The Construction of Numerical Mathematical Software for the AMT DAP by Program Transformation.

James M. Boyle ‡[1], Maurice Clint, Stephen Fitzpatrick [2], Terence J. Harmer

The Queen's University of Belfast
Department of Computer Science
Belfast BT7 1NN
Northern Ireland

‡Mathematics and Computer Science Division
Argonne National Laboratory
Argonne IL60439
USA

1 Introduction

The general advantages of functional programming - naturalness of expression (in some problem areas), ease of proof, and clarity - are well known. In the field of numerical mathematical computation, however, these advantages are normally outweighed by functional programming's main disadvantage - inefficient execution. In this paper, we show how a functional style can be used for the elegant specification of numerical algorithms while still obtaining highly efficient implementations through the application of program transformations.

2 Notation: A Functional Specification Language

For our functional specification language we use a (very) small subset of the language constructs available in Standard ML (SML) [9]. The specifications that we express in this language are high-level. They are, however, algorithmic and, in fact, executable. Indeed, we occasionally execute them in specification form in order to carry out rapid prototyping.

We provide a standard library of vector/matrix operations to facilitate the specification of numerical mathematical algorithms. The transformational derivation must provide an implementation for each primitive operation tailored for the hardware architecture in use. In addition, a numerical mathematical algorithm is expressed using the following general-purpose vector/matrix functions.

Generate :$shape \times (Index \rightarrow \alpha) \rightarrow Matrix(shape, \alpha)$

where $shape$ is a descriptor of vector/matrix of α type elements and Index is position in that vector/matrix shape. The $generate$ function returns a vector/matrix with shape given by its shape argument and where each element of the vector/matrix is defined by the function that is an argument to the $generate$ function.

Map :$Matrix(shape, \alpha) \times (\alpha \rightarrow \beta) \rightarrow Matrix(shape, \beta)$

$map(V, \lambda x.B) \overset{\text{def}}{=} generate(Shape(V), \lambda k.(\lambda x.B)(element(V, k)))$

[1]This work was supported by the Applied mathematical sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

[2]This work is supported by a research studentship from Department of Education, Northern Ireland.

Reduce :$Shape \times (Index \to \alpha) \times (\alpha \times \alpha \to \alpha) \times \alpha \to \alpha$

A $reduce$ function combines the elements of a vector/matrix by means of a binary function to produce a single value.

3 The Parallel Orthogonal Transformation(POT) Algorithm

The POT algorithm calculates the eigenvectors, Q, and eigenvalues, Λ of a real symmetric matrix A. The algorithm is based on the construction of a sequence of symmetric matrices as shown below:

1. $U_0 = I$
2. $U_{k+1} = ortho(A.U_k.transform(B_k), diagonal(B_k)), k \geq 0$, where the sequence B_k is constructed as follows:
3. $B_0 = A$
4. $B_k = U_k^T.A.U_k$

Then $\lim_{k\to\infty} U_k = Q$ and $\lim_{k\to\infty} B_k = \Lambda$ [7]. The function $ortho$ orthonormalizes the columns of its non-singular matrix argument using the standard Gram-Schmidt method. An SML function Pot may be defined thus:

```
fun Pot( A: real Array, U: real Array) : real Array * real Array =
 let B = multiplymatrix(transpose(U), multiplymatrix(A, U))
 in
  if ( is_satisfactory(B) )
  then (U, B)
  else
    Pot(A,
      orthogonalize(multiplymatrix(A, multiplymatrix(U, transform(B))),
            diagonal(B))
 end;
```

The operation *transform* produces from its matrix argument a matrix, T_k, each column of which is an approximation to an eigenvector of B_k. The components of T_k are computed by an SML function thus:

```
fun Transform( M : real Array) : real Array
 = let
  fun Calculate(M: real Array, i: int, j: int) : real
   = let val d = M@[j, j] - M@[i, i]
    in
     2*M@[i,j]/(d+sign(d)*sqrt(sqr(d)+4*sqr(M@[i,j])))
    end
   in
    generate(Shape(M), fn(i, j)=>
    if ( i > j ) then Calculate(M, i, j)
    else if i = j then 1.0
    else ˜Calculate(M, j, i))
 end;
```

A $generate$ function is used to construct the transformation matrix from its argument matrix, M. A local function $Calculate$ defines the value of the (i, j)th element of the transformation matrix. The generating function embodies the cases required by the algorithm specification.

A similar development may be used for the ortho function. We claim that we have not intentionally biased these definitions. Indeed, we would further claim that they represent a natural SML formulation of the algorithm specification.

