 0 \tag{3.30}$$

$$\frac{\partial h}{\partial t} + H\frac{\partial u}{\partial x} + c\frac{\partial h}{\partial x} = 0 \tag{3.31}$$

we might wish to use one scheme for the advection terms and another scheme for the gravity wave terms. In such a situation, even though both of the schemes to be used are stable if considered separately, we can not be sure that the scheme obtained as a combination of the two will also be stable. These problems can be avoided by using the splitting method. The idea of this method is to construct schemes for a complex system of equations so that within each time-step the system is split into a number of simpler subsystems, which are then solved consecutively. In the case of (3.30) and (3.31), within a given time-step, we could first solve the system of

advection equations:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \tag{3.32}$$

$$\frac{\partial h}{\partial t} + c\frac{\partial h}{\partial x} = 0 \tag{3.33}$$

Denote the provisional values h^{n+1}, u^{n+1} obtained in this way by u^*, h^*. Use these values at the beginning of the time-step for solving the remaining subsystem:

$$\frac{\partial u}{\partial t} + g\frac{\partial h}{\partial x} = 0 \tag{3.34}$$

$$\frac{\partial u}{\partial t} + H\frac{\partial u}{\partial x} = 0 \tag{3.35}$$

The values h^{n+1}, u^{n+1} obtained after solving also this other subsystem, are now taken as actual approximate values of these variables at time instance $n+1$. The procedure is repeated in each following time-step.

When applying the splitting method, we do not necessarily have to use equal time-steps for each of the subsystems. This may well be the main advantage of the splitting method: we can choose a relatively long time step for the subsystem modeling a slow process, advection in the present example, and then use a number of smaller steps to calculate the faster process. Since the advection process is the most expensive in computation time, within the primitive equations, significant economies can be accomplished in this way. A disadvantage of this method is that calculation of the effects of different physical factors one at a time usually leads to an increase in the truncation error[2, p58-59].

Chapter 4

The Hirlam Model

For those who are not familiar with the Hirlam model it might be easier to read appendix C before reading this chapter.

The HIRLAM LEVEL 1 forecast model is a numerical weather prediction model based on the Swedish and Danish limited area model (S/D-LAM). The model includes the integration of the primitive equations and consequently space differencing. Because these calculations are performed in the procedure DYN, DYN is of our main interest in this chapter.

The Hirlam model has been totally recoded in order to improve the efficiency on vector-machines and to prepare the dynamical part so that alternative numerical methods can be implemented more easily[10].

4.1 The Continuous Hirlam Model

Horizontal coordinate system The model is programmed for a spherical rotated coordinate system (λ, θ) (longitude, latitude), but in the formulation two metric coefficients (h_x, h_y) (functions of θ and λ respectively), have been introduced. This is done to prepare the model for any orthogonal coordinate system or map projection with axis (x, y)[10, p1.5].

Vertical coordinate The model is formulated in the general hybrid-coordinate in the vertical

$$\eta = \frac{A}{p_o} + B \tag{4.1}$$

where p_o the reference pressure for the vertical coordinate. The sigma-coordinate is obtained by substitution of $A = p$, $p_o = p_s$ and $B = 0$ into (4.1) (p_s is the surface pressure). Thus the sigma-coordinate is pressure normalized with surface pressure. We are expected to give speed-up expectations for Hirlam using sigma-coordinates.

Prognostic variables For each time instance, in the integration procedure of the primitive equations, the following variables are calculated:

- p_s - surface pressure
- T - temperature
- q - moisture
- u - wind in the east direction
- v - wind in the north direction

4.1.1 The Continuous Equations

The continuous equations are given, in Eulerian formulation, for a spherical rotated coordinate system and the general vertical hybrid coordinate η. An explanation of the symbols can be found in the Documentation Manual of the Hirlam Level 1 Forecast Model. The continuous equations are:

- The equations of motion:

$$\frac{\partial u}{\partial t} = (f+\xi)v - \dot{\eta}\frac{\partial u}{\partial \eta} - \frac{R_d T_v}{a h_x}\frac{\partial ln(p)}{\partial x} - \frac{1}{a h_x}\frac{\partial}{\partial x}(\phi + E) \tag{4.2}$$

$$\frac{\partial v}{\partial t} = -(f+\xi)u - \dot{\eta}\frac{\partial v}{\partial \eta} - \frac{R_d T_v}{a h_y}\frac{\partial ln(p)}{\partial y} - \frac{1}{a h_y}\frac{\partial}{\partial y}(\phi + E) \tag{4.3}$$

where

$$\xi = \frac{1}{a h_x h_y}\left(\frac{\partial}{\partial x}(h_y v) - \frac{\partial}{\partial y}(h_x u)\right) \tag{4.4}$$

$$\phi = gz \tag{4.5}$$

$$E = \frac{1}{2}\left(u^2 + v^2\right) \tag{4.6}$$

The linear and non-linear horizontal advection terms are contained in the first and fourth sub-term of (4.2) and (4.3), because:

$$\xi v - \frac{1}{a h_x}\frac{\partial E}{\partial x} = -\frac{v}{a h_y}\frac{\partial u}{\partial y} - \frac{u}{a h_x}\frac{\partial u}{\partial x} \tag{4.7}$$

$$-\xi u - \frac{1}{a h_y}\frac{\partial E}{\partial y} = -\frac{u}{a h_x}\frac{\partial v}{\partial x} - \frac{v}{a h_y}\frac{\partial v}{\partial y} \tag{4.8}$$

The remaining part of the fourth subterms is a result of the vertical coordinate transformation as is pointed out in appendix C. The second sub-terms represent linear vertical advection terms and the third the horizontal pressure

force terms. The Coriolis terms are contained in the first sub-term of (4.2) and (4.3). The quantities ξ, $f+\xi$ are called the relative and absolute vorticity, respectively. Equations (4.2), (4.3) and (4.13) are equivalent with the primitive (three-dimensional) equations of motion.

- The continuity equation:

$$\frac{\partial}{\partial \eta}\frac{\partial p}{\partial t} + \nabla \cdot \left(\vec{V}\frac{\partial p}{\partial \eta}\right) + \frac{\partial}{\partial \eta}\left(\dot{\eta}\frac{\partial p}{\partial \eta}\right) = 0 \tag{4.9}$$

where the divergence operator is defined as:

$$\nabla \cdot \vec{V} = \frac{1}{a h_x h_y}\left(\frac{\partial}{\partial x}(h_y u) + \frac{\partial}{\partial y}(h_x v)\right) \tag{4.10}$$

For a derivation of (4.9) we refer to appendix C.

- The thermo dynamical equation:

$$\frac{\partial T}{\partial t} = -\frac{u}{a h_x}\frac{\partial T}{\partial x} - \frac{v}{a h_y}\frac{\partial T}{\partial y} - \dot{\eta}\frac{\partial T}{\partial \eta} + \frac{\kappa T_v w}{(1+(\delta-1)q)p} + P_T + K_T \tag{4.11}$$

- The moisture equation:

$$\frac{\partial q}{\partial t} = -\frac{u}{a h_x}\frac{\partial q}{\partial x} - \frac{v}{a h_y}\frac{\partial q}{\partial y} - \dot{\eta}\frac{\partial q}{\partial \eta} + P_q + K_q \tag{4.12}$$

The first three sub-terms of (4.11) and (4.12) are the linear advection terms, where the transported variables are temperature and moisture, respectively.

The hydrostatic assumption is formalized by the hydrostatic equation:

$$\frac{\partial \phi}{\partial \eta} = -\frac{R_d T_v}{p}\frac{\partial p}{\partial \eta} \tag{4.13}$$

For a derivation of (4.13) we refer to appendix C.

Sigma-coordinates Substitution of the sigma-coordinate $\sigma = \frac{p}{p_s}$ for η in (4.9) gives:

$$\frac{\partial p_s}{\partial t} = -\nabla \cdot (p_s \vec{V}) - \frac{\partial}{\partial \sigma}(p_s \dot{\sigma}) \tag{4.14}$$

The boundary conditions on "the vertical velocity",

$$\dot{\sigma} = \frac{d\sigma}{dt}$$

are:

- At the surface; $p = p_s$; hence $\sigma = 1$ and $\dot{\sigma} = 0$.

- Top of the atmosphere; $p = 0$; hence $\sigma = 0$ and $\dot{\sigma} = 0$.

With these conditions (4.14) may be integrated over the entire atmosphere, or over part of it, as follows

$$\frac{\partial p_s}{\partial t} = -\int_0^1 \nabla \cdot (p_s \vec{V}) d\sigma \tag{4.15}$$

which is an equation for the surface pressure tendency.

Integrating from $\sigma = 0$ to an arbitrary level σ will give an equation for the vertical velocity $\dot{\sigma}$ at level σ

$$\sigma \frac{\partial p_s}{\partial t} = -\int_0^\sigma \nabla \cdot (p_s \vec{V}) d\sigma - p_s \dot{\sigma} \tag{4.16}$$

If the value of $\frac{\partial p_s}{\partial t}$ is obtained from (4.15), it can be used in (4.16) to calculate $\dot{\sigma}$, which in turn can be used in the momentum equations, the thermo-dynamical equation and the moisture equation. These steps are part of the procedure to determine the local time changes of $\vec{V}$, T and q in the Hirlam numerical weather prediction model.

The reason that $\dot{\sigma} = 0$ at the surface, is that the flow along the surface must be horizontal (i.e. $\sigma = 1$). It is this simplified lower boundary condition that provides the advantage of the σ-coordinate. An immediate result is a concise form (4.15) for computing the surface pressure tendency.

4.2 The Discrete Hirlam Model

The discretization of the continuous model is performed by replacing a bounded three dimensional continuous domain by a three-dimensional grid (34x34x9). Thus the finite-difference method is used, although the discrete Hirlam model was originally proposed to be a spectral model.

Horizontal grid structure The horizontal grid-structure, a staggered Arakawa C-grid, is depicted in figure 4.1.

Vertical grid structure The vertical grid structure is depicted in figure 4.2.

Space difference scheme Hirlam uses centered space differencing and therefore the accuracy of the finite difference approximations is second order in space.

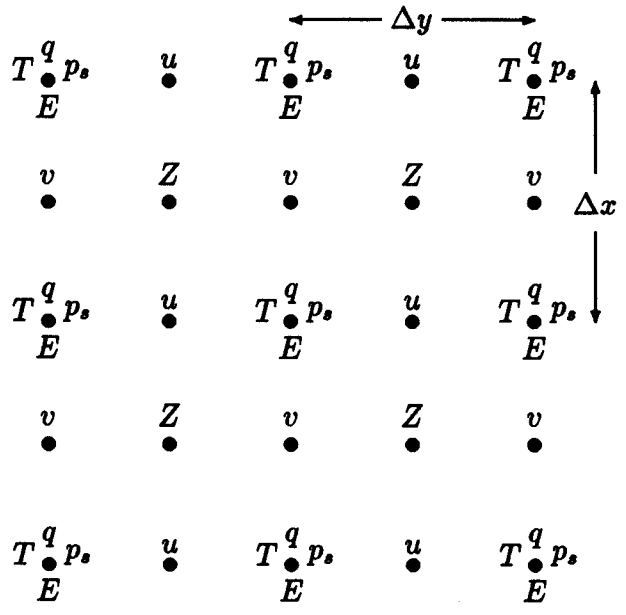


Figure 4.1: Horizontal grid of the HIRLAM LEVEL 1 limited area model

Time difference scheme Hirlam uses leap-frog time-differencing (chapter 3) to calculate the (explicit) values for the prognostic variables, explicit splitting (chapter 3) and optionally implicit adjustment (appendix B) as a correction to explicit values. Finally the Asselin time-filter (appendix C) is used to remove the computational mode in time associated with the leap-frog scheme. Because of using leap-frog time differencing, the accuracy of finite-difference approximations is second order in time as well.

Boundary treatment Boundary relaxation (appendix C) with a 6-hourly data input interval and relaxation-factor α chosen as

$$\alpha = 1 - tanh\left(\frac{2j}{N-4}\right)$$

where j is the number of gridpoints from the boundary point and N being the width of the boundary relaxation zone.

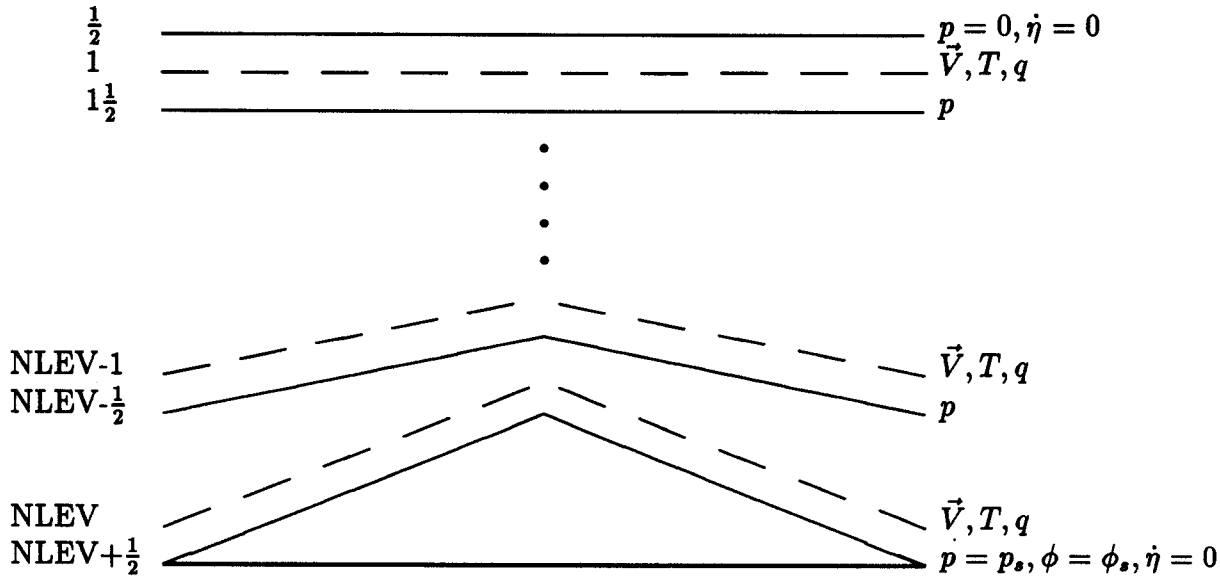


Figure 4.2: Vertical structure of the HIRLAM LEVEL 1 limited area model

4.3 The Sequential Hirlam Program

4.3.1 Program structure

begin (* main *)
read slabs from start data file unit
read slabs from boundary file unit
create first pair of boundary data sets
check humidity for critical values
if *nlsimp* **then** store $ln(p_s)$ (* instead of p_s *)
calculate boundary relaxation function
copy start data to time-step n
compute coriolis parameter and mapping factors
initialize semi-implicit scheme
for *nsteps*:=1 **to** *nstop+1* **do**
begin (* time-loop *)
(* leapfrog (semi-implicit) scheme cycle *)
(* run one extra time step for time-filtering of data *)
(* dynamical process calculations - DYN *)
computation of surface pressure tendency
for k:=lowest level **to** highest level **do**
begin (* big outer loop *)
(* calculate new values of p_s, T, u, v and q *)
compute constants

```
        virtual temperature at level k
        compute geopotential at level k
        compute divergence at level k
        computation of vertical advection terms on T, q, u and v
        compute omega for energy conversion term
        compute absolute vorticity + energy
        add horizontal advection to the tendencies
        update geopotential part two from level k to level k - 1/2
        update sum of divergence from lowest level to level k
    end(* DYN *)
    if not nlphys then make an explicit time-step in tstep on T, u, v and q
    else physical process calculations (* PHCALL *)
    if nlhdif then horizontal diffusion on T, u, v and q (* HDIFF *)
    (* method is non linear fourth order diffusion *)
    if nlsimp then semi-implicit calculations and boundary relaxation (* SICALL *)
    (* correct the adiabatic, explicit values *)
    else boundary relaxation
    (* the explicit case *)
    if nlstat then compute statistics (* STATIS *)
    Asselin time-filter on p_s, T, u, v and q
    (* to remove computational mode associated with the leap frog scheme *)
    copy n + 1 to n on p_s, t, u, v and q
    print out statistics
    check writeup time
    check boundary time
    if nlsimp then store ln(p_s) (* instead of p_s *)
    check forecast time at time-step nstop
  end (* time loop *)
end (* main *)
```

4.3.2 Statistics

Each time-step procedure STATIS is invoked, which outputs the CPU-time and the percentage of total CPU-time of the five most time-consuming procedures. Figure 4.3 shows partial output, which is generated by running the original Hirlam program on the HCX-9.

The current Hirlam program has a bug (by the time of this writing) which causes a crash in time-step 3. This bug appeared when we ran the program, with the implicit adjustment calculations (SICALL) turned off.

```
MAIN DIMENSIONS:
----------------
 NLON  =  34
 NLAT  =  34
 NLEV  =   9
 NSLAB =  16

NSTEP    0 BEGINS HERE:
   TDATA       NBDTIM       NBDDIF
      .0          900        43200

DYN   :        2.2833328247 S       18.3892574310 P
PHCALL:        8.7833337784 S       70.7382583618 P
HDIFF :         .6833324432 S        5.5033483505 P
SICALL:         .3500003815 S        2.8187949657 P
STATIS:         .2500000000 S        2.0134227276 P
1 TIMESTEP TOOK       12.4166669846 SECONDS

NSTEP    1 BEGINS HERE:
   TDATA       NBDTIM       NBDDIF
   900.0         1800        43200

DYN   :        2.1999988556 S       17.9347763062 P
PHCALL:        8.6166667938 S       70.2445755005 P
HDIFF :         .6999988556 S        5.7065134048 P
SICALL:         .3333339691 S        2.7173969746 P
STATIS:         .2666664124 S        2.1739113331 P
1 TIMESTEP TOOK       12.2666645050 SECONDS

NSTEP    2 BEGINS HERE:
   TDATA       NBDTIM       NBDDIF
  1800.0         2700        43200

DYN   :        2.2000026703 S       17.9348030090 P
PHCALL:        8.6499977112 S       70.5162734985 P
HDIFF :         .6833343506 S        5.5706596375 P
SICALL:         .3166656494 S        2.5815131664 P
STATIS:         .2666664124 S        2.1739106178 P
1 TIMESTEP TOOK       12.2666683197 SECONDS

NSTEP    3 BEGINS HERE:
   TDATA       NBDTIM       NBDDIF
  2700.0         3600        43200

Negative argument for ALOG        -.2243472636   19
```

Figure 4.3: Partial output of the original Hirlam program.

4.3.3 Computer code organization

During the design of the model it was decided to keep all three instances of time of the prognostic variables in core. The main reason for this was to have a more flexible data-organization compared with the old code which had only one latitude line in core. It has the additional advantage that the model will run efficiently on those vector machines which need very long inner loops. Moreover, it was necessary to do so to prepare the model for planned future experimentation which needs the whole horizontal field in core.

This will to some extent make a limitation for the size of the area (or rather the number of horizontal points and levels) and two steps have been taken to counteract this. First, only those points which are needed in the boundary fields are stored and pay the little extra overhead due to inconvenient data storage. Second, the dynamic and physical part of the model are also organized somewhat differently in order to minimize the needs of the workspace. In the dynamics (DYN) the computations are done mainly layer by layer scanning through all the horizontal points in the innermost loops. However, for the physical subroutine (PHYS), only a so-called "slab-area", which cover a number of latitude lines and all levels, is considered and the physical subroutine is called several times during one time-step to complete the computations for all slab-areas. In this way the size of the slab-area can be varied so that the workspace for the physics will run efficiently with long inner loops[10, p3.1].

Chapter 5

The Transputer

5.1 Why Transputers?

Like many other simulation problems, numerical weather prediction requires vast amounts of processing. Furthermore, the demand for processing power is hard to fulfil, because more processing power can always be used to increase accuracy or response times.

Nowadays many extensive grid-based computations are performed using vectorprocessing supercomputers. These machines are in general very expensive and therefore only a few big corporations or authorities can afford such machines.

For many grid-based computations, the capacity of a "super computer" can be achieved by using a large number of small computers, each working on a small area of the grid, interconnected by a network. The performance/price ratio for small computers is, in general, better than for supercomputers. If the overhead due to inter-processor communication is limited, a network af small computers could provide the performance of a much bigger computer at lower costs.

The *transputer* microprocessor is especially well suited for network applications. It has been designed to make the use of parallel processes and inter-processor communication easy to use and fast. Moreover, transputers are relatively cheap and very little extra hardware is necessary to build a network of them. Therefore, networks of low cost are easy to build using transputers.

Each transputer in a transputer-network can possibly run a different program. This gives such a system a great flexibility, which could for instance be used to perform different calculations at the border of a grid. With vector machines such exceptions are clumsy and inefficient to implement.

Computations with embedded conditions also cause problems for vector computers, but not for a transputer-network. The conditional expressions at different gridpoints possibly evaluate to different truth values. A vector-computer therefore has to perform both branches of a conditional statement for each gridpoint and must select the desired one afterwards. This inefficiency can be avoided with a distributed

system of transputers.

