Abstract Level Parallelization of Finite Difference Methods

EDWIN VOLLEBREGT

Faculty of Technical Mathematics and Informatics, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands; tel: +31-15 278 58 05; fax: +31-15 278 72 09; e-mail: edwin@pa.twi.tudelft.nl

ABSTRACT

A formalism is proposed for describing finite difference calculations in an abstract way. The formalism consists of index sets and stencils, for characterizing the structure of sets of data items and interactions between data items ("neighbouring relations"). The formalism provides a means for lifting programming to a more abstract level. This simplifies the tasks of performance analysis and verification of correctness, and opens the way for automatic code generation.

The notation is particularly useful in parallelization, for the systematic construction of parallel programs in a process/channel programming paradigm (e.g., message passing). This is important because message passing, unfortunately, still is the only approach that leads to acceptable performance for many more unstructured or irregular problems on parallel computers that have non-uniform memory access times. It will be shown that the use of index sets and stencils greatly simplifies the determination of which data must be exchanged between different computing processes.

1 INTRODUCTION

One of the main causes for the complexity of parallel programming is the necessity of *data locality*. Truly scalable and cost-effective parallel computers will have a physically distributed memory organization. This causes memory access times to be non-uniform, such that memory can be said to be close to or far from a processor. The key to high performance on these machines is to optimally exploit the structure of the memory system, for instance by minimizing the number of non-local data items that are needed by a processor.

Control over data locality is maximized if a process/channel parallel programming paradigm is used in which computing processes work on private data only, and exchange data with each other through communication [12]. At the same time this programming paradigm is also the most difficult to manage for more irregular applications. It requires that a program is written for each computing process that explicitly states which data items must be exchanged between the processes and at which times. Unfortunately this (currently) is the only approach that leads to acceptable performance for many more unstructured or irregular problems.

The development of software with a process/channel paradigm can be simplified by lifting the programming to a higher level of abstraction. Whereas many numerical applications are nicely structured on a conceptual level (as in a finite element mesh), this structure may have been obscured in a computer program. Determining data dependences and corresponding communication required is much easier at the higher level. Therefore a means is required for specifying the program at the conceptual level and then methodically transforming it into the desired form.

Received December 1996
Revised November 1997

ISSN 1058-9244/97/$8
Scientific Programming, Vol. 6, pp. 331–344 (1997)

In this work we describe our approach for the parallelization of a complex finite difference application with irregularly shaped grids. This approach has been applied for parallelization of the operational 3D shallow water simulation program TRIWAQ of the Dutch National Institute for Coastal and Marine Management, Rijkswaterstaat/RIKZ. The parallelization is based on PVM and has yielded good results, e.g., almost ideal speed up on 48 processors of a Cray T3E parallel computer [22, 23]. This article describes the theoretical foundation underlying some of the parts of the parallel software, which are in the public domain.

The application TRIWAQ, too, is provided with a nice mathematical structure, through the usage of an internally regular grid. However, the irregular boundaries of the domain (coastlines) necessitate usage of an indirect addressing mechanism, i.e., putting all active grid points consecutively in a 1D array. This mismatch between mathematical and implementation structures becomes important in parallelization, because distribution of the implementation structure does not preserve data locality.

Essential abstractions in our approach are index sets and stencils (Section 2) for *precisely* and *compactly describing* for which grid points calculations are carried out and which data items are involved. It can be argued that this is valuable also in sequential computation, because it gives additional insights in the numerical method, helps preventing and detecting errors in the final source code, and can be a starting point for developing tool-support. The notation is much more indispensible in parallel computation. This is because it allows for the specification of inherently data parallel operations in a parallel way, and greatly simplifies the determination of which data must be exchanged between neighbouring subdomains.

The index sets and stencils in our formalism are in essence the same as *index domains* and *communication forms* in Crystal [7], and *index spaces* and *communication topologies* in PROMOTER [14, 20]. Although developed independently, this work can be viewed as a case study to the use of these formalisms for a complex finite difference application. We show that the formalism leads not only to concise documentation but also contributes to better understanding of the numerical method. In parallelization it enables handsome optimizations in the amount of communication. In the manipulations that are required we use knowledge of the problem domain, such that this work cannot be automated easily in a compiler-system such as Crystal or PROMOTER.

This clearly illustrates the different scopes of both Crystal and PROMOTER and that of the current work: whereas the former intend to provide a general purpose programming model/compiler system, our formalism is oriented towards a specific application domain and to parallelization of sequential Fortran code. Important aspects in our case are the re-use of program-code, algorithms and data structures, because of the huge investments made in the program and in pre- and post-processing facilities. A commonality is the requirement that increasing the level of abstraction should not be achieved at the cost of efficiency.

Other solutions that have been proposed for simplifying the programming of parallel applications are: automatic transformation of abstract specifications into executable program code [10, 11, 19], formalization of the software development process [4, 5, 17], and the use of skeletons [8, 9]. These approaches also work by increasing the level of abstraction of parallel programming, and are supplementary to our work. Firstly because our communication operations (Subsection 4.6) can be seen as a kind of skeletons, secondly, our notation allows for formalization of some part of the development process and finally, our description clearly shows starting-points for automatic code generation.

2 THE FORMALISM IN A GENERAL SETTING

The purpose of the concepts introduced below is to describe explicitly the structure and interactions of data items in a numerical simulation algorithm. For instance we want to capture the structure of an irregularly bounded grid such as is shown in Figure 1. This is done with an *index set*:

Definition 1 An n-dimensional index set is an arbitrary subset of $\mathbb{Z}^n$.

This concept provides a generalization of conventional arrays, for which the index set is a Cartesian product: the index set of an $m \times n$ array is $\{1, \ldots, m\} \times \{1, \ldots, n\}$. Calligraphic letters are used to denote index sets. For simplicity we mainly restrict ourselves to two-dimensional index sets.

The structure of the interactions between data items is non-trivial for instance in the matrix-vector multiplication $y := Ax$ for a sparse matrix $A \in \mathbb{R}^{m \times n}$. A question is which elements x_j of x are required for calculation of a given set of elements y_i for $i \in \mathcal{I}_s \subset \{1, \ldots, m\}$. The set of indices j for which x_j is required is called $\mathcal{J}_s$. It is determined by the non-zeros in A:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \tag{1a}$$

$$\rightarrow \mathcal{J}_s = \left\{ j \in \{1, \ldots, n\} \mid \exists i \in \mathcal{I}_s \colon a_{ij} \neq 0 \right\}. \tag{1b}$$

In this example the structure of the operation is determined by the non-zero pattern of A. This pattern is given

FIGURE 1 Typical grid in shallow water applications (Wadden sea, the Netherlands, 400 × 270, 45,000 active points).

by the pairs

$$\mathcal{A} = \big\{(i, j) \in \{1, \ldots, m\} \times \{1, \ldots, n\} \mid a_{ij} \neq 0\big\}. \quad (2)$$

Index set $\mathcal{A}$ is called the *interaction set* of the operation (the *communication product* of [20]). It describes the interactions between the elements of x and y (data dependences of the operation) in a compact way.

Definition 2 An interaction set is an arbitrary subset of the Cartesian product of two index sets.

In many large scale computing problems the non-zero pattern of a sparse matrix (at least at the conceptual level) can be characterized by a sparsity structure. In finite difference applications the structure arises through the application of *stencils* or computational molecules [1, 16]. An example is the well-known five-point stencil for the Laplace operator,

$$\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}, \quad (3)$$

which implies that the equation for an internal grid point takes the values of its four "neighbouring" grid points minus four times the value of the grid point itself. The stencil is determined by the discretization techniques that are used (e.g., central, 1st order upwind, 2nd order upwind). In our case a stencil is not used for describing the mutual influences of neighbouring grid points (through weights such as in Equation (3)), but only for characterizing the pattern of the interactions:

Definition 3 A two-dimensional stencil is an arbitrary collection of offsets $(\delta m, \delta n) \in \mathbb{Z}^2$.

A list of stencils that are used in this work is given in Table 1.