4 Deriving Efficient Programs from Functional Specifications

Of course, our goal is not simply to execute our functional specifications as *ML programs*, where execution can often be excruciatingly slow. POT has been designed with large matrices in mind and is interesting because it permits implementations tailored for execution on computers having novel advanced architectures [6, 10, 11, 7]. Efficient ML implementations are not likely to be available for such computers.

We use the TAMPR program transformation system [3, 4] to apply a sequence of sets of program transformations that produce an efficient Fortran or C program from the higher-order functional specification. Most of the transformations are basic; that is, they can be employed in the efficient implementation of any functional specification. These transformations form the framework for the derivation. (Used alone, they are highly effective [4].) We intersperse the basic set of transformation with a few sets of other transformations that perform either 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 exploit the specialized hardware of the ADT DAP.

4.1 DAP-Specific Transformations

Earlier we defined two primitive vector/matrix functions: $generate$ and $reduce$. These functions do not have equivalents in DAP Fortran: in order to implement an *arbitrary* $generate$ or $reduce$ in DAP Fortran we must use an explicit loop where expressions involve manipulations of individual matrix elements. However, certain simple forms of $generate$ and $reduce$ *do* have equivalents in DAP Fortran (and in other languages for array processors). Also, to make the most effective use of the processor array the aim is to produce an implementation where whole vector/matrix manipulations are used rather than giving alternative manipulations expressed as a sequence of simple vector/matrix element operations.

The strategy that the transformation set for the DAP adopts is *to propagate* $generate$ *functions into expressions until the function argument of each* $generate$ *is a value that is either independent of the generation index, or is a vector/matrix element operation*. This strategy aims to remove occurrences of vector/matrix element operations and arises from the desire to realize computations as whole vector/matrix operations.

The strategy requires an algebra that defines how a $generate$ may be propagated into expressions. This propagation algebra has 5 rules:

1. A $generate$ applied to a function the value of which is the vector/matrix element at the generating index is replaced by the vector/matrix - the identity $generate$; $generate(Shape(M), \lambda((i, j).M@[i, j])) \Longleftrightarrow M.$
2. A $generate$ applied to a function the value of which is independent of the generating index is replaced by an $expand$ function that creates a vector/matrix in which all the elements have that independent value; e.g., $generate(Shape(M), \lambda((i, j).2)) \longrightarrow expand(2, n, n)$, where matrix M is of shape $n \times n.$
3. A $generate$ with a scalar generating function that has a componental equivalent is converted to that componental equivalent. The arguments to which the componental function is applied must be vectors/matrices, so the $generate$ is propagated inward and applied to the arguments of the componental function; e.g.,
$generate(Shape(A), \lambda((i, j).A@[i, j] + A@[i, j])) \longrightarrow$
$generate(Shape(A), \lambda((i, j).A@[i, j])) + generate(Shape(A), \lambda((i, j).A@[i, j]))$
4. A $generate$ of a conditional expression becomes an application of a data-parallel conditional - a $join$ function. The $generate$ is propagated inwards to the limbs of the conditional and applied to the conditional guards and the guarded results. The $join$ function combines ($joins$) the results to form a single composite result for the conditional expression.
5. A $reduce$ for which an aggregate function is defined is replaced by an application of that aggregate function.

The transformation strategy uses the a number simplification/optimization transformations that remove duplicate computation and to make use of standard arrangement/rearrangement being performed to the vector/matrix elements (- the interested reader is referred to [5]).

The transformations outlined above are not specific to the DAP or any other array processor architecture: they target the set of operations that may be implemented efficiently on architectures of this kind. Thus, most of the transformations above may be used in a derivation aimed at deriving efficient code for the Cray X-MP.

5 Derived DAP Code

Space does not permit us to trace the operations of the transformations that implement this algebra - the interested reader is referred to [5]. The derived DAP code for the $Transform$ function given above that is the result of their application is given in Figure 1. The variable g22 contains the transformation matrix. The resulting program may appear to be rather ugly, but **it is not intended that *this* form should be read**. The identifier names are chosen for convenience rather than for understandability or readability.

6 Comparison with Hand-Crafted Versions of POT

There are two versions of DAP Fortran: Fortran-Plus [1] and Fortran-Plus Enhanced [2]. The former requires all arrays to be the size of the processor grid (32 by 32); the latter allows any size of array to be used, with the compiler automatically sectioning arrays into 32 by 32 arrays.

```
 g371598 = patlowertri( n ) .and. .not. patunitdiag( n )
 g371651 = patlowertri( n )
 g371652 = patunitdiag( n )
 g22( g371651 .and. g371652 ) = 1
 g371651( g371652 ) = .false.
 g371652 = g371598
 g371597 = matr( g371600 , n ) - matc( g371600 , n )
 g371629 = 1
 g371629( g371597 .lt. 0 ) = - 1
 g22( g371651 .and. g371652 ) = ( - 2 * b ) / (
&     g371597 + g371629 * sqrt( g371597 * g371597 +
&     4 * ( b * b ) ) )
 g22( .not. patlowertri( n ) ) = - tran( g22 )
```

Figure 1: Derived DAP Code for the Transform Function.

