*A practical functional program for the CRAY X-MP**

JAMES M. BOYLE
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, USA

TERENCE J. HARMER
The Queen's University of Belfast, Department of Computer Science, Belfast BT7 1NN, Northern Ireland

Abstract

One can have all the advantages of functional programming – correctness, clarity, simplicity, and flexibility – without *any* sacrifice in performance, even for a scientifically significant computation on a supercomputer. Therefore, why use Fortran? We demonstrate *parity* – equality of speed and storage use – between a program generated automatically from a functional specification and a program written by hand in the procedural style. To our knowledge, this demonstration of parity is the first for a program that solves a scientifically significant problem – quasi-linear hyperbolic partial differential equations – on a scientifically interesting supercomputer – the CRAY X-MP. We use pure Lisp, including higher-order functions, to express the functional specification for the PDE solver. We designed this specification for maximal clarity and flexibility, rather than for efficiency. Nevertheless, we obtain a highly efficient program to solve the PDEs: automated program transformations put back the missing efficiency as they produce an executable Fortran program from the specification. The generated Fortran program vectorizes on the CRAY X-MP and runs about 4% faster than a handwritten Fortran program for the same problem. We describe the problem and the specification, and some of the problem-domain-specific and hardware-specific transformations that we use to obtain the high-efficiency program.

1 Introduction

A recent report (FCCSET, 1987) of the US Federal Coordinating Council for Science, Engineering and Technology (FCCSET) outlines a national strategy for high-performance computing and recommends research 'to improve basic tools, languages, algorithms, and associated theory for the scientific "grand challenges"'. These grand challenges are fundamental problems of science and engineering involving computations that are at or just beyond the current limits of the possible. Such problems range over all the sciences and include problems in computational fluid dynamics, electronic structure of materials, plasma dynamics, and quantum

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, US Department of Energy, under Contract W-31-109-Eng-38.

theory, as well as symbolic computations in natural language recognition, computer vision, automated reasoning, and computer-aided design, manufacturing, and simulation.

It is unlikely that existing computer software can be used to perform the grand challenge computations. Not only are the problems themselves new (at least on the scale proposed in the report), but new means are also required to solve them, including the use of supercomputers having new advanced-vector or massively parallel architectures. Thus, solving grand challenge problems will almost certainly entail the development of both new algorithms and new computer programs to implement these algorithms.

What is the likelihood that functional programming techniques will be used to develop the programs required to solve these grand challenge problems? Functional programming offers significant advantages over programming in conventional, procedural languages: clarity, conciseness, modularity, ease of proof, ease of introducing problem-oriented notation, and independence from hardware peculiarities, to name a few (see Boyle, 1980; Bird and Wadler, 1988; Field and Harrison, 1988; Kelly, 1989; Burton and Kollias, 1989). These advantages are especially important for the solution of grand challenge problems, which will require programs that are significantly more complex than existing ones – programs that involve not only advanced numerical methods but also symbolic computation and complicated data- and control-structures. Unfortunately, however, functional programming seems unlikely to be used in the solution of grand challenge problems, because it is perceived to obtain its advantages at the price of (perhaps greatly) decreased execution speed and (perhaps greatly) increased storage use when compared to conventional procedural programs (see, for example, Bloss *et al.*, 1989; Kelly, 1989).

One can imagine that the scientists and applications programmers involved will argue along the following lines: 'We believe that solving our problem will require months of computation on the most advanced available hardware using the most efficient available computational methods. We can't afford to lose a factor of 10, or even of 2, in computation speed because we employed functional programming. We'll use Fortran, thank you!' (Of course, this argument completely ignores both the possibility that the programs developed in Fortran may contain undetected bugs that could render their months of computation worthless – bugs that might be avoided if functional programming were to be used – and the possibility that the total time to solution of the problem might be reduced if functional programming were to be used.)

We believe that such arguments can be defused by (several) successful demonstrations of parity *between functional and procedural programs.* A demonstration of parity is an experiment that shows that using a functional program for a significant scientific application can result in code that executes (at least) as fast and uses (at least) as little storage on a high-performance computer as does a typical handwritten procedural program for the same application on that hardware. Our goal of making functional programs execute efficiently on widely available hardware architectures is shared by at least two other functional programming research groups, the one at Yale (Bloss *et al.*, 1988, 1989; Kelsey and Hudak, 1989) and the one at Lawrence Livermore National Laboratory and Colorado State University (Feo *et al.*, 1990).

This, then, is what our paper is about: a demonstration of parity between an automatically derived implementation of a functional program and a handwritten Fortran program for solving a practical fluid-dynamics problem on the CRAY X-MP computer. We have achieved parity for this problem by using automatic program transformations to derive an efficient, vectorizable Fortran program from a (higher-order) functional program, which serves as a specification. To achieve this result, we use mainly basic, general-purpose transformations that implement functional specifications in Fortran, with a few transformations that perform either problem-domain-oriented or hardware-oriented optimizations interspersed. Indeed, this is one of the important advantages of using program transformations to perform the derivation: they make it easy for one to incorporate problem-domain-dependent and hardware-dependent optimizations to whatever extent is necessary to achieve the desired level of performance in the implemented program.

We are the first to admit that the problem for which we have demonstrated parity is still far from the scope and complexity of a grand challenge problem. Nevertheless, we have taken a step in the direction of demonstrating parity for such problems. We hope that our results will encourage others to attempt similar demonstrations.

2 The problem

The solution of problems in fluid flow is the subject of the functional specification and vectorizable code developed in this case study. Technically, the equations to be solved are conservation laws, or, more generally, first-order, quasilinear, hyperbolic partial differential equations. The solution of hyperbolic PDEs arises in many practical applications that involve wave phenomena, including acoustics, elasticity, and electromagnetism.

In all problems involving hyperbolic PDEs, *characteristics* play an important role. Characteristics are curves in space-time along which information propagates from the initial data. In the case of nonlinear hyperbolic PDEs, characteristics may intersect. When such intersections occur, the hyperbolic problem no longer has a unique solution, and a 'shock' forms. Shocks, and the discontinuities that accompany them, can be observed in everyday phenomena – traffic flow (the bunching of cars as faster moving vehicles overtake slower moving ones), waves breaking, and sonic booms.

The mathematical discontinuity at a shock, of course, creates difficulties for the designer of algorithms for solving hyperbolic PDEs. In general, algorithms based on usual mathematical techniques, such as the method of characteristics, provide fast, accurate solutions in regions well away from shocks but encounter both mathematical and programming difficulties when shocks occur. On the other hand, algorithms based on discrete approximations to the molecular dynamics of the physical problem, such as cellular automata, deal well with shocks, but their convergence is difficult to prove mathematically.

This complementarity of features and deficiencies makes combining the two methods attractive: use a cellular automaton to model the regions where shocks occur, but let the behaviour of the cell be that given by the method of characteristics in the remaining regions. Such an algorithm has other properties that are interesting for

high-performance computing – the algorithm exhibits data parallelism, and (as a result) it also vectorizes well.

A new algorithm for solving hyperbolic PDEs based on this combined approach is the subject of active research by Garbey and Levine (1990). As in other algorithms based on the method of characteristics, this combined algorithm does not compute the solution of the hyperbolic PDEs directly but rather computes the field of characteristics, from which one can easily derive the solution. In many problems, this approach is computationally more efficient than computing the solution directly.

This algorithm is an interesting choice for an attempt to achieve parity for a number of reasons:

- The algorithm is numerically intensive. Such algorithms are generally regarded as being outside the area of application of functional programming. Indeed, at a recent supercomputing working group meeting where our investigation of a functional solution for this problem was mentioned, the investigation was regarded as a waste of time, if not a joke.
- The algorithm is still subject to refinements and changes. To permit easy evaluation of these refinements and changes, minimizing the cost of recoding the executable program to incorporate them is important.
- The performance of the algorithm is being evaluated. Meaningful performance evaluation demands the use both of large grids (grids with many cells) and of many time steps, which in turn lead to long execution times. This circumstance rules out using the functional specification in a rapid prototyping mode, because naïve execution of the specification not only would be slow but also would require repeated duplication and copying of the grid. Note that in this situation, even if the execution time of the functional specification were within a factor of, say, two of the hand-coded program (regardless of the elegance or ease of modification of the functional specification), the specification would not be useful, because employing it would double the cost of evaluating the algorithm on expensive machines. Moreover, we lay great importance on retaining the freedom to write the functional specification in the clearest possible form, in order to make it as simple and understandable as possible. Thus, having to 'tailor' the functional specification to increase the economy of its direct execution is unacceptable.
- The algorithm, although conceptually simple, requires some difficult coding and optimization. These difficulties are particularly knotty in the natural extensions of the algorithm for two and three dimensions, in which the geometry of the grid (hexagonal in the two-dimensional case) requires intricate calculations.
- The algorithm is a candidate not only for vector, but also for parallel – and, in particular, data-parallel – implementations, because of the large grids it uses. Indeed, handwritten Fortran implementations of the algorithm were prepared both for the data-parallel Connection Machine 2 and for the vector architectures of the Alliant FX/8 and CRAY X-MP. Thus, multiple implementations of the same specification, each tailored to a particular

architecture, are required. Again, because this is an experimental algorithm, the effort expended to produce these different versions must be minimized to enable the algorithm to be evaluated on different hardware.

- The algorithm does not require floating-point arithmetic for its execution. Thus, it can be run with our current transformations and run-time support system (based on LISP F3 [Nordstrom, 1978]), which were developed to support functional specifications for symbolic computations and do not support the floating-point type. (We could have added support for floating-point data but did not wish to delay this demonstration of parity by doing so.)

In short, we believe that the cellular automation algorithm for hyperbolic PDEs is difficult enough to represent a good test of the applicability of functional programming to scientific computation, and thus is a good example for our case study. The requirements of ease of modification, efficient execution, and multiple executable realizations of the same specification provide opportunities for functional programming to aid the developers of the algorithm. Moreover, the complexity of the algorithm in the higher-dimensional cases should provide an opportunity to demonstrate the ability of the modularity inherent in functional programs to 'factor out' complexity. If functional programming is to gain credibility, it should be demonstrably effective on this sort of problem and in this type of situation.

2.1 Cellular automaton solution of a hyperbolic PDE

The algorithm we are specifying uses a cellular automaton (CA) technique to compute the characteristics of the solution to a hyperbolic PDE. Cellular automata have applications ranging from modelling snowflakes to computing a solution (as in this case) of a partial differential equation. In fluid dynamics computations, CA methods are usually used as a direct approximation to the molecular dynamics of the problem, because the automaton gives a simple model for a complex physical process. That is, each cell in the automaton models the behaviour of a 'molecule' in the fluid, and the transition rules of the automaton correspond to the possible results of collisions with other molecules. (Wolfram, 1986, discusses many interesting applications of cellular automata.)

A cellular automaton model consists of many identical cells, each having simple, locally determined behaviour. Nevertheless, the combined behaviour of the cells can be complex. The grid for a one-dimensional hyperbolic PDE is a cellular automaton consisting of a line of cells, each of which has information about the physical state of the problem. Updating the state information of each cell through a sequence of discrete time steps computes the characteristics of the solution to the hyperbolic PDE. At a given time step, the state information of all cells is updated simultaneously according to the same rule. (In technical terms, the updating of all cells simultaneously based only on values from the preceding iteration makes this a Jacobi, as opposed to a Gauss–Seidel, method.) The update rule is said to be 'local' because it depends only on the state information at the cell being updated and that of some set of its neighbouring cells.

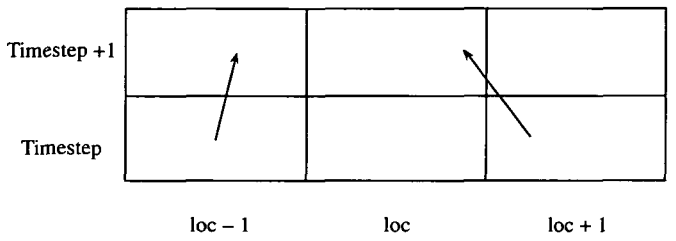

Fig. 1. A characteristic entering an empty cell at loc from its east neighbour.

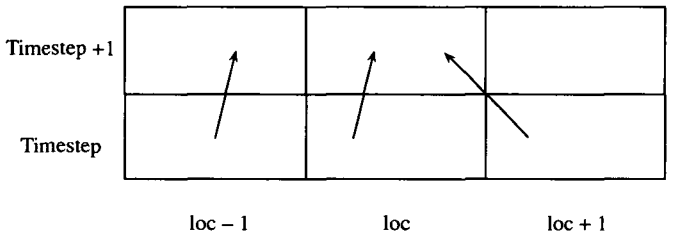

Fig. 2. Shock resulting from a characteristic entering an occupied cell at loc.

