A Family of Intermediate Forms[1]

M. Clint, Stephen Fitzpatrick, T.J. Harmer, P. Kilpatrick, J.M. Boyle[†]

boyle@mcs.anl.gov, {M.Clint, S.Fitzpatrick, P.Kilpatrick, 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.

November 1993

Technical Report 1993/Nov-MC.SF.TJH.PLK.JMB, Computer Science, QUB, November 1993
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1993/Nov-MC.SF.TJH.PLK.JMB/>
<URL:ftp://ftp.cs.qub.ac.uk/pub/TechReports/1993/Nov-MC.SF.TJH.PLK.JMB.ps.gz>

Keywords: program transformation, program derivation, intermediate forms, canonical forms, functional specification

1 Introduction

A good programmer attempts to minimize architecture specific detail when writing a sequential implementation of an algorithm. Such good programming practice makes it possible to transport the implementation to other hardware architectures and thus minimize programmer effort. Indeed, high-level programming languages attempt to hide the detail required by particular machine architectures and thus make it easier to construct programs and transport them.

When using traditional methods to implement an algorithm for an advanced parallel architecture, the programmer faces the dilemma of

- writing the algorithm implementation in an architecture independent way and thereby, unfortunately, achieving disappointing execution performance (when compared to the architecture's theoretical execution performance); or
- writing the algorithm implementation in a way that exploits the capabilities of the hardware architecture and achieving good execution performance, but in the process producing an implementation dedicated to a particular parallel architecture.

With the scientific community's insatiable desire for performance, a programmer will generally choose the latter course. However, the increasing variety of parallel computer architectures and the speed of technological change make this course an expensive one, since it requires a new implementation to be prepared when a new parallel architecture is considered.

The problem that we address in this paper is how to enable a programmer to obtain high performance from an implementation without having to write a low-level, machine specific implementation. The approach that we use is to write high-level, machine independent specifications of algorithms in a pure, functional programming language.

Such *functional specifications* can be executed, and indeed often are executed in the initial stages of development to give confidence that an algorithm has been correctly described. However, a functional specification of an algorithm is intended as a clear statement of the algorithm, that captures the essence of

[1]This work is supported by SERC Grant GR/G 57970, by a research studentship from the Department of Education for Northern Ireland and by the Applied mathematical sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38

A simplified version of this report was presented at HPCN '94: see 'Proceedings of High Performance Computing and Networking', Volume II pp 457-462, Springer-Verlag, Lecture Notes in Computer Science 797, April 1994; Wolfgang Gentzsch and Uwe Harms (Ed.).

the algorithm without consideration for execution efficiency. As such, a specification executes prohibitively slowly using available functional language compilers, much more slowly than could be expected of a hand-crafted, imperative implementation of the algorithm.

So, rather than compile a functional specification, we *derive* imperative implementations of the specification, normally expressed in some dialect of Fortran. Many implementations may be derived from a single specification; each implementation is tailored for the hardware to be used and the data that are to be manipulated.

The derivation of an implementation from a specification is performed by automatically applying *program transformations*. Each transformation is a simple rewrite rule that produces a change in a program; normally, a single application of a transformation produces a minor, local change; a derivation effects the implementation of a functional specification by applying many transformations, each applied many times. We have demonstrated that it is possible to derive highly efficient implementations from our functional specifications; indeed, these implementations are comparable in performance to implementations written by a programmer using a conventional approach.

2 Functional Specifications

The language we use for functional specifications is (a small subset of) the SML functional language [7]. The important features of this language are:

- algorithms are denoted as pure expressions; executing an algorithm is performed by evaluating expressions;
- modularity is supported through functional decomposition and a *structure* mechanism for defining abstract data types;
- functions and operators can be overloaded to allow vector/matrix operations to be expressed in a manner similar to standard mathematics.

We use a library of vector/matrix operations that commonly occur in numerical/scientific algorithms. These library functions are defined in terms of a small number of *primitive* functions which we discuss below.