Another advantage of such a network is the ability to increase the performance gradually by adding more computers. However, this ability is possibly limited by the number of gridpoints, because a number of extra communications may become necessary when the calculations for a gridpoint are divided across different transputers.

In the next sections, we will look at transputers in greater detail.

5.2 What is a Transputer?

The British chip manufacturer INMOS recently developed a new range of microprocessors [13]. These microprocessors are especially designed for building parallel computers. These chips were named *transputers*, because they are, according to INMOS, similar to transistors in the sense that both are elementary building blocks for complex systems. The first transputers were produced in 1985.

At present there are three different types of transputers available. The transputer T212 is a 16-bit processor, which is mainly meant to be used in controller-applications. Two other members of the transputer family are the T414 and the T800; they are both 32-bit processors. Transputers have a RISC-architecture (*Reduced Instruction Set Computer*) and are available with performances of 5 to 10 MIPS. Transputers have a number of nice features that make them especially well suited for multiprocessor environments.

A transputer is in fact a complete computer on one chip. It contains a processing unit, 4K of very fast (50 ns) static RAM (2K for T414), 2 timers, 4 high speed serial links with DMA (*Direct Memory Access*) capability, operating bidirectionally at 10 or 20 Mbit/s and a memory interface for controlling up to 4 Gbyte of external memory. As an extra the T800 has an integrated, very fast (1.5 Mflops for a 20 MHz device [15]) floating point unit.

5.3 Properties of Transputers

What makes a transputer so well suited for building distributed systems are the four integrated serial links. These links enable high-speed communication and synchronization between different transputers. Because the links are serial, no more than a simple cable is needed to connect two transputers.

Communication via these links is completely controlled by hardware. After a process has set up a link-communication, the communicating process is suspended until the communication is completed. The hardware takes care of reading and writing data from and to memory by DMA. This DMA does hardly decrease processing speed, even when all four links operate at the same time.

Another nice feature is the on-chip RAM. This memory is fast enough to keep

up with the processing unit, whereas external memory references will slow down the processor. For some applications, it might not even be necessary to use external memory. In that case, the cost for building a network of transputers is greatly reduced. The required interconnections for such a network are minimal; only the power supply, clock and reset signal and of course the four links have to be interconnected. No extra components are necessary in this case.

A transputer can boot from a ROM or from a link. For members of a transputer network, the latter method has the advantage of not requiring a boot ROM for every transputer. The desired boot method is selected by wiring a pin to logic "0" or "1". When booting from a link is selected and the transputer is reset, the first block of data the transputer receives via one of its links is loaded into the internal RAM and executed. To start up a whole network, one must first boot one transputer which has a connection with the host. After that, this transputer is instructed to send a boot-program to its neighbors, etc.

The transputer has the built-in capability to manipulate processes. There are instructions to start and stop processes [14]. These instructions manipulate a linked list which contains all the processes which are ready to execute. A simple form of time-sharing is also built-in; processes that are running longer than some fixed time-interval are moved to the end of the list and the next process is picked from the beginning of the list for execution. When a process has to wait for a timer or for a communication, it is temporarily removed from the list of processes and the next one is started. A context-switch between two processes takes, because it is built-in, less than 1 μs. It follows that the use of processes is very simple and also does not lead to much overhead.

Communication between processes is also very simple. With a single instruction one can send or receive a message, using a *channel.* Communication via a channel takes place unidirectionally, according to the so-called *rendez-vous* principle: the sender and receiver should simultaneously engage in the communication. If one of them is not ready to communicate, the other must wait. A waiting processes will automatically be removed from the list of ready processes. Communication is the only way to synchronize processes. Because an identification of the suspended process is stored in a memory word associated to the channel, it is not possible for more than one process at either side to use a channel at the same time.

This form of asynchronous communication gives rise to data-driven execution. Although asynchronous communication is not as efficient as synchronous communication between processes which are synchronous in time, it is a convenient programming tool, which enables the programmer to abstract from instruction timings.

There are two types of channels: *internal* channels for communication between processes on the same transputer; *external* channels for communication between processes on different transputers. External channels are mapped onto a link; internal channels are mapped onto an arbitrary memory word. The same communication instructions can be used for internal and external channels; for the program, the difference is transparent.

Because of the hardware restrictions mentioned above, at most one external channel in each direction can be mapped onto a link. In order to get more "external" channels, it is necessary to implement a number of "virtual" channels on one link. Virtual channels can be implemented by using "multiplexer" and "demultiplexer" processes on each side of the link. A multiplexer process communicates at one side with a number of processes via internal channels and at the other side with a demultiplexer on another transputer via the link.

The communication between a multiplexer and the corresponding demultiplexer at the other side of a link, can be done in two different ways. The first approach is to send an identification of the virtual channel with each value. Another approach is to agree on sending values in a fixed order. Of course, the latter method can only be used if it is known beforehand how the virtual channels will be used.

Chapter 6

Grid-Based Computations on a Network of Transputers

6.1 Choosing a Network Topology

When designing a network for a parallel computation, one should keep in mind two important objectives:

1. It should be possible to divide the computation in such a way that each processor has to do approximately the same amount of work, because the time to complete the whole computation equals the time the longest computation part needs to complete.

2. The need for communications between different processors, and in particular the waiting time caused by these communications should be minimized.

In the case of a computation on a grid, the first objective is achieved easily. The calculation times at the gridpoints will in many cases be approximately the same, so the grid points can be distributed evenly across the available processors

The second objective might cause more problems, because it is sometimes possible to avoid communications by modifying the algorithm. Communications are then replaced by some extra calculations. Thus there is a tradeoff between communication and computation. In order to achieve maximum performance, one must know precisely what the communication costs and the cost of the extra calculations are, and how they relate to each other. Moreover, one must take into account that communication between two processors which are not directly connected requires the processors in between to involve in the communication. This places an extra burden on these processors.

Many algorithms on grids, including the finite differencing methods used for numerical weather prediction require only local information on each grid point; that is information stored at that grid point, or information from one of its direct "neighbors". For these cases the obvious network architecture is a grid of processors, where

each processor will do the calculation for some rectangular area. It depends on the relationship between the amount of vertical and horizontal communications needed, what the optimal shape of these rectangular areas is. Although other distributions of gridpoints over the available processors are possible, they will result in more external communications, and thus in more overhead.

Because a transputer has only four links, it is not possible to form a grid of transputers of dimension 3 or more. A more-dimensional grid however, can always be projected onto a 2-dimensional grid, which can then be distributed over a 2-dimensional grid of transputers.

We will use only orthogonal projections. The choice which dimension(s) to project depends on a number of properties of the program, such as:

- The amount of communication in each direction. The dimension in which the most communication takes place, is a good candidate for projecting, because this gives the greatest saving of communications between transputers. External communication is always slower than internal communication, so lowering the number of these communications has a positive effect on the speedup.

- The shape of the space of gridpoints. By choosing the dimensions with the least number of gridpoints for projecting, the resulting 2-dimensional grid has a maximum number of gridpoints. As a result, more transputers can be used without being forced to split gridpoints computations over different transputers, which possibly results in a large number of extra communications.

- The use of global data. To avoid data-duplication, gridpoints using the same global data should preferably reside on the same transputer.

In the remainder of this chapter, we will consider grid-based computations on a network of transputers, in which only local communications take place. The grid may be 1- or 2-dimensional, but we will concentrate on 2-dimensional grids. The total number of gridpoints will be denoted by N, the total number of transputers by P. P will never be greater than N.

Each transputer will do the computations for some rectangular subgrid of the total grid. We will call these gridpoints the *local grid* of a transputer. The actions needed for calculating the new state of one gridpoint consist of a sequence of communication steps and calculation steps. The communication steps are needed if the next calculation step needs some value from a "neighbor" gridpoint. Because the actions for each gridpoint are equal, this implies that at the same time the neighbor on the opposite side also needs a value from this gridpoint. Thus one communication generally involves a send and a receive action with two opposite neighbors.

In the next sections we will first treat the impact of staggering on projecting a physical grid on a grid of transputers. After that, we will look at different ways of calculating the gridpoints of a local grid on one transputer. In the last sections we will draw our attention to the effect of the number of transputers on the total execution time.

6.2 Implementation of Staggering on a Grid of Transputers

The proposed implementation of staggering in section 3.3.1 yields an efficient program for both vector computers and a grid of transputers. When calculating a space centered (with respect to a staggered variable) finite-difference scheme, only one neighbor variable is needed. Thus besides a gain of a factor two in computation time by removing the computational mode, the proposed programming model reduces both internal and external communication by a factor two compared to the program that implements staggering directly.

In this chapter we will use the term "gridpoint" to denote a *logical* gridpoint, as is introduced in section 3.3.3.

6.3 Two Computational Models

Each transputer has to do the calculations for the gridpoints of its local grid. It is, in general, not possible to calculate the new state of each gridpoint one after the other. The calculations are mutually dependent, so they must be performed quasi parallel, with interleaved communication steps.

The most obvious way to implement these calculations on a transputer is to create a separate process for each gridpoint. The exchange of values between neighbor gridpoints can be accomplished by using internal "channels" for inter-process communication. This model will be called the parallel model.

The other possibility is to use one process for all gridpoints on a transputer, by storing the gridpoint data in arrays indexed by the relative position of the gridpoint. Each calculation step will typically be a loop which iterates over the grid and communications are realized simply by using array subscripts. We will refer to this second model as the sequential model. In the following sections, we will discuss these two different models in detail.

6.3.1 The Parallel Model

The parallel model is a conceptual simple one. One writes a program to be executed at one gridpoint using only local variables. Values from neighboring gridpoints are requested using communication via channels.

For each gridpoint a process executing this program has to be created and run in parallel with the others. Using the built in timesharing facilities of a transputer this is easy to do. There is no need to control or synchronize these processes, they synchronize automatically through their communications.

Figure 6.1 shows as an example a parallel implementation of a 2-dimensional grid-based computation.

```
FOR t := 1 TO Nsteps
   BEGIN
      BEGINPAR
         BEGIN
            SEND U TO east;
            SEND U TO west;
            SEND U TO north;
            SEND U TO south
         END;
         BEGIN
            RECEIVE U_west FROM west;
            RECEIVE U_east FROM east;
            RECEIVE U_south FROM south
            RECEIVE U_north FROM north;
         END
      ENDPAR;
      U := f(U, U_west, U_east, U_south, U_north)
   END
```

Figure 6.1: Parallel implementation of a 2-dim. grid-based computation

The gridpoints processes at the boundary of a local grid, have to communicate with their counterparts on a neighbor transputer. Because there is only one communication link at each side, it is necessary to implement a number of "virtual channels" on one physical link by using multiplexers (see section 5.3). Each gridpoint process at the boundary will then communicate with its corresponding gridpoint process on another transputer via a virtual channel.

Multiplexers at the boundary of the global grid must be special. They should only simulate a multiplexer process and should not communicate over their link. These dummy multiplexers should ignore values they receive from the gridpoint processes and send values corresponding to the boundary conditions used to a gridpoint processes requesting a value.

Communications deserve special attention. As noted before, each communication step consists of a send action and a receive action from the opposite direction. Consider what happens when these two actions are performed sequentially. When for example each process starts with a send to their right neighbor, they all have to wait until their right neighbor performs a receive, which results in a deadlock situation. This situation can be prevented by executing the send and the receive actions concurrently.

Another approach is to run two different variants of the program for alternating gridpoints. The only difference between these two variants should be that the first

always starts with send actions, while the second starts with the receive actions.

When using the first approach, it is possible to create the two processes for the send and receive actions afresh for each communication, but this causes some overhead. The alternative is to create a send and a receive process once, and let them do all communications.

The problem with the latter approach is the synchronization needed between the calculations and the two communication processes. The only way to do this is using communications via a channel. Thus the second alternative saves two process creations per communication step, but at the cost of two extra channel communications.

A disadvantage of the parallel model is the data duplication caused by the communications. Each communication of a value in a particular direction requires an extra variable at each gridpoint. In section 6.4.1, the different variants of the parallel model are compared.

6.3.2 The Sequential Model

In the sequential model there is only one process per transputer and the data for all local gridpoints is stored in arrays. Each calculation step is performed for all local gridpoints, typically by iterating over the arrays. Internal communication steps are not necessary, because "neighbor values" can be found in the arrays. Only neighbor values that reside on another transputer deserve special attention. The simplest solution is to extend all arrays by one column and one row on every side. A communication step then consists of a copy af a complete row or column from a neighbor transputer to this extended border.

An important issue is the order of the iteration over the arrays. When for instance the calculation of some variable depends on the previous value of this variable at the gridpoint to the left, then one clearly should not calculate new values left-to-right but right-to-left. If the calculation depends on both the previous left and right values, we are forced to use a temporary array for the results of one row and copy the contents of this array back to the original array when the row is completely calculated.

Compared with the parallel model, there is some overhead caused by array subscripting and loop control. The overhead can be reduced by copying values that will be used a number of times into a plain variable, which can then be used instead of the original array subscript. Other possible optimizations are for instance the use of arrays of arrays instead of multidimensional arrays. In section 6.4.2, different optimizations of the sequential model are compared.

Figure 6.2 shows a sequential implementation of a 2-dimensional grid-based computation.

```
FOR t := 1 TO Nsteps
   FOR x := 1 TO GridX
      FOR y := 1 TO GridY
         BEGIN
            U_west := U[x-1][y];
            U_east := U[x+1][y];
            U_south := U[x][y-1];
            U_north := U[x][y+1];
            U[x][y] := f(U[x][y], U_west, U_east, U_south, U_north)
         END
```

Figure 6.2: Sequential implementation of a 2-dim. grid-based computation

6.4 Comparing the two Models

In order to compare the execution speed of the two models, we should construct test programs which perform the same computations on the same grid size, but using the two different computational models.

In these test programs, all variables used should be forced into the external RAM, for instance by declaring a dummy variable of 4K first. The internal RAM is much faster than external RAM, so the test results will not be accurate if some variables are in internal RAM while others are in external RAM. Furthermore, many applications, including the Hirlam program, require much more memory than 4K, so most variables will be in external RAM anyway.

Because the amount of link communication is the same for both models, it is possible to compare the two models using only one transputer. The external communications using links will therefore be omitted, but the parallel test program will include the four multiplexer processes.

Internal communications will take more time in the parallel model than in the sequential model. On the other hand, variable accesses in the sequential model will be slower. The decision which model to choose for a particular problem therefore depends on the amount of communications versus the amount of variable references. We will define Q to be the ratio of the number of references to local variables, and the number of communications.

The test programs will perform calculations on a 2-dimensional grid of size 10×10. Each test program consists of a communication step followed by a calculation step, which are performed 10 times in order to get more reliable results. The communication step consists of communications in all four directions. The calculation step is a loop with a calculation using four local variables. Q will be a parameter of the test programs; it will control the number of iterations of this loop. The total execution time is plotted as a function of Q.

The resulting graph will consist of a straight line, where the slope of this line gives information about the time needed to perform one iteration of the calculation loop. The vertical offset is the time needed to perform the communications (plus a little loop overhead).

As said before, a number of optimizations are possible for both models. For a comparison of the models, we should choose the fastest variants of the two models. This will be done in the next two sections.

6.4.1 The Optimal Parallel Model

The program `par.c` which can be found in appendix D, measures the time for three different variants of a parallel implementation of a grid-based computation. These three variants are implemented with the functions `par1`, `par2` and `par3`. The time is measured as a function of Q.

For the first two variants, the send and receive actions are performed by separate processes. There are two different ways to achieve this. The first method is to create these processes anew for every communication step; the second method is to create them outside the outer loop. The latter method has to use two extra channels per gridpoint to synchronize these processes with the calculation process. The third variant uses explicit alternating communication, caused by executing different code on adjacent gridpoints.

The reported times are net times; the constant startup time is subtracted from each measuring, so only the time for 10 loop-iterations is counted. The results in figure 6.3 show that for all values of Q, the function `par3` performs slightly better. The reason why the slopes of the plotted lines are not exactly parallel, is caused by the fact that the variables in the three functions do not have the same position in the function's "workspace". References to variables outside the first 16 words of the workspace are slower because they need an extra "prefix" instruction.

6.4.2 The Optimal Sequential Model

For the sequential model, a number of optimizations are possible. In the program `seq.c` (see appendix D), three different approaches are compared. The first approach (function `seq1`) uses normal 2-dimensional array subscripts to access a variable. The second approach (function `seq2`) uses a 1-dimensional array which is aliased to a column of the 2-dimensional array at the start of each column. The functions `seq3a` and `seq3b` implement yet another optimization. In these functions no array subscripts are used, but pointers instead. These pointers are initially set to point to the first array element and are incremented each iteration.

When the same array subscript or pointer dereference is used a number of times, it might be attractive to make a copy of the value into a plain variable, and to use this variable instead. The number of occurrences of a variable, above which this approach results in a gain in efficiency, depends strongly on the compiler used and

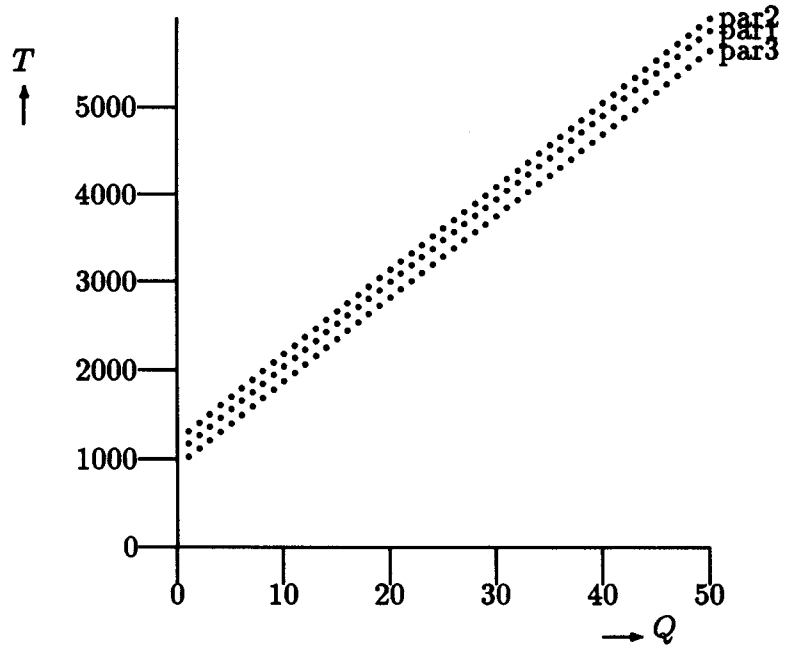


Figure 6.3: Results of Parallel Program

on the question whether the value is modified during the computations, in which case it has to be copied back.

With the program `optim.c` (appendix D), the minimum number of occurrences of a variable, for which this optimization is advantageous is determined. The values determined are valid only for the compiler we used [18]. Separate values are determined for 2-dimensional arrays, 1-dimensional arrays and pointers. Both the case that a variable is only read, and the case that a variable is modified and has to be copied back are covered. The next table shows the results:

	2-dim	1-dim	pointer
read-only	3	3	4
read/write	4	4	6

In the remainder of this section, we will assume that the number of times the same variable is used, is too small for these optimizations.

There are two versions of optimization using pointers, for the number of times a variable is used has a great impact on performance. This difference is caused by the increment operations which are needed for every variable used. The two functions therefore differ in the number of times a variable is used. The function `seq3a` simulates a computation where each variable is used only once; the function `seq3b` does the same for two references to each variable, so the number of increment operations is halved.

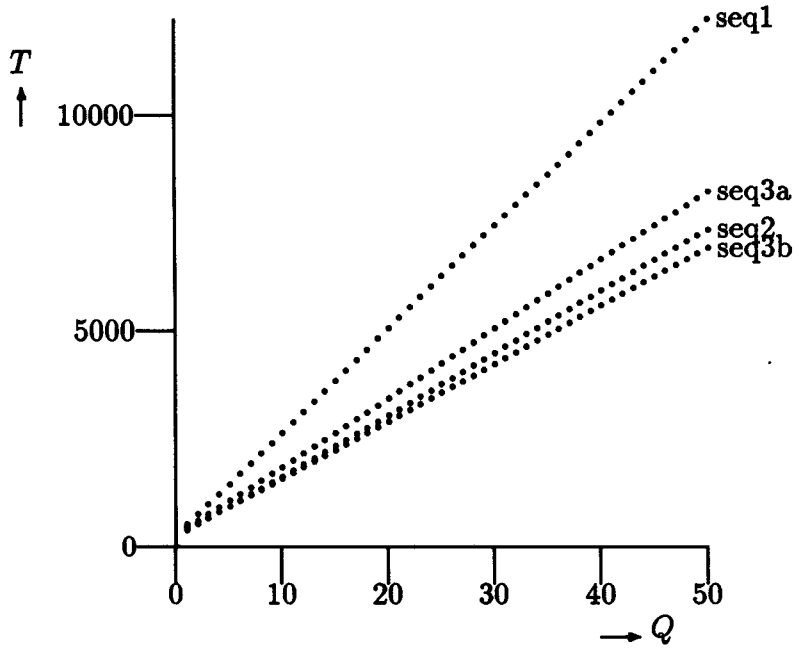


Figure 6.4: Results of Sequential Program

The results of the four different methods used in this program are plotted in figure 6.4. The function **seq3b** is a little faster than **seq2**, but **seq3a** is slower, although there is little difference. Because the performance of **seq2** is independent of the number of times each variable is used, this seems to be the most appropriate variant of the sequential implementation model.