In the hyperbolic problem, the state information of each cell in the grid contains a dependent variable that represents some aspect of the physical state of the system (velocity, density, etc.). Characteristics move across the grid at a specified speed and in a specified direction. To model the hyperbolic PDE, each cell holds four pieces of information:

- a u value, which is the value of the dependent variable in the cell;
- an x value, which is the position of the moving characteristic within the cell;
- a *state*, which denotes whether a shock has occurred in the cell (as discussed in the following paragraph); and
- a *slope*, which denotes the speed and direction of propagation of the characteristic.

In the specification for the one-dimensional hyperbolic PDE problem, the directions are west (left) and east (right). In the current specification, the direction of the slope is kept in a separate component of the cell, called the *sign*, thereby simplifying certain computations.

The CA algorithm employs a further discretization in addition to that obtained by using a grid of cells. Each cell contains a discrete number of internal points at which the characteristic can be located, as denoted by its x value. In our specification, each cell contains 100 such points. At each time step, the x value is updated by adding or subtracting (depending on *sign*) the value of *slope* to obtain a new position (time steps are normalized to 1).

Of course, the interesting behaviour of this simple CA model concerns what happens when the new x value lies outside the current cell, possibly causing the characteristics in different cells to intersect and produce a shock. A characteristic leaves a cell when its x value exceeds the maximum number of internal points in the cell. For the one-dimensional problem, three specific cases are associated with a characteristic leaving a cell, two of which lead to the formation of a shock:

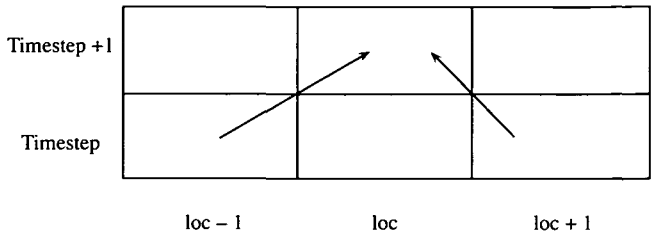

Fig. 3. Shock resulting from two characteristics entering an empty cell at loc.

(1) A cell that does not have a characteristic may obtain one if a characteristic moves into the cell from a neighbouring cell. This case is pictured in fig. 1.
(2) A shock occurs when one cell contains more than one characteristic. This situation may arise under one of two conditions: when a characteristic enters a cell that already contains a characteristic (and that characteristic is not leaving on the same time step), or when two characteristics enter the same cell at the same time step. These conditions are pictured in figs. 2 and 3.
(3) A shock ('crossing' shock) also occurs when two characteristics cross one another on a cell boundary at some time step. This condition is special because at no time do the two characteristics actually occupy the same cell. Nevertheless, this condtion is a shock because more than one characteristic would have occupied a cell if the cell boundaries of the grid had been differently positioned. This condition is pictured in fig. 4.

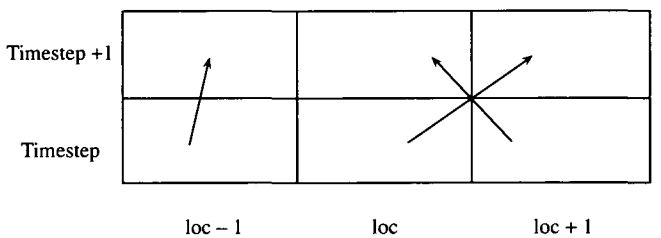

Fig. 4. Crossing shock resulting from two characteristics crossing on the east boundary of a cell at loc.

In addition to these cases, the update rule for cells also has a case for no shock – the simple movement of the characteristic within a given cell, as shown in fig. 5. This computation of the movement corresponds to that of the ordinary (non-CA) method of characteristics.

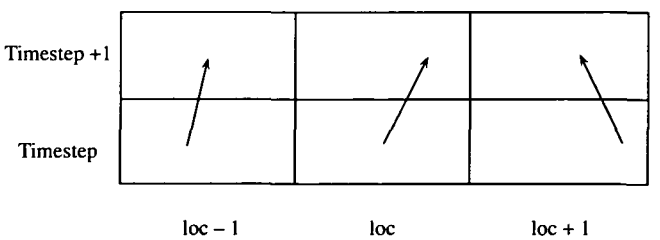

Fig. 5. Characteristic moving within a cell at loc.

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

2.2 Pure Lisp with data abstraction as a specification language

For our functional specification language we use pure Lisp (essentially a form of Church's lambda calculus [Church, 1941]), together with data abstraction. This specification language is similar to the pure functional subset of Scheme, as described in the 'Little Lisper' (Friedman and Felleisen, 1986). The specifications that we express in this language are high-level but still algorithmic and, in fact, executable. Indeed, we occasionally execute them in specification form in order to carry out rapid prototyping.

Pure Lisp is simple, even minimalist. It relies on just four basic constructs: conditional expressions, lambda abstraction (abstraction of an expression with respect to specified variables), application of lambda abstractions to arguments, and naming of lambda abstractions (to create recursive functions). Of course, higher-order functions (functions that take functions as arguments, or that are results of functions) are included. The great advantage of such a minimalist functional language is that, at least conceptually, it is easy to transform into an implementation; only the small number of constructs sketched above need be implemented.

We do not use the native data types of Lisp (except for Boolean and numeric types) in the upper levels of our specifications. Rather, we use data abstractions appropriate to the problem being specified. Moreover, we treat these data abstractions as if they were abstract data types, and we intend that our pure Lisp specifications be strongly typed, even though pure Lisp itself does not require strong typing.

2.3 A functional specification for the CA algorithm

We present some of the upper-level functions from the pure Lisp specification for the cellular automaton hyperbolic PDE algorithm outlined in section 2.1, together with commentary, in the remainder of this section. The complete specification appears in Appendix A.

One solves a hyperbolic problem by applying a function *steptime* to an initial grid for a specified problem of a specified size, a set of boundary values, an initial time, and a number of time steps to be performed:

steptime (*initgrid* (*problemtype*, *gridsize*), *bv*, *1*, *maxsteps*)

The *initgrid* function (see Appendix A) defines an initial grid representing suitable initial conditions for the hyperbolic problem. (For our tests, we used Riemann initial conditions.)

The result of taking a time step is the argument *grid* if the preceding time step was the last one; otherwise, the result is that obtained by taking another time step on an updated grid:

steptime (*grid*, *bv*, *step*, *maxsteps*) ≡
  if *step* > *maxsteps* **then**
    grid
  else
    steptime (*updategrid* (*grid*, *bv*), *bv*, *step* + *1*, *maxsteps*)

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

Mapping a local update rule for a cell over all the cells of the grid produces an updated grid:

updategrid (*grid*, *bv*) ≡
  mapgrid (λ *grid*, *loc* . *updatecell* (*grid*, *loc*, *bv*), *grid*)

The *mapgrid* operation applies a function (*mapgrid*'s first argument) to each cell in a grid (*mapgrid*'s second argument). For each cell in the grid, *mapgrid* applies the function to a pair of arguments – the grid and the location of the cell in the grid. Note that, under the semantics traditionally associated with '*map*' functions, the specification at this point is committed to a Jacobi method, in which the update of a cell depends only on the values of the cells in the argument grid and not on any newly updated cells. To specify a Gauss–Seidel method would require a more complicated *mapgrid* operation; in particular, the order of visiting cells in the grid would have to be specified, and *mapgrid* would have to apply its function argument not just to the grid but also to the (partially completed) new grid.

Updating a cell happens in one of two ways, depending on whether the cell is a boundary cell or an interior cell:

updatecell (*grid*, *loc*, *bv*) ≡
  if *isonboundary* (*loc*, *grid*) **then**
    updateboundarycell (*cellat* (*loc*, *grid*),
                          whichboundary (*loc*, *grid*), *bv*)
  else
    updateinteriorcell (*cellat* (*loc*, *grid*),
                          neighborsat (*loc*, *grid*))

This separation is convenient because the rules for updating boundary cells are significantly different from those for updating interior cells. The function for updating a boundary cell uses the boundary values provided as input to the problem to inject additional characteristics into the model when required (see Appendix A). Placing the test for being on a boundary here appears to be inefficient – the test is made within the *mapgrid* loop. However, as we discuss in section 3.2.2, transformations 'hoist' this test out of the loop, removing the apparent inefficiency.

Note how the modularity inherent in functional programming can be used (by means of the function *neighborsat*) to ensure that the updating of a cell depends only on the characteristic values of that cell and its neighbours, not on the exact location of the cell in the grid. Note also that at this level of abstraction we have not yet committed to implementing a cellular automaton algorithm. The preceding functions can be used just as well in the specification for a traditional 3-point (or, in two dimensions, 5-point) difference method.

Updating interior cells follows the cases outlined in section 2.1. If a shock or a crossing shock occurs, the result is an empty cell marked appropriately. If no shock occurs and a characteristic is entering the empty cell from one of its neighbours, then the cell was empty, and the result is a cell whose state reflects that a characteristic has

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

entered. Otherwise, the result is a cell whose state is computed according to the method of characteristics:

```
updateinteriorcell (cell, neighbors) ≡
  if isshocked (cell, neighbors) then
    emptymarkedcell (shock ())
  else if iscrossingshocked (cell, neighbors) then
    emptymarkedcell (crossingshock ())
  else if isenteringfrom (neighbors) then
    neighborenteredcell (neighbors)
  else
    timestepedcell (cell)
```

Note that thus far the dimensionality of the grid has played no role in the specifications of the functions. Thus, none of the preceding functions need be altered in going to a specification for a higher-dimensional problem.

The dimensionality of the grid does enter, however, in the specification of the functions *isenteringfrom* and *neighborenteredcell*. For a one-dimensional grid, each cell has two neighbours, west (left) and east (right). A characteristic is entering a cell from one of its neighbours (in the one-dimensional case) if the characteristic is leaving the west neighbour going east or leaving the east neighbour going west:

```
isenteringfrom (neighbors) ≡
  isexitingeast (west (neighbors)) | isexitingwest (east (neighbors))
```

If the characteristic is leaving the west neighbour of a cell, that characteristic is the basis for the updated state of the cell; otherwise, the characteristic from the east neighbour is the basis:

```
neighborenteredcell (neighbors) ≡
  if isexitingeast (west (neighbors)) then
    movedintocell (west (neighbors))
  else
    movedintocell (east (neighbors))
```

A shock occurs in a cell

(1) if the cell will have a characteristic on the next iteration (the cell's characteristic is not moving to a neighbouring cell) and if the characteristic of one (or both) of the cell's neighbours is entering it; or
(2) if (the cell will not have a characteristic on the next iteration, but) the characteristics of both of the cell's neighbours are entering it:

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
isshocked (cell, neighbors) ≡
  (hasstatenextiteration (cell)
& (isexitingeast (west (neighbors))
  | isexitingwest (east (neighbors))))
| (isexitingeast (west (neighbors))
  & isexitingwest (east (neighbors)))
```

A crossing shock occurs in a cell if the characteristic of the cell is entering one of its neighbours at the same time as the characteristic of that neighbour is entering the cell:

```
iscrossingshocked (cell, neighbors) ≡
  (isexitingwest (cell) & isexitingeast (west (neighbors)))
  | (isexitingeast (cell) & isexitingwest (east (neighbors)))
```

Note that, as far as the top level of the specification that we discuss in this section is concerned, the grid itself is still an abstract object; no implementation decisions have been made about it.

At the next level of detail of this specification, however, we do make an implementation decision to use an array to implement the grid. As one consequence, we define the *mapgrid* function in terms of a primitive function *maparraywithindex*. Primitive functions are ones (such as *car* and *cdr*) that are supplied by the implementation. In the case of *maparraywithindex*, transformations insert a form of the implementation that is tailored to be efficient on the target hardware. (For example, a *maparraywithindex* implementation tailored for a parallel machine might divide the grid into several blocks, instead of constructing a vector loop, as is done here.)

Specification of the remainder of the cellular automaton hyperbolic PDE solver proceeds in a similar manner until all functions (not only computational functions but also those implementing data abstractions) have been specified; Appendix A contains the complete specification.

We claim that this specification is a simple and natural one for this problem; indeed, we even believe that this specification is transparently clear. Moreover, we claim that we have not knowingly biased the specification in the direction of an efficient final implementation. Indeed, we have tried always to choose naturalness and clarity over efficiency, as discussed in conjunction with the placement of the *isonboundary* test in the *updatecell* function.

In writing this specification, we have also attempted to make the upper-level functions independent of the dimensionality of the problem, as pointed out in the discussion of *updateinteriorcell*. However, since preparing this paper, we have discovered a way to make the specification almost completely independent of dimensionality, by representing the neighbours of a point as an index set. We compare and contrast this new form of the specification to the form presented here in Boyle and Harmer (1991).

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

3 The transformational derivation

From the simple, pure functional specification of the preceding section we automatically derive an efficient implementation for computers having vector hardware, such as the Alliant FX/80 or the CRAY X-MP. We use the TAMPR program transformation system (Boyle, 1970, 1989) to apply a sequence of sets of program transformations that derive an efficient Fortran program from the higher-order functional specification. As we indicated in the introduction, most of these are basic transformations – ones that are needed to implement any functional specification in Fortran. These transformations form the framework for the derivation. Used alone, they do a highly competent job; we have transformed a number of functional specifications for both numeric and nonnumeric problems and have yet to see an example for which the derived program was not noticeably faster than compiled Franz Lisp (Boyle, 1989). Nonetheless, our basic transformations are applicable to a broad class of functional specifications, and the implementations they produce, still have the characteristics of implemented functional code: extensive copying and use of fresh storage, explicit use of a stack (or heap, if functions are used as first-class objects) to implement recursion, etc.

Such an implementation performs well, but in our view it cannot hope to equal the performance of good handwritten imperative Fortran or C code. For example, without further optimization such an implementation is prohibitively inefficient in terms of storage consumption when large arrays are used. Even if general remedies for such inefficiencies could be found, we believe that the speed of well-written code comes from taking advantage of properties of both the problem being solved and the target hardware. It is just this type of knowledge that we can capture and codify in sets of transformations that are problem-domain-oriented or hardware-oriented (but not both). We can then apply the problem-domain-oriented transformations as part of any derivation starting from a specification in the problem area, or we can apply the hardware-oriented transformations as part of any derivation ending in a program for the target hardware, to produce high-performance, automatically generated programs. (Over time, libraries of such transformations will accumulate, enabling efficient realizations of functional specifications from various problem areas to be produced easily for many different types of hardware, simply by drawing appropriate sets of transformations from the library.)

Thus, to transform the specification of the hyperbolic PDE solver into the implementation that achieves parity, we intersperse into the outline formed by the basic sets of transformations a few sets of transformations that perform problem-domain-oriented or hardware-oriented optimizations. These transformations, few in number but powerful in effect, guide the derivation in the direction of producing code that will vectorize and that will, when compiled by the Cray Fortran compiler, run efficiently on the CRAY X-MP hardware.

It is these sets of transformations that we emphasize in this section; however, we begin with a brief overview of the basic ones to provide a framework for discussion.

3.1 General outline of the Lisp-to-Fortran derivation

The sets of transformations implementing the basic pure-Lisp-to-Fortran derivation are designed to carry a pure Lisp specification into a Fortran program in a sequence of about 20 minor steps. The minor steps can be characterized broadly as belonging to five major steps:

(1) Canonicalizing the pure Lisp specification (standardization).
(2) Preparing pure Lisp for transformation to Fortran.
(3) Transforming prepared pure Lisp to structured, recursive Fortran.
(4) Transforming recursive Fortran to nonrecursive Fortran.
(5) Cleaning up and implementing the remaining abstractions.

3.1.1 Basic Lisp-to-Fortran transformations

The first major step consists of transformations whose primary effect is to simplify later transformation steps. Examples include converting multiple-variable lambda expressions to nests of single-variable ones, and renaming lambda variables to avoid name clashes (α-conversion).

The second major step uses the algebraic properties of the lambda calculus to manipulate the pure Lisp specification into a form in which all function applications have simple arguments (arguments that are variables or constants). This step uses formal algebraic properties and identities of the lambda calculus to introduce the temporary variables that are needed to hold the results of functions, conditional expressions, etc., that are themselves arguments to functions (Boyle, 1989).

The third major step replaces each Lisp function definition with a Fortran one, and assigns the Lisp expression representing the body of the definition to the Fortran function identifier. Transformations based on distributive laws for assignment then change this single assignment statement involving a complicated Lisp expression into a sequence of assignment statements involving only simple expressions. After this major step, the transformed program looks like Fortran, but it still assumes that functions can be called recursively.

The fourth major step implements recursion by introducing a stack to hold function arguments, local variables, and an indication of the point to which each function application should return. (This stack is, of course, ultimately represented by a Fortran array of fixed size; with tail recursion removed, the implementation of this specification requires only a few dozen stack cells.) The return point is represented by an integer that is used as an index in a Fortran computed go-to statement.

The fifth major step completes the Fortran implementation by inserting definitions for certain Lisp primitives, by implementing such 'structured Fortran' constructs as do-end loops, and by tidying up some debris (such as multiple labels on the same point in the program) left behind by the earlier transformations. At this stage, the program is expressed completely in standard, nonrecursive Fortran, which for our purposes plays the role of a portable, high-level assembly language.

A somewhat more detailed discussion of these transformation steps, including example code fragments at each stage, is given in Boyle (1989); an earler version

appears in Boyle and Muralidharan (1984). Of course, our derivation is only one of many possible approaches to converting pure Lisp into procedural code. One might conduct most of the derivation within the Lisp language, converting to Fortran only at the final step. For example, steps three and four outlined above could be carried out using Lisp notation; Harrison and Padua (1989) discuss such derivations. However, in our view, the virtues of the Lisp language lie in its functional subset; we see no advantage to using its procedural constructs over those of Fortran or C. The TAMPR transformation system provides structured extensions to Fortran in its subject language, and we choose to use this notation for the procedural stages of the derivation. Regardless of the notation used, the procedural code derived from pure Lisp can be guaranteed to have certain properties, which can be used to simplify the later steps of the derivation.

In a similar vein, one might express tail recursion in terms of continuations, and then translate the continuations to Fortran. While this approach is certainly feasible, we again do not see that it offers advantages over our proven approach.

3.1.2 Optional additional Lisp-to-Fortran transformations

Additional sets of transformations of a general nature may be inserted at various points in this basic derivation to perform desired optimizations. We add three such sets of transformations: one unfolds data abstractions; another performs tail-recursion elimination; and the third optimizes arithmetic operations.

As mentioned earlier, a major aspect of our approach to specification is the almost complete use of data abstractions. That is, except possibly at the lowest level, our specifications do not use list-processing primitives (for example). At the lowest level of specification, we may use such primitives to implement the accessors and constructors that we define for problem-oriented data abstractions. Alternatively, in a particular derivation we may define the implementation of some accessors and constructors directly by transformations. This approach has the effect of producing a very flexible specification for which one can easily change the target implementation at will.

Of course, such flexibility appears to have a price: if one naïvely implements the specification, the resulting program is spectacularly inefficient, because every data access incurs the overhead of at least one function application. As a first step in improving the efficiency of the implementation of such specifications, we apply transformations that result in the unfolding (copying) of the definitions of most accessors and constructors for data abstractions into the text of the program. The transformations that do so are general in that they are not keyed to specific data abstractions, but rather unfold the definitions of all simple functions in place of their applications.

We apply the transformations that unfold data abstractions fairly early in the derivation, between steps 1 and 2 discussed in the preceding section. The choice of when in the derivation to unfold data abstractions must balance two conflicting demands. On the one hand, eliminating data abstractions early has the advantage of reducing the number of constructs in the program, because several different abstract

data types may be implemented in terms of the same primitives, such as those for lists. This reduction in variety means that simplifications needed in the remainder of the derivation can be performed by a small set of transformations operating on primitive functions, rather than by a large set operating on the data abstraction functions. On the other hand, discarding the (variety of) data abstractions eliminates problem-oriented information from the derivation. Thus, the definitions of data abstractions must not be unfolded before problem-domain-oriented transformations that need this information have had a chance to take advantage of it.

Unfolding the definitions of data abstractions has significance for efficiency far beyond the elimination of a level of function application, important though that may be. Unfolding frequently makes other simplifications (optimizations) possible. In the case of the specification for the hyperbolic PDE solver, the effect of unfolding data abstractions is dramatic; it reduces the 69 functions in the specification to a driver program and a single tail-recursive function that updates the grid in a sequence of time steps. Thus, essentially all of the optimizations are made possible, or at least detectable, by the transformations that perform unfolding. (In this respect, our implementation transformations can be thought of as having the goal of converting a functional specification into a single function (or a very small number of functions) that has been tailored to a particular specification, rather than having the goal of making function application highly efficient, as do, for example, Bloss *et al.* (1988).)

Another important and well-known general transformation for improving efficiency is tail-recursion elimination. Because tail-recursion elimination introduces assignment, this transformation must be delayed until those transformations that rely for their correctness on properties of pure functional code have been applied. Thus, tail-recursion elimination can be inserted just before step 3 of the main derivation, which introduces assignment statements in converting to Fortran.

Finally, very late in the derivation, during step 5, we apply transformations that eliminate, where possible, type checking for arithmetic operations. Our underlying Lisp implementation, based on that of LISP F3 (Nordstrom, 1978), encodes the type tag (list, atom, or integer) of a variable as part of its value. These optimizing transformations eliminate the type-tag manipulations internal to arithmetic expressions. (In the present implementation, however, we do maintain the tags for integer values that are the final results of expressions, because such results are placed on the stack where they could be examined by the garbage collector. An optimization we might implement in the future would be to eliminate type tags in those derivations, such as this one, that are both known to be strongly typed and known not to require type information for garbage collection. Alternatively, switching to a modern, strongly-typed functional language for our specification language would potentially remove the need for type-tag optimizations altogether.)

These three optional sets of transformations are useful in many derivations. They are problem- and hardware-independent in the sense that they represent optimizations that a competent programmer should apply during coding, even if he is familiar with neither the problem domain nor the target hardware. Implementing these somewhat tricky optimizations by transformations guarantees that they will be performed during the derivation, and performed correctly.

3.2 Transformations specific to the problem domain and target hardware

The transformations that we discuss in this section comprise those that are problem-domain-oriented and those that are hardware-oriented, plus a few that are as general-purpose as those discussed in the preceding section, but whose development was occasioned by our work on this algorithm and specification.

3.2.1 Problem-domain-oriented transformations

The most important problem-oriented data abstraction in this specification is the grid on which the algorithm solves the hyperbolic PDE. A natural implementation for the grid is an array of structures with each structure representing the state of a cell in the grid. In fact, this implementation is so natural that when one codes a program for such a problem by hand, one immediately thinks in terms of arrays. Nevertheless, an abstract conception of grid is useful, not least because abstraction keeps the dimensionality of the problem from entering the specification too early.

Once one has decided to implement the grid as an array, the array transformations discussed in this section construct a loop to implement the higher-order *maparraywithindex* function and its function argument; this loop is so constructed that it ultimately can be vectorized by the CRAY X-MP compiler. In addition, the transformations handle the important (and general-purpose) matter of arranging to reuse the old copy of the array instead of allocating a new array for each time step. Thereby, they reduce the storage complexity of the specification by a factor of $O(maxsteps)$ in the implementation.

The first step in implementing the loop that updates the array representing the grid is to split the *maparraywithindex* operation of the specification into two parts: a *maparraywithindex′* operation that takes an existing array as an additional argument, and an *allocatearray* function that creates an array. (Of course, later in the derivation we intend to implement *maparraywithindex′* so that it stores its result in the additional array, and to implement *allocatearray* to reserve storage for an array. For the present, however, we can still think of *maparraywithindex′* and *allocatearray* as being pure functions.) Initially, after the definitions of *updategrid* and *mapgrid* have been unfolded, the *steptime* function has the form

```
steptime (grid, bv, step, maxsteps) ≡
  if step > maxsteps then
    grid
  else
    steptime
      (maparraywithindex
        (λ grid, loc . updatecell (grid, loc, bv), grid),
      bv, step + 1, maxsteps)
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

After introduction of *maparraywithindex'* and *allocatearray*, the function has the form

```
steptime (grid, bv, step, maxsteps) ≡
  if step > maxsteps then
    grid
  else
    steptime
      (maparraywithindex'
        (λ grid, loc . updatecell (grid, loc, bv), grid,
        allocatearray(sizeof(grid))),
      bv, step + 1, maxsteps)
```

The transformations now add a formal argument to *steptime* that serves as a local name for the new array; an application of *allocatearray* becomes the corresponding actual argument, giving

```
steptime' (grid, bv, step, maxsteps, newgrid) ≡
  if step > maxsteps then
    grid
  else
    steptime'
      (maparraywithindex'
        (λ grid, loc . updatecell (grid, loc, bv), grid, newgrid),
      bv, step + 1, maxsteps, allocatearray(sizeof(grid)))
```

Finally, because aliasing of the argument grid (the 'old grid') takes place in neither the initial nor recursive applications of *steptime'*, there exist no references to the old grid after the recursive application. Hence, the array to which the variable *grid* can points can be used as the actual argument corresponding to *newgrid* in the recursive application. This change gives the form

```
steptime' (grid, bv, step, maxsteps, newgrid) ≡
  if step > maxsteps then
    grid
  else
    steptime'
      (maparraywithindex'
        (λ grid, loc . updatecell (grid, loc, bv), grid, newgrid),
      bv, step + 1, maxsteps, grid)
```

Thus, after these transformations there need be only two copies of the array representing the grid (one created in *initgrid* and one in the initial invocation of *steptime'*), provided that *maparraywithindex'* is given an imperative implementation that actually stores the updated values of the grid cells in the preallocated array

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

represented by the argument *newgrid*. (One can see this step as a generalized form of the *finite differencing* transformation (Paige and Koenig, 1982), which replaces an expensive computation that is repeated each time a loop is executed with a cheaper incremental computation. At present, we are unsure how general this point of view is; it is obviously a topic for further investigation.) We apply the transformations that implement these important storage-usage improvements immediately before step 2 of the basic derivation.

