Deriving DAP Implementations of Numerical Mathematical Software through Automated Program Transformation[1]

James M. Boyle†[2], Maurice Clint, Stephen Fitzpatrick[3], Terence J. Harmer

boyle@mcs.anl.gov, {M.Clint, S.Fitzpatrick, T.Harmer}@cs.qub.ac.uk

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

†Mathematics and Computer Science Division,
Argonne National Laboratory,
Argonne IL 60439, USA.

July 1992

Technical Report 1992/Jul-JMB.MC.SF.TJH, Computer Science, QUB, July 1992.
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1992/Jul-JMB.MC.SF.TJH/>
<URL:ftp://ftp.cs.qub.ac.uk/pub/TechReports/1992/Jul-JMB.MC.SF.TJH.ps.gz>

1 Introduction

At first sight, functional programming languages seem to be not at all appropriate for expressing solutions to numerical mathematical problems. On the one hand, there is a long-established tradition of expressing such algorithms in procedural languages; in fact, the earliest procedural languages, Fortran and Algol 60, were specifically designed with numeric computations in mind. On the other hand, functional languages appear to be ill-suited to the specification of computations that operate on data represented by arrays - a data structure that is the dominant one in mathematical software.

If one sets aside these preconceptions, however, closer examination reveals that functional languages do have advantages for describing numerical computations. The general advantages of functional programming - naturalness of expression (in some problem areas), ease of proof, and clarity - are well known. When one uses functional programming in a numerical mathematical context, one finds that many numerical computations are more naturally expressed by using recursion than by using the iterative constructs which are often employed when using procedural languages.

Nevertheless, these desirable features of functional specifications for numerical problems count for nothing if the specified computations cannot be executed efficiently. The very problems and algorithms for which the advantages of functional specification are most important are those that require prodigious amounts of computation and for which even relatively minor inefficiencies cannot be tolerated.

We have been investigating the generation of efficient programs from functional specifications for a range of numerical algorithms. We have succeeded in using transformations to derive automatically a vectorizable implementation of a hyperbolic PDE algorithm whose performance is more efficient than that of the handwritten program on the CRAY X-MP [8].

We consider a functional specifications as just that - an *abstract specification* of how to solve a problem, *not* a *concrete program* to be run in order to compute a solution. By adopting this point of view, we liberate the style of the functional specification from any demands for efficient execution. We can write the specification in the style and with the techniques that produce the greatest degree of clarity, guarantee correctness and facilitate adaptability. We then address the question of efficiency during the process of creating an executable concrete implementation by automated program transformation.

[1] A simplified version of this report was presented at CONPAR 92-VAPP V: see 'Proceedings of CONPAR VAPP V' pp 761-767, Springer-Verlag, Lecture Notes in Computer Science 634, September 1992; L. Bouge, M. Cosnard, Y. Robert and D. Trystram (Ed.)

[2] 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.

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

In this paper we describe recent work on deriving implementations tailored for execution on the AMT DAP computer. We apply the transformation methods in the derivation of an efficient implementation of an eigenvalue algorithm.

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) [12]. 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.

2.1 Primitive Vector and Matrix Functions

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 the discussion below we give definitions of the vector functions; each has a natural analogue for matrix arguments.

2.1.1 $Generate$

Definition 1 $generate : Vector(\beta, N) \times ([0 \ldots N-1] \to \alpha) \to Vector(\alpha, N)$
where $Vector(\alpha, N)$ *is a vector of* N *elements of type* α

The $generate$ function is applied to two arguments: a vector (referred to as the base vector) and a function (referred to as the generating function) and produces a result vector. The result vector is of the same size as the base vector and each of its elements is obtained by applying the generating function to the index for that element.

2.1.2 Map

Definition 2 $map : Vector(\alpha, N) \times (\alpha \to \beta) \to Vector(\beta, N)$
$map(V, \lambda x \cdot B) \stackrel{\text{def}}{=} generate(V, \lambda k \cdot (\lambda x.B)(element(V, k))$

For many vector expressions the evaluation of the generating function for a particular index requires only the k^{th} element of the base vector. (Such expressions have a generating function that can be written in the form $\lambda k \cdot ((\lambda x \cdot B)(element(V, k)))$, where the body B of the inner λ abstraction has no occurrences of the $element$ function.) Such expressions occur frequently enough to justify the definition of a simpler form of $generate$ called map.