6.4.3 Comparison of the two Optimal Models

When we compare the methods **par3** and **seq2** by drawing their results together in figure 6.5, we see that there is no absolute "winner". Function **seq2** performs better for $Q < 15$ and function **par3** performs better for $Q > 15$.

The figure also shows that the communication costs for **par3** are approximately three times as high as for **seq2** , but this is compensated by faster variable references for larger values of Q.

It is good to note however, that these results depend strongly on the compiler used. Because the slopes of the two lines in the figure are so close, the intersection point will rapidly move if another compiler generates different code.

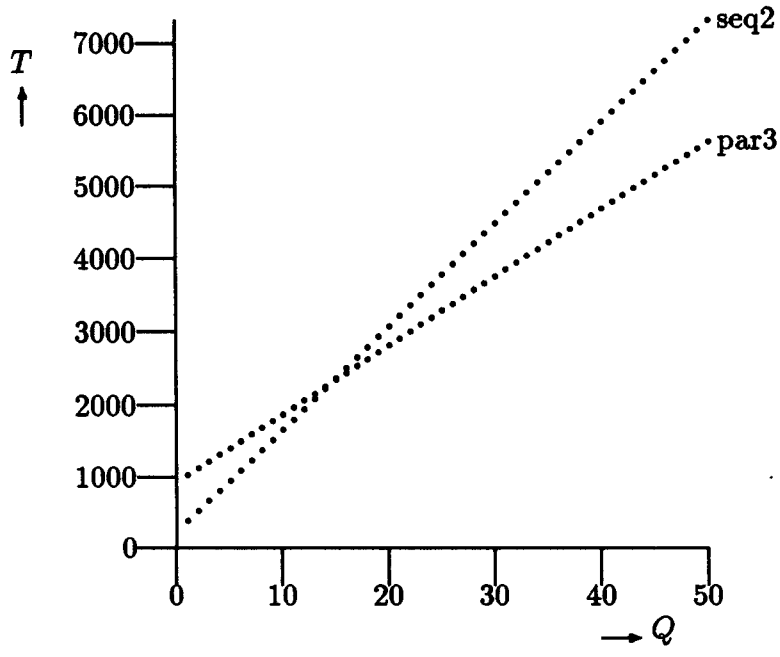


Figure 6.5: Comparison between parallel and sequential model

6.5 Using More Transputers

So far, we only looked at the organization of the calculations for the local grid of one transputer. Now we will take a look at the behavior of a network of transputers, together performing the computations for the whole grid.

First we need a few definitions, which are listed in table 6.1. We will assume

N_x	number of gridpoints in horizontal direction
N_y	number of gridpoints in vertical direction
N	total number of gridpoints; equal to $N_x \cdot N_y$
P_x	number of transputers in horizontal direction
P_y	number of transputers in vertical direction
P	total number of transputers; equal to $P_x \cdot P_y$
x	size of the local grid in horizontal direction
y	size of the local grid in vertical direction

Table 6.1: Definitions for Transputer Grids

that N_x is dividable by P_x, and N_y is dividable by P_y. For grid sizes where this is not the case, it is always possible to extend the grid at the boundary with dummy gridpoints. Figure 6.6 shows an example of a grid of size 8 × 8, which is divided

among 4 × 4 transputers. For this example, $N_x = N_y = 8$, $P_x = P_y = 4$ and $x = y = 2$.

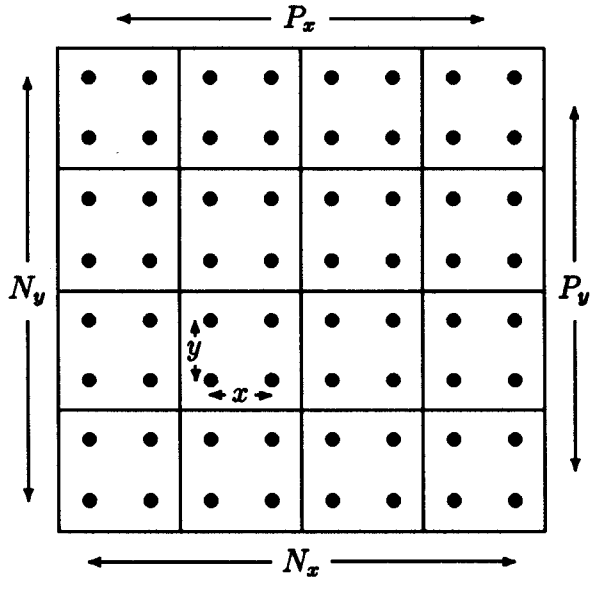


Figure 6.6: Example of a transputer-grid

Because all transputers perform the same calculations, they will run almost synchronously. Therefore, it suffices to use the execution time of one processor as a measure of the execution time of the complete transputer network. It is possible to express this time for the optimal variants of the two implementation models from the previous section as a function of the grid sizes x and y and a number of other parameters. These parameters apply to the local grid of size $x \times y$ of one transputer; they are listed in table 6.2.

We can now give a formula for the time needed to perform the computations for the optimal variant of the parallel model (par3):

$$\begin{aligned} T^{par} &= T_{calc} \cdot x \cdot y \\ &+ C_x \cdot T_{chan} \cdot x \cdot y \\ &+ C_y \cdot T_{chan} \cdot x \cdot y \\ &+ (C_x \cdot y + C_y \cdot x)(T_{chan} + F \cdot T_{link}) \qquad (6.1) \\ &= xy(T_{calc} + C_{xy}T_{chan}) + C_{ext}(T_{chan} + FT_{link}) \qquad (6.2) \end{aligned}$$

The first term of (6.1) represents the total calculation time for all local gridpoints. The second and third term give the time needed to do the internal communications in horizontal respectively vertical direction, including the communications with the multiplexer processes.

Application Dependent Constants

C	total number of calculation steps
C_x	number of communications in horizontal direction per gridpoint
C_y	number of communications in vertical direction per gridpoint
C_{xy}	total number of communications per gridpoint (abbreviation for $C_x + C_y$)
C_{ext}	total number of external communications (abbreviation for $C_x \cdot y + C_y \cdot x$)
V	number of subscripts of 1-dimensional arrays, needed only in the sequential model, per gridpoint ($V = Q \cdot C_{xy}$)
F	fraction of link-communication time which is not overlapped with internal processing ($0 \leq F \leq 1$)
T_{calc}	total calculation time for one gridpoint

Application Independent Constants

T_{subs}	additional costs for referencing a 1-dimensional array instead of a plain variable
T_{array}	time needed to access a "neighbor value" by doing a subscript in a 2-dimensional array
T_{chan}	time for performing one internal communication (a send plus the corresponding receive)
T_{loop}	time for performing one iteration of a loop over the local grid
T_{link}	additional costs for doing a link-communication instead of an internal communication

Table 6.2: Definitions for Execution Time Calculations

The last term denotes the time needed for external communications in horizontal and vertical direction, which are done by the (de)multiplexer processes. The parameter F has a value between 0 and 1. The value of T_{link} is approximately the time the hardware needs to send and receive a value over the links. Because the link-hardware works concurrently with the processor, link-communication is partly overlapped with processing. If $F = 0$, link-communication is completely overlapped; if $F = 1$, there is no overlapping at all.

If $x = 1$ or $y = 1$, the vertical respectively the horizontal multiplexer processes can be eliminated. In that case y can be replaced by $y - 1$ in the third term, respectively x can be replaced by $x - 1$ in the second term.

For the chosen sequential model (**seq2**), a similar formula can be derived:

$$\begin{aligned} T^{seq} &= (T_{calc} + V \cdot T_{subs} + C \cdot T_{loop}) \cdot x \cdot y \\ &+ (C_x + C_y) \cdot T_{array} \cdot x \cdot y \\ &+ (C_x \cdot y + C_y \cdot x) \cdot (T_{chan} + T_{link}) \qquad (6.3) \\ &= xy(T_{calc} + VT_{subs} + CT_{loop} + C_{xy}T_{array}) + C_{ext}(T_{chan} + T_{link}) \qquad (6.4) \end{aligned}$$

As with T^{par}, the first term of (6.3) gives the calculation time. Additional time compared with the parallel model is needed for subscripting (as a result of the optimization in an 1-dimensional array), and for iterating. The number of loops over the grid equals the number of calculation steps C, which also equals the number of communication steps.

The second term represents internal communication which is done by subscripting in a 2-dimensional array. The third term is like the last term of (6.1), except that the parameter F has disappeared. This is because in the sequential model communication is done sequentially, by the same process which does the calculations, so communication and calculation do not overlap. Depending on the application it might be possible to overlap some calculations with external communication. However, this is the responsibility of the programmer, whereas in the parallel model it is inherent.

As can be seen in the formulas (6.2) and (6.4), the time needed for external communications is proportional to $C_x y + C_y x$, while the time for internal processing is proportional to $x \cdot y$. To minimize the overhead caused by external communications, the shape of a local grid should be chosen in such a way that $C_x y + C_y x$ is minimized with respect to $x \cdot y$. So for the case that $C_x \approx C_y$, x and y should be chosen approximately equal.

6.6 Speedup Calculations

To see what the effect on the processing time of adding extra transputers is, we will compare the processing times for different numbers of transputers, using the same global grid in all cases. When the number of transputers in the horizontal direction

(P_x) is increased by a factor p and the number of transputers in the vertical direction (P_y) is increased by a factor q, the speedup for the parallel model is given by:

$$\begin{aligned} & S^{par}(p,q) \\ &= \frac{T^{par}(px,qy)}{T^{par}(x,y)} \\ &= \frac{pqxy(T_{calc}+C_{xy}T_{chan})+(C_xqy+C_ypx)(T_{chan}+FT_{link})}{xy(T_{calc}+C_{xy}T_{chan})+(C_xy+C_yx)(T_{chan}+FT_{link})} \\ &= pq-\frac{((pq-q)C_xy+(pq-p)C_yx)(T_{chan}+FT_{link})}{xy(T_{calc}+C_{xy}T_{chan})+(C_xy+C_yx)(T_{chan}+FT_{link})} \end{aligned} \tag{6.5}$$

The speedup calculation for the sequential model is analogous:

$$\begin{aligned} & S^{seq}(p,q) \\ &= \frac{T^{seq}(px,qy)}{T^{seq}(x,y)} \\ &= \frac{pqxy(T_{calc}+VT_{subs}+CT_{loop}+C_{xy}T_{array})+(C_xqy+C_ypx)(T_{chan}+T_{link})}{xy(T_{calc}+VT_{subs}+CT_{loop}+C_{xy}T_{array})+(C_xy+C_yx)(T_{chan}+T_{link})} \\ &= pq-\frac{((pq-q)C_xy+(pq-p)C_yx)(T_{chan}+T_{link})}{xy(T_{calc}+VT_{subs}+CT_{loop}+C_{xy}T_{array})+(C_xy+C_yx)(T_{chan}+T_{link})} \end{aligned} \tag{6.6}$$

As the number of transputers increases, the size of the local grid decreases. For small local grids, the time devoted to external communications will increase with respect to processing time. When the amount of external communication dominates the total processing time, $C_{ext}(T_{chan}+T_{link})$ respectively $C_{ext}(T_{chan}+FT_{link})$ will be much greater than the other terms in the denominators. In this case, the speedup for both models reduces to:

$$S(p,q) \approx pq-\frac{(pq-q)C_xy+(pq-p)C_yx}{C_xy+C_yx} \tag{6.7}$$

This formula gives a lowerbound for the speedup attainable. If we choose p and q equal, this reduces to:

$$S(p,p) \approx p \tag{6.8}$$

For the case that $C_x \approx C_y$, x and y should be chosen approximately equal, so that (6.7) reduces to:

$$S(p,q) \approx \frac{p+q}{2} \tag{6.9}$$

6.7 Variables for Speedup Calculation

A number of variables in the speedup formulas do not depend on the type of computation performed, but they only depend on the hardware and the compiler used.

These variables are: T_{subs}, T_{array}, T_{chan}, T_{loop} and T_{link}. The first four variables can simply be measured using an appropriate benchmark program running on a single transputer. To determine T_{link}, a network of transputers is needed.

The program `bench.c` (see appendix D) determines the first four variables for the parallel C compiler from UNICOM [18] generating code for an T414, running at 20 MHz. The value for T_{subs} is determined by subtracting the time needed for a fixed number of variable references from the time for the same number of 1-dimensional array references. For the other variables, the constant overhead is subtracted from each measure, so all values are net.

To determine T_{link}, a separate program named `link.c` was written (see appendix D). This program runs on two adjacent transputers. The second transputer does nothing but echoing values it receives via one of its links. The other transputer sends values to this transputer and waits until they return. In this way the time for two link-communications can be measured. To determine T_{link}, the value of T_{chan} has to be subtracted from this time.

The next table shows the results of these benchmarks:

T_{subs}	=	0.61	μ seconds
T_{array}	=	3.36	μ seconds
T_{chan}	=	6.03	μ seconds
T_{loop}	=	0.57	μ seconds
T_{link}	=	5.15	μ seconds

6.8 Conclusion

Two different implementation models for grid-based computations were introduced, together with a criterion by which the fastest model for a given application can be selected, on a basis of the ratio of local variable usage and the number of communications.

Formulas were derived to calculate the execution time for each model, using a number of program dependent constants, and a number of hardware/compiler dependent constants. The latter constants for our configuration were measured in section 6.7. With these formulas, expressions for the speedup of both models were derived, together with a lowerbound for the speedup.

Inspection of the formulas for the execution time shows that to minimize the overhead caused by external communications and thus maximize speedup, the shape of a local grid should be chosen in such a way that $C_x y + C_y x$ is minimized with respect to $x \cdot y$.

Chapter 7

Parallelization of the Hirlam Program

In the Hirlam program some terms of the primitive equations are solved implicitly. This results in numerically solving a Helmholz equation and the need of global communication. These calculations are performed in procedure SICALL. It is possible to leave out these calculations, but then the time step has to be chosen smaller (approximately a factor 3) to avoid instability. It is the aim of this project that this increase in computation time can be gained back by parallelizing the Hirlam program.

7.1 Communication in the Hirlam Program

The Hirlam program uses a 3-dimensional space of gridpoints to model the behavior of the atmosphere above a rectangular area on earth. These gridpoints (except the ones at the boundary) have to communicate with their neighbor gridpoints in all six directions. As was explained in section 6.1, we have to project one of these dimensions onto the other two.

In the Hirlam program, communication takes place in all three dimensions, but the amount of vertical communication exceeds the amount of horizontal communication. Also, in typical applications of the Hirlam program, the number of gridpoints in the vertical direction is much lower than in both horizontal directions. Furthermore, a great number of constants and variables used do not depend on the height. Obviously, the best solution is to project the vertical dimension onto the horizontal plane, so the gridpoints modeling a vertical column of air are always calculated on the same transputer.

The remaining point is how to distribute these "columns" over the transputer grid in the case the number of columns exceeds the number of transputers. In section 6.5 we have seen that when each transputer takes care of a local grid of size $x \times y$, the external communication time is minimized if $(C_x \cdot y + C_y \cdot x)$ is minimal with

respect to $x \cdot y$.

Because the behavior of the atmosphere is inherently symmetric in the two horizontal directions, the number of communications in both directions C_x and C_y will be approximately the same in the Hirlam program. Performance therefore will be maximal when x and y are chosen approximately equal. Of course, the sizes of the local grids of the different transputers should all be roughly the same, because the speed of the total system is dictated by the slowest transputer.

At a few spots in the Hirlam program, diagonal communications take place. These communications have to be replaced by two-step communications. To avoid unnecessary waiting as a result of diagonal communication, the programmer should take care that the particular information is transferred in the previous communication phase to a real neighbor.

Figure 7.1 shows the structure of the Hirlam program. Upper case names denote subroutines. The program starts with reading the input data and initializing variables. After that, a big loop is entered, in which a new state is calculated each iteration. Every six hours, new boundary data is read in.

In the figure, the subroutines in which communications take place, are marked. The subroutines DYN, VDIFFX and HDIFF contain local communication, whereas the subroutine STATIS contains global communication. This subroutine contains the calculations of some statistical values, like the average pressure tendency. Because these calculations are not essential, STATIS can be omitted in a transputer implementation. Consequently, only local communications are needed in a transputer implementation of the Hirlam program in explicit mode. The subroutine BDMAST is a replacement for the subroutine SICALL in case the implicit calculations are turned off.

7.2 Choosing the Implementation Model

In the previous section, we saw that the Hirlam program contains communication at four stages during each time step. The decision which implementation model is the most appropriate for the Hirlam program, depends on the value of Q, introduced in the previous chapter. We therefore have to take a closer look at the number of array references and the number of communications in the Hirlam program. The subroutine with the most communication is DYN, as a result of the integration of the primitive equations, so we will look at DYN first.

7.2.1 Calculation of Q for Subroutine DYN

DYN mainly consists of two loops over the (vertical) levels, the first of which is used for initializing. Inside these two loops there are various loops over the horizontal grid. As a result of optimizations, all 2-dimensional variables are "flattened" and stored in 1-dimensional arrays, so these loops are also 1-dimensional.

```
* read start data
* read boundary data 1
* read boundary data 2
* initialization :
   * BDINIT
   * MAPFAC
   * INIPHY
* loop for each time-step :
   * DYN (local communication)
   * TSTEP
   * PHCALL:
      * HYBRID
      * RADIA
      * VDIFFX(1) (local communication)
      * VDIFF
      * VDIFFX(2) (local communication)
      * KUO
      * COND
      * QNEGAT
   * HDIFF (local communication)
   * BDMAST
   * STATIS (global communication)
   * array copying
   * PRSTAT
   * 6 hourly input of new boundary data
   * output results STATIS
```

Figure 7.1: Structure of the Hirlam Program

loop	variable	direction W	E	S	N	SW	SE	NW	NE
Communication step 1									
FOR K := 1 TO NLEV									
Communication step 2									
260	DPK		N		N				
END FOR									
280	UU	N							
"	VV			N					
"	HYU	N							
"	HXV			N					
Communication step 3									
FOR K := 1 TO NLEV									
Communication step 4									
550	DPK		N						
560	DPK				N				
Communication step 5									
570	UU	N							
"	VV			N					
620	DPK		O		O				
"	EDPDE		N		N				
Communication step 6									
650	DPK		O		O				
"	EDPDE		N		N				
720	TV	N	N	N	N				
"	PP	N	N	N	N				
"	UU	O							
"	VV			O					
800	UZ	N							
"	VZ			N					
"	HYU	O							
"	HXV			O					
810	ZAHXHY		N		N				N
"	VZ		N						
"	UZ				N				
"	RHYV		N						
"	RHXU				N				
"	DPK		O		O				N
Communication step 7									
820	ZK			N					
"	VV		N	O		N			
"	PHI		N						
"	EK		N						
"	TV		O						
"	PP		O						
"	ZK	N							
"	UU	O			N			N	
"	PHI				N				
"	EK				N				
"	TV				O				
"	PP				O				
"	UU	O							
"	VV			O					
"	TZ(K)	N	N	N	N				
"	UU	O							
"	VV			O					
"	QZ(K)	N	N	N	N				
END FOR									

Communication step 1:

W_HYU, S_HYU,
W_UZ, N_UZ,
S_VZ, E_VZ,
E_RHYV, N_RHXU,
W_TZ[ ], E_TZ[ ], S_TZ[ ], N_TZ[ ],
W_QZ[ ], E_QZ[ ], S_QZ[ ], N_QZ[ ],
E_ZAHXHY, N_ZAHXHY.

Communication step 2:

E_DPK, N_DPK.

Communication step 3:

W_UU, S_VV, N_E_ZAHXHY.

Communication step 4:

E_DPK, N_DPK.

Communication step 5:

W_UU, S_VV,
E_EDPDE, N_EDPDE.

Communication step 6:

E_EDPDE, N_EDPDE,
W_TV, E_TV, S_TV, N_TV,
W_PP, E_PP, S_PP, N_PP,
N_E_DPK.

Communication step 7:

W_ZK, S_ZK,
N_UU, N_W_UU,
E_VV, W_S_VV,
E_PHI, N_PHI,
E_EK, N_EK.

Figure 7.2: Communication in Subroutine DYN

Figure 7.2 shows the communication pattern of DYN. Listed are the neighbor values needed in a particular loop. A mark 'O' after a variable name indicates that the neighbor value is "old", which means that this value was used before. It is possible to avoid the "O-communications" by saving the received neighbor values for subsequent use. A mark 'N' means "new", so a communication is always necessary here. A neighbor value is marked with 'N' the first time it is needed, or if the variable was recomputed after the last usage.