A stencil is always used in the meaning that a calculation for each central point (offset $(0, 0)$) requires values from all points referred to by the stencil. In some cases we want to describe an interaction with opposite direction (information of the central point is needed by all other points indicated by the stencil), therefore we introduce:

Definition 4 The negative of a (two-dimensional) stencil $\mathcal{S}$ is the stencil

$$-\mathcal{S} = \big\{(\delta m, \delta n) \in \mathbb{Z}^2 \mid (-\delta m, -\delta n) \in \mathcal{S}\big\}. \quad (4)$$

With these definitions we can construct the interaction set for many finite difference operations. However, we are mostly interested in determining data requirements such as in Equation (1b). In such a case one index set ($\mathcal{I}_s$) is used as the *range* of the computation (also called *iteration set* conform [6]); a computation is performed for all elements (y_i) in this set using neighbouring values specified by a stencil. Which values (x_j) are required is now easily determined by adding all offsets in the stencil to all elements in the range of the computation. This operation is called *dilatation* and is denoted by the symbol $\otimes$.

Table 1. Definition of a Number of Stencils

stencil	collection of offsets	stencil	offsets
$\mathcal{S}_1$	$(0,-1),(-1,0),(0,0),(1,0),(0,1)$		
$\mathcal{S}_{1x}$	$(-1,0),(0,0),(1,0)$	$\mathcal{S}_{1y}$	$(0,-1),(0,0),(0,1)$
$\mathcal{S}_{-1x}$	$(-1,0),(0,0)$	$\mathcal{S}_{-1y}$	$(0,-1),(0,0)$
$\mathcal{S}_{+1x}$	$(0,0),(1,0)$	$\mathcal{S}_{+1y}$	$(0,0),(0,1)$
$\mathcal{S}_{uv}$	$(-1,0),(0,0),(-1,1),(0,1)$	$\mathcal{S}_{vu}$	$(0,-1),(0,0),(1,-1),(1,0)$

Definition 5 The dilatation of an index set $\mathcal{I}_s \subset \mathcal{I}$ with stencil $\mathcal{S}$ is

$$\begin{aligned}\mathcal{J}_s &= \mathcal{I}_s \otimes \mathcal{S} \\ &= \big\{(\tilde{m},\tilde{n}) \in \mathcal{J} \mid \exists (m,n) \in \mathcal{I}_s \wedge \exists (\delta m, \delta n) \in \mathcal{S}: \\ &\qquad \tilde{m} = m + \delta m,\ \tilde{n} = n + \delta n\big\}. \end{aligned} \tag{5}$$

Although the index set $\mathcal{J}$ for which the input data is defined need not be the same as the index set of the result variable ($\mathcal{I}$), it is clear that they must have the same dimension. From Definition 5 follows further that the dilatation operation is *commutative* and *associative*:

$$\mathcal{I} \otimes \mathcal{S} = \mathcal{S} \otimes \mathcal{I}, \tag{6a}$$

$$(\mathcal{I} \otimes \mathcal{S}_1) \otimes \mathcal{S}_2 = \mathcal{I} \otimes (\mathcal{S}_1 \otimes \mathcal{S}_2). \tag{6b}$$

Also note the direction of the stencil: offsets in the stencil are *added* to each point of the iteration set. If instead we want to determine the set of elements y_i that are influenced by elements x_j in a subset $\mathcal{J}_s \subset \mathcal{J}$ of the input data, then we must calculate the dilatation of $\mathcal{J}_s$ with stencil $-\mathcal{S}$:

$$\begin{aligned}\mathcal{I}_s &= \mathcal{J}_s \otimes -\mathcal{S} \\ &= \big\{(m,n) \in \mathcal{I} \mid \exists (\tilde{m},\tilde{n}) \in \mathcal{J}_s \wedge \exists (\delta m, \delta n) \in \mathcal{S}: \\ &\qquad m + \delta m = \tilde{m},\ n + \delta n = \tilde{n}\big\}. \end{aligned} \tag{7}$$

Finally the interaction set for the entire operation is given by

$$\begin{aligned}&\big\{(m,n,\tilde{m},\tilde{n}) \in \mathcal{I} \times \mathcal{J} \mid \exists (\delta m, \delta n) \in \mathcal{S}: \\ &\qquad \tilde{m} = m + \delta m,\ \tilde{n} = n + \delta n\big\}. \end{aligned} \tag{8}$$

The concepts of index sets and interaction sets are applicable in many fields in scientific computing. For instance in finite element applications we can distinguish the sets of vertices, edges and elements $\mathcal{V}, \mathcal{E}, \mathcal{T}$. For an unstructured mesh the index set $\mathcal{V}$ will be one-dimensional, simply giving the node number. The set of edges $\mathcal{E}$ can be defined as a subset of $\mathcal{V} \times \mathcal{V}$, specifying a relation between two vertices. Similarly triangular elements can be defined as points in the set $\mathcal{V} \times \mathcal{V} \times \mathcal{V}$. Interactions can be defined in terms of these relations between nodes, edges and elements. For instance a computation for an element might require information from the nodes that constitute the element. In another computation information might also be required from "neighbouring" nodes, which can be stated precisely using index set $\mathcal{E}$.

Using index sets we can also give a nice characterization of data parallelism: a data parallel operation, also called super-step, consists of more or less elementary calculations performed independently for all elements of an iteration set. Examples are found in parallel constructs such as the `forall`-loop in parallel Fortran languages [6]. A data parallel program consists of a sequence of such data parallel operations. A single thread of control is visible to the programmer; parallelism is explicit in each data parallel operation. A program in this form can be parallelized by applying multiple workers to each step (agenda-parallel way of working, see [3]), and can be converted into an SPMD form with multiple processes. Where possible this should be left to a compiler; in the following sections however it will be done manually in a systematic manner.

At the conceptual level, the main issue in parallelization is that of determining a suitable distribution of all computations over a set of processors. In data parallel algorithms, the data dependence graph of the algorithm is presented in a highly structured form through the list of super-steps. A partitioning for the entire algorithm can then be constructed by giving a partitioning for each of the super-steps. Of course the objective is to determine the separate partitionings such that the overall performance is maximized, which requires optimal utilization of the processors as well as minimization of the amount of data movement. This implies among others that different steps using the same result variable (iteration set) should be partitioned in the same way if possible. Furthermore different index sets which are used together in super-steps should be partitioned such that as many interacting indices are assigned to the same processor as possible. These issues are also visible in High Performance

Fortran (HPF, [6, 15]), in the problems of *data distribution* and re-distribution and of *alignment* of different arrays. Index sets and interaction sets thus provide generalizations of the regular array structures in HPF.

3 APPLICATION TO SHALLOW WATER SIMULATION PROBLEMS

We have applied our formalism to the documentation and parallelization of the complex three-dimensional shallow water simulation program TRIWAQ [22, 23]. For matter of exposition we restrict ourselves in this section to a numerical model for simulation of the two-dimensional (depth-integrated) shallow water equations (SWE). The characteristics of the problem that are essential for the mathematical structure of the application are maintained, namely the use of irregularly bounded staggered grids and a time-dependent computational domain due to drying and flooding. However, a large number of practical (engineering) aspects are abstracted from, such as the use of curvi-linear grids, density differences due to temperature and salinity variations, and solution of non-linear equations etc. An introduction to SWE and their numerical treatment is given in [24]. An introduction to finite difference methods can be found in [1, 16].

We show how a finite difference method can accurately be described on a high level of abstraction by means of index sets and stencils. All the parallelism of the numerical method is still present here. Therefore this level is well suitable for analysis purposes and for starting the parallelization. In Section 4 index sets and stencils are used for describing the parallelization of the simulation method. It will be shown that communication requirements can be determined easily and manipulated using extra index sets for grid points near subdomain boundaries.

3.1 A numerical model for SWE

As a test case for illustrating the use of our documentation techniques we use the numerical method of Yu [21, 25]. Because this problem is two-dimensional and due to the largely linear and explicit treatment this method can be explained somewhat easier than other shallow water models. This method is especially attractive for large scale models such as the continental shelf, although it has also been used for the Belgium coast and for lab-scale models.