In Table 1 the execution time (for each approximation) [3] for the implementation of POT derived by automatic program transformation is shown together with times for two hand-crafted versions - the first is written in Fortran-Plus and the other written in Fortran Plus Enhanced. These hand-crafted versions have been analysed in [8, 11]. A hand-crafted Fortran-Plus version of POT is between 12% and 13% faster than a hand-crafted Fortran-Plus Enhanced version. As reported in [8] the code produced by the Fortran-Plus Enhanced compiler for frequently occurring linear algebra operations (e.g. matrix product) is very efficient but it is less so on more specialized operations (e.g. Gram-Schmidt orthogonalization).

The hand-crafted and automatically derived versions have execution times that are almost identical. For the large matrix examples the derived implementation is marginally slower than the time for the hand crafted version (between 0.1% and 0.6%). This discrepancy arises from a minor optimization made possible by the particular way the hand-crafted version produces the $transform$ matrix. We could write a transformation that could perform this optimization, but it appears to be too special-purpose to be of general use.

Matrix Size	*Time per iteration (sec)*		
	Hand Crafted Fortran Plus	Hand Crafted Fortran Plus Enhanced	Automatically Derived Fortran Plus Enhanced
64	1.2	1.35	1.35
128	8.23	9.30	9.31
256	60.92	69.86	70.30

Table 1: Hand-Crafted POT versus Mechnically Derived POT

[3]The time is constant for generation of each approximation

7 Conclusion

We have shown that it is possible mechanically to produce a highly efficient implementation tailored for execution on the AMT DAP 510 of a high-level functional specification. The functional specification is not biased in ways that would permit its efficient execution on a particular machine architecture, but is expressed in a way that gives a clear statement of the algorithm. Indeed, the functional specification may be used as the starting point for producing implementations tailored for execution on other machines (and will be used in this way in future investigations).

The transformations used to produce the implementation discussed in this paper are not particular to this problem and are currently being applied in the transformation of functional specification for other algorithms where an implementation tailored for the DAP processor is required. However, there is still development work to be undertaken for these derivations. This includes tailoring the generated code for the compiler (for example, producing sectioned array operations) and tailoring for particular data sets (for example, sparse matrices or banded matrices).

References

[1] *Fortran-Plus Language*, AMT, man 00202, 1988.

[2] *Fortran-Plus Language Enhanced*, AMT, man 102.01, 1988.

[3] *A Transformational Component for Programming Language Grammar*, J. M. Boyle, ANL-7690 Argonne National Laboratory, July 1970, Argonne, Illinois.

[4] *Abstract programming and program transformations - An approach to reusing programs.* James M. Boyle, Editors Ted J. Biggerstaff and Alan J. Perlis in Software Reusability, Volume I, Pages 361-413, ACM Press (Addison-Wesley Publishing Company), New York, NY, 1989

[5] *Deriving efficient programs for the AMT DAP 510 using Program transformation*, J.M. Boyle, M. Clint, S. Fitzpatrick and T.J. Harmer, QUB Techical Report, June 1992.

[6] *Towards the construction of an eigenvalue engine*, Clint M. et al, Parallel Computing, 8, 127-132, 1988.

[7] *A Comparison of two Parallel Algorithms for the Symmetric Eigenproblem*, Clint M. et al, Intern'l Journal of Computer mathematics, 15, 291-302, 1984.

[8] *Fortran-Plus v. Fortran-Plus Enhanced: A comparison for an Application in Linear Algebra*, M. Clint et al, QUB Technical Report, 1991 (submitted for publication).

[9] *Functional Programming using Standard ML*, Wilstöm, A, Prentice Hall, London 1987.

[10] *The parallel computation of eigenvalues and eigenvectors of large hermitian marices using the AMT DAP 510*, Weston J. et al, Concurrency Practice and Experience, Vol 3(3), 179-185, June 1991.

[11] *Two algorithms for the parallel computation of eigenvalues and eigenvectors of large symmetric matrices using the ICL DAP*, Weston J., Clint M., Parallel Computing, 13, 281-288, 1990.