The remaining step in implementing the grid is to transform the *maparraywithindex'* operation into a Fortran loop. The transformations construct the loop control statement so that the loop iterates over the elements of the array in some convenient order, and construct the body of the loop so that the body applies the first argument of *maparraywithindex'* (the function being mapped) to each element of the array. This change, of course, eliminates the overhead of a true higher-order implementation of *maparraywithindex'*. The change is an example of how our transformations use information in the specification to construct an implementation that is tailored to the specification. We apply these transformations during step 3 of the basic derivation.

It is worth remarking that, in comparison with the Fortran implementation of the CA hyperbolic PDE solver, the indirection (use of a pointer to a value rather than the value itself) permitted in the implementation of functional languages works to our advantage in handling the interchange of the grids. In our implementation, the arrays that ultimately represent grids are aggregated into structures to which the variables *grid* and *newgrid* point. Simply interchanging the roles of these two pointers (which serve as base addresses for the arrays in the final implementation) thus effects interchange.

In contrast, the programmer writing Fortran code has at hand no simple mechanism to aggregate arrays. He can use the indirection mechanism of our implementation, which involves allocating all problem arrays as subarrays of a single large array, but this approach quickly makes his program very difficult to write and to read. (The lack of clarity is no problem in our implementation, because one reads the abstract program, not the implemented code.) The Fortran programmer is thus usually forced to choose a relatively high-cost alternative. Among these alternatives are unrolling the loop to a depth of two, making the second copy of the loop body use the new and old grids in the opposite sense to the first; replacing the body of the loop by a (nonrecursive) subroutine call, unrolling the loop to a depth of two, and using the parameter mechanism of the subroutine call to effect the interchange; increasing the dimensionality of the arrays used by one (from one to two in the case of a one-dimensional grid), so that interchanging subscripts effects the interchange; or copying the grid. The handwritten code of Garbey and Levine (1990) uses a hybrid form of the last mechanism, which avoids some of the copying overhead at considerable expense in clarity.

3.2.2 Hardware-oriented transformations

In this section we turn to a discussion of the hardware-oriented transformations developed for this derivation, many of which, as noted earlier, are in fact generally useful transformations. To provide the background for understanding the design of,

and the motivation for, these transformations, we discuss briefly both the form of the *mapgrid* loop after the program has undergone unfolding and conversion to recursive Fortran, and the form that this loop must have in order to vectorize well.

We restrict our discussion to the vectorization of the relatively simple loops that arise when solving PDEs using a Jacobi method. Such loops contain no data dependencies; that is, at a given time step, the computation of values for the cells of the new grid depends only on the values of the cells of the old grid and, hence, the values in the new grid are independent of the order in which they are computed. In the cellular automaton algorithm, the main loop resulting from *mapgrid* consists of several cases represented as a nest of ***if-then-else*** statements. These cases, of course, arise from the alternatives in the state transition table of the cellular automaton.

After application of the transformations through step 3 of the derivation, including the problem-domain-oriented ones discussed in the preceding section, the *mapgrid* loop of the hyperbolic PDE solver has a form whose key structural features are illustrated by the code skeleton

```
do i = 1, n
  if (i .eq. 1) then
    ⟨update west boundary⟩
  else if (i .eq. n) then
    ⟨update east boundary⟩
  else if ((P11(i) .and. (P12(i) .or. P13(i))) .or. P14) then
    ⟨shock case⟩
  else if (P21(i)) then
    ⟨crossing shock case⟩
  else if (P12(i) .or. P13(i)) then
    if (P12(i)) then
      ⟨move in west neighbor case⟩
    else
      ⟨move in east neighbor case⟩
    end
  else
    ⟨time step cases⟩
  end
end
```

Here, *P11*, *P12*, etc., are predicates that represent the conditions for executing the cases. Each of these predicates may require several arithmetic or logical operations for its evaluation and, hence, each may represent a nontrivial amount of computation.

The first step in improving the efficiency of this form of the *mapgrid* loop is to apply the well-known general-purpose transformation of hoisting the computation of the boundary conditions out of the loop. This transformation reduces execution time on

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

sequential as well as vector machines. It applies to a loop that contains conditional statements whose conditions represent a simple partition of the index set of the loop. In effect, the transformation distributes the conditional statements out of the loop and uses them to partition the index set. This transformation thus produces several shorter (in this case, degenerate) loops and eliminates testing the conditions within the loop. In the case of the example skeleton above, this transformation produces the form

```
⟨update west boundary⟩ (i = 1)
⟨update east boundary⟩ (i = n)
do i = 2, n−1
  if ((P11(i) .and. (P12(i) .or. P13(i))) .or. P14) then
    ⟨shock case⟩
  else if (P21(i)) then
    ⟨crossing shock case⟩
  else if (P12(i) .or. P13(i)) then
    if (P12(i)) then
      ⟨move in west neighbor case⟩
    else
      ⟨move in east neighbor case⟩
    end
  else
    ⟨time step cases⟩
  end
end
```

We apply the hoisting transformation after the *maparraywithindex'* operation has been implemented as a loop (by the transformations added to step 3 of the basic derivation).

Must anything further be done to enable the loop that remains after the boundary updates have been hoisted out to vectorize efficiently? The answer is definitely yes – experiment showed that the loop does not vectorize at all under version 3.1.2 of the Cray CFT77 compiler. The loop fails to vectorize because it contains conditional statements nested in the ***then***-parts of other conditional statements. The compiler deems a loop containing such statements too complex to vectorize; it vectorizes only loops containing 'flat' nests of conditional statements. (Since we began these experiments, version 4.0 of the CFT77 compiler has been released. It does vectorize loops containing conditionals nested in the *then*-parts of other conditionals. Note, however, the advantage of using transformations – by writing transformations to flatten conditionals, we were able to obtain a vectorizable loop within a couple of hours instead of having to wait months for a new release of the compiler.)

As we discuss in more detail below, one reason for nesting conditional statements ('conditionals') in the ***then***-parts of other conditionals is to protect a test in the nested

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

conditional from evaluation in cases where the test would be undefined; protecting against division by zero and protecting against dereferencing the empty list are typical examples. In the case of the example program above, however, the nesting is not in any way fundamental; if the nested conditional evaluates the predicate *P12*, this predicate has already been evaluated as part of the expression *P12* **.or.** *P13*. In the specification for the hyperbolic PDE solver, the nesting arises simply from the way we have chosen to write the specification, specifically, from our desire to make it highly modular.

The next step is thus to flatten the conditionals. The algebra of conditionals can be used to show that the conditional within the loop in the preceding example is equivalent to the flattened conditional in the following program skeleton:

```
do i = 2, n−1
  if ((P11(i) .and. (P12(i) .or. P13(i))) .or. P14) then
    ⟨shock case⟩
  else if (P21(i)) then
    ⟨crossing shock case⟩
  else if ((P12(i) .or. P13(i)) .and. P12(i)) then
    ⟨move in west neighbor case⟩
  else if ((P12(i) .or. P13(i)) .and. .not. P12(i)) then
    ⟨move in east neighbor case⟩
  else
    ⟨time step cases⟩
  end
end
```

(Here and for the rest of the discussion, we omit from the example the boundary updates, which have been hoisted out of the loop.) Identities from Boolean algebra and the algebra of conditionals simplify the expressions used as conditions to produce

```
do i = 2, n−1
  if ((P11(i) .and. (P12(i) .or. P13(i))) .or. P14) then
    ⟨shock case⟩
  else if (P21(i)) then
    ⟨crossing shock case⟩
  else if (P12(i)) then
    ⟨move in west neighbor case⟩
  else if (P13(i)) then
    ⟨move in east neighbor case⟩
  else
    ⟨time step cases⟩
  end
end
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

As this code illustrates, the flattened conditionals may have conditions that share common subexpressions. In the case of the hyperbolic PDE solver, these common subexpressions are relatively expensive computations. One might hope the loop could be left in this form, expecting the CFT77 compiler to perform common-subexpression elimination on these computations. Although CFT77 does perform common-subexpression elimination on arithmetic expressions in a loop, experiment with the transformed code supports the hypothesis that the compiler does not do so for logical expressions. Therefore, we can increase the execution speed of the program by using transformations to perform common-subexpression elimination on the logical expressions used as conditions.

Applying the common-subexpression-elimination transformations brings the example program to the form

```
do i = 2, n-1
   112(i) = P12(i)
   113(i) = P13(i)
   if ((P11(i) .and. (112(i) .or. 113(i))) .or. P14) then
      ⟨shock case⟩
   else if (P21(i)) then
      ⟨crossing shock case⟩
   else if (112(i)) then
      ⟨move in west neighbor case⟩
   else if (113(i)) then
      ⟨move in east neighbor case⟩
   else
      ⟨time step cases⟩
   end
end
```

Is this form of the program optimal? No. Experiments with this form show that speed can be increased even further by extending 'common' subexpression elimination to the point of precomputing all of the conditions, so that each **if**-test consists of a simple test of a logical array element:

```
do i = 2, n-1
   112(i) = P12(i)
   113(i) = P13(i)
   11(i) = (P11(i) .and. (112(i) .or. 113(i))) .or. P14
   12(i) = P21(i)
   if (11(i)) then
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      ⟨shock case⟩
    else if (12(i)) then
      ⟨crossing shock case⟩
    else if (112(i)) then
      ⟨move in west neighbor case⟩
    else if (113(i)) then
      ⟨move in east neighbor case⟩
    else
      ⟨time step cases⟩
    end
  end
```

This form of the program performs more computation than does the preceding one (all of the logical expressions are evaluated every iteration, whereas *P21* was not in the preceding form), but this form is the more efficient one when vectorized. This efficiency is not surprising, because a vector processor such as the CRAY X-MP is, after all, a kind of shared-memory SIMD parallel computer. Hence the general principle for taking advantage of vectorization is similar to that for taking advantage of a SIMD machine: increasing the amount of work a program does may nevertheless increase the program's speed, if adding the work increases the uniformity of execution of the program. In this case, because of hardware features of the CRAY X-MP such as the availability of gather- and scatter-under-mask and compressed-vector instructions, and the need to optimize use of memory bandwidth (issues too complicated to discuss further here), precomputing the conditions for the conditionals reduces the number of distinct cases to be handled in the loop, making it more SIMD-like; hence, precomputation increases speed.

Unlike flattening conditionals and eliminating common subexpressions, transforming a loop to precompute all conditions is not possible for all loops. In our example, the form of the program that does not precompute conditions evaluates the expression *P21*(*i*) only for values of *i* for which the expression represented by *11*(*i*) is false, whereas the form with precomputation evaluates *P21*(*i*) for all *i*. Thus, the form with precomputation could be undefined in cases where the earlier form is well defined.

Actually, the problem runs deeper than the preceding paragraph suggests. Lisp semantics for ***and*** and ***or*** require that they be evaluated by using 'short-circuit' evaluation. That is, the operands of ***and*** and ***or*** must be evaluated left to right, stopping as soon as the value of the connective is determined (as soon as evaluation produces an operand that is false for ***and*** or true for ***or***). Our standard Lisp-to-Fortran transformations implement these semantics by converting the ***and*** and ***or*** connectives to nested conditional expressions.

If we consider the specification for the hyperbolic PDE solver literally to be written in Lisp and let these standard transformations apply, they would convert uses of ***and*** and ***or*** to even more deeply nested conditionals than illustrated in the preceding

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

example. In fact, the number of alternatives would be so large that each would contribute so few elements to the final result vectors that vectorization would produce little improvement over sequential execution. Thus, to produce a loop that will vectorize, we must avoid (if possible) the requirement for short-circuit evaluation of ***and*** and ***or***.

Fortunately, in the specification for the hyperbolic PDE solver, none of the tests actually used in any condition serves to protect the evaluation of any other test. Thus, we can implement the specification using non-short-circuit semantics. Interestingly, we can achieve this effect simply by *leaving out* the set of transformations that reexpresses ***and*** and ***or*** in terms of nested conditionals. Of course, we must add a set of transformations later in the derivation (late in step 3) to convert the Lisp ***and*** and ***or*** functions to the Fortran ***.and.*** and ***.or.*** operators, respectively.

Once we have established that none of the tests in any condition requires short-circuit evaluation, we can apply transformations that introduce precomputation of the complex conditions of the ***if-then-else*** statements to the beginning of the loop. (This introduction of precomputation is essentially similar to eliminating common subexpressions, except that we introduce precomputation even if there is only one instance of the expression.) The transformations that introduce precomputation add Fortran logical array temporaries to hold the results. These temporaries do not need to be allocated on the heap as do the arrays representing the grid because the logical arrays are reused every iteration and need not be interchanged. (In what appears to be a spectacular inefficiency engendered by being forced to communicate with CFT77 in Fortran 77 instead of in a higher-level notation, the thousands of words of storage represented by these logical temporary arrays are never actually used. The vector processor executes the loop as a sequence of loops, each computing 64 elements of the result vectors. In so doing, it is able to keep the values of the logical temporaries in vector registers, to use them as masks, and never to store them, because they are never used again. Fortunately, CFT77 recognizes this situation and does not actually reserve storage for these arrays.)

One more hardware-oriented transformation remains to be discussed. We apply it during step 3 of the derivation to change the representation of the grid. In the specification, the grid is a structure of cells, in which each cell has several components. The natural array implementation of such a grid is a multidimensional array, an array in which elements of a given kind, such as the x values, are not contiguous. Because access to an array of contiguous elements can be faster on the Cray than access to elements separated by a nonunit 'stride', we apply transformations that change the grid representation from an array of structures (the cells) to a structure of arrays.

4 Performance of the derived program

The Fortran program that the transformations outlined in the preceding section produce automatically from the specification given in Appendix A is shown in Appendix B. This program is, at least in principle, easily understandable in terms of the specification and the transformations, and so we do not discuss its structure further. This version of the program is targeted to a CRAY X-MP 1/8 system having

the extended memory address, compressed index/gather-scatter, and vector population count hardware features.

However, as discussed in the preceding sections, this version of our program is not the first version whose performance we measured. Rather, this version is the final result of a sequence of experiments. We began by writing a few sets of 'hardware-oriented' transformations (such as those discussed in the preceding section) based on our understanding of the operation of the CRAY X-MP. (We put quotation marks around 'hardware-oriented' in the preceding sentence because it soon became apparent that the lower levels of the hardware-oriented transformations must cater to the peculiarities of the Cray CFT77 compiler rather than simply to those of the raw CRAY X-MP hardware.) We then began experimenting. In each experiment, we derived a Fortran program from the specification using the current version of the transformations, measured that program's performance, altered or added hardware-oriented transformations to produce what we hoped would be a more efficient version of the program, and performed another experiment.

Do we imagine, based on the experimental approach just described, that every program specifier who uses the abstract programming and program transformation methodology will develop his own optimizing transformations? No, of course we do not; it is not worthwhile to develop special transformations to optimize specifications for simple problems. However, when using this methodology to solve large, long-running problems, such as the grand challenges, it may well be worthwhile for transformation specialists to develop transformations specific to the particular problem or hardware. Moreover, we believe that developing such optimizing transformations and adding them to a transformational derivation when using the TAMPR program transformation system is much easier than would be developing and adding them to a conventional compiler (which is essentially impossible), for several reasons:

- TAMPR exposes the optimization process. It permits one to examine the result of each step of the derivation of a program at the source-language level. Thus, it is fairly easy to determine where needed transformations should be inserted into the derivation.
- TAMPR transformations are specified in terms of the syntax of the source language (see Boyle, 1989, for some example transformations). Thus, one need not understand a complex, compiler-dependent representation of the program in order to add optimizing transformations to a derivation.
- TAMPR transformations are rewrite rules. Therefore, each is fairly simple. Hence, it is easy to see whether a transformation preserves the correctness of the program being transformed. We have found this property of simplicity to be a major help in writing correct optimizations.

Finally, problem-oriented transformations tend to be independent of the target hardware, while hardware-oriented transformations tend to be independent of a particular problem. For example, flattening conditionals and precomputing logical expressions are general strategies that will improve the performance on the Cray of any specification that uses conditionals, regardless of the problem it solves. These

independence properties permit problem-oriented transformations to be reused in derivations for many types of hardware, and hardware-oriented transformations to be reused in derivations for many types of problem. This reuse permits the effort required to develop sets of transformations to be amortized over a large number of derivations.

After a number of experiments, we obtained the set of transformations discussed in the preceding sections and the derived program given in Appendix B. Table 1

Table 1. *Program times on CRAY X-MP 1/8 for 16,384 cells, 1000 time steps, Riemann initial conditions*

Program version	CRAY X-MP time (sec)
Functional specification transformed to Fortran	7·283
Handwritten Fortran	7·551

presents the timing comparison between this version of our derived program and Garbey and Levine's (1990) program. While Garbey and Levine did not make a large investment in tuning their program for the CRAY X-MP, they did write the program with the architecture in mind and with the intent of getting decent vector performance on that machine.

The data in Table 1 were obtained using the Cray Fortran compiler CFT77, version 4.0.1.11, on a CRAY X-MP 1/8. We omitted tail-recursion elimination from the derivation of the timed program, because we discovered that with tail recursion eliminated the program runs slightly slower than the recursive version! (Do these data imply that Garbey and Levine's program would run faster if they recoded it so that the time-step loop is represented by simulated recursion? We do not know...)

These data show that our version certainly achieves parity with the handwritten program. In fact, our version is about 4% faster than the handwritten program. Determining exactly why our version is faster is difficult without a detailed timing analysis of the compiler-generated assembly language code. A cursory reading of the assembly language indicates, however, that our derived program is better than the handwritten program at overlapping the use of the multiple functional units of the Cray during the precomputation of the conditions for the **if**-statements in the main loop.

Of course, use of the experimental approach we have described invites the question of whether we have simply 'lucked out' in achieving parity by managing to manipulate the specification into a form that takes advantage of the sophisticated optimizations performed by the Cray CFT77 compiler. Could we do as well generating code for a less sophisticated compiler? One way (at least partially) to answer this question is to compare the performance of the derived program with that of the handwritten one on a typical sequential machine. (Of course, one should not set too much store by such a comparison, because *both* programs are, in fact, tuned for the Cray vector architecture.) We performed this experiment on a Sun 3/110C

Table 2. Program times on Sun 3/110C (16·67 MHz) for 16,384 cells, 100 time steps, Riemann initial conditions

Program version	Sun 3/110C time (sec)
Functional specification transformed to Fortran	136·880
Handwritten Fortran	131·712

Table 3. Program times on Sun 3/110C (16·67 MHz) and CRAY X-MP for 16,384 cells, 100 time steps, Riemann initial conditions

Program version	Sun 3/110C time (sec)	CRAY X-MP time (sec)
Functional specification transformed to Fortran	136·880	0·729
Functional specification, transformed to Fortran without redundant pre-computation of predicates	119·560	0·829
Handwritten Fortran	131·712	0·757

with 68020 and 68881 processors, running Sun OS 4.2 and the Unix f77 Fortran compiler. Table 2 gives the timing results for these runs. (Note that we have timed the programs for 100 time steps in Table 2 rather than for 1000 as in Table 1, in order to obtain reasonable running times on the Sun.) Indeed, on the Sun our program is about 4% slower than the handwritten program.

Do these results show that we have achieved our goal of catering to some of the properties of the Cray hardware and compiler in our hardware-oriented transformations? Or, could it not be that they simply produce very poor code, which the Cray compiler 'covers' with its sophisticated optimizations, but which the relatively unsophisticated Sun compiler exposes? Our first guess, based on the latter hypothesis, was that the most obvious inefficiencies in our program over the handwritten one are repeated references to the stack and repeated *car* operations within the loop. We prepared a version of our program with common subexpression elimination performed on these operations, but this version was slightly slower than the Cray version on the Sun. Evidently we had to look elsewhere for the cause of the inefficiency.

What is another possible explanation for this outcome? We have pointed out that some of our Cray optimizations increase the amount of work performed by the Cray-tailored version of the program. These optimizations result in fast execution on the Cray, but ought to be 'pessimizations' on the Sun. To test this hypothesis, we rederived our program, omitting the transformations that introduce redundant

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

precomputation of the complicated predicates in the **if**-conditions. The middle line of Table 3 gives the performance of this version on both the Sun and the Cray.

Comparing the Sun times for the version without redundant precomputation to those for the handwritten program shows that the version without redundant precomputation exceeds parity on the Sun. On the other hand, comparing its times on the Cray with those of our Cray-optimized version shows that the version without redundant precomputation has much-reduced performance on the Cray; in fact, this version has fallen substantially behind the performance of the handwritten program on the Cray. These results indicate that *the code-movement transformation* is the key hardware-oriented transformation that we perform – it is an optimization on the Cray and a pessimization on the Sun.

5 Other demonstrations of parity

We know of no other attempts to demonstrate parity between pure functional and handwritten code on vector supercomputers such as the CRAY X-MP, although the SISAL (Streams and Iterations in a Single-Assignment Language) project (Feo *et al.*, 1990) has attempted a parallel (nonvector) implementation on this machine and plans to construct a vectorizer for the SISAL compiler. Of course, one of the advantages of transforming a functional specification into a widely available language such as Fortran or C is that implementing the resulting program on any machine having the required compiler is simple. Moreover, transforming to a widely-available language enables one to reuse, instead of having to reproduce, the immense effort that has gone into producing sophisticated optimizing compilers for these languages on supercomputers.

Leaving aside the question of performance of functional programs on vector supercomputers for the moment, one may ask whether other approaches to implementing functional languages have demonstrated parity, regardless of the target computer architecture. There are two principal approaches to implementing functional languages: one involves direct implementation of lambda expressions by transformations or conventional compiler techniques; the other is based on combinatoric graph-reduction techniques.

Simon Peyton-Jones's (1987) book contains an excellent survey and description of combinatoric graph-reduction implementation techniques. These techniques have a strong theoretical foundation from the lambda calculus and combinatory logic, and they were originally thought to promise high efficiency, especially when implemented on special-purpose hardware (the 'G-machine') designed to carry out graph reductions efficiently.

Although attractive in principle, very few special-purpose graph-reduction machines have actually been built, and none is widely available. Even if a successful graph-reduction machine were built and achieved parity with good procedural programs, such a machine is unlikely to be a *cost-effective* alternative to conventional machines. In our opinion, barring a nearly complete conversion to the use of functional programming (a desirable but improbable event!), the inability to amortize the design, development, and software costs of special-purpose architectures over

large production runs will always prevent such machines from being competitive with general-purpose commodity chip sets.

If one accepts that existing von Neumann architectures must be employed because of their cost-effectiveness, graph-reduction implementation techniques can still be used, with the graph-reduction engine simulated by interpretation or compilation. Interpretive execution of graph-reduction implementations is exceedingly inefficient; a straightforward implementation executes about two orders of magnitude more slowly than conventional procedural code (Augustsson and Johnsson, 1989). Peyton-Jones discusses compilation of G-machine code into native code for conventional hardware; such an implementation of course performs better than an interpreter. The fastest implementation based on graph-reduction techniques of which we are aware is the Chalmers Lazy ML compiler (Augustsson and Johnsson, 1989). In effect, this system produces from each functional program a compiled special-purpose graph reducer tuned to that program. This compiler has achieved performance within factors of 1·5 to 5 of procedural (C language) code for simple benchmarks (8 queens, Fibonacci, prime numbers to 300, KWIC index). Such performance is impressive, especially in light of the fact that LML is a functional language having not only higher-order functions, but also lazy evaluation and pattern matching.

Among techniques based on transformation and compilation of lambda expressions are the work described here and that of the Yale Lisp and Functional Programming Research Group. The Yale group has developed a number of compilers for various functional languages. One is the Orbit compiler (Kranz, 1988; Kranz *et al.*, 1986) for T, a lexically scoped dialect of Scheme. This compiler has achieved parity with procedural (Pascal) code for several benchmarks similar to those used to benchmark LML.

Another Yale compiler is the one (Bloss *et al.*, 1988, 1989) for ALFL, a nonstrict functional language similar to SASL. The most recent version of this compiler, which incorporates strictness analysis and collected termination analysis, has achieved near-parity (within factors of 1 to 2) with the T compiler, which in turn is at or near parity with procedural code.

The ALFL compiler and our approach can be compared by observing that in ALFL the goal is to make the application of a function (including higher-order functions) as efficient as possible, whereas in our approach the goal is to eliminate as many function applications as possible (by unfolding). Of course, this observation is an oversimplification; our implementation does not eliminate all function applications, and the ALFL compiler does eliminate some.

Compiling optimally efficient code for function applications requires extensive global analysis of the program. Our approach of unfolding function definitions at points of application avoids the need for such analysis, at the expense of expanding the size of the generated code. Fortunately, unfolded function definitions can often be simplified, so that the code growth is not so great as might be expected. (In practice, we find that the code often shrinks!) Moreover, because our transformations unfold and simplify function applications, we can use such higher-order constructs as *map*, *reduce*, and *filter* in specifications without incurring any overhead at all. Each use of such a construct is replaced by code tailored to the particular function that is being

mapped, performing the reduction, or acting as the filter. On the other hand, in some situations producing tailored versions of unfolded functions requires not only unfolding, but also *folding*. Our current transformations do not attempt to fold simplified function definitions; indeed, folding is notoriously difficult to control in automatic transformation.

Work at MIT and UC Berkeley by Arvind, Nikhil, Traub, Hicks, Culler, and others on the Id language and its compilation aims at generating efficient code for parallel machines, especially dataflow machines, from flexible, high-level specifications. The Id language emphasizes explicit use of arrays, records, and other 'data-parallel' structures. In the dataflow context, Id treats the computation of each element of such a structure as an independent parallel process (Nikhil, 1991). In addition to functional representations for such structures, Id provides two other representations, M-structures and I-structures (Barth *et al.*, 1991). M-structures and I-structures are non-functional constructs. Compared to functional constructs, the use of M-structures and I-structures in some parallel applications can lead to sequential bottlenecks that limit parallel execution, while in others these structures may permit increased parallel execution.

Recently, this group has begun to investigate the problem of compiling Id into efficient code for non-dataflow architectures. To provide good performance on sequential machines, an Id compiler must perform sequentialization analysis and a type of flow analysis somewhat similar to strictness analysis. The compiler uses the results of these analyses to coalesce instructions into sequential threads. One Id compiler for sequential machines has achieved speeds within a factor of 2–5 of programs written in C (Culler *et al.*, 1991), excellent performance for code generated from a non-strict specification.

Another project having the goal of achieving parity for significant scientific computations on non-special-purpose hardware is the SISAL project. The SISAL language permits one to write code that looks much like ordinary procedural code while still guaranteeing that the code is referentially transparent.

Most of the work on SISAL has been directed toward constructing parallel implementations of SISAL programs. A highly optimizing compiler for the Sequent Balance shared-memory MIMD parallel processor has been implemented. This compiler makes extensive use of program analysis techniques to remove storage copying operations, etc. Of course, the necessary analysis is greatly facilitated by the referential transparency of the SISAL language. (We are indebted to an anonymous referee for pointing out the humour of this statement: it is because functional languages are referentially transparent that the analysis facilitated by this transparency is needed!) This compiler has achieved or exceeded parity with Fortran code using one processor for several of the Livermore Loops benchmark programs, which represent kernels of typical scientific calculations. Moreover, when run in parallel on the Sequent, these programs achieve good speedups. The current version of SISAL does not include higher-order functions, however, and makes some other concessions to efficient implementation (Feo *et al.*, 1990).

We have also applied our transformational techniques to the derivation of parallel realizations of functional specifications. As in this demonstration of parity, our

transformations have produced excellent results for shared memory machines – using 16 processors, speedups of 8 over the best sequential code, for real applications (Dritz and Boyle, 1987; Boyle, 1989).

6 Conclusions, critique, and further work

In the preceding sections, we have discussed how we used functional programming coupled with program transformation to enjoy the advantages of writing a clear, simple functional specification for a significant scientific computation without having to pay *any* price in terms of supercomputer performance compared to a handwritten Fortran program for the same application. In fact, the performance of the program derived from our functional specification slightly outstrips that of the handwritten program on the CRAY X-MP vector processor. This better-than-parity performance may be partly a matter of luck – the compiler is able to implement our derived program in a way that fortuitously overlaps certain computations to avoid delays in waiting for the operands to arrive.

Or is it luck? Because of the flexibility and ease of writing program transformations, we were able to experiment with a number of variant implementations of our specification. This experimentation enabled us to find one with excellent performance.

Such experimentation can be very difficult, if not impossible, when writing programs in procedural languages. From the writing of the first few lines, a Fortran-level program is committed to representations of the data and control flow that necessarily must be chosen in advance of any experimentation with that program's behaviour. Only rarely, and then only at high cost, is it possible to alter significantly these decisions once experiments have been performed. In contrast, writing a functional specification and implementing it by program transformations commits one only gradually to representations of data and control. Moreover, the various commitments tend to be independent of one another, and hence one can be easily modified without sacrificing the work done on the others.

Did we did purchase the advantages of deriving an efficient program from a functional specification at a high price in human effort? No. Preparing the problem-domain-oriented and hardward-oriented transformations that we wrote specifically for this application required less than one man-week. (This time does not include the time required to perform timing experiments on the Cray, which would be necessary no matter what methodology were used, assuming such experiments could be done at all.)

What type of expertise is required to develop program transformations? The principal skill required to write transformations is the ability to generalize – to express one's knowledge of how to write a particular program in such a way that that knowledge applies to all similar programs, not just to the one. Two important principles can be used to guide the development of the transformations: they must preserve correctness, and they must make progress toward a goal (by constructing canonical forms; see Boyle, 1989).

Beyond this skill, developing the hardware-oriented transformations for vector processors, and developing those specifically for the CRAY X-MP, requires

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

knowledge of the operation of the hardware and compiler. But, this knowledge is *no different* from that which an experienced Cray programmer must apply while writing a program in order to obtain peak performance from this machine. Using this knowledge to write transformations, however, has a *much more valuable* result than using it to write a program, because the hardware-oriented transformations can be applied in the derivation of many different programs. They thus provide a way to capture hardware-specific knowledge and make it available for reuse in other applications.

Have we biased the outcome of this experiment, either by the way we wrote our functional specification or by the way we wrote the transformations? In regard to the specification, we believe the answer is a resounding no! The specification is available for examination in its entirety in Appendix A. We believe that an examination will show that where we faced a choice between a specification having clarity, simplicity, and generality and one having efficiency, we have chosen the former.

The test for bias, or specificity, in the transformations is more subtle, because we do believe that some transformations need to embody problem-domain-dependent information. Is it possible, however, that we wrote transformations so specific to the hyperbolic PDE solver that they would not be applicable to any other specification? One way to try to answer this question is to determine whether the transformations can be used to obtain similar performance from specifications for other PDE solution methods. We have striven to make the transformations as general as possible (except where explicitly noted in the preceding sections), and we are in the process of writing a functional specification for another PDE solver to evaluate the generality of the transformations.

Where do we plan to go from here? We are beginning to modify the lower levels of the specification for solving the one-dimensional hyperbolic problem to produce a specification for solving the two-dimensional problem. One of the challenges in this problem is representing the computations in the hexagonal geometry of the two-dimensional grid. We expect to have to develop a few additional transformations, or variants of some of the ones we have implemented, to obtain efficient performance from this specification.

Modifying the specification is not the only direction for future work. We can also modify the transformations. Using program transformations to derive a program from a specification makes it easy to retarget the derivation to other computer architectures radically different from the Cray. In the case of this hyperbolic PDE solver, Garbey and Levine's original motivation for developing the algorithm was to produce a program that could take advantage of the SIMD architecture of the Thinking Machines Corporation Connection Machine 2 (CM-2). As time permits, we plan to write alternative transformations for the later stages of our derivation to produce code for the CM-2. Of course, such transformations, if properly written, would be useful not only for the specification for the hyperbolic PDE solver, but also for functional specifications for other computations.

In conclusion, we believe that studies demonstrating parity between functional and handwritten programs on significant problems are important steps toward a goal – the goal of making functional programming useful to the wide audience of scientists

and engineers badly in need of techniques to help them quickly write clear, correct, and efficient programs. We invite others to report on similar experiments.

Appendices

A Functional specification

The complete specification for the cellular automaton hyperbolic PDE solver, given in ordinary Lisp notation, rather than in the infix notation used in section 2.3, is