The partial differential equations that are considered here are:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + g\frac{\partial \zeta}{\partial x} - fv + \lambda u = F_x, \quad (9a)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + g\frac{\partial \zeta}{\partial y} + fu + \lambda v = F_y, \quad (9b)$$

$$\frac{\partial \zeta}{\partial t} + \frac{\partial Hu}{\partial x} + \frac{\partial Hv}{\partial y} = 0. \quad (9c)$$

In these equations, ζ denotes the water elevation with respect to a plane of reference and u and v are the flow velocities in x and y-directions. $H = \zeta + d$ is the total water depth, with d representing the bottom profile. F_x, F_y are the components of the wind-stress vector in x and y-directions, g is the acceleration of gravity, f is the Coriolis parameter and λ is the bottom friction coefficient. We have omitted among others eddy-viscosity effects and atmospheric pressure because they are not needed for showing the use of index sets and stencils.

Equations (9a)–(9c) are numerically integrated with respect to time by means of an Alternating Direction Implicit (ADI) method. This means that a time-step is split into two halves, first from time t to $t + \delta t/2$ and then from $t + \delta t/2$ to $t + \delta t$. In the first half step the spatial derivatives concerning water-elevation are taken explicitly in y-direction and implicitly in x-direction. This step is given in Equations (10a)–(10c), with a prime denoting values at the new time-level.

$$\frac{u' - u}{\delta t/2} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + g\frac{\partial \zeta'}{\partial x} - fv' + \lambda u = F'_x, \quad (10a)$$

$$\frac{v' - v}{\delta t/2} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + g\frac{\partial \zeta}{\partial y} + fu + \lambda v' = F'_y, \quad (10b)$$

$$\frac{\zeta' - \zeta}{\delta t/2} + \frac{\partial(\zeta + d)u'}{\partial x} + \frac{\partial(\zeta + d)v'}{\partial y} = 0. \quad (10c)$$

The second half step is largely the same, except that the role of x- and y-directions and u- and v-velocities is interchanged. In each half time step the calculations consist of first calculating values for the flow velocity in one direction at the new time-level, then solving a collection of tridiagonal systems for the water levels ζ, and finally computing the other flow velocity.

3.2 Finite difference grids and index sets

In shallow water simulations the use of irregularly bounded grids is inevitable, see Figure 1. An important aspect is furthermore that the domain is varying in time, as a result of drying and flooding due to tidal motion. In the Wadden sea shown in Figure 1 this effect is quite extreme: more than half of the grid points can be taken out of the computation due to drying.

Another form of irregularity arises through the use of *staggered grids*. This means that the different quantities in the PDEs (velocities, water-levels) are discretized on grids that are displaced with respect to each other by

one half mesh width. The use of staggered grids has several advantages over standard grids in the area of SWE: a decrease in storage requirement by a factor of four is achieved without noticeable loss of accuracy, the implementation of boundary conditions is simplified and spurious oscillations ("$2\delta x$ waves") are prevented, see, e.g., [24].

The finite difference grids are constructed by first approximating the physical domain by a collection of cells, with cell-faces parallel to the coordinate directions (Figure 2, left). The computational domain, i.e., the collection of cells, is represented by a two-dimensional index set:

$$\textit{Computational Domain} = \mathcal{D} = \big\{(m,n)\big\} \subset \mathbb{Z}^2. \quad (11)$$

Then the water level ζ is approximated in the centres of all cells and velocities are approximated on cell-faces. For instance u is approximated in the centres of cell-faces parallel to the y-axis. The nodes in which the unknowns are approximated thus form three different grids that are displaced with respect to each other: a staggered grid (Figure 2, right). The u-point with coordinates $((m+1/2)\delta x, n\delta y)$ is given the index (m,n), just as v-point $(m\delta x, (n+1/2)\delta y)$.

The index sets $\overline{\mathcal{G}}^*_u, \overline{\mathcal{G}}^*_v, \mathcal{G}_s$ for u, v- and ζ-grids are formally defined as:

$$\begin{aligned}
\overline{\mathcal{G}}^*_u &= \mathcal{D} \otimes \mathcal{S}_{+1x} \\
&= \big\{(m,n) \in \mathbb{Z}^2 \mid \\
&\qquad (m,n) \in \mathcal{D} \vee (m+1,n) \in \mathcal{D}\big\}, \qquad (12a)\\
\overline{\mathcal{G}}^*_v &= \mathcal{D} \otimes \mathcal{S}_{+1y} \\
&= \big\{(m,n) \in \mathbb{Z}^2 \mid \\
&\qquad (m,n) \in \mathcal{D} \vee (m,n+1) \in \mathcal{D}\big\}, \qquad (12b)\\
\mathcal{G}_s &= \mathcal{D}. \qquad (12c)
\end{aligned}$$

The adopted notation indicates that boundary points are included in the sets of velocity-points, but not in the set of water-level points. For the implementation of boundary conditions an extra ring of virtual ζ-points is created around the domain which is denoted by the index set $\partial\mathcal{G}_s = \partial\mathcal{G}^x_s \cup \partial\mathcal{G}^y_s$.

$$\begin{aligned}
\partial\mathcal{G}^x_s &= \big\{(m,n) \in \mathbb{Z}^2/\mathcal{D} \mid \\
&\qquad (m-1,n) \in \mathcal{D} \vee (m+1,n) \in \mathcal{D}\big\}, \quad (12d)\\
\partial\mathcal{G}^y_s &= \big\{(m,n) \in \mathbb{Z}^2/\mathcal{D} \mid \\
&\qquad (m,n-1) \in \mathcal{D} \vee (m,n+1) \in \mathcal{D}\big\}. \quad (12e)
\end{aligned}$$

The collection of all water-level points is denoted by $\overline{\mathcal{G}}_s = \mathcal{G}_s \cup \partial\mathcal{G}_s$. In the first phase of the ADI scheme only the boundary points in x-direction $\partial\mathcal{G}^x_s$ are used, in the second phase those in y-direction. Therefore we introduce index set $\overline{\mathcal{G}}^x_s = \mathcal{G}_s \cup \partial\mathcal{G}^x_s$ and likewise $\overline{\mathcal{G}}^y_s$. Note that a boundary point can be needed in x-direction as well as in y-direction, see for instance index $(m+1, n+1)$ in Figure 2 (right). Therefore $\partial\mathcal{G}^x_s$ and $\partial\mathcal{G}^y_s$ are overlapping if there are diagonal boundaries.

Only velocity points are subject to drying and flooding in the numerical method that is considered. A point is taken out of the computation as soon as the water-height drops below a threshold and is inserted again if the water-level rises above a (larger) threshold. The actual (time dependent) set of wet u-points at a given time, denoted by $\overline{\mathcal{G}}_u$, can thus be smaller than index set $\overline{\mathcal{G}}^*_u$ defined in Equation (12a).

Mapping the three grid points $(m\delta x, n\delta y)$, $((m+1/2)\delta x, n\delta y)$ and $(m\delta x, (n+1/2)\delta y)$ onto the same index (m,n) amounts to *aligning* the different grids. The three points together form one grid point of the staggered grid. The collection of all grid points in the staggered grid (the union of $\overline{\mathcal{G}}^*_u$, $\overline{\mathcal{G}}^*_v$ and $\overline{\mathcal{G}}_s$) is denoted by $\mathcal{H}$. It has a similar function as a *template* in HPF.

Calculations in the ADI scheme are sometimes based on the rows and columns of the computational domain, for instance in the definition of tridiagonal systems of equations. A row is a set of cells with identical n index and consecutive m indices. The index set of all rows is defined by

$$\begin{aligned}
\mathcal{R} = \big\{&(n, m_0, m_1) \in \mathbb{Z}^3 \mid \\
&m_0 \leqslant m_1 \wedge (m_0 - 1, n) \notin \mathcal{D} \wedge \\
&(m_1 + 1, n) \notin \mathcal{D} \wedge \forall m \in \{m_0, \ldots, m_1\}: \\
&(m,n) \in \mathcal{D}\big\}. \qquad (13)
\end{aligned}$$

The collection of columns $\mathcal{C}$ is defined similarly. There can be several rows with the same n coordinate if there are holes in the domain.