2.1.3 $Reduce$

Definition 3 $reduce : Vector(\beta, N) \times ([0 \ldots N-1] \to \alpha) \times (\alpha \times \alpha \to \alpha) \times \alpha \to \alpha$

A $reduce$ function combines the elements of a vector by means of a binary function to produce a single value. Common examples of reductions are calculation of the sum of the elements of a matrix of numbers and conjunction of the elements in a boolean matrix.

The $reduce$ function is applied to a base vector, a generating function, a (binary) reducing function and an initial value. The generating function is applied to each element in the vector and the values returned by the generating function are combined by the reducing function to form the final result.

We define a simpler form of $reduce$ called $fold$ that applies a function to the elements of a vector in the traditional manner. The function $fold$ has the same relation to $reduce$ as does map to $generate$; that is, it is a shorthand notation for a frequently used operation.

3 The Parallel Orthogonal Transformation(POT) Algorithm

The eigenvalues Λ and eigenvectors Q of a matrix A of order n satisfy the equation: $A.Q = Q.\Lambda$, where Λ is a diagonal matrix with the eigenvalues $\lambda_1, \ldots, \lambda_n$ of A as its diagonal elements. If A is symmetric Q is guaranteed to be non-singular and is, in addition, orthogonal.

The POT algorithm is based on the construction of a sequence of similar 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 $\lim_{k\to\infty} B_k = \Lambda$ [10]. The function $ortho$ orthonormalizes the columns of its non-singular matrix argument using the standard Gram-Schmidt method.

The POT algorithm may be realised by a Standard ML function of the form:

```
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;
```

This definition follows directly from the description of POT given above: if $B_k = U_k^T.A.U_k$ is sufficiently diagonal then the columns of U_k and the diagonal elements of B_k are the eigenvalues; otherwise a more accurate approximation to Q is derived and POT is applied using this new approximation.

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 as follows:

$$t_{ij} = \frac{2b_{ij}}{d_{ij} + sign(d_{ij})\sqrt{d_{ij}^2 + 4(b_{ij})^2}} \qquad \begin{array}{c} where\ i > j and \\ let\ d_{ij} = b_{jj} - b_{ii} \end{array} \qquad t_{ii} = 1, t_{ij} = -t_{ji}$$