Four values from diagonal neighbors are needed. These values must be send in two steps, so these communications must be counted twice.

Without the 'O' values, the total number of horizontal communications in the east-west direction, C_x, and in the north-south direction, C_y, are both $2+21 \cdot \text{NLEV}$, where NLEV is the number of vertical levels.

In the figure we also see that there are seven communication steps, two of which are outside the vertical loops. The number of communication steps therefore is $2 + 5 \cdot \text{NLEV}$. The values exchanged during each communication step are listed at the right of the figure. A prefix 'W_' means that the value of the following variable in westward direction is needed; the prefixes 'E_', 'N_' and 'S_' have analogous meaning. For diagonal communications, a double prefix is used. For instance, 'N_E_ZAHXHY' denotes the value of the variable ZAHXHY of the eastward neighbor of the northward neighbor.

The value of V, which stands for the number of array references, is for DYN $21 + 187 \cdot \text{NLEV}$. The value of Q, computed with these values, is 4.5, independent of the value of NLEV. Applying the results of section 6.4.3, we can expect that the best implementation model for DYN will be the sequential model.

7.2.2 Estimation of Q for the Hirlam Program

The question now arises, whether the sequential model is also the best choice for the rest of the Hirlam program. Of course, the sequential parts are better implemented parallel, but mixing the two implementation models in the same program causes extra overhead due to the switching back and forth between the two implementations. For a switch from a sequential part to a parallel part, array elements have to be copied into plain variables, while for a switch back, all variables have to be copied back (if they were modified).

To calculate the value of Q for the whole program, we have to know the values of C_{xy} and V. The number of communications in the subroutines VDIFFX and HDIFF are much lower than in DYN. VDIFFX(1) contains $2 \cdot \text{NLEV}$ communications, VDIFFX(2) $12 \cdot \text{NLEV}$ communications and HDIFF contains $6 + 16 \cdot \text{NLEV}$ communications. The total number of communications in the Hirlam program therefore is:

$$C_{xy} = 10 + 72 \cdot \text{NLEV}$$

To get a rough estimate of the value V for the Hirlam program, it is possible to

extrapolate the value for DYN to the whole program. As can be seen on the sample output in figure 4.3 of the Hirlam program running on the HCX-9, DYN takes about 18.5 % of the time used for one time step (STATIS not counted). When we assume that the other parts of the Hirlam program contain approximately the same mean number of array references per second, an estimate for V will be:

$$V \approx \frac{187 \cdot \mathrm{NLEV}}{18.5\%} \approx 1011 \cdot \mathrm{NLEV}$$

Using this value for V and the above value for C_{xy}, an estimate for Q can be computed:

$$Q \approx \frac{1011 \cdot \mathrm{NLEV}}{10 + 72 \cdot \mathrm{NLEV}} \approx 14$$

For this value of Q, the sequential model is slightly better, although the difference with the parallel model is minimal.

7.3 Performance Expectations

We would like to know what running times we have to expect, when running the Hirlam program on a grid of transputers. In chapter 6, formulas were given for the execution time as a function of a number of variables. One of these variables is T_{calc}, the total calculation time for one gridpoint. Because this variable will probably dominate total processing time in the Hirlam program, it is particularly important to determine T_{calc}.

7.3.1 Determining T_{calc} for subroutine DYN

First T_{calc} will be determined for subroutine DYN. The values of C_x, C_y and V for DYN are known, so T_{calc} can be determined by implementing DYN on a transputer and measuring the running time, for it is the only unknown in the formulas for the execution time (6.2,6.4).

We have implemented subroutine DYN in C on one transputer, without external communication and using the sequential model with pointers. This program can be found in appendix E. The reason we used the variant with pointers instead of the variant with 1-dimensional arrays is that in DYN, many variables are used more than once, so that this variant is probably the fastest, although the formula for it would be slightly more complicated than for variant **seq2**. The running time on a T414 was 120 ms per local gridpoint, with NLEV = 9, so $\frac{T^{seq}}{xy} = 120ms$.

We recall the formula for the execution time of the sequential model (6.4):

$$T^{seq} = xy(T_{calc} + VT_{subs} + CT_{loop} + C_{xy}T_{array}) + C_{ext}(T_{chan} + T_{link}) \qquad (7.1)$$

Because the program does not use external communications, the second term of this formula can be dropped. To determine T_{calc} we must subtract $(VT_{subs} + CT_{loop} +$

$C_{xy}T_{array}$) from the measured 120 ms. The above formula is actually for the variant with 1-dimensional arrays and not for the pointer variant. However, as can be seen in figure 6.4, the execution times of these two variants do not differ too much for $Q = 4.5$, so we can use it anyway.

The three terms to subtract can be computed, using the constants determined in section 6.7, as follows (assuming NLEV = 9):

$$\begin{aligned} VT_{subs} &= (21 + 187 \cdot \text{NLEV}) \cdot 0.61\mu s = 1039\mu s \\ CT_{loop} &= (3 + 17 \cdot \text{NLEV}) \cdot 0.57\mu s = 89\mu s \\ C_{xy}T_{array} &= (4 + 42 \cdot \text{NLEV}) \cdot 3.36\mu s = 1284\mu s \end{aligned}$$

The value C generally is the number of calculation steps, because this normally equals the number of loops over the grid. In the current implementation of the Hirlam program, there are more loops than calculation steps and because we did not change the loop structure during conversion to C, the number of loops is greater than necessary. Therefore, the real number of loops is used here, instead of the number of communication steps (which is $3 + 5 \cdot \text{NLEV}$).

Surprisingly, the three values just computed, are very small compared to the measured running time of 120 ms, only about 2 % of it! The value of T_{calc} for DYN therefore is

$$T_{calc} = 120ms - 1039\mu s - 89\mu s - 1284\mu s = 118ms$$

The communication time for subroutine DYN, in case external communications do take place, is

$$\begin{aligned} C_{ext}(T_{chan} + T_{link}) &= (C_x y + C_y x)(T_{chan} + T_{link}) \\ &= (x + y)(2 + 21 \cdot \text{NLEV})(6.03\mu s + 5.15\mu s) \\ &= (x + y)2135\mu s \end{aligned}$$

which is about 3.5 % of the total processing time, for a 1×1 local grid, 1.7 % for a 2×2 grid, etc. Clearly, external communications are not a bottleneck in DYN.

7.3.2 Estimating T_{calc} for the Hirlam Program

If we want to know T_{calc} for the whole Hirlam program, we have to implement it completely on a transputer. However, as was done in section 7.2.2, it is possible to get an estimate of T_{calc} by extrapolating the T_{calc} for DYN to a T_{calc} for the whole program. The assumption that T_{calc} is approximately proportional to the measured execution time on the HCX-9 is quite reasonable. The value of T_{calc} for the Hirlam program under this assumption becomes:

$$T_{calc} \approx \frac{118ms}{18.5\%} \approx 638ms$$

Using the estimated value of V and counted values C, C_x and C_y for the whole program, we can also compute the values for the other three subterms of the first term of (7.1). The value for C which is used here, is the minimum number of calculations steps needed for the implementation of the Hirlam program. The current implementation in Fortran uses more steps (loops). The values for NLEV = 9 are:

$$\begin{aligned} VT_{subs} &= (1011 \cdot \text{NLEV}) \cdot 0.61\mu s = 5550\mu s \\ CT_{loop} &= (4 + 10 \cdot \text{NLEV}) \cdot 0.57\mu s = 54\mu s \\ C_{xy}T_{array} &= (10 + 72 \cdot \text{NLEV}) \cdot 3.36\mu s = 2211\mu s \end{aligned}$$

Also, the second term of (7.1) can now be computed:

$$\begin{aligned} C_{ext}(T_{chan} + T_{link}) &= (C_x y + C_y x)(T_{chan} + T_{link}) \\ &= (x + y)(5 + 36 \cdot \text{NLEV})(6.03\mu s + 5.15\mu s) \\ &= (x + y)3678\mu s \end{aligned}$$

When we substitute all these values into (7.1), we get:

$$T^{seq} \approx xy(638ms + 5550\mu s + 54\mu s + 2211\mu s) + (x + y)3678\mu s \qquad (7.2)$$

From this equation, we can conclude that internal communication cost ($2211\mu s$) is very low, compared to calculation time ($638ms$). Also, the external communication is not costly. It accounts for only 1.1 % of the total processing time for a local grid of size 1×1, 0.6 % for a grid of size 2×2, etc. External communication therefore, will not be a bottleneck in the Hirlam program.

The total running time of a transputer implementation of the Hirlam program, for a $34 \times 34 \times 9$ grid is plotted in figure 7.3, as a function of the total number of transputers $P_x \cdot P_y$, using (7.2). The values for P_x and P_y in this figure are equal. When P_x and P_y do not divide 34, not all transputers process the same number of gridpoints.

To reveal more detail, the values from this figure are plotted a second time in figure 7.4, this time without the first two points.

7.4 Performance with T800 transputers

In the preceding section, we have seen that the calculation time dominates total execution time. Acceleration of the calculations will therefore have a direct effect on the execution time for the Hirlam program. Most calculations in the Hirlam program are floating point calculations. Floating point operations are relatively slow on a T414, as compared to a T800 transputer, so performance will be much higher on a network of T800 transputers than on a T414 network.

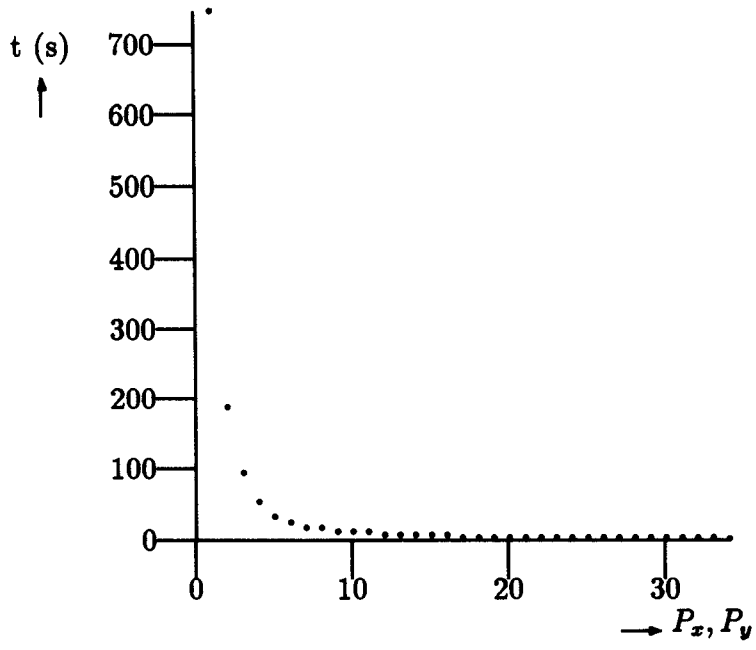


Figure 7.3: Estimated Running time for Hirlam on a T414 network

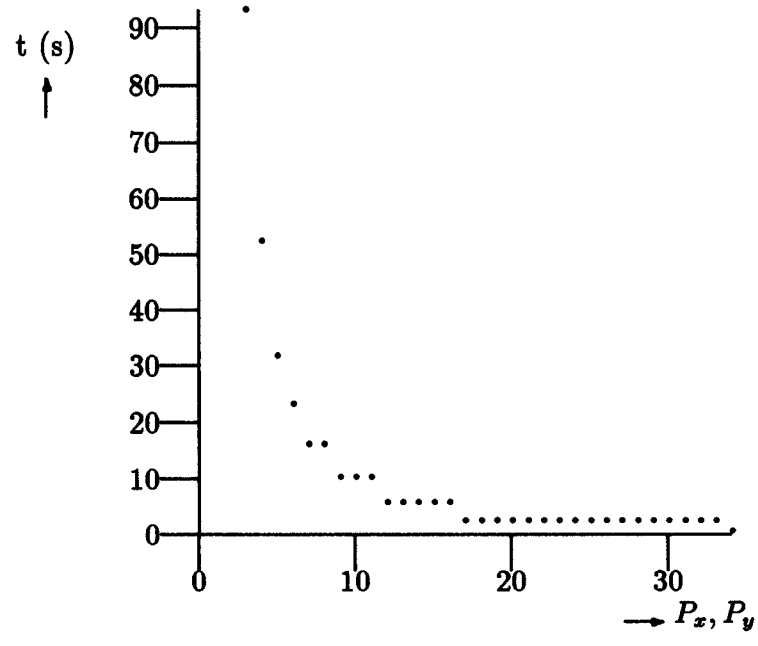


Figure 7.4: Estimated Running time for Hirlam on a T414 network (enlarged)

To get an indication of the running time of the Hirlam program on a grid of T800 transputers, we have determined a factor by which some typical floating point calculations (taken from DYN) are performed faster on a T800 than on a T414.

Because our transputer system did not contain T800 transputers, and the C compiler we used is not (yet) able to produce code for a T800, we had to use another system for this test. This system is an IBM-AT with a slot-card containing two T800 transputers, which can be programmed in OCCAM-2, using the MEGATOOL programming environment.

The determined speedup for using OCCAM-2 on a T800 instead of C on a T414 was approximately 16. However, the OCCAM-2 compiler seems to produce better code than the C compiler, because a comparison between the two systems, using a version of the program with integers instead of floating point variables, showed that the OCCAM-2 version was approximately 10 % faster.

When we decrease the value of T_{calc} in formula (7.2) by this factor, we get:

$$T^{seq} \approx xy(40ms + 5550\mu s + 54\mu s + 2211\mu s) + (x + y)3678\mu s \qquad (7.3)$$

From this formula, it can be calculated that for a grid of T800 transputers the external communication time will account for at most 13 % (1×1 local grids) of the total execution time.

The estimated running time for the Hirlam program on a grid of T800 transputers, is plotted in the figures (7.5) and (7.6).

7.5 Speedup Calculations for the Hirlam Program

Instead of plotting the total execution time in figure 7.5, it is also possible to plot the speedup as a function of the total number of transputers, which is done in figure 7.7. The speedup values for all possible values of $P_x \cdot P_y$ are plotted. In the case where the same total number of transputers can be formed by different combinations of P_x and P_y, the combination with the smallest difference between P_x and P_y is chosen.

This figure clearly shows the almost linear speedup for the Hirlam program as the number of transputers is increased, up to one transputer for every gridpoint. The point in the upper right corner corresponds to the speedup for $P_x = P_y = 34$, so there is one transputer per gridpoint. For the points with a speedup of about 500, the maximum local grid-size is 1×2; for a speedup of approximately 250, the maximum local grid-size is 2×2, etc.

Each of these "lines" in the figure corresponds to a particular maximum local grid-size. To maximize efficiency, P_x and P_y should be chosen in such a way that the corresponding speedup value is the very first point of a "line". For these values of P_x and P_y, the total number of transputers used for a computation with a particular maximum local grid-size is minimal, and the speedup is almost equal to $P_x \cdot P_y$.

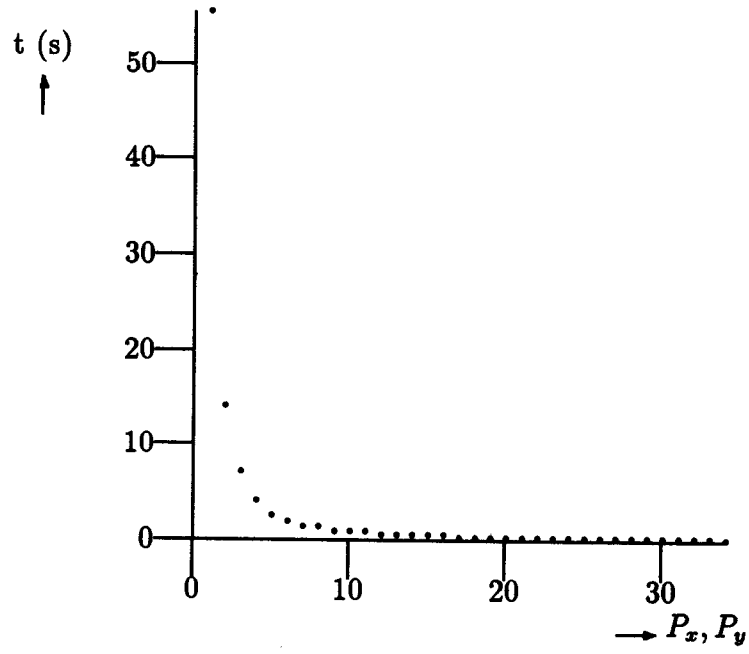


Figure 7.5: Estimated Running time for Hirlam on a T800 network

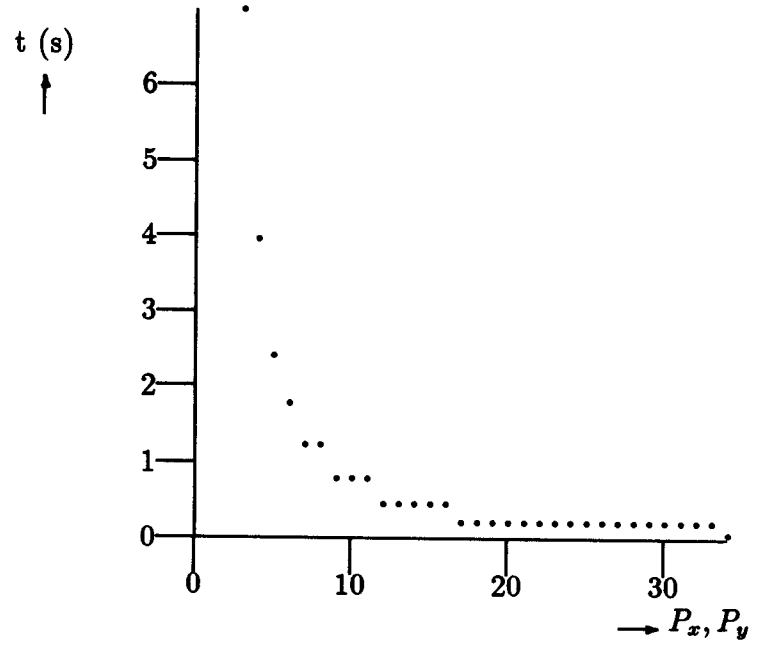


Figure 7.6: Estimated Running time for Hirlam on a T800 network (enlarged)

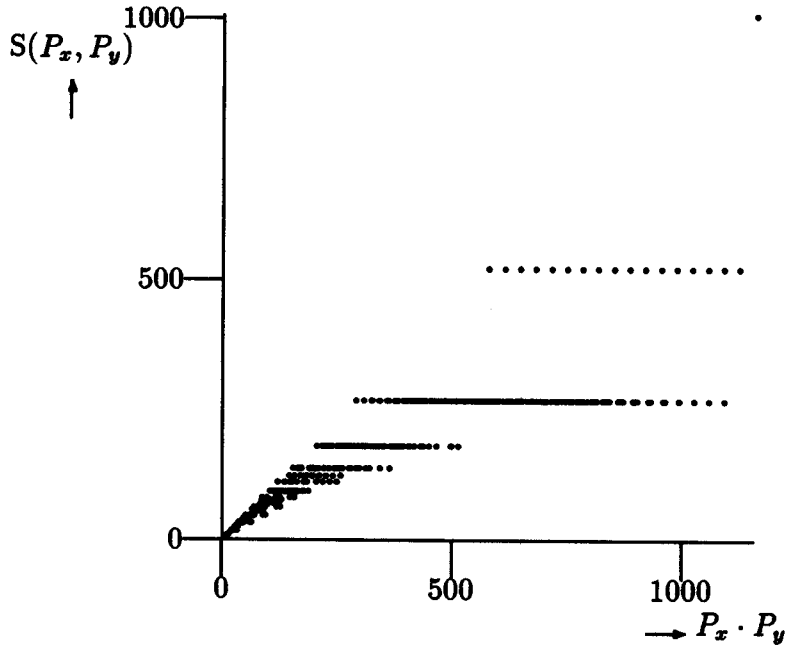


Figure 7.7: Estimated Speedup for Hirlam on T800 Transputers

7.6 Conclusion

Section 7.2.1 shows that the parts of the Hirlam program where communication takes place, such as the subroutine DYN, are best implemented using the sequential model. As was argued in section 7.2.2, implementing parts of a program sequentially and other parts parallel, may cause much extra overhead. The estimated value of Q for the Hirlam program indicates that both implementation models will be approximately equally fast.

Furthermore, one subroutine of the Hirlam program (DYN), was implemented on a transputer in order to derive an estimate for the running time of the whole program on a grid of transputers. On a network of T414 transputers, both the amount of internal and external communication turned out to be neglectable with respect to the calculation time. With T800 transputers, the overhead due to external communication will also be moderate. As a result, speedup for the Hirlam program will be almost linear in the number of transputers.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

The question to be answered by this report is whether it is feasible to implement a weather prediction model like HIRLAM on a network of transputers and if so, whether it can be done in such a way that performance is linear in the number of transputers involved in the calculation.