Definition (13) gives another representation of the computational domain because

$$\begin{aligned}
\mathcal{D} \equiv \big\{&(m,n) \in \mathbb{Z}^2 \mid \exists(\tilde{n}, m_0, m_1) \in \mathcal{R}: \\
&\tilde{n} = n \wedge m_0 \leqslant m \leqslant m_1\big\}. \qquad (14)
\end{aligned}$$

This relation forms the basis of the actual implementation in Fortran. An operation for a specific index set is realised by performing a double loop over all rows in $\mathcal{R}$ and all points (m,n) within each row. A calculation for wet points only is realised by selection with a mask array.

The implementation is not trivial however, but requires detailed knowledge of and is tailored to the application. For instance a calculation for all ζ-points in $\overline{\mathcal{G}}^x_s$ requires a loop over $m_0 - 1, \ldots, m_1 + 1$ instead of $m_0, \ldots, m_1$, because $\overline{\mathcal{G}}^x_s = \mathcal{D} \otimes \mathcal{S}_{1x}$:

$$\begin{aligned}
&\forall(n, m_0, m_1) \in \mathcal{R}: \\
&\quad \forall m \in \{m_0 - 1, m_0, \ldots, m_1 + 1\}: \\
&\qquad \text{calculate for grid point } (m,n). \qquad (15)
\end{aligned}$$

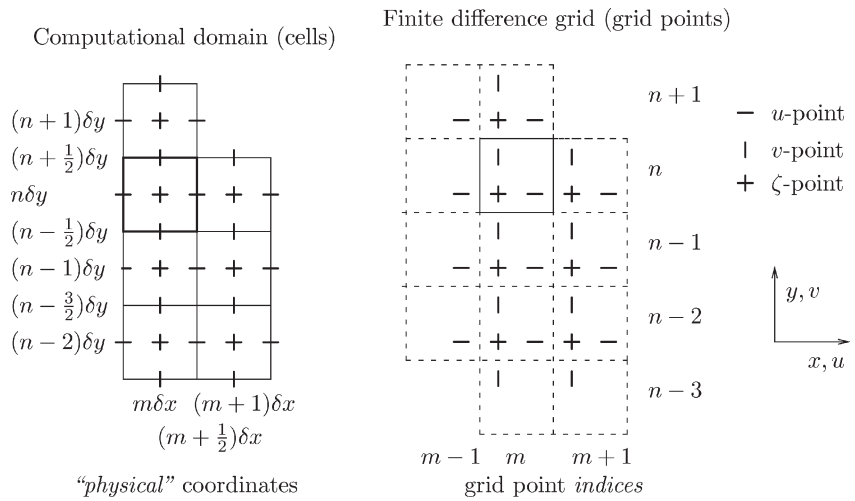

FIGURE 2 Computational domain versus staggered grid.

With respect to $\overline{\mathcal{G}}_s^x$, a row $r = (\tilde{n}, m_0, m_1)$ therefore consists of the grid points

$$gp(r) = \big\{(m,n) \in \overline{\mathcal{G}}_s^x \mid n = \tilde{n} \wedge m_0 - 1 \leqslant m \leqslant m_1 + 1\big\}. \quad (16)$$

This relation also describes the interaction set between $\overline{\mathcal{G}}_s^x$ and $\mathcal{R}$. With this definition two rows can have a grid point in common if the left boundary point of one row is the right boundary point of another ($n = n'$, $m_0 - 1 = m_1' + 1$). This may be problematic if the indices are also used as array-indices in the implementation. It can be prevented by imposing a restriction on the shape of $\mathcal{D}$.

The index sets that have been defined above characterize the mathematical structure of the data items in the application, and provide a natural starting point for choosing an implementation structure. For instance it is very convenient to have an array that contains the index set $\mathcal{R}$. Then a specification of a program in terms of index sets can be transformed automatically into program code by a simple preprocessor, see Figure 3.

This way of automatization allows for programming on a higher level of abstraction, leading to a more concise and readable program. It prevents *off-by-one* errors in loop bounds, because the programmer must only concentrate on which range is intended (`range_g_sx`), and not on how to realize this calculation. This way of working also enables choosing different realizations on different platforms. For instance on vector computers a useful alternative to the double loop implementation is to loop over all points in the rectangular hull around the domain, because this leads to longer vectors. Also it is often advantageous to have unit stride in computations, e.g., in the presence of cache memory, such that it might be better to access grid points column-wise instead of row-wise.

Abstract program code:

```
c$ix  for (m,n) in range_g_sx
      Fortran code for point (m,n)
c$ix  endfor
```

Fortran implementation with `do`-loops:

```
      do 20 ix_r = 1, norows
         n     = rowtbl(ix_r, 1)
         ix_m0 = rowtbl(ix_r, 2)
         ix_m1 = rowtbl(ix_r, 3)
         do 10 m = ix_m0-1, ix_m1+1
            Fortran code for
                    grid point (m,n)
 10      continue
 20   continue
```

FIGURE 3 Illustration of a simple mechanism for providing tool-support, for lifting the programming to a higher level of abstraction.

3.3 Spatial discretizations and stencils

All differentials in Equations (10a)–(10c) are replaced (approximated) by difference quotients. This is done in such a way that the values of unknowns are only required in the points where they are approximated. Due to the grid-staggering, this sometimes requires interpolation or averaging. For example in the discretization of (10b) in wet v-points in $\overline{\mathcal{G}}_v$, approximation of the Coriolis term fu requires u in v-points:

$$fu|_{v\text{-point}(m,n)} \approx f\frac{u_{m-1,n} + u_{m,n} + u_{m-1,n+1} + u_{m,n+1}}{4}. \quad (17)$$

The values that are required for evaluation of this formula are characterized by stencil $\mathcal{S}_{uv}$ (see Table 1). If Equation (17) is applied for iteration set $\overline{\mathcal{G}}_v$, then the set of grid points for which u-values are required is $\overline{\mathcal{G}}_v \otimes \mathcal{S}_{uv}$. But the value of u is undefined for indices in this set that do not belong to $\overline{\mathcal{G}}_u^*$. Therefore Equation (17) cannot be applied in v-points (m, n) for which

$$\exists(\delta m, \delta n) \in \mathcal{S}_{uv}\colon\ (m + \delta m, n + \delta n) \notin \overline{\mathcal{G}}_u^* \iff \{(m, n)\} \otimes \mathcal{S}_{uv} / \overline{\mathcal{G}}_u^* \neq \emptyset. \tag{18}$$

This occurs frequently if larger stencils (e.g., five points wide) are used. Therefore different approximations must be used near boundaries and in other special cases (e.g., for drying and flooding). Equation (18) illustrates the possibility for tool-support for automatic verification of the correctness of the discretizations that are used, for checking whether the stencils used in discretization need only data for grid points in the domain.

3.4 The solution algorithm

The steps in the solution algorithm have been described after the specification of temporal discretizations in Equations (10a)–(10c). In this subsection a more detailed description is presented for the first half of each time step.

The first step in the solution algorithm is to determine v' from the spatially discretized version of

$$\forall(m, n) \in \overline{\mathcal{G}}_v\colon\ v'_{m,n} = v_{m,n} + \frac{\delta t}{2(1+\lambda)}\left(F'_y - u\frac{\partial v}{\partial x} - v\frac{\partial v}{\partial y} - g\frac{\partial \zeta}{\partial y} - fu\right), \tag{19a}$$

$$\forall(m, n) \in \overline{\mathcal{G}}_v^* / \overline{\mathcal{G}}_v\colon\ v'_{m,n} = 0. \tag{19b}$$

Then u' and ζ' are calculated by solving Equations (10a) and (10c). For this, Equation (10a) is rewritten such that $u'_{m,n}$ is expressed in $\zeta'_{m+1,n} - \zeta'_{m,n}$, which is used to symbolically eliminate $u'_{m,n}$ from (10c). This leads to tridiagonal systems of equations for ζ' for all rows of the grid. For all ζ-points in $\overline{\mathcal{G}}_s^x$ the corresponding equation is written as

$$a_{m,n}\zeta'_{m-1,n} + b_{m,n}\zeta'_{m,n} + c_{m,n}\zeta'_{m+1,n} = d_{m,n}. \tag{20}$$

Of course at the start of a row where index $(m - 1, n) \notin \overline{\mathcal{G}}_s^x$ boundary conditions are inserted and the term $a_{m,n}\zeta'_{m-1,n}$ is dropped, and similarly at the end of a row. The second step in the solution algorithm is now to set up these equations:

$$\forall(m, n) \in \overline{\mathcal{G}}_s^x\colon \text{ calculate } a_{m,n},\ b_{m,n},\ c_{m,n} \text{ and } d_{m,n}. \tag{21}$$

These calculations are independent for all grid points. The formulas that must be evaluated can be found by performing the symbolic transformations described above.