From this definition the SML function $Transform$ may be coded thus:

```
fun Transform ( M : real Array) : real Array =
    let fun Calculate (M: real Array, i: int, j: int) : real =
            let val d = element(M, j, j) - element(M, i, i)
            in
             2*element(M,i,j)/(d+sign(d)*sqrt(sqr(d)+4*sqr(element(M,i,j))))
```

```
        end
in
  generate(fn(i, j)=>
              if ( i > j ) then Calculate(M, i, j)
              else if i = j then 1.0
              else ~Calculate(M, j, i), M)
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 The AMT DAP 510

The DAP 510 has a two-dimensional (32×32) array of processors which operate in a single instruction, multiple datastream mode. The instructions for the processor array are supplied by a scalar processor. The data for each processor can emanate from:

- The scalar processor which may broadcast scalar data;
- The local memories of the processors of the array.
- Hardware channels that permit nearest neighbour communication and the rapid duplication of a vector of values in the rows or columns of the array.

4.1 DAP Fortran

The language in which we express our final implementation is DAP Fortran, an extension to standard Fortran that allows the array processor to be used efficiently. DAP Fortran supports two types, scalar vector and scalar matrix, which are similar to one dimensional and two dimensional array, respectively, in standard Fortran, but supplies additional operations that use the processor array.

Each element of a vector or matrix is manipulated by its own processor, but corresponding components of different vectors and different matrices share a processor. The size of vectors or matrices is limited only by the the amount of memory available, not by the size of the processor array. DAP Fortran subdivides a vector or matrix whose dimensions are larger than those of the processor array into sections each of which is the size of the array (if necessary, it pads the edges of the vector or matrix to make the size a multiple of the processor array size).

The features of DAP Fortran that are important to us are:

- *Componental functions* - scalar functions applied to each element of a vector/matrix and scalar functions applied to corresponding elements of a pair of vector/matrices; these include simple arithmetic and logical operations. Applications of componental functions are evaluated in constant time.
- *Aggregate functions* which perform certain elementwise reductions on a vector/matrix. Such reductions can be performed in $O(log(N))$ steps.
- Vector/matrix assignment.
- Vector/matrix assignment controlled by a boolean vector/matrix, or mask. The assignment affects only those elements of the variable for which the corresponding element of the mask is $true$.
- Functions to create vector/matrix masks having $true$ elements arranged in certain commonly used patterns.

- Geometric functions to re-arrange the order of elements in a vector/matrix; for example, functions to form the transpose of a matrix or to reverse the order of elements in a vector.
- Extraction functions that return a vector with elements equal to the elements of a given row or column of a matrix.
- Masked extraction functions.

To runn efficiently on the DAP, a program must be expressed almost entirely in terms of these features.

5 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 [9, 13, 14, 10]. Efficient ML implementations are not likely to be available for such computers. Naturally, the question to be posed is: how can we obtain, from functional specifications, programs that execute efficiently and exploit high-performance computers? Our answer is to use program transformations.

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

5.1 Sketch of the Transformational Derivation

The derivation of an efficient implementation for the AMT DAP 510 consists of about 12 major transformational steps. Each of these steps belongs to one of three types: domain-dependent transformations that apply to *matrix/vector problems* (marked with a single bullet, in the list below); hardware-specific transformations that direct the derivation toward code that is efficient on the DAP (marked with two bullets, below); and general purpose functional-to-procedural transformations, such as might be incorporated in a compiler for functional languages (marked by +, below).

The major steps in the derivation are as follows:

+ Translating the ML specification into a canonical form(standardization)
+ Unfolding abstract data types and non-recursive function definitions and simplifying
+ Reducing the complexity of storage usage (implementing reuse where possible)
•• Converting expressions into applications of array processor functions
+ Performing common subexpression removal
•• Converting applications of array processor functions that have componental equivalents in DAP Fortran into the DAP Fortran versions
+ Preparing the ML for transformation to Fortran
+ Eliminating tail recursion
+ Transforming the prepared ML to structured, recursive Fortran
•• Implementing the data-parallel conditional

+ Transforming recursive Fortran to non-recursive Fortran

•• Eliminating common logical subexpressions

Each TAMPR transformation rule is literally a rewrite rule, having a pattern and a replacement each of which is specified in terms of the grammar of the programming language. Typically, fewer than ten of these transformation rules are required to implement each of the major steps in the derivation. However, these rules are applied many times; for this specification, the entire derivation from ML to DAP Fortran requires about 10000 rewrites. Clearly, it is vital that TAMPR provides the ability to apply rules *automatically*. No one could afford to apply thousands of transformations by hand, or even to guide their application.

A somewhat more detailed discussion of the basic transformation steps, including some example code fragments generated at various stages, is given in [4] (see also an earlier version in [5]).

5.2 DAP-Specific Transformations

Earlier we defined two primitive vector/matrix functions: $generate$ and $reduce$. A $generate$ function produces a vector/matrix the value of each element of which is specified by a function argument of $generate$. A $reduce$ function produces a scalar value that is an aggregate of its vector/matrix argument. A function argument of $reduce$ specifies the combining expression to be used when computing the aggregate value.

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 $generate$; $generate(\lambda((i,j).M(i,j), M) \Longleftrightarrow M$.
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., $generate(\lambda((i,j)2), M) \longrightarrow expand(2, n, n)$, where matrix M is of order $n \times n$.
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., $generate(\lambda((i,j)A(i,j) + A(i,j)), A) \longrightarrow$ $generate(\lambda((i,j)A(i,j), A)) + generate(\lambda((i,j)A(i,j)), A)$
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 following 6 simplifications/optimizations:

1. A $join$ with duplicate computation is optimized to remove the duplicated evaluation. For example, a $join$ may create a symmetric or anti-symmetric matrix. Thus, only the values in the lower (upper) triangle of the matrix need be calculated, with the values in the upper(lower) triangle of the matrix being *copied* from the lower (upper) triangle of the matrix.
2. A $generate$ the effect of which is a predefined rearrangement of the elements of a vector/matrix is replaced by an application of the standard rearrangement function; e.g. creating the transpose of a matrix.
3. A $generate$ that is a predefined vector part of a matrix is replaced by an application of a standard vector extraction function; e.g. column of a matrix.
4. A $generate$ function the result of which is a boolean vector/matrix with $true$ values in a predefined arrangement is replaced by an application of that standard arrangement function; e.g. the value $true$ in the diagonal elements of a matrix.
5. A vector $reduce$ is moved outside of the scope of a matrix $generate$ (this is possible only if the $reduce$ function has a componental equivalent).
6. Expressions that are independent of $generate/reduce$ indices are moved outside the scope of the generation/reduction.

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. They, therefore, define (partially) an *abstract* array processor architecture and the program that is the result of applying these transformation rules is refined further (by transformation) to produce a program that executes (efficiently) on a particular array processor.

5.3 Deriving an Implementation for the $Transform$ Function of POT

To illustrate the use of the above transformations, we consider, in detail, the $transform$ function from POT illustrating the translation from SML specification to code tailored for execution on the AMT DAP.

After unfolding and simplifying $transform$ has the form:

```
generate (
 lambda ( ( i,  j )
      if  i > j then 2 * element(M, i, j) ....
      else if i = j then 1
      else if true ~(2 * element(M, i, j) ....)
 , M)
```

The $generate$ is anti-symmetric, so this annotated as an $antisymmetric$ expression (Optimization 1):

```
antisymmetric (
 generate (
  lambda ( ( i,  j ),
           if   i = j then 1
   else if true then negative ( quotient ( ... ) )
         ), M)
```

(Note: this implies that the $generate$ function is free to *compute* the elements of the upper triangle; however, later transformations remove this apparent inefficiency)

By algebra rule 4 (propagation through conditional expressions) this becomes:

```
antisymmetric (
 join (
  if generate (lambda ( ( i, j ), i = j ), M ) then
         generate (lambda ( ( i, j ) 1 ), M )
  else if generate (lambda((i,j) true), M) then
         generate (lambda ( ( i, j ), ~ ( 2 * .... )), M) ))
```

The first predicate of the $join$ corresponds with one of the pattern functions, the unit diagonal pattern; so simplification rule 4 applies. The $generate$s in the first result expression and in the second conditional guard are independent of both generating indices, so algebra rule 2 applies. Thus we get:

```
antisymmetric (
  join (
    if  patdiag ( n ) expand ( 1, n, n )
    else if ( expand ( true, n, n ) ) then
                generate (lambda ( ( i, j ),
                           ~ ( 2 * ... ) ) )))
```

where $expand(a, b, c)$ creates a matrix of size $b \times c$ with each component having value a. The õperator has a componental equivalent, so algebra rule 3 applies.

```
   negative ( generate ( lambda ( ( i, j ), 2 * .... ) ) )
```

The $generate$ is propagated into the expression until the remaining $generate$s have bodies whose forms are either $element(m, i, i)$, $element(m, j, j)$ or $element(matrix, m, j, i)$. The first two of these correspond to one of the extraction functions: $diagonal$ (rule 5). The third corresponds to the $transpose$ function (rule 4).