`element:`α `array` $\times$ `index` $\rightarrow$ α

The function `element` is applied to an array of arbitrary dimensions (vectors and matrices being particular examples) and returns the value of the element stored at the location specified by the `index`. For convenience, various notations can be used for `element`. For example, the i^{th} element of a vector V can be denoted as `V@[i]` — read as ‘V at i’ — and the $(i,j)^{th}$ element of a matrix A as `A@[i,j]`.

`generate:shape` $\times$ `(index` $\rightarrow$ α`)` $\rightarrow$ α `array`

An application of the `generate` function evaluates to an array. The set of indices over which the array is defined is specified as a `shape`. The values of its elements are defined by a *generating function* which takes an `index` as argument: the value of an element at a particular location is given by applying the generating function to the `index` for that location.

As with `element`, we allow several forms of `generate` that are convenient for vector and matrix operations. For example, the vector of length n and with element i equal to i can be defined as

```
generate(n,fn(i:int)=>i)
```

where the expression `fn(i)=>expression` defines a function over i. The transpose of a matrix A of dimensions $m\times n$ can be defined as

```
generate([n,m],fn(i:int,j:int)=>A@[j,i]).
```

`reduce:shape` $\times$ `(index` $\rightarrow$ α`)` $\times$ `(`α $\times$ α $\rightarrow$ α`)` $\times$ α $\rightarrow$ α

The `reduce` function is used to combine a set of values into a single value by the repeated application of some binary function. For example, a function to sum the elements of a vector V can be defined as

```
fun sum(V:real vector):real = reduce(shape(V),fn(i)=>V@[i],+,0.0).
```

Here `reduce` adds the values of `fn(i)=>V@[i]` for all indices in `shape(V)`, i.e. it is equivalent to $V[1] + V[2] + \cdots$. (An identity element, `0.0`, is specified to make the definition of reduce valid even when the set of values contains only one element).

These functions can be used to define the elementwise multiplication of two vectors:

```
fun times(U:real vector, V:real vector):real vector
   = generate(shape(U),fn(i:int)=>U@[i]*V@[i]).
```

This can then be combined with the summation function above to obtain more complex matrix and vector operations:

```
fun innerproduct(U:real vector, V:real vector):real
   = sum(times(U,V))

fun mvmult(A:real matrix, V:real vector):real vector
  = generate(size(A,0),fn(i:int)=>innerproduct(row(A,i),V)).
```

We believe that functional definitions should be high-level: they should express operations in the most natural way, with no consideration given to efficiency or implementation concerns such as vectorization. However, functional definitions are algorithmic and can be executed for the purpose of rapid prototyping. We provide library definitions for a collection of useful functional specification primitives. When textually included with a specification, the elements of this library provide a complete executable definition of a computation.

It is our experience that a competent mathematician can be taught to write functional specifications, in the style discussed above, in a few hours. Indeed, our style of algorithm specification is close to that used by mathematicians when describing algorithms in text books. Interestingly, the difficulty many have in initially learning to produce good specifications is in trying to ignore the efficiency considerations that are important when writing traditional Fortran implementations of algorithms.

3 A Family of Transformational Derivations

3.1 Transformations

A transformation is a rewrite rule, consisting of a pattern and a replacement defined using a wide-spectrum grammar. The transformations used for the work reported here are applied by the TAMPR transformation system [1],[3]. When a transformation is applied to a program, all sections of the program that match the pattern are replaced by the replacement.

For example, consider the transformation

```
<term>"1"+<term>"1" ==> 2*<term>"1"
```

(the non-terminal symbol `<term>` may informally be thought of as an expression):

- the pattern is `<term>"1"+<term>"1"`;
- the replacement is `2*<term>"1"`;
- the label `"1"` on the non-terminals in the pattern indicates that each `<term>` must match the same expression for the pattern as a whole to be a match;

- the `<term>"1"` in the replacement stands for whatever is matched by `<term>"1"` in the pattern.

This transformation would convert the expression `x+x` into `2*x`, and `(1/2)+(1/2)` into `2*(1/2)`; but it would not convert the expression `x+y`, since `x` and `y` are different expressions.