The third step consists of solving the tridiagonal systems, for instance with Thomas' algorithm ("Gaussian elimination without pivoting", see for instance [16]).

$$\forall r \in \mathcal{R}\colon\ \forall(m, n) \in gp(r)\colon \text{ determine } \zeta'_{m,n} \text{ from (20).} \tag{22}$$

Here the first loop over all rows describes independent calculations, whereas the second part consists of the solution of one tridiagonal system of equations.

Finally u' can be calculated from

$$\forall(m, n) \in \overline{\mathcal{G}}_u\colon\ u'_{m,n} = u_{m,n} + \frac{\delta t}{2}\left(F'_x - u\frac{\partial u}{\partial x} - v\frac{\partial u}{\partial y} - g\frac{\partial \zeta'}{\partial x} + fv' - \lambda u\right), \tag{23a}$$

$$\forall(m, n) \in \overline{\mathcal{G}}_u^* / \overline{\mathcal{G}}_u\colon\ u'_{m,n} = 0. \tag{23b}$$

The description above shows how index sets are used for specifying the range of a computation, which causes information to be recorded that is otherwise not documented. For instance in Equation (23a) it is made explicit that wet u-points are treated differently than dry u-points (Equation (23b)). Also in Equation (22) it is explicitly specified where new ζ-values are calculated, namely in all points in $\overline{\mathcal{G}}_s^x$ (see Equation (14)), and not in y-boundary points $\partial\mathcal{G}_s^y / \partial\mathcal{G}_s^x$ (note that $\partial\mathcal{G}_s^x$ and $\partial\mathcal{G}_s^y$ need not be disjoint, see Section 3.2). This way of working therefore leads to less mistakes during software development. More information regarding data dependences can be recorded using stencils, as will be shown in the following section.

The solution algorithm in steps (19), (21), (22) and (23) can be further refined into a sequence of more elementary calculations for all grid points. This will give a detailed but also still abstract specification of the numerical method, concentrating only on the mathematical structure of the algorithm. This level is suitable for manipulations such as algebraic simplification (expanding or rewriting expressions) and common subexpression elimination (calculating averaged u-values only once from (17), use twice in (19a)), see [10]. In this way the computational complexity of the algorithm can be reduced. Similarly the data dependences between different supersteps may be optimized.

4 APPLICATION IN PARALLELIZATION

In the previous section we have shown our way of working in documentation of a finite difference method. In the

description of the solution algorithm much attention is paid to precisely specifying for which grid points calculations are carried out and which data items are involved. This helps preventing and detecting errors in the final source code and may lead to new insights in the numerical method. In this section we consider the application of the extra documentation for parallelization of the numerical method.

The parallelization is based on the agenda parallel approach described in Section 2: the super-steps of the solution algorithm are carried out in order, and the work in each step is carried out by multiple processes. The distribution of work in the steps is based on a partitioning of the staggered grid $\mathcal{H}$, and each worker process is responsible for the computations for one subgrid (subdomain). For the kind of grids that occur in shallow water applications the subdomains are necessarily irregularly shaped. This implies that determination of which data must be exchanged between neighbouring subdomains can be an extremely difficult task. However, it is greatly simplified by the use of index sets and stencils, by performing the parallelization first on the conceptual level and then working this down to the source code.

The notation that is used for describing the partitioning of the grid and the related index sets is introduced in Subsection 4.1 below. Then this notation is used for transforming the solution algorithm into a form with multiple computing processes. The amount of data exchange between these processes can be derived easily. This allows for comparison of the performance of alternative parallelization strategies for the super-steps, see Subsections 4.4 and 4.5. Finally a set of high-level collective communication subroutines is introduced with which the program can be converted easily into process/channel form, by inserting communication between the steps where necessary.

4.1 Partitioning of index sets

The distribution of work and data over the processors is based on a partitioning of the staggered grid $\mathcal{H}$. This means that each grid point is assigned to a processor. The subdomains (subgrids) are denoted by $\mathcal{H}_p$, with subdomain number $p \in \mathcal{P}$. The sets of u, v- and ζ-points are partitioned accordingly. This gives for example for the set of internal ζ-points:

$$\mathcal{G}_s = \bigcup_{p\in\mathcal{P}} \mathcal{G}_{s,p} \quad \text{with } \mathcal{G}_{s,p} \cap \mathcal{G}_{s,\tilde{p}} = \emptyset \text{ if } p \neq \tilde{p}. \quad (24)$$

Similarly for boundary points:

$$\partial\mathcal{G}_s = \bigcup_{p\in\mathcal{P}} \partial\mathcal{G}_{s,p} \quad \text{with } \partial\mathcal{G}_{s,p} \cap \partial\mathcal{G}_{s,\tilde{p}} = \emptyset \text{ if } p \neq \tilde{p}. \quad (25)$$

Thus the set of boundary points of subdomain p consists only of real boundary points and does not include points at interface boundaries between subdomains. Like before we define $\overline{\mathcal{G}}_{s,p} = \mathcal{G}_{s,p} \cup \partial\mathcal{G}_{s,p}$.

For denoting the range of a calculation or the domain for which a variable is required we must also describe the interface points of subdomains. The points just outside a subdomain p are called *external* interface points and are referred to by $\mathcal{J}_p$ and points inside subdomain p near the interface are called *internal* interface points $\mathcal{I}_p$. A precise specification of which points are "near" the interface is made with stencils:

Definition 6 The external interface $\mathcal{J}_{p,\mathcal{S}}$ of subdomain p with respect to stencil $\mathcal{S}$ is the collection of all grid points that do not belong to subdomain p and which can be obtained from a grid point in $\mathcal{H}_p$ plus an offset in $\mathcal{S}$:

$$\mathcal{J}_{p,\mathcal{S}} = (\mathcal{H}_p \otimes \mathcal{S})/\mathcal{H}_p. \quad (26)$$

Definition 7 The internal interface $\mathcal{I}_{p,\mathcal{S}}$ of subdomain p with respect to stencil $\mathcal{S}$ is the set of grid points in $\mathcal{H}_p$ for which one of the points referred to by $\mathcal{S}$ is outside $\mathcal{H}_p$:

$$\mathcal{I}_{p,\mathcal{S}} = (\mathcal{H}/\mathcal{H}_p \otimes -\mathcal{S}) \cap \mathcal{H}_p. \quad (27)$$

These definitions can be generalized easily for excluding certain interface points, for instance if a calculation is performed for interior ζ-points or wet u-points only.

Note that three different index sets are partitioned at once by partitioning of $\mathcal{H}$. This idea is carried further in the parallelization of the full application TRIWAQ. There, we have identified about 15 different index sets such as for source points or for open boundary points. All these index sets can be related to the grid through grid point numbers, which implicitly defines interaction sets. Quantities for these additional index sets are stored in different arrays than the fields u, v and ζ. Therefore direct partitioning of all separate arrays, on the level of the implementation structures, would not preserve data locality. However, with our approach of partitioning all index sets through partitioning of $\mathcal{H}$, all related data are assigned to the same subdomain.

Optimization of the partitioning requires that the performance for a given partitioning can be estimated beforehand. This is done by defining a workload for each grid point in $\mathcal{H}$, and by estimating the amount of communication overhead. A good approximation of the workload per domain is given by the number of active grid points, i.e., by the cardinality $\#\mathcal{H}$. Estimates for the communication overhead are derived from the stencils that are used, for instance by analyzing the cardinalities $\#\mathcal{J}_{p,\mathcal{S}}$ and $\#\mathcal{I}_{p,\mathcal{S}}$ for a few elementary domains (e.g., rectangles). In this work we do not consider load balancing any further, i.e., the partitioning method that is used is completely left open. For an overview of partitioning and partitioning methods see for instance [18].