In chapter 6, two implementation models for grid-based computations, the parallel and the sequential model, were presented, together with a number of possible optimizations. Which model is best suited for a given application depends on the ratio of the number of variable references and the number of communications. With suitable test programs, a criterion is determined by which the fastest model can be selected for a given application. For the Hirlam program, it was estimated that both implementation models are approximately equally fast.

From the formulas for the execution time in section 6.5 it follows that for inherently symmetric computation models like atmospheric models, the shape of the subgrid processed by one transputer, should be square, in order to minimize inter processor communication, and therefore maximize speedup. The shape of the global grid is irrelevant here.

With appropriate benchmark programs, a number of hardware/compiler dependent constants were measured, which occur in the formulas for the execution time. With these constants, it was estimated that for the Hirlam program, running on a grid of T414 transputers, the time used for internal communication is less than 1 % of the total processing time, and the time used for external communication is maximal 1 %, decreasing when the local grid gets bigger. As a consequence, the speedup will be almost linear in the number of transputers. Also, performance will hardly decrease if x and y are not chosen equal.

Because floating point operations on a T800 are approximately 16 times faster than on a T414, the relative overhead due to external communication is somewhat larger for a grid of T800 transputers. However, in the worst case (1×1 local grids),

it only accounts for approximately 13 % of the total execution time, so even when using T800 transputers, external communication will not be the bottleneck.

Besides the principal questions answered by this report, the feasibility study has initiated several spinoffs like the development of transputer support software, the starting of transputer related projects and extensive contacts with some companies.

8.2 Future Work

Several points of further research, have become clear during the feasibility study:

- Further development is needed in implementing HIRLAM completely on transputers, taking the results of this report as a starting point. The next question to be answered concerns the impact of global communication on a transputer implementation of the Hirlam model.
- A programming environment for transputers and, more general, parallel systems needs to be developed. The emphasis will be on the development of (specification) languages for writing software and operating software.
- It needs to be investigated if the concept of time staggering instead of time filtering, is advantageous for an implementation of the Hirlam model.
- More investigation has to take place to find out if and how various classes of differential equations with a numerical solution algorithm on a grid can automatically be implemented on a grid of transputers.

Appendix A

Mathematical Principles used for Finite Differencing

A.1 Finite Differences

Partial derivatives can be approximated by finite-differences in many ways. All such approximations introduce errors, called truncation errors. By development of the Taylor series for $u(x + \Delta x, y)$ about (x, y) we get a forward difference equation:

$$u(x + \Delta x, y) = u(x, y) + \Delta x \frac{\partial u}{\partial x}(x, y) + \frac{(\Delta x)^2}{2!} \frac{\partial^2 u}{\partial x^2}(x, y) + \frac{(\Delta x)^3}{3!} \frac{\partial^3 u}{\partial x^3}(x, y) + O(\Delta x)^4 \tag{A.1}$$

Equation (A.1) gives:

$$\frac{u(x + \Delta x, y) - u(x, y)}{\Delta x} = \frac{\partial u}{\partial x}(x, y) + \frac{(\Delta x)}{2!} \frac{\partial^2 u}{\partial x^2}(x, y) + \frac{(\Delta x)^2}{3!} \frac{\partial^3 u}{\partial x^3}(x, y) + O(\Delta x)^3 \tag{A.2}$$

Taking the left-hand side of (A.2) as an approximation to $\frac{\partial u}{\partial x}(x, y)$ we introduce a truncation error, usually written in the asymptotic O notation, $O(\Delta x)$. By this way we get a first order forward finite-difference approximation.

A.2 Computational Errors

Strictly speaking the truncation error belongs to the finite-difference schemes and not to the solution. This point is emphasised here, because as in analytic methods, the boundary conditions are essential to the proper solution. These boundary conditions must be approximated by finite-differences, thereby introducing an additional or boundary truncation error[1, p23-24].

The error in the solution due to the replacement of the continuous problem by the discrete model is called the discretization error. The over-all discretization error is the smallest order of all those approximations used unless they are somehow related.

When the discrete equations are not solved exactly, an additional error is introduced, the round-off error. The sizes of the discretization interval affects the discretization error and the round-off error in opposite sense. That is the reason why in general one can not assert that decreasing the mesh size always increases the accuracy[1, p23-24].

Finally, discrete approximations contain an aliasing error. Knowing only the values at grid-points waves with wavelength smaller than $2\Delta x$ are falsely represented as waves with wave length bigger than $2\Delta x$. To understand how such a wave is falsely represented we can make the following analysis:

$$\sin(kx) = \sin((2k_{max} - (2k_{max} - k))x$$

where $k_{max} = \frac{2\pi}{\Delta x}$, the maximum wave number that can be represented on a grid with grid-length Δx. Applying the difference formula for the sinus and substituting $k_{max} = \frac{\pi}{\Delta x}$:

$$\sin kx = \sin \frac{2\pi}{\Delta x} x \cos\left(\frac{2\pi}{\Delta x} - k\right) x - \cos \frac{2\pi}{\Delta x} x \sin\left(\frac{2\pi}{\Delta x} - k\right) x$$

Substituting $x = j\Delta x$:

$$\sin kj\Delta x = -\sin\left(2k_{max} - k\right) j\Delta x$$

Thus a wave with wavenumber $k > k_{max}$ is falsely represented as a wave with wave number $k^* = 2k_{max} - k < k_{max}$.

A.3 Consistency, Convergence and Stability

Computational errors may lead to numerical instability. Any numerical scheme which allows the growth of an (initial) error, is unstable. Stability of a scheme is a property of great practical significance. Second order approximation schemes do not necessarily lead to better numerical results than the first order approximation schemes. In certain applications a second order approximation is an unstable method, while a first order approximation is stable[1, p17].

Consistency A finite-difference scheme is said to be consistent with the differential equation if the truncation error tends to zero (as $\Delta x, \Delta y \to 0$). There are consistent schemes, of a high order of accuracy, that still give solutions diverging unacceptably fast from the true solution. Thus consistency is not strong enough to characterize correctness of finite-difference schemes.

Following Richtmeyer and Morton (1967) we ask two questions:

1. What is the behavior of the error $u_j^n - u(j\Delta x, n\Delta t)$, when, for a fixed total time $n\Delta t$, the increments $\Delta x, \Delta t$ tends to zero.

2. What is the behavior of the error $u_j^n - u(j\Delta x, n\Delta t)$, when, for a fixed values of Δx and Δt, the number of time steps n increases.

Convergence The answer to the first question depends on convergence of the numerical solution; if the error tends to zero as the grid is redefined (as $\Delta x, \Delta t \to 0$) the solution is called convergent. If a scheme gives a convergent solution for any initial condition, then the scheme is also called convergent[2, p5-6]. Consistency of a scheme does not guarantee convergence, because of the possibility that the characteristic defining the true solution is outside the domain of dependency (all gridpoints that are involved in the finite-difference calculation) of the approximated solution of a gridpoint. From this we infer that a gridpoint at the origin (containing an initial condition) can be outside that domain and therefore can not affect the numerical solution. Then the error can be arbitrarily large.

Stability The answer on the second question depends on stability of the numerical solution. When we know that the true solution is bounded, as in the equations we are interested in, we can use a definition referring to the boundedness of the error $u_j^n - u(j\Delta x, n\Delta t)$. We say that a solution u_j^n is stable if this error remains bounded for increasing n, with fixed values of $\Delta x, \Delta t$. A finite-difference scheme is stable if it gives a stable solution for any initial condition[2, p5-6].

The Lax equivalence theorem Lax studied the relation between consistency, stability and convergence of finite-difference approximations of linear initial value problems. The major result of that study is called the Lax equivalence theorem: Given a properly posed initial boundary value problem and a finite difference approximation to it that satisfies the consistency condition, then stability is a necessary and sufficient condition for convergence.

Non-linear instability In a numerical approximation for non-linear terms, using a grid with grid point distance Δx, the wave number that can be represented is bounded to a maximum value $k_{max} = \frac{\pi}{\Delta x}$. By substitution of a sinusoid:

$$u(x,t) = Re\left[U(t)e^{ikx}\right]$$

with wave number k, $\frac{1}{2}k_{max} \leqq k \leqq k_{max}$, in the non-linear advection term:

$$u\frac{\partial u}{\partial x}$$

a wave with wave number $2k$ is introduced:

$$u\frac{\partial u}{\partial x} = U(t)^2 ike^{i2kx}$$

Consequently the introduced wave (by a non-linear interaction) has a wavelength that is too short to be represented on a grid with gridpoint distance Δx.

Appendix B

Implicit and Semi-Implicit Schemes

An inconvenient feature in case of gravity waves is the long computer time required for a solution using explicit schemes for time differencing. The time-step imposed by the stability criterion for explicit schemes is generally considered to be much less for an accurate integration of slower quasi geostrophic motion. With these steps the errors due to space differencing are much greater than those due to time differencing[1, p62].

A forward-backward scheme is comparable in computation time with the leap-frog time differencing by the Eliassan grid, but with time-steps twice those allowed for the leap-frog scheme. Even these time-steps are considerably shorter than that required for accurate integration of quasi-geostrophic motions and even with these economical schemes the time differencing error is still much less than the space differencing error for typical current atmospheric models. Therefore we consider implicit schemes which are stable for any choice of time-step, so that the choice of time-step is solely based on accuracy.

B.1 Semi-Implicit Scheme

The semi-implicit scheme goes a step nearer to a fully implicit scheme by treating implicitly those terms in the equations of motion that are primarily responsible for the propagation of gravity waves. In a semi-implicit scheme the advection terms are treated in explicit fashion.

B.2 Implicit Scheme

To apply an implicit method it is necessary to solve the difference system for all variables at instance of time $n+1$ simultaneously. Consider again the two-dimensional

system of linearized shallow water equations (3.27), (3.28) and (3.23). By applying the trapezoidal time differencing scheme we get:

$$u^{n+1} = u^n - g\Delta t\frac{1}{2}(\delta_x h^n + \delta_x h^{n+1}) \tag{B.1}$$

$$v^{n+1} = v^n - g\Delta t\frac{1}{2}(\delta_y h^n + \delta_y h^{n+1}) \tag{B.2}$$

$$h^{n+1} = h^n - H\Delta t\frac{1}{2}\left[(\delta_x u + \delta_y v)^n) + (\delta_x u + \delta_y v)^{n+1})\right] \tag{B.3}$$

where δ_x is a centered space differencing operator.

The quantities $\delta_x(u^{n+1})$ and $\delta_y(v^{n+1})$ can be eliminated from the third equation by applying operators δ_x and δ_y to equations (B.1) and (B.2) respectively, and by substituting the result into equation (B.3). This gives a finite difference approximation to the Helmholz equation:

$$(\nabla)^2 h + ah + b(x,y) = 0 \tag{B.4}$$

for the height which is to be solved, where $\nabla^2 h$ is approximated by the finite difference Laplacian $\delta_x(\delta_x h) + \delta_y(\delta_y h)$. To solve such an elliptic equation, it is necessary to know the values of $h(x,y)$ at the boundaries of the computation region. For a numerical solution we write a finite-difference approximation of the Helmholz equation at each of interior gridpoint where the variable h is carried.

In this way we obtain a system with one equation for each interior gridpoint. In each of the equations, except the equations for points adjacent to the boundary there are five of these unknowns. There are no difficulties in principle in solving such a system of linear equations, but, for efficiency reasons, sometimes the relaxation method is preferred.

B.2.1 Relaxation Method

A widely used standard method to solve the Helmholz equation numerically is the relaxation method. The essence of the relaxation method is to start an iteration process with an arbitrary tentative solution to approximate the true solution. This method consists of the following steps:

1. An arbitrary guess is made for the field h^{n+1}. Usually the field of the preceding time step h^n, is taken as a first guess.

2. At each of the grid points the value h^{n+1} is changed so as to satisfy the finite-differenced Helmholz equation. These changes can be made simultaneous at all gridpoints (simultaneous or Richardson relaxation), or sequentially (sequential or Liebmann relaxation).

3. The preceding step is repeated as many times as needed to make the change at every point less than some preassigned small value.

Simultaneous relaxation calculates guess values for all gridpoints before continuing. Sequential relaxation calculates a guess value for one gridpoint and this guess value is used to determine the guess value in the next grid point. To be useful the iteration must converge but it is not considered to be effective unless the convergence is rapid. Experience shows that the convergence is faster for sequential relaxation. On the other hand simultaneous relaxation can be expressed as a vector operation and therefore computed fast on vector machines.

The Helmholz equation can also be numerically solved by direct methods, like the Gauss elimination method. A direct method can be more efficient than the relaxation method, especially when it is difficult to start with a good guess. Therefore they are typically used when relaxation requires a very long computation time, as may happen, for example, in convection studies. When implicit schemes are used for simulation or prediction of large scale atmospheric motions, the time needed for relaxation is several times less than the time needed for other steps of the integration procedure, so that only a small fraction of the total computer time will be saved by using a faster direct method. For that reason the use of direct methods, requiring a larger programming effort, is not popular for these models.

With the semi-implicit scheme it is also possible to construct an economical grid analogues to the Eliassan grid for the leap-frog scheme; the appropriate space-time staggering of the variables was pointed out in [7].

Implicit and semi-implicit schemes are undoubtedly the most efficient schemes used in atmospheric models. To achieve this efficiency we have to put additional effort into solving an elliptic equation. Furthermore, they are associated with an appreciable deceleration of gravity waves. Thus, the implicit schemes do not seem suitable for the study of details of the geostrophic adjustment process. On the other hand, this deceleration does not appear particularly harmful for the simulation and prediction of the large scale geostrophic motions compared to other sources of error that are normally present in numerical models. The computation time saved by implicit differencing can be used to reduce the grid size of the computation. This would decrease the phase speed error for all the waves, including the gravity waves.

Appendix C

Modeling the Dynamics of the Atmosphere

As a propagation problem, dynamical numerical weather forecasting requires a closed set of appropriate physical laws expressed in mathematical form, suitable initial and boundary conditions and an accurate numerical method of integrating the system of equations forward in time. In this appendix we will discuss some aspects of these topics that are relevant in the context of this report. For a more rigorous treatment of atmospheric modeling the book by Haltiner and Williams[12], probably the most widely available text, can be mentioned.

C.1 The Primitive Equations

The primitive equations, which models the behavior of the atmosphere are:

1. The equations of motion
2. The continuity equation
3. The equation of state
4. The first law of thermo-dynamics
5. The moisture equation

The equations of motion The equations of motion based on Newton's second law, which states that the acceleration per unit mass equals the sum of the forces per unit mass. The principle forces in atmospheric motion are the pressure-, the gravitation-, the coriolis- and friction force. The equations of motion can be expressed in the form:

$$\frac{du}{dt} = -\frac{1}{\rho}\frac{\partial p}{\partial x} - f_h u + F_x \tag{C.1}$$

$$\frac{dv}{dt} = -\frac{1}{\rho}\frac{\partial p}{\partial y} - f_h v + F_y \tag{C.2}$$

$$\frac{dw}{dt} = -\frac{1}{\rho}\frac{\partial p}{\partial z} - f_v w + g + F_z \tag{C.3}$$

where u, v and w components of the wind in east, north, and vertical direction respectively; f_h, f_v the horizontal and vertical component of the Coriolis force; F_x, F_y and F_z components of the friction force ; p is the pressure. From (C.1), (C.2) and (C.3) we can deduce the hydrostatic one layer equation by assuming hydrostatic equilibrium.

The hydrostatic assumption For large scale motions of the air, the atmosphere may be assumed to be in hydrostatic equilibrium, that is, vertical acceleration may be neglected along with the vertical component of the Coriolis force. By assuming hydrostatic equilibrium, it follows immediately that for any point in the fluid:

$$g\rho(h - z) = p$$

where h is the height of the free surface. Therefore:

$$g\frac{\partial h}{\partial x} = \frac{1}{\rho}\frac{\partial p}{\partial x}$$

With this equation, neglecting friction terms, we can deduce the hydro-static one layer equations:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} - fv + g\frac{\partial h}{\partial x} = 0 \tag{C.4}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} - fu + g\frac{\partial h}{\partial y} = 0 \tag{C.5}$$

where u, v represent velocity components in the x and y direction. These equations are sufficient to describe horizontal frictionless motion. Because of the hydrostatic assumption (and constant density), the horizontal pressure force is independent of height. By assuming that the velocity field is initially independent of height, it will remain so; thus the vertical advection terms have been omitted in (C.4) and (C.5).

The continuity equation The continuity equation expresses conservation of mass:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = -\frac{1}{\rho}\frac{\partial \rho}{\partial t} \tag{C.6}$$

With the incompressibility assumption, (C.6) is linear:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{C.7}$$

Integrating (C.7) with respect to z gives:

$$\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) h + w_h - w_0 = 0 \tag{C.8}$$

In accordance with the boundary condition w must vanish at the lower boundary(i.e. $w_0 = 0$). On the other hand, the vertical velocity $w = \frac{dz}{dt}$ at the upper boundary represents the rate at which the free surface is rising. Thus $w_h = \frac{dh}{dt}$, and (C.8) becomes:

$$h\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) = -\frac{dh}{dt} = -\left(\frac{\partial h}{\partial t} + u\frac{\partial h}{\partial x} + v\frac{\partial h}{\partial y}\right) \tag{C.9}$$

Equations (C.4), (C.5) and (C.9) constitute a system of three unknowns, u, v and h. These equations are called the shallow water equations.

The other primitive equations The equation of state expresses the relationship between the three thermodynamic variables p, ρ and T. The first law of thermodynamics expresses the principle of conservation of energy. These four primitive equations suffice to model a dry atmosphere. For a more realistic model of the atmosphere the moisture equation needs to be included.

C.2 Coordinate System

Application of the equations of motion to real problems, in particular, numerical weather prediction, requires a map in terms of a coordinate system. Since the large-scale motions of the atmosphere are quasi-horizontal with respect to the earth's surface, spherical coordinates are quite useful. Choosing the right vertical coordinate is more delicate as will be made clear in the next subsection.

C.2.1 Vertical Differencing of the Primitive Equations

In the primitive equations there are non-linear terms associated with horizontal advection and vertical advection. In addition, it is generally attempted to incorporate a realistic topography of the earth's surface into the system as the lower boundary condition.

These situations make vertical (and horizontal) differencing of the primitive equations particularly difficult. Therefore we must properly choose a vertical coordinate, a vertical grid structure, a vertical difference scheme satisfactory for all possible types of motion, the vertical extent of the model, and finally, vertical spacing and resolution of the grid[6].

Vertical coordinate The height z is not the most convenient vertical coordinate for many purposes. Vertical coordinates that have been used with advantage are: pressure p; $ln\frac{p}{p_s}$; pressure normalized with surface pressure, $\sigma = \frac{p}{p_s}$ or the more general hybrid η-coordinate (Hirlam). By using an alternate vertical coordinate the partial differential equations can be transformed to a more convenient formulation.

As an example, using a η-coordinate, we are able to make the following transformations. First we need some expressions for transformation of z to η coordinate:

$$\nabla_\eta A = \nabla_z A + \frac{\partial A}{\partial \eta}\frac{\partial \eta}{\partial z}\nabla_\eta z \tag{C.10}$$

$$\frac{\partial A}{\partial z} = \frac{\partial A}{\partial \eta}\frac{\partial \eta}{\partial z} \tag{C.11}$$

$$\left(\frac{\partial A}{\partial t}\right)_\eta = \left(\frac{\partial A}{\partial t}\right)_z + \frac{\partial A}{\partial \eta}\frac{\partial \eta}{\partial z}\left(\frac{\partial z}{\partial t}\right)_\eta \tag{C.12}$$

$$\frac{dA}{dt} = \left(\frac{\partial A}{\partial t}\right)_\eta + \vec{V}\nabla_\eta A + \dot{\eta}\frac{\partial A}{\partial \eta} \tag{C.13}$$

where $\vec{V}$ is the horizontal velocity, $\dot{\eta}$ the vertical velocity in the η-system and A can be a scalar or a vector. (C.13) is the total derivative in η-coordinates.

- The equations of motion: By virtue of (C.11) the horizontal pressure force transforms as follows:

$$-\frac{1}{\rho}\nabla_z p = -\frac{1}{\rho}\nabla_\eta p + \frac{1}{\rho}\frac{\partial p}{\partial z}\nabla_\eta z = -\frac{1}{\rho}\nabla_\eta p - \nabla_\eta \phi \tag{C.14}$$

  where $\phi = gz$ is the geopotential. Equation (C.14) can be recognized as part of the equations of motion of the Hirlam model.