Many of the transformations used in practice are no more complex than the above example; the power of transformations comes from the cumulative effect of perhaps thousands of applications of dozens of transformations, each application performing some simple, local change to a program section. Since the application is performed entirely automatically, the number of applications is of little significance to the program developer.

What is significant is the simplicity of each transformation; usually it is trivial to see that an application of a transformation preserves the meaning of a program. (As transformations are formal, formal proofs can be used to guarantee the correctness of more complex transformations.) And if each application preserves meaning, then their combined application also preserves meaning. Thus, an implementation is known to be equivalent to the specification from which it was derived.

Often, several transformations are required (or are convenient) to perform some change to a program, so transformations can be combined into transformation sets; the application of a transformation set is the exhaustive application of all the transformations in the set. A TAMPR derivation is a sequence of such transformation sets. Each set is applied in turn to a program.

3.2 A Family of Data Parallel Derivations

The implementation of a functional specification is a complex task; its complexity may be reduced by identifying *intermediate forms* between the specification language and the final implementation language. For example, rather than make the transition directly from SML to Fortran, a specification may first be converted into the λ-calculus, then into Fortran:

$$\text{SML} \longrightarrow \lambda\text{-calculus} \longrightarrow \text{Fortran.}$$

Each of these transitions may also be sub-divided into further intermediate forms, to whatever level is convenient.

A derivation to implement a specification is correspondingly divided into *sub-derivations*; one sub-derivation being used to create each intermediate form.

There are advantages to such division beyond simplifying the task of implementation: the sub-derivation that creates the λ-calculus form is independent of the final implementation form, and so may be combined with another sub-derivation to create, say, an array processor implementation (for example, for the AMT DAP). Similarly, the λ-calculus to Fortran sub-derivation is independent of how the λ-calculus form was created, and may be combined with another sub-derivation that converts another specification language into the λ-calculus.

$$\left.\begin{array}{c}\text{SML}\\ \text{Lisp}\\ \text{Miranda}\end{array}\right\} \longrightarrow \lambda\text{-calculus} \longrightarrow \left\{\begin{array}{c}\text{Fortran}\\ \text{DAP Fortran}\end{array}\right.$$

Further, sub-derivations can be added to optimize implementations by performing, for example, function unfolding or common sub-expression elimination. Other sub-derivations can be added to tailor an implementation when data sets are known to have particular properties, such as a matrix being sparse. Figure 1 is a (somewhat simplified) illustration of the relationships among various intermediate forms created by such sub-derivations.