4.2 Distribution with an owner computes rule

A partitioning of $\mathcal{H}$ into disjoint subsets $\mathcal{H}_p$ is nothing more than that, and does not command anything about the mapping of operations onto the processors. A common approach for making a distribution of work out of a partitioning is by means of an owner computes rule. This rule is here formulated as "a calculation is assigned to the processor that owns the index of the result variable".

As an example we consider the following super-step in the evaluation of the terms between brackets in Equation (19a) in auxiliary variable $e_{m,n}$:

$$\forall (m,n) \in \overline{\mathcal{G}}_v\text{: calculate}$$
$$e_{m,n} := e_{m,n} - \left(g \frac{\zeta_{m,n+1} - \zeta_{m,n}}{\delta y} \right). \qquad (28)$$

This super-step can be parallelized by noting that $e_{m,n}$ belongs to grid point (m,n) and assigning the update of $e_{m,n}$ to the owner of this grid point. This results in the following super-step:

$$\forall p \in \mathcal{P}\text{: } \forall (m,n) \in \overline{\mathcal{G}}_{v,p}\text{: calculate}$$
$$e_{m,n} := e_{m,n} - \left(g \frac{\zeta_{m,n+1} - \zeta_{m,n}}{\delta y} \right). \qquad (29)$$

This super-step is called parallel because it is explicitly subdivided into groups of calculations for different processes. The required data items for each group are determined by the dilatation of the subdomain with stencil $\mathcal{S}_{+1y}$, the stencil of the calculation:

$$\forall p \in \mathcal{P}\text{: } \textit{input}\text{: } \zeta_{m,n} \text{ for } (m,n) \in \overline{\mathcal{G}}_{v,p} \otimes \mathcal{S}_{+1y}. \qquad (30)$$

The amount of data exchange between different processes can be determined by counting the number of non-local grid points in (30). An estimate is obtained by noting that $\overline{\mathcal{G}}_{v,p} \otimes \mathcal{S}_{+1y} \subset \overline{\mathcal{G}}^{y}_{s,p} \cup \mathcal{J}_{p,+1y}$, using (26), with the difference residing in dry v-points and boundary effects. So the number of values that must be obtained by processor p from other processors is roughly equal to $\#\mathcal{J}_{p,+1y}$.

Subgrids can be represented by a collection of rows in the same way as the entire grid (Equation (14)). Calculations for subdomain index sets can therefore also be implemented with a double loop over rows and grid points per row, and the same ideas for automatic code generation are valid. In determining which points belong to a row (loop bounds, see Equation (16)) we must now take into account whether a boundary point is an interface point or a true boundary point. Parallelization actually consists of adding a check on whether the calculation must be performed for the start and end-points of each row. This implies that the greater part of the sequential code can be re-used in the parallelization.

4.3 Parallel solution of tridiagonal systems

The solution of tridiagonal systems by means of Thomas' algorithm is purely sequential. Still there is parallelism in step (22), namely through the occurrence of multiple tridiagonal systems, one for each row in $\mathcal{R}$. Parallelization is therefore possible by means of partitioning of the set $\mathcal{R}$ over the processors, into disjoint subsets $\mathcal{R}_p$.

The processor that must solve the tridiagonal system for a row $r \in \mathcal{R}$ must have all the coefficients $a_{m,n}, \ldots, d_{m,n}$ for this row. The precise set of grid points for which data are required is determined with the set $gp(r)$, which contains all grid points that are associated with row r:

$$\forall p \in \mathcal{P}\text{: } \textit{input}\text{: } a_{m,n}, \ldots, d_{m,n} \text{ for } (m,n) \in \bigcup_{r \in \mathcal{R}_p} gp(r). \qquad (31)$$

From the viewpoint of data locality it is now advantageous to choose the partitionings of $\mathcal{H}$ and $\mathcal{R}$ such that the overlap between $\mathcal{H}_p$ and $\bigcup_{r \in \mathcal{R}_p} gp(r)$ is maximized. However, in the second phase of the integration scheme tridiagonal systems have to be solved for all columns in $\mathcal{C}$. This imposes a requirement concerning overlap of $\mathcal{H}_p$ and $\mathcal{C}_p$ that is conflicting with the former requirement for $\mathcal{R}_p$; a row-wise distribution of $\mathcal{H}$ is required in the first phase and a column-wise distribution in the second. Therefore a complete data transposition step (similar to transposing a distributed matrix) is needed between the two solution phases.

The amount of communication in the transposition step is proportional to the number of grid points in $\mathcal{H}$. Therefore other parallelization approaches might be better than this "Thomas algorithm with data transposition" (TADT, partitioning of $\mathcal{R}$ and $\mathcal{C}$) approach. However, for the numerical model discussed in this work, this TADT approach outperforms a number of direct and iterative alternatives on an Intel iPSC/860 parallel computer for moderate (16–32) numbers of processors [21]. For other numerical methods the number of data transpositions can be larger, in which case other solvers are more appropriate [22].

The TADT approach in this subsection illustrates the application of general interaction sets (between $\mathcal{H}$ and $\mathcal{R}$, through $gp(r)$) for determining the data items that are needed in an operation. The specification of data requirements is done by adding an *input*-clause as annotation to each step. This has been applied also for non-data parallel operations such as the solution of a tridiagonal system. Further we have given an example of a solution algorithm that consists of super-steps for different index sets, for which parallelization requires the determination of an optimal alignment of the separate partitionings.

4.4 Distribution with redundant calculations

Use of the owner computes rule implies that a processor always calculates values for his own grid points only. This rule is easy to use, but may not be the most efficient in all cases. An alternative strategy is to allow *redundant calculations*. This means that a processor performs a calculation not only for its own grid points, but also for some external interface points. For indices close to subdomain boundaries the same value is then calculated independently by different processes. This may require more communication for one calculation, but may save communication in another place.

For instance it is possible to let all processors calculate v' for $(\mathcal{H}_p \cup \mathcal{J}_{p,-1y}) \cap \overline{\mathcal{G}}_v$ (all wet v-points in subdomain p and its external interface with respect to $\mathcal{S}_{-1y}$) instead of for their own grid points $\overline{\mathcal{G}}_{v,p}$ only (as in Equation (29)). The advantage is that v' does not have to be communicated for discretization of the term $\partial(\zeta + d)v'/\partial y$ in Equation (10c), (in coefficient $d_{m,n}$ in Equation (21)). Unfortunately v' is still required for stencil $\mathcal{S}_{vu}$ for evaluation of (23a) (see Equation (17)), such that this scheme does not save communication in this example. We have encountered other cases in which this technique does reduce the total amount of communication.

Redundant calculation of v' implies that step (28) is parallelized with

$$\forall p \in \mathcal{P}\colon \forall (m,n) \in (\mathcal{H}_p \cup \mathcal{J}_{p,-1y}) \cap \overline{\mathcal{G}}_v\colon \\ e_{m,n} := e_{m,n} - \left(g \frac{\zeta_{m,n+1} - \zeta_{m,n}}{\delta y} \right). \tag{32}$$

Therefore the ζ-values that are required are given by:

$$\forall p \in \mathcal{P}\colon \textit{input}\colon \zeta_{m,n} \text{ for} \\ (m,n) \in \left((\mathcal{H}_p \cup \mathcal{J}_{p,-1y}) \cap \overline{\mathcal{G}}_v \right) \otimes \mathcal{S}_{+1y}. \tag{33}$$

By manipulation with index sets, using that $\mathcal{S}_{1y} = \mathcal{S}_{-1y} \otimes \mathcal{S}_{+1y}$, it can be shown that the number of non-local ζ-values needed by processor p is now roughly $\#\mathcal{J}_{p,1y}$. This must be compared with $\#\mathcal{J}_{p,+1y}$ for the original approach. Redundant calculations always need more data than with an owner computes rule because calculations are done for a larger iteration set. However, depending on the situation these data might already be available from previous calculations or communication.

The discussion above shows that the analysis of redundant calculation schemes can be done using a high-level description of the numerical method in terms of index sets and stencils. The manipulation with index sets and derivation of the amount of communication can even be automated. This analysis would have been virtually impossible otherwise, on the level of the implementation structure, without (implicitly) using the notation.