- The hydrostatic equation:

$$\frac{1}{\rho}\frac{\partial p}{\partial z} + g = 0 \tag{C.15}$$

  can be rewritten, applying (C.11) into:

$$\frac{1}{\rho}\frac{\partial p}{\partial \eta}\frac{\partial \eta}{\partial z} + g = 0 \tag{C.16}$$

  This equation can be further rewritten as:

$$\frac{1}{\rho}\frac{\partial p}{\partial \eta} + g\frac{\partial z}{\partial \eta} = 0 \tag{C.17}$$

  and can be recognized as (4.13), the hydrostatic equation of the Hirlam model used in chapter 4. Using sigma-coordinates this reduces further to:

$$\frac{1}{\rho}p_s + \frac{\partial \phi}{\partial \sigma} = 0 \tag{C.18}$$

  where $\phi = gz$. Notice that (C.18) is linear.

- The continuity equation (C.6):

$$\frac{\partial \ln \rho}{\partial t} + \nabla_z \cdot \vec{V} + \frac{\partial w}{\partial z} = 0 \tag{C.19}$$

Application of (C.11) and (C.12) to the divergence terms of (C.19):

$$\nabla_z \cdot \vec{V} + \frac{\partial w}{\partial z} = \nabla_\eta \cdot \vec{V} - \frac{\partial \eta}{\partial z}\frac{\partial \vec{V}}{\partial \eta} \cdot \nabla_\eta z + \frac{\partial w}{\partial \eta}\frac{\partial \eta}{\partial z} \tag{C.20}$$

Applying (C.13) to $w = \frac{dz}{dt}$:

$$w = \frac{dz}{dt} = \left(\frac{\partial z}{\partial t}\right)_\eta + \vec{V} \cdot \nabla_\eta z + \dot{\eta}\frac{\partial z}{\partial \eta} \tag{C.21}$$

Hence:

$$\frac{\partial w}{\partial \eta} = \frac{\partial}{\partial t}\left(\frac{\partial z}{\partial \eta}\right) + \vec{V} \cdot \nabla_\eta \frac{\partial z}{\partial \eta} + \frac{\partial \vec{V}}{\partial \eta}\nabla_\eta z - \frac{\partial \dot{\eta}}{\partial \eta}\frac{\partial z}{\partial \eta} + \dot{\eta}\frac{\partial}{\partial \eta}\left(\frac{\partial z}{\partial \eta}\right) \tag{C.22}$$

Substituting the foregoing equation in (C.20):

$$\nabla_z \cdot \vec{V} + \frac{\partial w}{\partial z} = \nabla_\eta \cdot \vec{V} + \frac{\partial \eta}{\partial z}\left(\frac{\partial}{\partial t} + \vec{V}\nabla_\eta + \dot{\eta}\frac{\partial}{\partial \eta}\right)\frac{\partial z}{\partial \eta} + \frac{\partial \dot{\eta}}{\partial \eta} \tag{C.23}$$

Furthermore, since

$$\frac{\partial \eta}{\partial z} = (\frac{\partial z}{\partial \eta})^{-1}$$

the middle term is just

$$\frac{d\left(\ln\left(\frac{\partial z}{\partial \eta}\right)\right)}{dt}$$

which can now be combined with $\frac{d(\ln \rho)}{dt}$ in (C.19) to give:

$$\frac{d \ln\left(\rho\frac{\partial z}{\partial \eta}\right)}{dt} = \frac{d\left(\ln \frac{\partial p}{\partial \eta}\right)}{dt}$$

Thus the transformed continuity equation becomes:

$$\frac{d}{dt}\left(\ln \frac{\partial p}{\partial \eta}\right) + \nabla_\eta \cdot \vec{V} + \frac{\partial \dot{\eta}}{\partial \eta} = 0 \tag{C.24}$$

which can be rewritten into:

$$\frac{d}{dt}\left(\frac{\partial p}{\partial \eta}\right) + \nabla_\eta \cdot \vec{V}\frac{\partial p}{\partial \eta} + \frac{\partial \dot{\eta}}{\partial \eta}\frac{\partial p}{\partial \eta} = 0 \tag{C.25}$$

which can be furthur rewritten into the continuity equation (4.9) of the Hirlam model, which is used in chapter 4.

C.3 Time Filtering

Using finite-difference schemes for non-linear equations the separation of solutions at alternate time-steps generates two-grid-interval noise in time. This high frequency noise also appears in atmospheric models as a result of difficulties in observing initial conditions of large scale atmospheric motions. The observed initial conditions contain measuremental errors, are influenced by subgrid scale motions that are not resolved by the model grid, and, finally are completely absent over relative large areas of the globe.

To increase the damping of the noise in atmospheric models time filtering is used. This requires at least three consecutive values of the function $U(t)$ to be filtered are needed. If a filter is continually applied during a numerical integration, the value $U(t - \Delta t)$ has already been changed prior to changing $U(t)$. It is then appropriate to consider the filter:

$$U_{filtered}(t) = U(t) + \frac{1}{2}S(U_{filtered}(t - \Delta t) - 2U(t) + U(t + \Delta t))$$

where S is the filter parameter. This filter is called the basic time-filter.

An analysis of the effect of the time-filter for some particular choices of time differencing schemes (the leap-frog, implicit and semi-implicit schemes) can be found in a paper of Asselin[2, p61,62].

C.4 Limited Area Modeling

Most operational meteorological organizations use a limited area model both as an operational model and as a research tool. At the ECMWF(European Centre for Medium Range Weather Forecast), the LAM was implemented as a research vehicle to study the nature of particular phenomena, and to investigate and test a variety of new techniques before introducing them into the global operational model.

The solution of the mathematical problems and the fulfillment of technical requirements in using a LAM can be very difficult. The boundary values and a 'boundary scheme' for the treatment of the lateral boundary are the first requirement for a LAM. Many of the techniques and problems associated with limited area numerical weather prediction models are common to those of other atmospheric modeling efforts[8].

C.4.1 Boundary Relaxation

The effect of inadequate treatment of the lateral boundaries is the creation of instabilities that can destroy the forecast in quite a short period of time. While the first two methods can be shown to be inappropriate, the relaxation method has been found to be effective without any major drawback[9].

To illustrate this scheme consider the equation:

$$\frac{\partial X}{\partial t} + F(X) = 0$$

The time differencing scheme is the following:

$$\frac{X^{n+1} - X^{n-1}}{2\Delta t} + F(X^n) = -k(X^{n+1} - X_{prescribed}) \tag{C.26}$$

The term on the right-hand side determines the degree to which the solution relaxes towards the prescribed value $X_{prescribed}$. Therefore near the boundaries k is chosen to have a large value, whereas away from the boundaries k is small. Solving the associated homogeneous equation gives the explicit solution:

$$X^{n+1}_{solution} = X^{n-1} - 2\Delta t F(X^n)$$

Substitution in (C.26), gives:

$$X^{n+1} = (1 - \alpha)X^{n+1}_{solution} + \alpha X^{n+1}_{prescribed}$$

where

$$\alpha = \frac{2\Delta t k}{1 + 2\Delta t k}$$

The profile chosen for α is specified by:

$$\alpha = 1 - tanh(\frac{j}{2})$$

where j is the number of grid-lengths to the nearer boundary. With these relaxation factors, while the outermost value of X is replaced completely, the others are a combination of internal and external values; the importance of the latter decreasing rapidly. In every time-step linear interpolation is used to update the boundary data between the 12-hourly intervals data is taken[9].

Appendix D

Test Programs

D.1 Program par.c

```
/*
 * par.c
 */

#include <locsys.h>

#define Xgrid   10
#define Ygrid   10
#define REPEAT  10
#define Qmax    50

#define PARXY(f)        par { REP10(REP10,f); }
#define PARX(f)         par { REP10(APP,f); }
#define PARY(f)         par { REP10(APP,f); }
#define APP(f,x)        f(x)
#define REP10(a,b)      a(b,0);a(b,1);a(b,2);a(b,3);a(b,4);\
                        a(b,5);a(b,6);a(b,7);a(b,8);a(b,9)

char    Dummy[4096];

channel ch_right[Xgrid+1][Ygrid];       /* communication rightwards */
channel ch_left[Xgrid+1][Ygrid];        /* communication leftwards */
channel ch_up[Xgrid][Ygrid+1];          /* communication upwards */
channel ch_down[Xgrid][Ygrid+1];        /* communication downwards */

/* multiplexer / demultiplexer buffers: */
float   em_buf[Ygrid], ed_buf[Ygrid],   /* eastern buffers */
        wm_buf[Ygrid], wd_buf[Ygrid],   /* western buffers */
        nm_buf[Xgrid], nd_buf[Xgrid],   /* northern buffers */
        sm_buf[Xgrid], sd_buf[Xgrid];   /* southern buffers */

int     Q;
int     do_nothing;
```

```
#pragma par
par1 (x, y)
{       channel *FromWest, *FromEast, *FromNorth, *FromSouth;
        channel *ToWest, *ToEast, *ToNorth, *ToSouth;
        float   dummy, U_north, U_south, U_west, U_east, U;
        int     r, q;

        ToWest = &ch_left[x][y];
        FromWest = &ch_right[x][y];
        ToEast = &ch_right[x+1][y];
        FromEast = &ch_left[x+1][y];
        ToSouth = &ch_down[x][y];
        FromSouth = &ch_up[x][y];
        ToNorth = &ch_up[x][y+1];
        FromNorth = &ch_down[x][y+1];
        if (do_nothing)
                return;
        for (r = 0; r < REPEAT; r++) {
                par {
                        {       *ToSouth = U;
                                *ToNorth = U;
                                *ToEast = U;
                                *ToWest = U;
                        }
                        {       U_north = *FromNorth;
                                U_south = *FromSouth;
                                U_west = *FromWest;
                                U_east = *FromEast;
                        }
                }
                for (q = 0; q < Q; q++) {
                        dummy = U;
                        dummy = U;
                        dummy = U;
                        dummy = U;
                }
        }
}

par2 (x, y)
{       channel *FromWest, *FromEast, *FromNorth, *FromSouth;
        channel *ToWest, *ToEast, *ToNorth, *ToSouth;
        channel SendReady, RecReady;
        float   U_north, U_south, U_west, U_east, U;
        int     r1, r2, r3, q;

        ResetChan(& SendReady);
        ResetChan(& RecReady);
        ToWest = &ch_left[x][y];
        FromWest = &ch_right[x][y];
```

```
        ToEast = &ch_right[x+1][y];
        FromEast = &ch_left[x+1][y];
        ToSouth = &ch_down[x][y];
        FromSouth = &ch_up[x][y];
        ToNorth = &ch_up[x][y+1];
        FromNorth = &ch_down[x][y+1];
        if (do_nothing)
                return;
        par {
                for (r1 = 0; r1 < REPEAT; r1++) {
                        *ToSouth = U;
                        *ToNorth = U;
                        *ToEast = U;
                        *ToWest = U;
                        SendReady = 0;
                }
                for (r2 = 0; r2 < REPEAT; r2++) {
                        U_north = *FromNorth;
                        U_south = *FromSouth;
                        U_west = *FromWest;
                        U_east = *FromEast;
                        RecReady = 0;
                }
                for (r3 = 0; r3 < REPEAT; r3++) {
                        float   locdummy;
                        RecReady;
                        for (q = 0; q < Q; q++) {
                                locdummy = U;
                                locdummy = U;
                                locdummy = U;
                                locdummy = U;
                        }
                        SendReady;
                }
        }
}

par3 (x, y)
{       channel *FromWest, *FromEast, *FromNorth, *FromSouth;
        channel *ToWest, *ToEast, *ToNorth, *ToSouth;
        float   dummy, U_north, U_south, U_west, U_east, U;
        int     r, q;

        ToWest = &ch_left[x][y];
        FromWest = &ch_right[x][y];
        ToEast = &ch_right[x+1][y];
        FromEast = &ch_left[x+1][y];
        ToSouth = &ch_down[x][y];
        FromSouth = &ch_up[x][y];
        ToNorth = &ch_up[x][y+1];
        FromNorth = &ch_down[x][y+1];
```

```
        if (do_nothing)
                return;
        if ((x ^ y) & 1)
                for (r = 0; r < REPEAT; r++) {
                        *ToSouth = U;
                        *ToNorth = U;
                        *ToEast = U;
                        *ToWest = U;
                        U_north = *FromNorth;
                        U_south = *FromSouth;
                        U_west = *FromWest;
                        U_east = *FromEast;
                        for (q = 0; q < Q; q++) {
                                dummy = U;
                                dummy = U;
                                dummy = U;
                                dummy = U;
                        }
                }
        else
                for (r = 0; r < REPEAT; r++) {
                        U_north = *FromNorth;
                        U_south = *FromSouth;
                        U_west = *FromWest;
                        U_east = *FromEast;
                        *ToSouth = U;
                        *ToNorth = U;
                        *ToEast = U;
                        *ToWest = U;
                        for (q = 0; q < Q; q++) {
                                dummy = U;
                                dummy = U;
                                dummy = U;
                                dummy = U;
                        }
                }
}

test (f)
        void (*f)();
{       channel *In = ChannelFrHost(), *Out = ChannelToHost();
        int     overhead, time;

        do_nothing = TRUE;
        for (Q = 0; Q < 5; Q++) {
                overhead = timer;
                PARXY((*f))
                overhead = timer - overhead;
        }
        do_nothing = FALSE;
        for (Q = 1; Q <= Qmax; Q++) {
```

```
                time = timer;
                PARXY((*f))
                time = timer - time;
                *Out = (Q << 24) | (time - overhead);
        }
}

main ()
{       int     x, y;

        for (x = 0; x < Xgrid+1; x++)
                for (y = 0; y < Ygrid; y++) {
                        ResetChan(&ch_right[x][y]);
                        ResetChan(&ch_left[x][y]);
                }
        for (x = 0; x < Xgrid; x++)
                for (y = 0; y < Ygrid+1; y++) {
                        ResetChan(&ch_up[x][y]);
                        ResetChan(&ch_down[x][y]);
                }
        par {
                west_mux();
                west_demux();
                east_mux();
                east_demux();
                south_mux();
                south_demux();
                north_mux();
                north_demux();
                {       test(par1);
                        test(par2);
                        test(par3);
                }
        }
}

/*---------- multiplexers: ----------*/
#define east_get(y)     em_buf[y] = ch_right[Xgrid][y]
east_mux ()
{       while (TRUE) {
                PARY(east_get)
                /* send em_buf over right link */
        }
}

#define west_get(y)     wm_buf[y] = ch_left[0][y]
west_mux ()
{       while (TRUE) {
                PARY(west_get)
                /* send wm_buf over left link */
        }
```

```
}

#define south_get(x)     sm_buf[x] = ch_down[x][0]
south_mux ()
{       while (TRUE) {
                PARX(south_get)
                /* send sm_buf over lower link */
        }
}

#define north_get(x)     nm_buf[x] = ch_up[x][Ygrid]
north_mux ()
{       while (TRUE) {
                PARX(north_get)
                /* send nm_buf over upper link */
        }
}

/*---------- de-multiplexers: ----------*/
#define west_put(y)      ch_right[0][y] = wd_buf[y]
west_demux ()
{       while (TRUE) {
                /* receive wd_buf from left link */
                PARY(west_put)
        }
}

#define east_put(y)      ch_left[Xgrid][y] = ed_buf[y]
east_demux ()
{       while (TRUE) {
                /* receive ed_buf from right link */
                PARY(east_put)
        }
}

#define south_put(x)     ch_up[x][0] = sd_buf[x]
south_demux ()
{       while (TRUE) {
                /* receive sd_buf from lower link */
                PARX(south_put)
        }
}

#define north_put(x)     ch_down[x][Ygrid] = nd_buf[x]
north_demux ()
{       while (TRUE) {
                /* receive nd_buf from upper link */
                PARX(north_put)
        }
}
```

D.2 Program seq.c

```
/*
 * seq.c
 */

#include <locsys.h>

#define Xgrid   10
#define Ygrid   10
#define REPEAT  10
#define Qmax    50

char    Dummy[4096];

float   U[Xgrid+2][Ygrid+2];
int     Q;
int     do_nothing;

seq1 ()
{       int     x, y, r, q;
        float   dummy, U_north, U_south, U_west, U_east;

        if (do_nothing)
                return;
        for (r = 0; r < REPEAT; r++)
                for (x = 1; x <= Xgrid; x++)
                        for (y = 1; y <= Ygrid; y++) {
                                U_north = U[x][y-1];
                                U_south = U[x][y+1];
                                U_west = U[x-1][y];
                                U_east = U[x+1][y];
                                for (q = 0; q < Q; q++) {
                                        dummy = U[x][y];
                                        dummy = U[x][y];
                                        dummy = U[x][y];
                                        dummy = U[x][y];
                                }
                        }
}

seq2 ()
{       int     x, y, r, q;
        float   dummy, U_north, U_south, U_west, U_east;
        float   *Ux;

        if (do_nothing)
                return;
        for (r = 0; r < REPEAT; r++)
                for (x = 1; x <= Xgrid; x++) {
                        Ux = U[x];
```

```
                        for (y = 1; y <= Ygrid; y++) {
                                U_north = Ux[y-1];
                                U_south = Ux[y+1];
                                U_west = U[x-1][y];
                                U_east = U[x+1][y];
                                for (q = 0; q < Q; q++) {
                                        dummy = Ux[y];
                                        dummy = Ux[y];
                                        dummy = Ux[y];
                                        dummy = Ux[y];
                                }
                        }
                }
}

seq3a ()
{       int     x, y, r, q;
        float   dummy, U_north, U_south, U_west, U_east;
        float   *Up1, *Up2, *Up3, *Up4;

        if (do_nothing)
                return;
        for (r = 0; r < REPEAT; r++) {
                Up1 = Up2 =Up3 = Up4 = & U[0][0];
                for (x = 1; x <= Xgrid; x++) {
                        for (y = 1; y <= Ygrid; y++) {
                                U_north = U[x][y-1];
                                U_south = U[x][y+1];
                                U_west = U[x-1][y];
                                U_east = U[x+1][y];
                                for (q = 0; q < Q; q++) {
                                        dummy = *Up1++;
                                        dummy = *Up2++;
                                        dummy = *Up3++;
                                        dummy = *Up4++;
                                }
                        }
                }
        }
}

seq3b ()
{       int     x, y, r, q;
        float   dummy, U_north, U_south, U_west, U_east;
        float   *Up1, *Up2;

        if (do_nothing)
                return;
        for (r = 0; r < REPEAT; r++) {
                Up1 = Up2 = & U[0][0];
                for (x = 1; x <= Xgrid; x++) {
```

```
                        for (y = 1; y <= Ygrid; y++) {
                                U_north = U[x][y-1];
                                U_south = U[x][y+1];
                                U_west = U[x-1][y];
                                U_east = U[x+1][y];
                                for (q = 0; q < Q; q++) {
                                        dummy = *Up1;
                                        dummy = *Up1++;
                                        dummy = *Up2;
                                        dummy = *Up2++;
                                }
                        }
                }
        }
}

test (f)
        void (*f)();
{       channel *In = ChannelFrHost(), *Out = ChannelToHost();
        int     overhead, time;

        do_nothing = TRUE;
        overhead = timer;
        (*f)();
        overhead = timer - overhead;
        do_nothing = FALSE;
        for (Q = 1; Q <= Qmax; Q++) {
                time = timer;
                (*f)();
                time = timer - time;
                *Out = (Q << 24) | (time - overhead);
        }
}

main()
{       test(seq1);
        test(seq2);
        test(seq3a);
        test(seq3b);
}
```