```
(defun driver ()
  ((lambda (ol)
     (steptime
       (initgrid (car ol) (car (cdr ol)))
       (makebv (car (cdr (cdr (cdr ol)))) (car (cddr (cddr ol))))
       1
       (car (cdr (cdr ol)))
     )
   )
     (read)
   )
)

(defun initgrid (option gridsize)
  (mapgrid
    (lambda (grid loc)
      (riemann (quotient (car (cdr ol)) 2) grid loc))
    (newcellgrid gridsize)
  )
)

(defun riemann (midpoint grid index)
  (cond
    ((or (lessp index midpoint) (eq index midpoint))
      (composecell
        (withstate) (signforeast) (maxpointsincell)
        (maxpointsincell) (midpointincell)
      )
    )
    (t
      (composecell
        (withstate) (nullsign) (nullslope) (nullu) (zerox)
      )
    )
  )
)

(defun steptime (grid bv step maxsteps)
  (cond
    ((greaterp step maxsteps) grid)
    (t (steptime (updategrid grid bv) bv (plus step 1) maxsteps))
  )
 )
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
(defun updategrid (grid bv)
(mapgrid (lambda (grid loc) (updatecell grid loc bv)) grid)
)

(defun updatecell (grid loc bv)
    (cond
        ((isonboundary loc grid)
          (updateboundarycell (cellat loc grid) (whichboundary loc grid) bv)
        )
        (t (updateinteriorcell (cellat loc grid) (neighborsat loc grid)))
  )
)

(defun updateboundarycell (cell whichboundary bv)
  (cond
    ((iswestboundary whichboundary)
      (updatewestboundaryelement cell (westbv bv))
    )
    (t (updateeastboundaryelement cell (eastbv bv)))
  )
)

(defun updateinteriorcell (cell neighbors)
  (cond
    ((isshocked cell neighbors) (emptymarkedcell (shock)))
    ((iscrossingshocked cell neighbors) (emptymarkedcell (crossingshock)))
    ((isenteringfrom neighbors) (neighborenteredcell neighbors))
    (t (timestepcell cell))
  )
)

(defun timestepcell (cell) (timestepelement cell))

(defun isenteringfrom (neighbors)
  (or (isexitingeast (west neighbors)) (isexitingwest (east neighbors)))
)

(defun neighborenteredcell (neighbors)
  (cond
    ((isexitingeast (west neighbors)) (moveintocell (west neighbors)))
    (t (moveintocell (east neighbors)))
  )
)

(defun moveintocell (cell) (moveelement cell))

(defun isshocked (cell neighbors)
  (or (and (hasstatenextiteration cell)
        (or (isexitingeast (west neighbors))
          (isexitingwest (east neighbors))))
    (and (isexitingeast (west neighbors))
      (isexitingwest (east neighbors)))
  )
)
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
(defun iscrossingshocked (cell neighbors)
  (or
    (and (isexitingwest cell) (isexitingeast (west neighbors)))
    (and (isexitingeast cell) (isexitingwest (east neighbors)))
  )
)

(defun isonboundary (loc grid)
    (or (iswestelement loc grid) (iseastelement loc grid))
)

(defun whichboundary (loc grid)
  (cond
    ((iswestelement loc grid) (signforwest))
    (t (signforeast))
  )
)

(defun iseastboundary (whichboundary) (eq whichboundary (signforeast)))

(defun iswestboundary (whichboundary) (eq whichboundary (signforwest)))

(defun isexitingwest (cell) (isgoingwest cell))

(defun isexitingeast (cell) (isgoingeast cell))

(defun updatewestboundaryelement (element bvw)
  (cond
    ((or (hasnostate element)
         (isleavingcell (newxvalue element))
         (and (hasstate element) (greaterp bvw (uof element)))
      )
      (composecell (withstate) (signforeast) bvw bvw (smallestpartofcell))
      )
    )
    (t
      (composecell (stateof element) (signof element)
        (slopeof element) (uof element) (newxvalue element)
      )
    )
  )
)