4.5 Distribution with an input-owner computes rule

In determining data requirements for the matrix-vector multiplication in (1b) we have actually assumed that an owner computes rule is used. Another parallelization strategy is to define the distribution of work not on basis of the elements y_i of y but instead on the elements x_j of x. This can be viewed as a column-wise distribution of the matrix A, where the original scheme implies a row-wise distribution of A.

Similarly, an alternative to (29) can be based on the indices of ζ-values in (28) instead of the indices of e. This leads to the parallel calculations (for all $p \in \mathcal{P}$)

$$\forall (m,n) \in \overline{\mathcal{G}}^y_{s,p}\colon \text{ if } (m,n-1) \in \overline{\mathcal{G}}_v\colon \\ \text{calculate } e_{m,n-1} := e_{m,n-1} - g\zeta_{m,n}/\delta y, \tag{34a}$$

$$\forall (m,n) \in \overline{\mathcal{G}}^y_{s,p}\colon \text{ if } (m,n) \in \overline{\mathcal{G}}_v\colon \\ \text{calculate } e_{m,n} := e_{m,n} + g\zeta_{m,n}/\delta y. \tag{34b}$$

Here we violate the owner computes rule of Subsection 4.2 because there are indices $(m,n) \in \overline{\mathcal{G}}^y_{s,p}$ such that processor p is not the owner of result variable $e_{m,n-1}$, corresponding to v-point $(m,n-1)$. The distribution of work is no longer based on the result variable, but instead on one of the input variables. Therefore this way of work-distribution is called an "input-owner computes rule". The correctness of this realization can be shown by manipulation of the index sets that are involved.

A complication in implementation of (34a) and (34b) is that specific values $e_{m,n}$ are modified by more than one processor. In a parallel programming paradigm in which all data is shared by all computing processes this requires some form of synchronization for enforcing mutual exclusion. In a process/channel paradigm, where processes only have local memory, processes cannot access all variables that they contribute to. Then it is useful to introduce temporary variables $e^{(p)}_{m,n}$ on all processors p that contribute to the result value for index (m,n). The index set for which temporary variables must be introduced is given by

$$\forall p \in \mathcal{P}\colon \textit{output}\colon e^{(p)}_{m,n} \text{ for} \\ (m,n) \in \left(\overline{\mathcal{G}}^y_{s,p} \otimes \mathcal{S}_{-1y} \right) \cap \overline{\mathcal{G}}_v. \tag{35}$$

Communication is then used for summing up the different contributions and recovering the final value, see Subsection 4.6 below. On the other hand, no communication is required for ζ because it is required for grid points in $\overline{\mathcal{G}}^y_{s,p}$ only. Therefore the situation is reversed in comparison to the original owner computes rule, where communication is needed for the input variable and not for the output variable.

4.6 Abstract communication operations

In the previous subsections we have shown how to specify distributions of work at the mathematical level in terms of subdomain and interface index sets, thereby disregarding whether a processor can access the required data or not. A distribution of work should be specified for each super-step separately. Then the data that are needed and produced by an operation can be derived immediately from the iteration set and stencil that are used. This information has been indicated by annotating super-steps with *input* and *output*-clauses. We have shown how the data requirements can be influenced by choosing different distributions of work. After all super-steps have been parallelized in this way, communication must be inserted between them for ensuring that all required data are indeed available.

For this we introduce two communication steps by means of their input- and output-clauses. The first one is `update`, with as function to make information of each subdomain available to neighbouring subdomains. On input, a grid function $d_{m,n}$ is stored in a distributed way in the local memories of all computing processes. Auxiliary variables are introduced for storing local copies of $d_{m,n}$ for indices "just outside" each subdomain (a so-called *guard band*). The update procedure is used for refreshing these auxiliary variables.

Procedural abstraction P1. Update of local copies $tmp^{(p)}_{m,n}$ of global values $d_{m,n}$ for stencil $\mathcal{S}$,

$$\forall p \in \mathcal{P}\text{: } \textit{input}\text{: } d_{m,n} \text{ for } (m,n) \in \mathcal{H}_p, \tag{36a}$$

$$\forall p \in \mathcal{P}\text{: } \textit{output}\text{: } tmp^{(p)}_{m,n} = d_{m,n} \text{ for } (m,n) \in \mathcal{H}_p \cup \mathcal{J}_{p,\mathcal{S}}. \tag{36b}$$

In an implementation, a single array can be used for storing both $d_{m,n}$ in subdomain p and $tmp^{(p)}_{m,n}$. This array then provides different views on the data alternately, namely as part of a global grid function or as individual local grid functions. In the definition of P1 above these views have been separated.

The second communication step is `accumulate`, which is used for summing up contributions from different processors on the processor that owns an index. This operation is in many respects the dual of the `update`-operation. It is not required if the owner computes rule is applied consistently.

Procedural abstraction P2. Accumulation of local contributions $tmp^{(p)}_{m,n}$ to values $d_{m,n}$ with respect to stencil $\mathcal{S}$,

$$\forall p \in \mathcal{P}\text{: } \textit{input}\text{: } tmp^{(p)}_{m,n} \text{ for } (m,n) \in \mathcal{H}_p \cup \mathcal{J}_{p,-\mathcal{S}}, \tag{37a}$$

$$\forall p \in \mathcal{P}\text{: } \textit{output}\text{: } d_{m,n} = \sum_{\tilde{p} \in pr(m,n)} tmp^{(\tilde{p})}_{m,n} \text{ for } (m,n) \in \mathcal{H}_p, \quad \text{with}$$

$$pr(m,n) = \left\{ \tilde{p} \in \mathcal{P} \mid (m,n) \in \mathcal{H}_{\tilde{p}} \cup \mathcal{J}_{\tilde{p},-\mathcal{S}} \right\}. \tag{37b}$$

The stencil $-\mathcal{S}$ is used in the definition of the accumulate operation because we adhere to the conventional meaning of a stencil in which central point is connected with the result variable. With this definition, values $e^{(p)}_{m,n}$ provided in (35) must then accumulated for stencil $\mathcal{S}_{+1y}$ instead of $\mathcal{S}_{-1y}$, which agrees with the stencil that is used in calculation (28).

By inserting communication a program in a process/shared data paradigm is transformed into a new program in which communication is taken into account. The task of determining where these communication operations are needed and for which variables and stencils is straightforward and is largely similar to data dependence analysis. It is amenable to automization especially if all super-steps in the solution algorithm are annotated with the required input and output data. Data dependence analysis might also be used for ordering the super-steps in an optimal way. The amount of communication overhead is reduced if different communications are taken together and if communication is overlapped with useful calculations. For example in step (23a) of the solution algorithm the communication for ζ' can be overlapped with the evaluation of the other terms in this equation.

Communication operations P1 and P2 are abstract in the sense that implementation aspects such as interconnectivity of subdomains, processor topology and message passing protocol are hidden. The interface is formulated in terms of entities that are meaningful to application programmers, which further simplifies their usage (see Figure 4). Therefore these operations can also be used by non-parallelization experts. At the same time they are very powerful, because they allow for arbitrarily shaped subgrids and allow for precise specification of what should be communicated through the index set and stencil used.

Portability is obtained by using a standard interface that can be implemented on all (distributed and shared memory) parallel computers, and by using the *de facto* message passing standard PVM [2, 13] in the implementation. Efficiency is achieved because of the inherent possibility of *message-vectorization* for minimization of latency effects, i.e., packing as many data items in one message as possible. Furthermore the detailed knowledge of the solution algorithm and required communication operations allows for *direct determination* of the sets of values (indices) to be sent to and received from other processes.

```
c partno=processor number,
c h_owner=specifying the partitioning,
c make_table: create the communication
c             table for a stencil
     call make_table(partno, proc_list,
    +          h_owner, stencil, table)
     call update(partno, in_field,
    +                 table, out_field)
```

FIGURE 4 Example calls to the communication subroutines.

In this way the *inspector*-phase is avoided that is often used in the compilation of HPF programs. The inspector-phase works by executing a loop without calculating anything, for determining what must be communicated. Then the communication takes place and finally the loop is executed with all calculations in an *executor*-phase [6].