```
antisymmetric (
 join(
  if patdiag(n) then expand(matrix,1,n,n))
  else if (expand(true,n ,n)) then
   negative (
    el_wise-quotient (
      el_wise-times (
   expand(2,n,n),b),
         el_wise-plus (
               el_wise-difference(expand(rows,diagonal(b),n),
                                  expand(cols,diagonal(b),n)),
               el_wise-times(
                    join(
                         if el_wise-less(
                              el_wise-difference(expand(rows,diagonal(b),n),
                                                 expand(cols,diagonal(b),n)),
                              expand(matrix,0,n,n))
                         then expand(- 1 ,n,n)
                         else expand(matrix,1,n,n)),
                    el_wise-sqrt(
                         el_wise-plus (
          el_wise-times(
                                   el_wise-difference(
                                        expand(rows,diagonal(b),n),
                                        expand(cols,diagonal(b),n)),
                                   el_wise-difference(
                                        expand(rows,diagonal(b),n),
                                        expand(cols,diagonal(b),n))),
                              el_wise-times (expand(matrix,4,n,n),
                                             el_wise-times (b,b))
                    )))))))
```

Where el_wise-'op' is the named operation, 'op', applied element-wise to matrix arguments. For example, $el\_wise-times$ is the arithmetic product operation applied between all corresponding pairs of elements of two matrices (- it is not standard matrix multiplication).

After common sub-expression elimination is perfromed the conversion of this DAP independent form to DAP Fortran may be undertaken. Those functions which have equivalent functions (or combinations of functions) in DAP Fortran are converted into their DAP Fortran forms. The program may then be processed by the standard functional language compiler transformations to produce the following code:

```
  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 .eq. 0 ) = 0
  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 )
```

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

In comparing the execution times for programs that execute on the AMT DAP it is necessary to distinguish two distinct Fortran Dialects in which a program may be written; namely Fortran-Plus [1] and Fortran-Plus Enhanced[2]. A *Fortran-Plus* program must express array operations in a size that matches the size of the processor array in use. Therefore, for the DAP used in these experiments, array operations are expressed as operations on 32×32 array. A *Fortran-plus Enhanced* program need not take account of the processor array size when an vector/matrix operation is used. The Fortran-Plus Enhanced language compiler will produce an implementation that expresses operations on arrays that are larger than the processor array size as a sequence of operations on processor array size sections of the array; hopefully, the compiler can do this with a degree of efficiency. It is possible to use a mixture of Fortran-Plus Enhanced features and the more primitive Fortran Plus language features in a program. In the discussion below: a fortran-Plus program is one that always expresses array operations in terms of the size of the processor array; and a Fortran-Plus Enhanced program does not take account of the processor array size in array operations.

In Table 1 the execution time (for each iteration) [4] 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 [11, 14]. 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 [11] the implementation produced by the Fortran-Plus Enhanced compiler for frequently occurring linear algebra operations (e.g. matrix product) is very efficient but 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. In (very) large matrix examples the derived implementation is marginally slower than time for the hand crafted version (about).

7 Conclusion

We have shown that it is possible automatically to produce a highly efficient implementation of a high-level functional specification tailored for execution on the AMT DAP 510. The functional specification is not baised in ways that would permit its efficient execution on a particular machine architecture, but is written

[4] The iteration time is constant for all iterations

in a way that gives a clear statement of the algorithm. Indeed, the functional specification of POT 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 research).

The transformations used in producing the implementation discussed herein are not particular to this problem and are being applied to functional specification for other algorithms where an implementation tailored for the DAP is required. However, there is still development work to be undertaken for this derivation: including tailoring the generated code for the compiler (such as producing sectioned array operations) and tailoring for particular data sets (such as 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] *Program reusability through program transformation*, J. M. Boyle and M. N. Muralidharan, IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, Sept. 1984, Pages 574-588.

[6] *Program adaptation and program transformation*, J. M. Boyle in Practice in Software Adaptation and Maintenance, Editor R. Ebert, J. Lueger, and L. Goecke, North-Holland Publishing Co., Amsterdam 1980, Pages 3-20.

[7] *Functional Specifications for Mathematical Computations*, Boyle, J. M. and Harmer, T. J., Constructing Programs from Specifications, Proceedings of the IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May, 1991, B. Möller, Ed., North-Holland, Amsterdam, 1991, pp. 205-224

[8] *A Practical Functional Program for the CRAY X-MP*, James M. Boyle and Terence J. Harmer, Preprint MCS-P159-0690, Argonne National Laboratory, Argonne, Illinois, July 1990, (To appear in *Journal of Functional Programming*.)

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

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

[11] *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).

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

[13] *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.

[14] *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.

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	

Table 1: Hand Crafted POT versus Automatically Derived POT