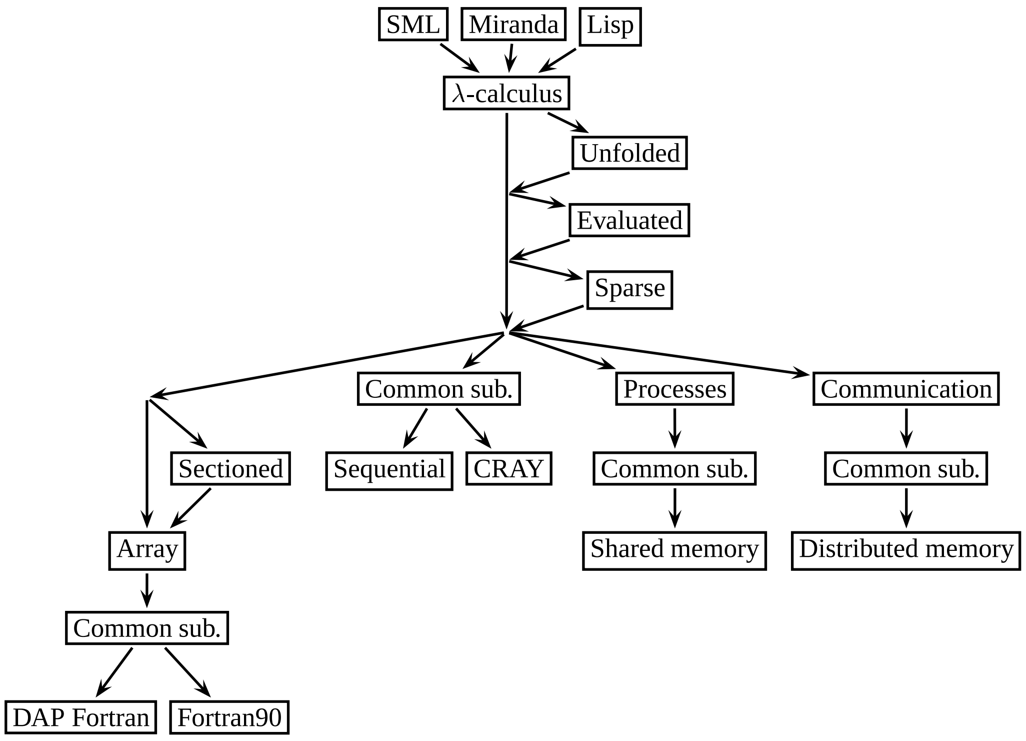

Figure 1: A family of derivations

4 Example

As an example of the various forms a specification may take during the implementation development process, we will consider the derivation from a functional specification of a simple, computationally efficient, functional form for matrix-vector multiplication. We will then consider the implementation of this form for a sequential processor, for a CRAY vector processor and for the AMT DAP array processor. We will then consider the derivation of a form optimized for a tridiagonal matrix, and the implementation of this sparse form for the same three architectures.

4.1 Specification of Matrix-Vector Multiplication

The standard ‘row into column’ definition of matrix-vector multiplication and some of the functions used in the definition are shown in figure 2.

```
fun times(U:real vector, V:real vector):real vector
   = generate(size(U), fn(i:int) => U@[i]*V@[i])
fun sum(U:real vector):real
   = fold(U, +, 0.0)
fun innerproduct(U:real vector, V:real vector):real
   = sum(times(U,V))
fun mvmult(A:real matrix, V:real vector):real vector
   = generate(size(A,0), fn(i:int) => innerproduct(row(A,i), V))
```

Figure 2: SML specification of matrix-vector multiplication

For this example, we assume that matrix-vector multiplication is used in an ‘algorithm’

```
fun init(n:int)
   = let
      val A:real matrix = read_matrix([n,n]);
      val V:real vector = read_vector([n])
   in
      A*V
   end
```

which we shall refer to as MVM, and shall consider the implementation of MVM.

4.2 Initial Stages

The initial stages of the derivation are applied to a specification to produce a computationally efficient form in a standard, simple notation.

4.2.1 λ-Calculus

The λ-calculus that we use is a simple extension of the standard λ-calculus [9], in which named function definitions are allowed. The most noticeable differences from SML are:

- function definitions have the form `name = open λ-expression`;
- function prototypes have been converted into open λ-expressions;
- local values have been converted into closed λ-expressions;
- operators have been converted into prefix functions (overloaded operators such as `*` have generic names such as `op.times`).

The λ-calculus form of MVM is shown in figure 3.

```
fun times
   = λU:real vector, V:real vector·
      generate(size(U),
         λi:int·op.times(element(U,[i]),element(V,[i])))
...
fun init
   = λn:int·
      λA:real matrix·λV:real vector·
         op.times(A,V)
      (read_vector([n]))
      (read_matrix([n,n]))
```

Figure 3: MVM: λ-calculus

4.2.2 Unfolded

Functions are unfolded: that is, an application of a function is replaced by a copy of the function's definition, with formal and actual arguments bound (this requires function overloading to be resolved using type information). For MVM, the entire specification collapses into a single function definition.

In addition, α-conversion and currying are performed. See figure 4. (From this point on, we do not show type information, though it is still present in programs.)

```
fun init
   = λn·
     λA·λV·
       generate (size(A) (0))(λi·
         λU1·
           λU2·
             reduce (size(U2))(λi2·element (U2)([i2]))(real.plus)(0.0)
           (generate (size(U1))(λi1·
             real.times (element (U1)([i1]))(element (V) ([i1])))
           )
         (generate (size(A) (1))(λj·element (A)([i,j])))
     (read_vector([n]))
     (read_matrix([n,n]))
```

Figure 4: MVM: unfolded

4.2.3 Statically Evaluated

Consider the unfolded form from an operational or computational perspective; that is, as instructions for calculating a value. The outermost generation specifies that, to calculate element i of the product:

- calculate `U1`, row i of A;
- calculate `U2`, the elementwise product of the row and V;
- sum the elements of `U2`.