The interface index sets that are required can be implemented by formulas or by tables. The latter is especially advantageous for irregular partitionings. The number of tables required is reduced by performing slightly more communication than strictly required, providing data for an interface index set slightly larger than specified in the input-clauses. For instance data can be sent for all interface points instead of wet interface points only. This yields a considerable gain because it requires the inspector-phase to be executed only once instead of each time that points are drying and flooding. Also instead of defining interface index sets for $\overline{\mathcal{G}}^*_{u,p}$, $\overline{\mathcal{G}}^*_{v,p}$ and $\overline{\mathcal{G}}_{s,p}$ we use $\mathcal{H}_p$ in the definition of $\mathcal{I}_{p,\mathcal{S}}$ and $\mathcal{J}_{p,\mathcal{S}}$, leading to slightly more communication near boundaries of the computational domain. To assess the amount of extra communication that is introduced in this way requires knowledge about the problem and therefore these modifications cannot be automated easily.

5 SUMMARY AND DISCUSSION

In this work we have introduced the concepts of index sets and stencils and shown their application for documenting and parallelizing a complex numerical model. Their essential contribution is to specify precisely the mathematical structure of the model, i.e., the domain of variables, the range of computations and the set of data items required in each operation. This extra information can be used throughout for structuring the development of numerical simulation software. Possibilities for tool-support have been indicated, such as devising a simple preprocessor for replacing calculations for index sets by do-loops. This allows for choosing different implementations on different types of computers and therefore enhances portability.

The notation is particularly useful in parallelization, in determining where and what to be communicated between processes. Namely, the index set for which data is required for an operation is the set of all points that are reached by applying the stencil of the calculation in all points in the range of the calculation. Parallelization by means of an owner computes rule has almost become a mechanical process of determining (and optimizing) data dependences and inserting communication, and is highly suitable for automatization. Other parallelization strategies have become manageable, whereas determination of the required communication would be virtually impossible otherwise. A few powerful subroutines can be devised for this communication that take care of all necessary message passing. Portability and efficiency are achieved by using a standard interface and providing efficient implementations on all platforms.

We have thus shown how message passing software can be developed in a systematic way for more unstructured or irregular problems. The success of our techniques is based on bringing the development to a more abstract level. At this level, the structure of a solution algorithm is made explicit through the interactions between different data items. This allows for improving the degree of *data locality* in a parallel implementation, by proper alignment of the partitionings of different index sets. In the comparison of alternatives application programmers can concentrate on relevant issues of the solution algorithm only. These issues do include *what* must be communicated, but not the precise details of sending messages, i.e., *how* it is implemented.

Concepts that are similar to our index sets and interaction sets are employed in Crystal [7] and PROMOTER [14, 20]. A difference is that our work is oriented towards a specific application (FDM on irregularly bounded staggered grids). It has been shown that complex algebraic manipulations with index sets $\mathcal{D}$, $\mathcal{G}$, $\mathcal{H}$ and $\mathcal{R}$ are possible and that a considerable performance gain can be achieved using knowledge of the application. For instance by communicating for all u-points ($\overline{\mathcal{G}}^*_u$) instead of wet u-points only ($\overline{\mathcal{G}}_u$) repeated analysis of the required communication can be avoided. It is therefore advisable to apply the concepts not only as entities in a programming model or compiler system, but to use them throughout the entire software development process.

6 OBTAINING THE COMMUNICATION SOFTWARE

The communication subroutines that implement the procedural abstractions of Subsection 4.6 are kindly made available to the public by the Dutch Rijkswaterstaat and

VORtech. They can be obtained via the world wide web at URL

`http://ta.twi.tudelft.nl/PA/VORtech/`

REFERENCES

[1] W. F. Ames, *Numerical Methods for Partial Differential Equations*, 3rd ed. New York: Academic Press, 1992.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam, "Recent enhancements to PVM," *International Journal of Supercomputer Applications and High Performance Computing*, Vol. 9, no. 2, pp. 108–127, 1995.

[3] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, Vol. 21, pp. 323–357, 1989.

[4] H. H. ten Cate and E. A. H. Vollebregt, "On the portability and efficiency of parallel algorithms and software," *Parallel Computing*, Vol. 22, no. 8, pp. 1149–1163, October 1996.

[5] K. M. Chandy and J. Misra, *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[6] B. Chapman, P. Mehrotra, and H. Zima, "High Performance Fortran languages: Advanced applications and their implementation," *Future Generation Computer Systems*, Vol. 11, pp. 401–407, 1995.

[7] M. Chen, Y. Choo, and J. Li, "Crystal: Theory and pragmatics of generating efficient parallel code," in B. K. Szymanski, Ed., *Parallel Functional Languages and Compilers*, Frontier Series. ACM Press, 1991, Ch. 7, pp. 255–308.

[8] M. I. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Cambridge, MA: MIT Press, 1989.

[9] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While, "Parallel programming using skeleton functions," in A. Bode, M. Reeve, and G. Wolf, Eds., *PARLE 93: Parallel Architectures and Languages Europe*. Berlin: Springer-Verlag, 1993.

[10] R. van Engelen, L. Wolters, and G. Cats, CTADEL: A generator of efficient code for PDE-based scientific applications," Technical Report 95-26, Department of Computer Science, Leiden University, 1995.

[11] S. Fitzpatrick, M. Clint, P. Kilpatrick, and T. J. Harmer, "The tailoring of abstract functional specifications of numerical algorithms for sparse data structures through automated program derivation and transformation," *The Computer Journal*, Vol. 39, no 2, pp. 145–168, 1996.

[12] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[13] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 user's guide and reference manual," Technical Report TM-12187, Oak Ridge National Laboratory, May 1994.

[14] W. K. Giloi and A. Schramm, "PROMOTER – an application-oriented programming model for massive parallelism," in W. K. Giloi, S. Jähnichen, and B. Shriver, Eds., *Massively Parallel Programming Models, Proc. Int. MPPM Conference*. IEEE-CS Press, 1993.

[15] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, Vol. 2, no. 1, pp. 1–170, 1993.

[16] C. Hirsch, *Numerical Computation of Internal and External Flows, Vol. 1: Fundamentals of Numerical Discretization*. Chichester: Wiley, 1988.

[17] P. Pepper, J. Exner, and M. Südholt, "Functional development of massively parallel programs," in D. Bjørner, M. Broy, and I. V. Pottosin, Eds., *Formal Methods in Programming and Their Applications. Proc. Int. Conf. Novosibirsk, 1993*, Lecture Notes in Computer Science, Vol. 735. Berlin: Springer-Verlag, 1993, pp. 217–238.

[18] M. R. T. Roest, *Partitioning for Parallel Finite Difference Computations in Coastal Water Simulation*. PhD thesis, Delft University of Technology, 1997.

[19] Th. Ruppelt and G. Wirtz, "Automatic transformation of high-level object-oriented specifications into parallel programs," *Parallel Computing*, Vol. 10, pp. 15–28, 1989.

[20] A. Schramm, "Irregular applications in PROMOTER," in W. K. Giloi, S. Jähnichen, and B. Shriver, Eds., *Massively Parallel Programming Models, Proc. Int. MPPM Conference*. IEEE-CS Press, 1995.

[21] Z. W. Song, *Parallelization of Hydrodynamic Models for Distributed Memory Computers*. PhD thesis, K.U. Leuven, 1995.

[22] E. A. H. Vollebregt, *Parallel Software Development Techniques for Shallow Water Models*. PhD thesis, Delft University of Technology, 1997.

[23] E. A. H. Vollebregt, M. R. T. Roest, H. H. ten Cate, and H. X. Lin, "The PARALLEL project: Parallel simulation of 3D flow and transport models," in L. Dekker, W. Smit, and J. C. Zuidervaart, Eds., *Int. EUROSIM Conference HPCN Challenges in Telecomp and Telecom*. Elsevier, 1996, pp. 479–486.

[24] C. B. Vreugdenhil, *Numerical Methods for Shallow-Water Flow*, Water Science and Technology Library, Vol. 13. Dordrecht: Kluwer Academic Publishers, 1994.

[25] C. S. Yu, *Modelling Shelf Sea Dynamics*. PhD thesis, K.U. Leuven, Belgium, 1993.