(defun updateeastboundaryelement (element bve)
  (cond
    ((or (hasnostate element)
         (isleavingcell (newxvalue element))
         (and (hasstate element) (lessp bve (uof element)))
      )
      (composecell (withstate) (signforwest) (difference 0 bve) bve (smallestpartofcell))
    )
    (t
      (composecell (stateof element) (signof element)
        (slopeof element) (uof element) (newxvalue element)
      )
    )
  )
)
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
(defun timestepelement (element)
  (cond
    ((hasstate element)
      (cond
        ((isleavingcell (newxvalue element))
          (composecell (nullstate) (nullsign) (nullslope)
            (nullu) (zerox)
          )
        )
        (t
          (composecell (withstate) (signof element)
            (slopeof element) (uof element) (newxvalue element)
          )
        )
      )
    )
    (t
      (composecell (nullstate) (nullsign) (nullslope) (uof element) (xof element))
    )
  )
)
(defun moveelement (element)
  (composecell (withstate) (signof element) (slopeof element)
    (uof element) (containedxvalue (newxvalue element))
  )
)
(defun hasstatenextiteration (element)
  (and (hasstate element) (not (isleavingcell (newxvalue element))))
)
(defun isgoingeast (element)
  (greaterp (times (newxvalue element) (signof element)) (eastmostpoint))
)
(defun isgoingwest (element)
  (lessp (times (newxvalue element) (signof element)) (westmostpoint))
)
(defun emptymarkedcell (mark) (emptymarkedelement mark))
(defun emptymarkedelement (mark)
  (composecell (nullstate) (nullsign) (nullslope) mark (zerox))
)
(defun newcellgrid (size) (newarray size nil))
(defun mapgrid (fn grid)
  (maparray-withparams-withindex grid (sizeofarray grid) fn)
)
(defun cellat (loc grid) (elementvalueofarray grid loc))
(defun neighborsat (loc grid)
  (makeneighborset (elementvalueofarray grid (westelement loc))
                   (elementvalueofarray grid (eastelement loc))
  )
)
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
(defun westelement (cellposition) (difference cellposition 1))

(defun eastelement (cellposition) (plus cellposition 1))

(defun iswestelement (elementposition elgrid) (eq elementposition 1))

(defun iseastelement (elementposition elgrid)
  (eq elementposition (sizeofarray elgrid))
)

(defun makeneighborset (westelement eastelement)
  (cons westelement eastelement)
)

(defun west (neighborset) (car neighborset))

(defun east (neighborset) (cdr neighborset))

(defun makebv (westelement eastelement) (cons westelement eastelement))

(defun westbv (bv) (car bv))

(defun eastbv (bv) (cdr bv))

(defun composecell (state sign slope u x)
  (cons state (cons sign (cons slope (cons u x))))
)

(defun stateof (cell) (car cell))

(defun signof (cell) (car (cdr cell)))

(defun slopeof (cell) (car (cddr cell)))

(defun uof (cell) (cadr (cddr cell)))

(defun xof (cell) (cddr (cddr cell)))

(defun nullslope ( ) 0)

(defun nullu ( ) 0)

(defun zerox ( ) 0)

(defun maxpointsincell ( ) 100)

(defun midpointincell ( ) 50)

(defun smallestpartofcell ( ) 10)

(defun isleavingcell (x) (greaterp x (maxpointsincell)))

(defun shock ( ) 12345)

(defun crossingshock ( ) 54321)

(defun nullsign ( ) 0)

(defun signforwest ( ) -1)

(defun signforeast ( ) 1)

(defun hasstate (element) (eq (stateof element) (withstate)))

(defun hasnostate (element) (not (eq (stateof element) (withstate))))

(defun withstate ( ) 1)

(defun nullstate ( ) 0)
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
(defun newxvalue (element) (plus (xof element) (slopeof element)))
(defun containedxvalue (xvalue) (difference xvalue (maxpointsincell)))
(defun eastmostpoint () 100)
(defun westmostpoint () -100)
```

B Derived code

The following code is the Cray Fortran program derived from the pure Lisp specification in Appendix A. We omit the definition for a large common block and various runtime support routines whose roles should be obvious from their names. Of course, this Fortran program is not intended to be readable or understandable; nevertheless, we display it simply to show the code that our transformations produce.

We omitted the transformations that eliminate tail recursion from the derivation of this program. For reasons we do not understand (but which presumably are related to the CFT77 compiler's global optimization strategy), the version with tail recursion eliminated runs slightly slower on the CRAY X-MP than does this version.

The comment lines containing *CDIR$ ivdep* instruct the Cray compiler to ignore what appear to be vector dependencies within a loop; without these directives, the compiler will not vectorize a loop containing such apparent dependencies. (The main loop that implements the optimized *mapgrid* function begins after the second of these directives.)

```
      subroutine driver
#include "common"
      integer plus , diff , times , quot , rmndr, time ,
     & second , cons
      logical g148 ( 16384 )
      integer g161
      integer g162
      integer g163
      integer g164
      integer g165
      integer g166
      logical g144 ( 16384 )
      logical g145 ( 16384 )
      logical g146 ( 16384 )
      logical g147 ( 16384 )
      logical g149 ( 16384 )
      logical g150 ( 16384 )
      integer g160
      integer g159
      integer g158
      integer g157
      integer g143
      mjp = jp
c     assert ( $stackref ( stack ( mjp ) ) )
      mjp = mjp - 11
      if ( mjp .le. ip ) call stkerx
      jp = mjp
      stack ( mjp ) = mknum ( - 100 )
      stack ( mjp + 1 ) = mknum ( - 1 )
      stack ( mjp + 2 ) = mknum ( 54321 )
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      stack ( mjp + 3 ) = mknum ( 12345 )
      stack ( mjp + 4 ) = mknum ( 10 )
      stack ( mjp + 5 ) = mknum ( 0 )
      stack ( mjp + 6 ) = mknum ( 2 )
      stack ( mjp + 7 ) = mknum ( 50 )
      stack ( mjp + 8 ) = mknum ( 100 )
      stack ( mjp + 9 ) = mknum ( 1 )
      stack ( ip + 1 ) = 1
      goto 138
 139 continue
      stack ( mjp + 10 ) = stack ( jp - 1 )
      mjp = mjp + 11
      jp = mjp
      return
c     assert ( g108 )
 138 continue
      ip = ip + 1
c     assert ( $stackref ( stack ( jp ) ) )
      jp = jp - 8
      if ( jp .le. ip ) call stkerx
      call read
      stack ( jp + 6 ) = stack ( jp - 1 )
      continue
c     assert ( maparray .ne. nil )
      stack ( jp + 4 ) = newarray ( car ( cdr ( stack (
     &        jp + 6 ) ) ) )
      stack ( jp + 3 ) = newarray ( car ( cdr ( stack (
     &        jp + 6 ) ) ) )
      stack ( jp + 2 ) = newarray ( car ( cdr ( stack (
     &        jp + 6 ) ) ) )
      stack ( jp + 1 ) = newarray ( car ( cdr ( stack (
     &        jp + 6 ) ) ) )
      stack ( jp ) = newarray ( car ( cdr ( stack ( jp
     &        + 6 ) ) ) )
      stack ( jp + 5 ) = cons ( stack ( jp + 4 ) ,
     &        cons ( stack ( jp + 3 ) , cons ( stack (
     &        jp + 2 ) , cons ( stack ( jp + 1 ) ,
     &        stack ( jp ) ) ) ) )
      g166 = stack ( jp + 6 )
      g165 = stack ( jp + 1 )
      g164 = stack ( jp )
      g163 = stack ( jp + 2 )
      g162 = stack ( jp + 3 )
      g161 = stack ( jp + 4 )
      if ( - numadd + car ( cdr ( stack ( jp + 6 ) )
     &     ) .lt. 1 ) goto 170
CDIR$ ivdep
      do 169 i = 1 , ( car ( cdr ( stack ( jp +
     &        6 ) ) ) - numadd )
      g148 ( i ) = i .ge. ( - numadd + ( car ( cdr (
     &        g166 ) ) ) ) / 2 .and. i .ne. ( - numadd +
     &        ( car ( cdr ( g166 ) ) ) ) / 2
      if ( g148 ( i ) ) goto 167
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      car ( g165 + i ) = 100 + numadd
      car ( g164 + i ) = 50 + numadd
      car ( g163 + i ) = 100 + numadd
      car ( g162 + i ) = 1 + numadd
      car ( g161 + i ) = 1 + numadd
      goto 168
167 continue
      car ( g165 + i ) = numadd
      car ( g164 + i ) = numadd
      car ( g163 + i ) = numadd
      car ( g162 + i ) = numadd
      car ( g161 + i ) = 1 + numadd
168 continue
169 continue
170 continue
      stack ( jp - 2 ) = stack ( jp + 5 )
      stack ( jp - 3 ) = cons ( car ( cdr ( cdr ( cdr
     &        ( stack ( jp + 6 ) ) ) ) ) , car ( cdr (
     &        cdr ( cdr ( cdr ( stack ( jp + 6 ) ) ) )
     &        ) ) )
      stack ( jp - 4 ) = 1 + numadd
      stack ( jp - 5 ) = car ( cdr ( cdr ( stack ( jp
     &        + 6 ) ) ) )
      stack ( jp - 6 ) = cons ( newarray ( car ( cdr (
     &        stack ( jp + 6 ) ) ) ) , cons ( newarray (
     &        car ( cdr ( stack ( jp + 6 ) ) ) ) , cons
     &        ( newarray ( car ( cdr ( stack ( jp + 6 )
     &        ) ) ) , cons ( newarray ( car ( cdr ( stack
     &        ( jp + 6 ) ) ) ) , newarray ( car ( cdr (
     &        stack ( jp + 6 ) ) ) ) ) ) )
      stack ( ip + 1 ) = 1
    write (*,98000) igetnm(car(cdr(stack(jp+6)))),
                    igetnm(car(cdr(cdr(stack(jp+6)))))
98000 format ('Using', i6, 'cells and performing', i10, 'iterations.')
    itime = isecnd(0)
      goto 140
 141 continue
    itime = isecnd (0) - itime
    write (*,98001) itime
98001 format ('Time in msec: ', i10)
      stack ( jp + 7 ) = stack ( jp - 1 )
      jp = jp + 8
      irlab = stack ( ip )
      ip = ip - 1
      goto ( 139 ) , irlab
c     assert ( steptime )
 140 continue
      ip = ip + 1
c     assert ( $stackref ( stack ( jp ) ) )
      jp = jp - 16 - 2
      if ( jp .le. ip ) call stkerx
      if ( stack ( jp + 14 ) .le. stack ( jp + 13 ) )
     &      goto 189
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      stack ( jp + 16 + 1 ) = stack ( jp + 16 )
      goto 190
  189 continue
      continue
c     assert ( maparray .ne. nil )
      stack ( jp + 1 ) = cdr ( cdr ( cdr ( cdr (stack
     &        ( jp + 12 ) ) ) ) )
      stack ( jp + 2 ) + car ( cdr ( cdr ( cdr ( stack
     &        ( jp + 12 ) ) ) ) )
      stack ( jp + 3 ) = car ( cdr ( cdr ( stack ( jp
     &        + 12 ) ) ) )
      stack ( jp + 4 ) = car ( cdr ( stack ( jp + 12
     &        ) ) )
      stack ( jp + 5 ) = car ( stack ( jp + 12 ) )
      stack ( jp ) = stack ( jp + 12 )
      stack ( jp + 6 ) = car ( stack ( jp + 16 ) )
      stack ( jp + 7 ) = cdr ( cdr ( cdr ( cdr ( stack
     &        ( jp + 16 ) ) ) ) )
      stack ( jp + 8 ) = car ( cdr ( cdr ( stack ( jp
     &        + 16 ) ) ) )
      stack ( jp + 9 ) = car ( cdr ( cdr ( cdr ( stack
     &        ( jp + 16 ) ) ) ) )
      stack ( jp + 10 ) = car ( cdr ( stack ( jp + 16
     &        ) ) )
c     assert ( left$boundary$condition )
      if ( car ( stack ( jp + 6 ) + 1 ) .eq. 1 +
     &        numadd .and. - numadd + car ( stack ( jp + 7
     &        ) + 1 ) - numadd + car ( stack ( jp + 8 )
     &        + 1 ) .le. 100 .and. ( car ( stack ( jp +
     &        6 ) + 1 ) .ne. 1 + numadd .or. car ( stack
     &        ( jp + 15 ) ) .le. car ( stack ( jp + 9 )
     &        + 1 ) ) ) goto 171
      car ( stack ( jp + 5 ) + 1 ) = 1 + numadd
      car ( stack ( jp + 4 ) + 1 ) = 1 + numadd
      car ( stack ( jp + 3 ) + 1 ) = car ( stack ( jp
     &        + 15 ) )
      car ( stack ( jp + 2 ) + 1 ) = car ( stack ( jp
     &        + 15 ) )
      car ( stack ( jp + 1 ) + 1 ) = 10 + numadd
      goto 172
  171 continue
      car ( stack ( jp + 5 ) + 1 ) = car ( stack ( jp
     &        + 6 ) + 1 )
      car ( stack ( jp + 4 ) + 1 ) = car ( stack ( jp
     &        + 10 ) + 1
      car ( stack ( jp + 3 ) + 1 ) = car ( stack ( jp
     &        + 8 ) + 1 )
      car ( stack ( jp + 2 ) + 1 ) = car ( stack ( jp
     &        + 9 ) + 1 )
      car ( stack ( jp + 1 ) + 1 ) = car ( stack ( jp
     &        + 7 ) + 1 ) - numadd + car ( stack ( jp +
     &        8 ) + 1 )
  172 continue
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
c     assert ( right$boundary$condition )
      if ( car ( stack ( jp + 6 ) - numadd + car ( car
     &        ( stack ( jp + 16 ) ) ) ) .eq. 1 + numadd
     &        .and. - numadd + car ( stack ( jp + 7 ) -
     &        numadd + car ( car ( stack ( jp + 16 ) ) )
     &        ) - numadd + car ( stack ( jp + 8 ) -
     &        numadd + car ( car ( stack ( jp + 16 ) ) )
     &        ) .le. 100 .and. ( car ( stack ( jp + 6 )
     &        - numadd + car ( car ( stack ( jp + 16 ) )
     &        ) ) .ne. 1 + numadd .or. cdr ( stack ( jp +
     &        15 ) ) .ge. car ( stack ( jp + 9 ) -
     &        numadd + car ( car ( stack ( jp + 16 ) ) )
     &        ) ) ) goto 173
      stack ( jp + 11 ) = - 1 + numadd
      car ( stack ( jp + 5 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = 1 + numadd
      car ( stack ( jp + 4 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = stack ( jp + 11
     &        )
      car ( stack ( jp + 3 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = 0 - ( - numadd
     &        + ( cdr ( stack ( jp + 15 ) ) ) ) +
     &        numadd
      car ( stack ( jp + 2 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = cdr ( stack (
     &        jp + 15 ) )
      car ( stack ( jp + 1 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = 10 + numadd
      goto 174
173   continue
      car ( stack ( jp + 5 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = car ( stack (
     &        jp + 6 ) - numadd + car ( car ( stack ( jp
     &        + 16 ) ) ) )
      car ( stack ( jp + 4 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = car ( stack (
     &        jp + 10 ) - numadd + car ( car (stack (
     &        jp + 16 ) ) ) )
      car ( stack ( jp + 3 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = car ( stack (
     &        jp + 8 ) - numadd + car ( car ( stack ( jp
     &        + 16 ) ) ) )
      car ( stack ( jp + 2 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = car ( stack (
     &        jp + 9 ) - numadd + car ( car ( stack ( jp
     &        + 16 ) ) ) )
      car ( stack ( jp + 1 ) - numadd + car ( car (
     &        stack ( jp + 16 ) ) ) ) = car ( stack (
     &        jp + 7 ) - numadd + car ( car ( stack ( jp
     &        + 16 ) ) ) ) - numadd + car ( stack ( jp
     &        + 8 ) - numadd + car ( car ( stack ( jp +
     &        16 ) ) ) )
174   continue
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      g143 = - numadd + car ( car ( stack ( jp + 16 )
     &       ) ) - 1
      g157 = stack ( jp + 7 )
      g158 = stack ( jp + 8 )
      g159 = stack ( jp + 10 )
      g160 = stack ( jp + 6 )
      g161 = stack ( jp + 5 )
      g162 = stack ( jp + 4 )
      g163 = stack ( jp + 3 )
      g164 = stack ( jp + 2 )
      g165 = stack ( jp + 1 )
      g166 = stack ( jp + 9 )
      if ( g143 .lt. 2 ) goto 188
CDIR$ ivdep
      do 187 i = 2 , g143
      g144 ( i ) = - numadd + car ( g157 + i ) -
     &       numadd + car ( g158 + i ) .le. 100
      g147 ( i ) = car ( g160 + i ) .eq. 1 + numadd
      g145 ( i ) = ( - numadd + ( car ( g157 + i - 1
     &       ) - numadd + car ( g158 + i - 1 ) ) ) *
     &       ( - numadd + ( car ( g159 + i - 1 ) ) )
     &       .le. 100
      g146 ( i ) = ( - numadd + ( car ( g157 + i + 1
     &       ) - numadd + car ( g158 + i + 1 ) ) ) *
     &       ( - numadd + ( car ( g159 + i + 1 ) ) )
     &       .ge. - 100
      g149 ( i ) = ( .not. ( g147 ( i ) .and. g144 ( i
     &       ) ) .or. g145 ( i ) .and. g146 ( i ) )
     &       .and. ( g145 ( i ) .or. g146 ( i ) )
      g150 ( i ) = ( ( - numadd + ( car ( g157 + i )
     &       - numadd + car ( g158 + i ) ) ) * ( -
     &       numadd + ( car ( g159 + i ) ) ) .ge. - 100
     &       .or. g145 ( i ) ) .and. ( ( - numadd + (
     &       car ( g157 + i ) - numadd + car ( g158 + i
     &       ) ) ) * ( - numadd + ( car ( g159 + i )
     &       ) ) .le. 100 .or. g146 ( i ) )
      if ( g149 ( i ) ) goto 185
      car ( g161 + i ) = numadd
      car ( g162 + i ) = numadd
      car ( g163 + i ) = numadd
      car ( g164 + i ) = 12345 + numadd
      car ( g165 + i ) = numadd
      goto 186
185   continue
      if ( g150 ( i ) ) goto 183
      car ( g161 + i ) = numadd
      car ( g162 + i ) = numadd
      car ( g163 + i ) = numadd
      car ( g164 + i ) = 54321 + numadd
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      car ( g165 + i ) = numadd
      goto 186
183 continue
      if ( g145 ( i ) ) goto 181
      car ( g161 + i ) = 1 + numadd
      car ( g162 + i ) = car ( g159 + i - 1 )
      car ( g163 + i ) = car ( g158 + i - 1 )
      car ( g164 + i ) = car ( g166 + i - 1 )
      car ( g165 + i ) = car ( g157 + i - 1 ) -
     &          numadd + car ( g158 + i - 1 ) - 100
      goto 186
181 continue
      if (g146 ( i ) ) goto 179
      car ( g161 + i ) = 1 + numadd
      car ( g162 + i ) = car ( g159 + i + 1 )
      car ( g163 + i ) = car ( g158 + i + 1 )
      car ( g164 + i ) = car ( g166 + i + 1 )
      car ( g165 + i ) = car ( g157 + i + 1 )
     &          numadd + car ( g158 + i + 1 ) - 100
      goto 186
179 continue
      if ( g147 ( i ) ) goto 177
      car ( g161 + i ) = numadd
      car ( g162 + i ) = numadd
      car ( g163 + i ) = numadd
      car ( g164 + i ) = car ( g166 + i )
      car ( g165 + i ) = car ( g157 + i )
      goto 186
177 continue
      if ( g144 ( i ) ) goto 175
      car ( g161 + i ) = numadd
      car ( g162 + i ) = numadd
      car ( g163 + i ) = numadd
      car ( g164 + i ) = numadd
      car ( g165 + i ) = numadd
      goto 186
175 continue
      car ( g161 + i ) = 1 + numadd
      car ( g162 + i ) = car ( g159 + i )
      car ( g163 + i ) = car ( g158 + i )
      car ( g164 + i ) = car ( g166 + i )
      car ( g165 + i ) = car ( g157 + i ) - numadd +
     &          car ( g158 + i )
186 continue
187 continue
188 continue
      stack ( jp - 2 ) = stack ( jp )
      stack ( jp - 3 ) = stack ( jp + 15 )
      stack ( jp - 4 ) = stack ( jp + 14 ) + 1
```

https://doi.org/10.1017/S0956796800000289 Published online by Cambridge University Press