D.3 Program optim.c

```
/*
 * optim.c
 */

#include <locsys.h>
#include <stdio.h>
```

```
char    Dummy[4096];

#define REPEAT  10000
#define TICK    64000   /* nano-seconds per tick */

float   mat2[10][10];
float   mat1[10];

sub2 (repeat, count)    /* subscript in 2-dim array */
{       int     r, c, i = 2, j = 2;
        float   dummy;

        for (r = 0; r < repeat; r++)
                for (c = 0; c < count; c++)
                        dummy = mat2[i][j];
}

sub2opt_ro (repeat, count)      /* 2-dim subscript with optimization r.o. */
{       int     r, c, i = 2, j = 2;
        float   copy, dummy;

        for (r = 0; r < repeat; r++) {
                copy = mat2[i][j];
                for (c = 0; c < count; c++)
                        dummy = copy;
        }
}

sub2opt_rw (repeat, count)      /* 2-dim subscript with optimization r/w */
{       int     r, c, i = 2, j = 2;
        float   copy, dummy;

        for (r = 0; r < repeat; r++) {
                copy = mat2[i][j];
                for (c = 0; c < count; c++)
                        dummy = copy;
                mat2[i][j] = copy;
        }
}

sub1 (repeat, count)    /* subscript in 1-dim array */
{       int     r, c, i = 2;
        float   dummy;

        for (r = 0; r < repeat; r++)
                for (c = 0; c < count; c++)
                        dummy = mat1[i];
}

sub1opt_ro (repeat, count)      /* 1-dim subscript with optimization r.o. */
```

```
{       int     r, c, i = 2;
        float   copy, dummy;

        for (r = 0; r < repeat; r++) {
                copy = mat1[i];
                for (c = 0; c < count; c++)
                        dummy = copy;
        }
}

subiopt_rw (repeat, count)      /* 1-dim subscript with optimization r/w */
{       int     r, c, i = 2;
        float   copy, dummy;

        for (r = 0; r < repeat; r++) {
                copy = mat1[i];
                for (c = 0; c < count; c++)
                        dummy = copy;
                mat1[i] = copy;
        }
}

poin (repeat, count)            /* pointer dereference */
{       int     r, c;
        float   dummy, *p;

        p = & dummy;
        for (r = 0; r < repeat; r++)
                for (c = 0; c < count; c++)
                        dummy = *p;
}

poinopt_ro (repeat, count)      /* pointer dereference with optimization r.o. */
{       int     r, c;
        float   copy, dummy, *p;

        p = & dummy;
        for (r = 0; r < repeat; r++) {
                copy = *p;
                for (c = 0; c < count; c++)
                        dummy = copy;
        }
}

poinopt_rw (repeat, count)      /* pointer dereference with optimization r/w */
{       int     r, c;
        float   copy, dummy, *p;

        p = & dummy;
        for (r = 0; r < repeat; r++) {
                copy = *p;
```

```
                for (c = 0; c < count; c++)
                        dummy = copy;
                *p = copy;
        }
}

int     time (f, count)
        void    (*f)();
{       int     t1, t2;

        t1 = timer;
        (*f)(REPEAT, count);
        t1 = timer - t1;
        t2 = timer;
        (*f)(2*REPEAT, count);
        t2 = timer - t2;
        return (TICK * (t2 - t1)) / REPEAT;
}

print (value)   /* simulate missing %f directive of printf */
{       value += 5;
        printf(" %2d.%d%d", value/1000, (value%1000)/100, (value%100)/10);
}

test (name, f)
        char    *name;
        void    (*f)();
{       int     i;

        printf("%s\t:", name);
        for (i = 0; i < 7; i++)
                print(time(f, i));
        printf("\n");
}

main ()
{       printf("======= optim.c ======\n");
        test("sub2\t", sub2);
        test("sub2opt_ro", sub2opt_ro);
        test("sub2opt_rw", sub2opt_rw);
        test("sub1\t", sub1);
        test("sub1opt_ro", sub1opt_ro);
        test("sub1opt_rw", sub1opt_rw);
        test("poin\t", poin);
        test("poinopt_ro", poinopt_ro);
        test("poinopt_rw", poinopt_rw);
        printf("====================\n");
}
```

D.4 Program bench.c

```
/*
 * bench.c
 */

#include <locsys.h>
#include <stdio.h>

char    Dummy[4096];

#define COUNT   50000
#define TICK    64000   /* nano-seconds per tick */

float   mat2[10][10];
float   mat1[10];
float   plain;
channel ch, *chp;

empty (count)   /* do nothing */
{       int     n;

        for (n = 0; n < count; n++)
                ;
}

subs (count)    /* 1-dimensional array access */
{       int     i, n;
        float   dummy;

        i = 2;
        for (n = 0; n < count; n++)
                dummy = mat1[i];
}

single (count)  /* plain variable access */
{       int     n;
        float   dummy;

        for (n = 0; n < count; n++)
                dummy = plain;
}

array (count)   /* get a neighbor value in a 2-dimensional array */
{       int     i, j, n;
        float   dummy;

        i = j = 2;
        for (n = 0; n < count; n++)
                dummy = mat2[i+1][j];
}
```

```
loop (count)    /* perform one loop over the local grid */
{       int     i, j, n;

        for (n = 0; n < count/25; n++)
                for (i = 0; i < 5; i++)
                        for (j = 0; j < 5; j++)
                                ;
}

chan_get (count)
{       int     n;
        float   dummy;
        channel *loc_in = chp;

        for (n = 0; n < count; n++)
                dummy = *loc_in;
}

chan_put (count)
{       int     n;
        float   dummy;
        channel *loc_out = chp;

        for (n = 0; n < count; n++)
                *loc_out = dummy;
}

#pragma par
chan (count)    /* perform one channel communication */
{       par {
                chan_get(count);
                chan_put(count);
        }
}

#pragma seq
int     time (f)
        void (*f)();
{       int     t1, t2;

        t1 = timer;
        (*f)(COUNT);
        t1 = timer - t1;
        t2 = timer;
        (*f)(2*COUNT);
        t2 = timer - t2;
        return (TICK * (t2 - t1)) / COUNT;
}

print (name, value)     /* simulate missing %f directive of printf */
```

```
        char    *name;
{       value += 5;
        printf("%s\t= %d.%d%d micro-seconds\n",
                name, value/1000, (value%1000)/100, (value%100)/10);
}

main ()
{       int     i;

        chp = & ch;
        ResetChan(chp);
        printf("======= bench.c ======\n");
        print("Tsubs", time(subs) - time(single));
        print("Tarray", time(array) - time(empty));
        print("Tchan", time(chan) - 2 * time(empty));
        print("Tloop", time(loop) - time(empty));
        printf("======================\n");
}
```

D.5 Program link.c

```
/*
 * link.c
 */

#include <locsys.h>
#include <stdio.h>

#define LEFTLINK  0
#define UPLINK    1
#define RIGHTLINK 2
#define DOWNLINK  3

char    Dummy[4096];

#define COUNT   50000
#define TICK    64000   /* nano-seconds per tick */

channel ch1, ch2;
int     link_time, chan_time;

echo (in, out, count)
        channel *in, *out;
{       int     n;
        float   dummy;
        channel *loc_in = in;
        channel *loc_out = out;

        for (n = 0; n < count; n++) {
```

```
                dummy = *loc_in;
                *loc_out = dummy;
        }
}

bench (in, out, count)
        channel *in, *out;
{       int     n;
        float   dummy;
        channel *loc_in = in;
        channel *loc_out = out;

        *loc_out = dummy;
        for (n = 0; n < count - 1; n++) {
                dummy = *loc_in;
                *loc_out = dummy;
        }
        dummy = *loc_in;
}

empty_echo (in, out, count)
        channel *in, *out;
{       int     n;

        for (n = 0; n < count - 1; n++)
                ;
}

empty_bench (in, out, count)
        channel *in, *out;
{       int     n;

        for (n = 0; n < count; n++)
                ;
}

int     time (f, in, out)
        void (*f)();
        channel *in, *out;
{       int     t1, t2;

        t1 = timer;
        (*f)(in, out, COUNT);
        t1 = timer - t1;
        t2 = timer;
        (*f)(in, out, 2*COUNT);
        t2 = timer - t2;
        return (TICK * (t2 - t1)) / COUNT;
}

#pragma par
```

```
tr00 ()
{       par {
                echo(&ch1, &ch2, 3*COUNT);
                {
                        link_time = time(bench, LINKIN(RIGHTLINK),
                                                LINKOUT(RIGHTLINK)) / 2;
                        chan_time = ( time(bench, &ch2, &ch1)
                                    - time(empty_echo, &ch2, &ch1)
                                    - time(empty_bench, &ch2, &ch1)
                                    ) / 2;
                }
        }
}

#pragma seq
tr10 ()
{       echo(LINKIN(LEFTLINK), LINKOUT(LEFTLINK), 3*COUNT);
}

print (name, value)     /* simulate missing %f directive in printf */
        char    *name;
{       value += 5;
        printf("%s\t= %d.%d%d micro-seconds\n",
                name, value/1000, (value%1000)/100, (value%100)/10);
}

main ()
{       int     *x, *y;

        ResetChan(&ch1);
        ResetChan(&ch2);
        GetCoordinates(LEFTLINK, RIGHTLINK, DOWNLINK, UPLINK, &x, &y);
        if (x == 0 && y == 0) {
                printf("======= link.c =======\n");
                tr00();
                print("link-time", link_time);
                print("chan-time", chan_time);
                print("Tlink", link_time - chan_time);
                printf("====================\n");
        } else
                tr10();
}
```