```
      stack ( jp − 5 ) = stack ( jp + 13 )
      stack ( jp − 6 ) = stack ( jp + 16 )
      stack ( ip + 1 ) = 2
      goto 140
142 continue
      stack ( jp + 16 + 1 ) = stack ( jp − 1 )
190 continue
      jp = jp + 16 + 2
      irlab = stack ( ip )
      ip = ip − 1
      goto ( 141 , 142 ) , irlab
    end
```

Acknowledgments

We are indebted to both David Levine and Marc Garbey for explaining their algorithm and the difficulties of implementing it, and to Hans Kaper for encouraging us to work on this problem.

References

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *Computer J.*, 32 (2): 127–41.

Barth, P., Nikhil, R. S. and Arvind. 1991. M-Structures: Extending a parallel, non-strict, functional language with state. MIT Computation Structures Group Memo 327, MIT/Laboratory for Computer Science, Cambridge, MA, USA. (To appear in *Proc. Functional Programming and Computer Architecture*, Cambridge, MA, USA. (To appear in *Proc. Functional Programming and Computer Architecture*, Cambridge, MA (Aug. 28–30 1991).)

Bird, R. and Wadler, P. 1988. *Introduction to Functional Programming*. Prentice-Hall, New York.

Bloss, A., Hudak, P. and Young, J. 1988. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1: 147–64.

Bloss, A., Hudak, P. and Young, J. 1989. An optimizing compiler for a modern functional language. *Computer J.*, 32 (2): 152–61.

Boyle, J. M. 1970. A Transformational Component for Programming Language Grammar. ANL-7690, Argonne National Laboratory, Argonne, IL, USA (July).

Boyle, J. M. 1980. Program adaptation and program transformation. In R. Ebert, J. Lueger, and L. Goecke (editors), *Practice in Software Adaptation and Maintenance*, pp. 3–20, North-Holland, Amsterdam.

Boyle, J. M. and Muralidharan, M. N. 1984. Program reusability through program transformation. *IEEE Trans. Software Eng.*, 10 (5): 574–88 (Sept.).

Boyle, J. M. 1989. Abstract programming and program transformations – an approach to reusing programs. In T. J. Biggerstaff and A. J. Perlis (editors), *Software Reusability, Volume I*, pp. 361–413, Addison-Wesley, New York.

Boyle, J. M. and Harmer, T. J. 1991. Functional specifications for mathematical computations. In B. Moeller (editor), *Proc. IFIP TC2/WG2.1 Working Conf. on Construction Programs from Specifications*, Pacific Grove, California (May 13–16), Elsevier, Amsterdam.

Burton, F. W. and Kollias, J. Y. G. 1989. Functional programming with quadtrees. *IEEE Software*, 6: 90–97 (Jan.).

Church, A. 1941. *The Calculi of Lambda Conversion.* Princeton University Press.

Culler, D. E., Sah, A., Schauser, K. E., von Eicken, T. and Wawrzynek, J. 1991. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems.*

Dritz, K. W. and Boyle, J. M. 1987. Beyond 'Speedup': Performance Analysis of Parallel Programs. Argonne National Laboratory Technical Report ANL-87-7, Argonne, IL, USA (Feb.).

Federal Coordinating Council for Science, Engineering and Technology. 1987. A research and development strategy for high performance computing. Executive Office of the President, Office of Science and Technology Policy, Washington, DC (Nov. 20).

Feo, J. T., Cann, D. and Oldehoeft, R. R. 1990. A report on the SISAL language project. *J. Parallel and Distributed Computing* (Dec.).

Field, A. J. and Harrison, P. G. 1988. *Functional Programming*, Addison-Wesley, New York.

Friedman, D. P. and Felleisen, M. 1986. *The Little LISPer*, Science Research Associates, Inc., Chicago, IL, USA.

Garbey, M. and Levine, D. 1990. Massively parallel computation of conservation laws. *Parallel Computing*, **16**: 293–304.

Harrison, L. and Padua, X. 1989. Parcel: Project for the automatic restructuring and concurrent evaluation of Lisp. In *Proc. 1988 Int. Conf. on Supercomputing*, pp. 527–38, ACM Press, New York.

Kelly, P. 1989. *Functional Programming for Loosely-Coupled Multiprocessors*, Pitman Publishing/MIT Press, London/Cambridge, MA, USA.

Kelsey, E. and Hudak, P. 1989. Realistic compilation by program transformation. In *Proc. 16th ACM Symposium on Principles of Programming Languages* (Jan.).

Kranz, D. 1988. ORBIT: An Optimizing Compiler for Scheme. Yale University Technical Report YALEU/DCS/RR-632, Yale University, New Haven, CT, USA.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. 1986. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices*, 21 (7) (July).

Nikhil, R. S. 1991. The parallel programming language Id and its compilation for parallel machines. In *Proc. Workshop on Massive Parallelism*, Amalfi, Italy. Academic Press, New York.

Nordstrom, M. 1978. LISP F3, Users Guide. Report, Datalogilaboratoriet, Uppsala University, Uppsala, Sweden (June).

Paige, R. and Koenig, S. 1982. Finite differencing of computable expression. *ACM Trans. Programming Lang. and Syst.*, 4 (3): 402–54 (July).

Peyton-Jones, S. L. 1987. *The Implementation of Functional Programming Languages*, Prentice-Hall, New York.

Wolfram, S. 1986. *Theory and Applications of Cellular Automata*, World Scientific, Singapore.