Appendix E

Implementation of DYN in C

E.1 Program DYN.c

```
/* ------------------------------------------------------------------ */
/*      SUBROUTINE DYN;                                               */
/*                                                                    */
/* **** DYN - SUBROUTINE TO PERFORM ALL DYNAMICAL COMPUTATIONS        */
/*                                                                    */
/*      BRONNO DE HAAN     HIRLAM         870207                     */
/*                                                                    */
/*      DYN CALCULATES NEW VALUES OF PS T U V AND Q                   */
/*                                                                    */
/* **   INTERFACE                                                     */
/*                                                                    */
/*      CALL DYN                                                      */
/*      DYN WILL BE CALLED IN GEMINI                                  */
/*                                                                    */
/* **   METHOD                                                        */
/*      RESEARCH MANUAL 3: ECMWF FORECAST MODEL                       */
/*      DYNAMICAL PART                                                */
/*      IN CONTRAST TO THE ORIGINAL ECMWF CODE THIS CODE IS BASED ON  */
/*      OPERATING WITH HORIZONTAL LAYERS. WE START FROM BELOW ( = LEV) */
/*      IN THE FIRST (SMALL) PART WE COMPUTE THE TENDENCY FOR PS , NEXT */
/*      IN THE (BIG) PART WE COMPUTE THE OTHER TENDENCIES.            */
/* ------------------------------------------------------------------ */
/* Converted to C 29-03-88 by Dick Streefland & Hans Middelkoop       */
/* ------------------------------------------------------------------ */

#include <stdio.h>

extern double atof();

#define GRIDLOOP for (x=1; x <= MLON; x++) for (y=1; y <= MLAT; y++)

#define MLON 34
#define MLAT 34
```

```
#define MLEV 9
#define MLONPT MLON+2
#define MLATPT MLAT+2
#define NLTVIR 1

#ifdef HARRIS
#define INITVAL 1E-13
#else
float INITVAL;
#endif

#ifdef HARRIS
extern  long    clock();
#endif

init1 (v)
        float   v[MLEV];
{       int     k;

        for (k = 0; k < MLEV; k++)
                v[k] = INITVAL/(k+1);
}

init2 (v)
        float   v[MLONPT][MLATPT];
{       int     i, j;

        for (i = 0; i < MLONPT; i++)
                for (j = 0; j < MLATPT; j++)
                        v[i][j] = INITVAL;
}

init3 (v)
        float   v[MLEV][MLONPT][MLATPT];
{       int     i, j, k;

        for (k = 0; k < MLEV; k++)
                for (i = 0; i < MLONPT; i++)
                        for (j = 0; j < MLATPT; j++)
                                v[k][i][j] = INITVAL;
}

float   *ALFAp, *ALNPSZp, *BETAp, *DIVKp, *DIVSUMp, *DLNPKp,
        *DPKp, *DPSDTp, *DQDTp, *DTDTp, *DUDTp, *DVDTp,
        *EDPDEp, *EKp, *FPARp, *HXVp, *HYUp, *OMEGAp, *PHISp,
        *PHIp, *PKMp, *PKPp, *PPp, *PSZp, *QZp, *QZpm, *QZpp, *RAHXHYp,
        *RDPKp, *RHXUp, *RHYVp, *TVp, *TZp, *TZpm, *TZpp, *UUp, *UZp,
        *UZpm, *UZpp, *VVp, *VZp, *VZpm, *VZpp, *ZAHXHYp,
        *ZKp, *ZLOGMp;

float   RAHXHY[MLONPT][MLATPT], ZAHXHY[MLONPT][MLATPT],
```

```
        HXV[MLONPT][MLATPT], HYU[MLONPT][MLATPT],
        RHXU[MLONPT][MLATPT], RHYV[MLONPT][MLATPT],
        DPK  [MLONPT][MLATPT], PHI [MLONPT][MLATPT],
        DIVSUM[MLONPT][MLATPT], PKM  [MLONPT][MLATPT],
        PKP [MLONPT][MLATPT] ,ZLOGM [MLONPT][MLATPT],
        ZLOGP[MLONPT][MLATPT], RDPK[MLONPT][MLATPT],
        PP [MLONPT][MLATPT], DLNPK[MLONPT][MLATPT],
        DIVK[MLONPT][MLATPT],OMEGA [MLONPT][MLATPT],
        ALFA [MLONPT][MLATPT], BETA[MLONPT][MLATPT],
        TV[MLONPT][MLATPT], EDPDE[MLONPT][MLATPT],
        DPSDT[MLONPT][MLATPT], DTDT[MLEV][MLONPT][MLATPT],
        DUDT[MLEV][MLONPT][MLATPT], DVDT[MLEV][MLONPT][MLATPT],
        DQDT[MLEV][MLONPT][MLATPT], UU[MLONPT][MLATPT],
        VV[MLONPT][MLATPT], ZK[MLONPT][MLATPT],
        EK[MLONPT][MLATPT], UZ[MLEV][MLONPT][MLATPT],
        VZ[MLEV][MLONPT][MLATPT], QZ[MLEV][MLONPT][MLATPT],
        TZ[MLEV][MLONPT][MLATPT], PHIS[MLONPT][MLATPT],
        PSZ[MLEV][MLONPT][MLATPT], FPAR[MLONPT][MLATPT],
        ALNPSZ[MLONPT][MLATPT], AHYB[MLEV+1], BHYB[MLEV+1],
        ADLNPK[MLEV];
main()
{       long    time;
        int x, y, K;

        float   AKM, BKM, CLN, CPD, FCPVD1, RDLO, RDLA, ZLNTOP,
                ZLNTP, ZRDLA, ZRDLAH, ZRDLAR, ZRDLO, ZRDLOH, ZRDLOR, ZREPM1,
                ZDLNPK, ZRDLO4, ZRDLA4, ZLNTP2, DAKH, DBK, DBKH, ZRGASH;

#ifdef HARRIS
#       define  C0_0    0.0
#       define  C0_125  0.125
#       define  C0_25   0.25
#       define  C0_5    0.5
#       define  C1_0    1.0
#       define  C2_0    2.0
#       define  C4_0    4.0
#       define  RDLAM   10E-5   /* dummy */
#       define  RDTH    10E-5   /* dummy */
#       define  CLN2R1  10.0    /* dummy */
#       define  REAR    6.371E+6
#       define  CAPPA   0.2857143
#       define  RGASD   287.04
#       define  RGASV   461.51
#       define  CPV     1869.46
#else
        float   C0_0    = atof("0.0");
        float   C0_125  = atof("0.125");
        float   C0_25   = atof("0.25");
        float   C0_5    = atof("0.5");
        float   C1_0    = atof("1.0");
        float   C2_0    = atof("2.0");
```

```
        float   C4_0    = atof("4.0");
        float   RDLAM   = atof("10E-5");
        float   RDTH    = atof("10E-5");
        float   CLM2R1  = atof("10.0");
        float   REAR    = atof("6.371E+6");
        float   CAPPA   = atof("0.2857143");
        float   RGASD   = atof("287.04");
        float   RGASV   = atof("461.51");
        float   CPV     = atof("1869.46");

        INITVAL = atof("1E-13");
#endif

        /* initialization */
        init1(AHYB); init1(BHYB); init1(ADLNPK);

        init2(RAHXHY); init2(ZAHXHY); init2(HXV); init2(HYU);
        init2(RHXU); init2(RHYV); init2(DPK); init2(PHI);
        init2(DIVSUM); init2(PKM); init2(PKP); init2(ZLOGM);
        init2(ZLOGP); init2(RDPK); init2(PP); init2(DLNPK);
        init2(DIVK); init2(OMEGA); init2(ALFA); init2(BETA);
        init2(TV); init2(EDPDE); init2(DPSDT); init2(UU);
        init2(VV); init2(ZK); init2(EK); init2(PHIS);
        init2(ALNPSZ); init2(FPAR);

        init3(DUDT); init3(DVDT); init3(DTDT); init3(DQDT);
        init3(VZ); init3(QZ); init3(UZ); init3(TZ); init3(PSZ);

/* ------------------------------------------------------------------ */
        printf("DYN starts\n");

#ifdef HARRIS
        time = clock();
#else
        time = timer;
#endif

        RDLO = RDLAM;
        RDLA = RDTH;


/*      INITIALISATION OF PHYSIC CONSTANTS                            */

        CPD  = RGASD / CAPPA;

/* ------------------------------------------------------------------ */

/*      COMPUTATION OF D PS / D T                                     */

        UUp = & UU[0][0];
        VVp = & VV[0][0];
```

```
        ZAHXHYp = & ZAHXHY[0][0];
        RAHXHYp = & RAHXHY[0][0];
        RHXUp = & RHXU[0][0];
        RHYVp = & RHYV[0][0];
        GRIDLOOP {
                 *UUp++ = C0_0;
                 *VVp++ = C0_0;
                 *ZAHXHYp = REAR / (*RHXUp++ *  *RHYVp++);
                 *RAHXHYp++ = C1_0  / *ZAHXHYp++;
        } /* GRIDLOOP */

        for (K = 0; K < MLEV; K++) {

                DAKH = (AHYB[K+1]-AHYB[K])*C0_5;
                DBKH = (BHYB[K+1]-BHYB[K])*C0_5;

                DPKp = & DPK[0][0];
                PSZp = & PSZ[K][0][0];
                GRIDLOOP {
                    *DPKp++ = DAKH + DBKH * *PSZp++;
                } /* GRIDLOOP */

                UUp = & UU[0][0];
                VVp = & VV[0][0];
                DPKp = & DPK[0][0];
                UZp = & UZ[K][0][0];
                VZp = & VZ[K][0][0];
                GRIDLOOP {
                    *UUp++ = *UUp - *UZp++ * (*DPKp+DPK[x+1][y]);
                    *VVp++ = *VVp - *VZp++ * (*DPKp+DPK[x][y+1]);
                } /* GRIDLOOP */

        } /* LEVELLOOP */

        UUp = & UU[0][0];
        VVp = & VV[0][0];
        DPSDTp = & DPSDT[0][0];
        HXVp = & HXV[0][0];
        HYUp = & HYU[0][0];
        RAHXHYp = & RAHXHY[0][0];
        GRIDLOOP {
                 *DPSDTp++ = *RAHXHYp++ *
    (  (   *UUp++ * *HYUp++ - UU[x-1][y] * HYU[x-1][y]  )* RDLO
     + (   *VVp++ * *HXVp++ - VV[x][y-1] * HXV[x][y-1]  )* RDLA  );
        } /* GRIDLOOP */

/*      END OF COMPUTATION OD D PS / D T                                  */

/* ------------------------------------------------------------------- */

/*      START BIG OUTER K LOOP                                            */
```

```
/*      INITIALISATION                                                  */

        PHIp = & PHI[0][0];
        PHISp = & PHIS[0][0];
        EDPDEp = &  EDPDE[0][0];
        PKMp = & PKM[0][0];
        PSZp = & PSZ[K][0][0];
        ALNPSZp = & ALNPSZ[0][0];
        ZLOGMp = & ZLOGM[0][0];
        DIVSUMp = & DIVSUM[0][0];
        GRIDLOOP {
            *PHIp++      = *PHISp++;
            *EDPDEp++    = C0_0;
            *PKMp++      = *PSZp++;
            *ZLOGMp++    = *ALNPSZp++;
            *DIVSUMp++   = C0_0;
        } /* GRIDLOOP */

/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
        for (K = MLEV-1; K >= 0; K--) {
/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */

        AKM = AHYB[K];
        BKM = BHYB[K];
        DBK = BHYB[K+1]-BHYB[K];

        PSZp = & PSZ[K][0][0];
        PKMp = & PKM[0][0];
        PKPp = & PKP[0][0];
        DPKp = & DPK[0][0];
        RDPKp = & RDPK[0][0];
        DTDTp = & DTDT[K][0][0];
        DUDTp = & DUDT[K][0][0];
        DVDTp = & DVDT[K][0][0];
        DQDTp = & DQDT[K][0][0];

        GRIDLOOP {
            *PKPp = *PKMp;
            *PKMp = AKM + BKM * *PSZp++;
            *DPKp = *PKPp++ - *PKMp++;
            *RDPKp++ = C1_0 / *DPKp++;

            *DTDTp++ = C0_0;
            *DUDTp++ = C0_0;
            *DVDTp++ = C0_0;
            *DQDTp++ = C0_0;
        } /* GRIDLOOP */

        PPp = & PP[0][0];
        DLNPKp = & DLNPK[0][0];
```

```
        ALFAp = & ALFA[0][0];
        BETAp = & BETA[0][0];
        ALNPSZp = & ALNPSZ[0][0];
/* ------------------------------------------------------------------ */
        if (K == 0) {
/* ------------------------------------------------------------------ */
        ZLNTOP = CLN2R1;
        ZLNTP2 = C2_0*ZLNTOP;

        GRIDLOOP {
                *PPp++ = *ALNPSZp++;
                *DLNPKp++ = ZLNTP2;
                *ALFAp++ = ZLNTOP;
                *BETAp++ = ZLNTOP;
        } /* GRIDLOOP */

/* ------------------------------------------------------------------ */
        } else {
/* ------------------------------------------------------------------ */
        ZDLNPK = ADLNPK[K];

        GRIDLOOP {
                *DLNPKp = ZDLNPK;
                *PPp++ = *ALNPSZp++;
                *ALFAp++ = *DLNPKp * C0_5;
                *BETAp++ = *DLNPKp++ * C0_5;
        } /* GRIDLOOP */

/* ------------------------------------------------------------------ */
        } /* if */
/* ------------------------------------------------------------------ */

/*      COMPUTE VIRTUAL TEMPERATURE AT LEVEL K                        */
/*      COMPUTE PHI AT FULL K LEVEL                                   */

        TVp = & TV[0][0];
        TZp = & TZ[K][0][0];
        QZp = & QZ[K][0][0];
        ALFAp = & ALFA[0][0];
        PHIp = & PHI[0][0];
        if (NLTVIR) {

        ZREPM1 = RGASV / RGASD - C1_0;

        GRIDLOOP {
                *TVp = *TZp++ * (C1_0 + ZREPM1 * *QZp++);
                *PHIp++ = *PHIp++ + *ALFAp++ * RGASD * *TVp++;
        } /* GRIDLOOP */

        } else {
```

```
        GRIDLOOP {
                *TVp = *TZp++;
                *PHIp++ = *PHIp+ *ALFAp+++ RGASD * *TVp++;
        } /* GRIDLOOP */

        } /* if */

        UUp = & UU[0][0];
        UZp = & UZ[K][0][0];
        DPKp = & DPK[0][0];
        HYUp = & HYU[0][0];
        GRIDLOOP {
                *UUp++ = C0_5 * ( DPK[x+1][y] + *DPKp++ * *UZp++ * *HYUp++);
        } /* GRIDLOOP */

        VVp = & VV[0][0];
        VZp = & VZ[K][0][0];
        DPKp = & DPK[0][0];
        HXVp = & HXV[0][0];
        GRIDLOOP {
                *VVp++ = C0_5 * ( DPK[x][y+1]+ *DPKp++ *  *VZp*  *HXVp++);
        } /* GRIDLOOP */

/*       COMPUTE DIVERGENCE AT LEVEL K                                    */

        UUp = & UU[0][0];
        VVp = & VV[0][0];
        RAHXHYp = & RAHXHY[0][0];
        DIVKp = & DIVK[0][0];

        GRIDLOOP {
                *DIVKp++ = *RAHXHYp++ * ( (  *UUp++ - UU[x-1][y] ) * RDLO
                                + (  *VVp++ -VV[x][y-1] ) * RDLA );
        } /* GRIDLOOP */

/*      COMPUTATION OF VERTICAL ADVECTION TERMS                           */

/*      INFLOW FROM BELOW                                                 */

/*      IF (K.EQ.NLEV) GOTO 630                                          */
        if (K != NLEV-1) {
        DTDTp = & DTDT[K][0][0];
        DUDTp = & DUDT[K][0][0];
        DVDTp = & DVDT[K][0][0];
        DQDTp = & DQDT[K][0][0];
        UZp = & UZ[K][0][0];
        UZpp = & UZ[K+1][0][0];
        TZp = & TZ[K][0][0];
        TZpp = & TZ[K+1][0][0];
        QZp = & QZ[K][0][0];
        QZpp = & QZ[K+1][0][0];
```

```
        VZp = & VZ[K][0][0];
        VZpp = & VZ[K+1][0][0];
        DPKp = & DPK[0][0];
        RDPKp = & RDPK[0][0];
        EDPDEp = & EDPDE[0][0];
        GRIDLOOP {
                *DTDTp++ = *DTDTp - C0_5 *  *RDPKp * *EDPDEp *
                                    (  *TZpp++ - *TZp++ );
                *DQDTp++ = *DQDTp - C0_5 *  *RDPKp++ * *EDPDEp *
                                    (  *QZpp++ - *QZp++ );
                *DUDTp++ = *DUDTp - C0_5 /(  *DPKp +   DPK[x+1][y] ) *
                                  ( *EDPDEp + EDPDE[x+1][y] ) *
                                    (  *UZpp++ - *UZp++ );
                *DVDTp++ = *DVDTp - C0_5 /(  *DPKp++ + DPK[x][y+1] ) *
                                  ( *EDPDEp++ + EDPDE[x][y+1] ) *
                                    (  *VZpp++ - *VZp++ );
        } /* GRIDLOOP */

        } /* IF */

/*      IF (K.EQ.1) GOTO 660                                              */
        if (K != 0) {

/*      IF  = NO INFLOW FROM ABOVE AND NO UPDATE NEED                     */

/*      UPDATE VERTICAL VELOCITY TERM                                     */

        DIVKp = & DIVK[0][0];
        DPSDTp = & DPSDT[0][0];
        EDPDEp = & EDPDE[0][0];
        GRIDLOOP {
                *EDPDEp++ = *EDPDEp + DBK * *DPSDTp++ + *DIVKp++;
        } /* GRIDLOOP */

/*      INFLOW FROM ABOVE                                                 */

        DTDTp = & DTDT[K][0][0];
        DUDTp =  & DUDT[K][0][0];
        DVDTp =  & DVDT[K][0][0];
        DQDTp =  & DQDT[K][0][0];
        UZp =  & UZ[K][0][0];
        UZpm =  & UZ[K-1][0][0];
        TZp =  & TZ[K][0][0];
        TZpm =  & TZ[K-1][0][0];
        QZp =  & QZ[K][0][0];
        QZpm =  & QZ[K-1][0][0];
        VZp =  & VZ[K][0][0];
        VZpm =  & VZ[K-1][0][0];
        DPKp =  & DPK[0][0];
        RDPKp =  & RDPK[0][0];
        EDPDEp =  & EDPDE[0][0];
```

```
        GRIDLOOP {
                *DTDTp++ = *DTDTp - C0_5 *  *RDPKp* *EDPDEp*
                                    (  *TZp++ -*TZpm++ );
                *DQDTp++ = *DQDTp - C0_5 *  *RDPKp * *EDPDEp *
                                    (*QZp++ - *QZpm++);
                *DUDTp++ = *DUDTp - C0_5 /(  *DPKp +  DPK[x+1][y] ) *
                                  ( *EDPDEp +EDPDE[x+1][y] ) *
                                    (  *UZp++ - *UZpm++ );
                *DVDTp++ = *DVDTp - C0_5 /(  *DPKp++ +  DPK[x][y+1] ) *
                                  ( *EDPDEp++ +EDPDE[x][y+1] ) *
                                    (  *VZp++ - *VZpm++ );
        } /* GRIDLOOP */

        } /* IF */

/*      COMPUTE OMEGA FOR ENERGY CONVERSION TERM                          */

        OMEGAp = & OMEGA[0][0];
        if (NLTVIR) {

        FCPVD1 = CPV/CPD-C1_0;

        QZp =  & QZ[K][0][0];
        GRIDLOOP {
                *OMEGAp++ = CAPPA / ( C1_0 + FCPVD1* *QZp++ );
        } /* GRIDLOOP */

        } else {

        GRIDLOOP {
                *OMEGAp++ = CAPPA;
        } /* GRIDLOOP */

        } /* if */

        OMEGAp =  & OMEGA[0][0];
        DPSDTp =  & DPSDT[0][0];
        TVp =  & TV[0][0];
        VVp =  & VV[0][0];
        UUp =  & UU[0][0];
        PPp =  & PP[0][0];
        DIVKp =  & DIVK[0][0];
        DIVSUMp =  & DIVSUM[0][0];
        RDPKp =  & RDPK[0][0];
        DLNPKp =  & DLNPK[0][0];
        BETAp =  & BETA[0][0];
        RAHXHYp = & RAHXHY[0][0];
        GRIDLOOP {
                *OMEGAp++ =
                *OMEGAp * *RDPKp++ * (   *TVp *
                   (    *DLNPKp++ * (  *DIVSUMp++ + *DPSDTp++ )
```

```
                    +   *BETAp++  *    *DIVKp++                )
      + C0_25 *  *RAHXHYp++ *
      ( (                          *UUp++   *
                (     TV[x+1][y]  +   *TVp ) *
                (     PP[x+1][y]  -   *PPp )
                +                     UU[x-1][y]     *
                (      *TVp +  TV[x-1][y]   ) *
                (      *PPp -  PP[x-1][y]   )   ) *  RDLO
      + (                          *VVp++   *
                (     TV[x][y+1]  +   *TVp ) *
                (     PP[x][y+1]  -   *PPp )
                +                     VV[x][y-1]     *
                (      *TVp++   + TV[x][y-1]   ) *
                (      *PPp++   - PP[x][y-1]   )   ) *  RDLA  )  );
        } /* GRIDLOOP */

/*      END OF COMPUTATION OF OMEGA                                          */

/*      COMPUTE ABS VORTICITY + ENERGY                                       */

        EKp =  & EK[0][0];
        UZp =  & UZ[K][0][0];
        VZp =  & VZ[K][0][0];
        HXVp =  & HXV[0][0];
        RHXUp =  & RHXU[0][0];
        HYUp =  & HYU[0][0];
        RHYVp =  & RHYV[0][0];
        GRIDLOOP {
                *EKp++ = C0_25 *
          ( ( UZ[K][x-1][y] * UZ[K][x-1][y] * HYU[x-1][y]
                +  *UZp * *UZp++ * *HYUp++
            ) * *RHYVp++ +
            ( VZ[K][x][y-1] * VZ[K][x][y-1] * HXV[x][y-1]
                +  *VZp * *VZp++ * *HXVp++
            ) * *RHXUp++
          );
        } /* GRIDLOOP */

        ZRDLO4 = C4_0*RDLO;
        ZRDLA4 = C4_0*RDLA;

        UZp =  & UZ[K][0][0];
        VZp =  & VZ[K][0][0];
        RHXUp =  & RHXU[0][0];
        RHYVp =  & RHYV[0][0];
        ZKp =  & ZK[0][0];
        FPARp =  & FPAR[0][0];
        ZAHXHYp =  & ZAHXHY[0][0];
        DPKp =  & DPK[0][0];
        GRIDLOOP {
                *ZKp++ =
```

```
            (      *FPARp++    *
              (   *ZAHXHYp++
                + ZAHXHY[x+1][y] + ZAHXHY[x][y+1] + ZAHXHY[x+1][y+1]
              ) +
              ( ZRDLO4 *
                 ( VZ[K][x+1][y] / RHYV [x+1][y] - *VZp++ / *RHYVp++ )
              - ZRDLA4 *
                 ( UZ[K][x][y+1] / RHXU [x][y+1] - *UZp++ / *RHXUp++ )
              )
            ) /
              (   *ZAHXHYp++ * *DPKp++
                    + ZAHXHY[x+1][y]*DPK[x+1][y]
                    + ZAHXHY[x][y+1]*DPK[x][y+1]
                    + ZAHXHY[x+1][y+1]*DPK[x+1][y+1]
              ) ;
        } /* GRIDLOOP */

/*      ADD HORIZONTAL ADVECTION TO THE TENDENCIES                          */

        ZRDLOR = RDLO/REAR;
        ZRDLAR = RDLA/REAR;
        ZRGASH = RGASD*CO_5;
        ZRDLOH = RDLO*CO_5;
        ZRDLAH = RDLA*CO_5;

        DUDTp =  & DUDT[K][0][0];
        DVDTp =  & DVDT[K][0][0];
        DQDTp =  & DQDT[K][0][0];
        DTDTp =  & DTDT[K][0][0];
        RHXUp =  & RHXU[0][0];
        RHYVp =  & RHYV[0][0];
        UUp =  & UU[0][0];
        VVp =  & VV[0][0];
        PPp =  & PP[0][0];
        TVp =  & TV[0][0];
        TZp =  & TZ[K][0][0];
        QZp =  & QZ[K][0][0];
        ZKp =  & ZK[0][0];
        EKp =  & EK[0][0];
        PHIp =  & PHI[0][0];
        RDPKp =  & RDPK[0][0];
        RAHXHYp =  & RAHXHY[0][0];
        OMEGAp =  & OMEGA[0][0];
        GRIDLOOP {
                *DUDTp++ = *DUDTp + *RHXUp++ *
                ( CO_125 * ( ZK[x][y-1]+ *ZKp )
                * ( *VVp + VV[x+1][y] + VV[x][y-1] + VV[x-1][y-1] )
                - ZRDLOR    *
               ( PHI[x+1][y] - *PHIp++ + EK[x+1][y] - *EKp
                + ZRGASH * (*TVp + TV[x+1][y]) * (PP[x+1][y] - *PPp)));
```

```
                *DVDTp++ = *DVDTp -  *RHYVp++ *
                ( CO_125 * ( ZK[x-1][y] + *ZKp++ )
                     * ( UU[x-1][y+1]+UU[x][y+1]
                        +UU[x-1][y]+ *UUp )
                + ZRDLAR    *
               (  PHI[x][y+1]- *PHIp++ +  EK[x][y+1]- *EKp++
                + ZRGASH * (*TVp++ + TV[x][y+1]) * (PP[x][y+1] - *PPp++)));

                *DTDTp++ = *DTDTp  -  *RDPKp *  *RAHXHYp *
           ( ZRDLOH * (  *UUp * ( TZ[K][x+1][y]- *TZp )
                              - UU[x-1][y] * ( TZ[K][x-1][y]- *TZp ) )
           + ZRDLAH * (  *VVp * ( TZ[K][x][y+1]- *TZp )
                              - VV[x][y-1] * ( TZ[K][x][y-1]- *TZp++ ) ) )
            +  *OMEGAp;

                *DQDTp++ = *DQDTp - *RDPKp++ * *RAHXHYp++ *

           ( ZRDLOH * (  *UUp++ * ( QZ[K][x+1][y] - *QZp )
                              - UU[x-1][y] * ( QZ[K][x-1][y] - *QZp ) )
           + ZRDLAH * (  *VVp++ * ( QZ[K][x][y+1] - *QZp )
                              - VV[x][y-1] * ( QZ[K][x][y-1] - *QZp++ ) ) );
        } /* GRIDLOOP */

/*      UPDATE PHI PART TWO      FROM LEVEL (K) TO LEVEL (K-1/2)          */
/*      UPDATE SUM OF DIVERGENCE FROM LEVEL NLEV TO LEVEL K               */

/*      -----------------                                                 */
        if (K > 0 ) {
/*      -----------------                                                 */
        PHIp =  & PHI[0][0];
        BETAp =  & BETA[0][0];
        TVp =  & TV[0][0];
        DIVSUMp =  & DIVSUM[0][0];
        DIVKp =  & DIVK[0][0];
        GRIDLOOP {
                *PHIp++ = *PHIp+ *BETAp++ * RGASD * *TVp++;
                *DIVSUMp++ = *DIVSUMp+ *DIVKp++;
        } /* GRIDLOOP */
/*      ------                                                            */
        } /* if */
/*      ------                                                            */
/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */
        } /* GRIDLOOP */
/* = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = */

/*      END OF BIG OUTER LOOP OVER K                                      */

#ifdef HARRIS
        time = clock();
#else
        time = (timer - time) * 64;
```

```
#endif
        printf("DYN ready.\n");
        printf("Time used    : %d milli-seconds \n", (time+500)/1000);
        time /= MLON * MLAT;
        printf("Per gridpoint: %d u-seconds \n", time);
} /* main */
```

Appendix F

Test and working environment

This appendix describes the environment in which the development of the test software w.r.t. the HIRLAM model has taken place.

F.1 Hardware

In order to support and stimulate small high technology companies it was decided to use hardware from the hi-tech company Parsec, selling special purpose systems as well as general purpose hardware related with transputers. The system finally delivered consists of the following items:

- A Rack with power supply and backplane.
- 4 Double eurocard processor boards. Each board has 4 transputers with 1 Mb of RAM for each transputer[1].
- 1 Double eurocard board containing a transputer to/from RS232 link adapter

Each processor-board offers the possibility to put either the T414 or the T800 version of a transputer on the board, the second being the fastest but not yet available in a reliable version. In this way the upgrade to the fastest possible configuration is relatively easy.

The links of each transputer can be freely connected to any other transputer, so any network structure can be built from the supplied hardware. In the case of the calculations involved with the weather prediction model a mesh[2] architecture was chosen, in order to obtain an easy mapping from the HIRLAM model to the actual hardware. One of the transputers at the edge of the network is connected via the RS232 adapter to the host machine, in this case a HCX-9, a UNIX machine from Harris.

The architecture of the test hardware amounts to the picture shown in figure F.1.

[1] At the time of this writing only one board has 4 T414 transputers.
[2] Network where transputers are connected like a square grid.

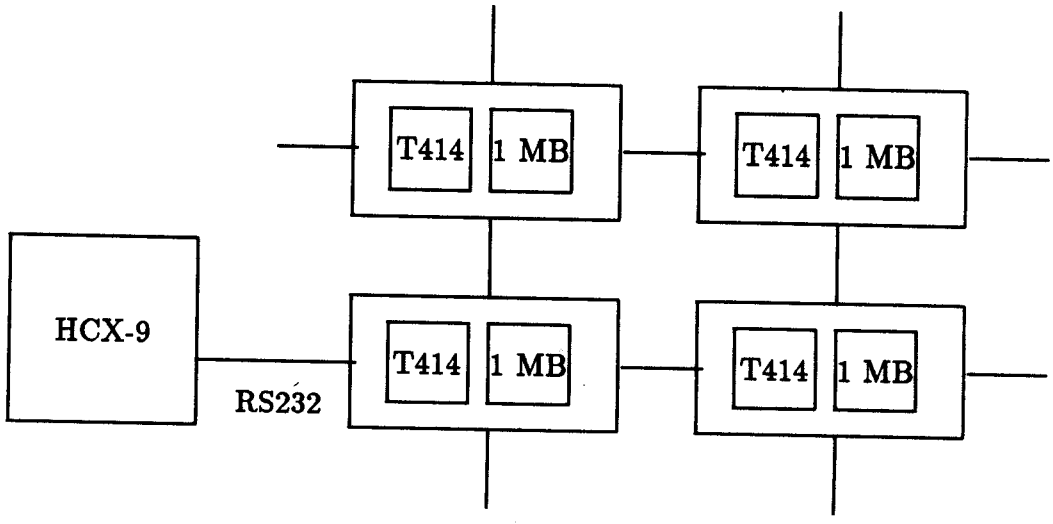


Figure F.1: Hardware Configuration

F.2 Software

Development of software was done on the HCX-9 connected to the transputer network. For the transputer system a cross-compiler in development for the language C was supplied by Parsec [18]. This compiler was used for further programming the transputers.

F.2.1 Support

In order to enhance the capabilities offered by the C-compiler, considerable effort was made to provide a reliable programming environment. Eventually the software development system included the following:

- C cross-compiler on the HCX-9.
- Bootstrapping software to load compiled C programs on a complete network of transputers.
- A monitor to communicate with the transputers and perform several other functions like booting, analyzing the network, and simulation of some C standard I/O routines on the HCX-9.

This development took place in close coöperation with Parsec, which also used the environment and a resulting demonstration program on the CE-BIT in Hannover. Together with Harris, Arcobel and Parsytec, a project was started in order to create a fast communication medium between the HCX-9 and the transputer network as a replacement for the slower RS232 connection.

Bibliography

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

[2] *Reader Numerieke Methoden in de Meteorologie en de Fysische Oceanografie*, University of Utrecht, D 85-3.

[3] Mesinger F. *Finite-difference methods for the linear advection equation*, Seminar 1983, Numerical methods for weather prediction: ECMFW Reading, Shinfield Park, Reading, sept. 5-9, 1984, vol 1.

[4] Janjic Z.I. and Mesinger F. *Finite-difference methods for the shallow water equations on various horizontal grids*, Seminar 1983, idem, vol 1.

[5] Arakawa A. *The use of integral constraints in designing finite-difference schemes for the two-dimensional advection equations*, Seminar 1983, idem, vol 1.

[6] Arakawa A. *Vertical differencing of the primitive equations*, Numerical methods for weather prediction, Seminar 1983, idem, vol 1.

[7] Gerrity J.P. and McPherson, 1971. *On an efficient scheme for the numerical integration of a primitive equation barotropic model*, Seminar 1983, idem, vol 1.

[8] Davies H.C. *Techniques for Limited Area Modeling*, Seminar 1983, idem, vol 2.

[9] Dell'Osso L. *A practical approach to the problems of limited area modelling*, Seminar 1983, idem, vol 2.

[10] *Documentation Manual of the HIRLAM LEVEL 1 Forecast Model*, KNMI, de Bilt, the Nederlands.

[11] Machenhauer B. *The spectral method* chapter 3 of GARP Publication Series No 17. Vol II.

[12] Haltiner, G.J., and R.T. Williams, 1980: *Numerical Prediction and Dynamic Meteorology* 2nd., Wiley, New York.

[13] INMOS, *transputer reference manual*, oct. 1986

[14] INMOS, *T2/T4 transputer instruction set*, july 1986

[15] INMOS, *IMS T800 transputer data sheet*, april 1987

[16] UNICOM, *Parallel C Compiler Features*, 1987

[17] Wijbrans, K. and Kurver, R., *The Development of a Parallel C Compiler*

[18] UNICOM, *Description of TCC version 1.10a*