Thus, the calculation of a single element of the product requires the construction of two intermediate vectors (`U1` and `U2`). However, it is possible to calculate the element without forming these intermediate vectors:

- there is no need for `U2`, as the elements of `U1` and `V` can be multiplied together as the sum is being formed;
- there is no need for `U1`, as the elements of a row of `A` can be accessed *in situ*.

The calculation of the product could thus be expressed as shown in figure 5. This form has a more efficient computational interpretation. It is produced from the unfolded form using the techniques of *static evaluation*, in which transformations evaluate expressions.

```
fun init
   = λn·
     λA·λV·
       generate ([n])
         (λi·reduce ([n])
           (λj·real.times (element (A) ([i,j])) (element (V) ([j])))
           (real.plus) (0.0)
         )
     (read_vector([n]))
     (read_matrix([n,n]))
```

Figure 5: MVM: statically evaluated

In the MVM example, the important evaluations are:

- The β-reduction of `U1` and `U2`.
- These β-reductions result in expressions of the form:

  `element (generate (size(A) (1))(λj·element (A)([i,j]))) ([i1])`

  which can be evaluated using an identity of `generate` to

```
element(A) ([i,i1]).
```

4.3 Dense Implementations

Having derived the statically evaluated form, sub-derivations can be applied to produce implementations for a variety of architectures. These implementations are discussed below.

4.3.1 Sequential

Implementing the evaluated form of MVM is straightforward, requiring only the creation of two loops to implement the generation and reduction. Normally the implementation is more complex; see [2] for a detailed discussion.

The Fortran77 [10] version of MVM is shown in figure 6.

```
      program init
      parameter(n)
      integer n
      real A(n,n)
      real V(n)
      real init(n)

      read_matrix(A)
      read_vector(V)
      DO 20 i=1,n,1
      init(i) = 0.0
      DO 10 j=1,n,1
      init(i) = init(i)+A(i,j)*V(j)
20    continue
10    continue
      end
```

Figure 6: MVM:Sequential and CRAY form

4.3.2 CRAY

The CRAY range of vector computers is programmed using Fortran77; the compiler converts certain types of loops into instructions for the hardware vector units. Thus, the CRAY implementation is similar to the sequential form, but with some tailoring performed. For example:

- Some versions of the CRAY Fortran compiler cannot vectorize a loop if it contains more than a certain number of conditional statements. Transformations can reorganize loop structures to assist the compiler.
- The CRAY software libraries contain highly optimized implementations of some commonly used matrix and vector operations; for example, the Basic Linear Algebra Subroutines (BLAS). Even a Fortran implementation of these routines may be unable to match the performance of the library implementations, which have been hand-crafted in low-level machine language. Thus, it may be preferable to use a library routine for an operation if such is available, rather than code the routine in Fortran. (It is not *always* preferable to use library routines: many library routines are designed to be of general use and generality frequently reduces efficiency. A less general Fortran routine may give better performance than a general purpose library routine.)

  The Fortran compiler recognizes certain simple forms of loops as applications of library routines and replaces the loops with calls to the library routine. Transformations perform similar substitutions for more complex forms that the compiler cannot recognize.

For the MVM example, the sequential form in figure 6 is suitable for vectorization; the compiler replaces the double loop with an application of the SGEMV routine.

4.3.3 DAP

The implementation of MVM for the AMT DAP array processor is performed in two stages: the statically evaluated form is converted into an array form, which is then converted into DAP specific Fortran.

Array Form

The array form is a functional abstraction of the operations that a typical array processor could be expected to perform; that is, it is a form that has an operational interpretation suited to array processors, but that is expressed entirely functionally. There are several reasons for introducing an array form, rather than going directly to an imperative form for, say, the AMT DAP:

- problem decomposition: the problem of converting into an imperative form is separated from the problem of converting into a data-parallel form;
- the array form should be reasonably suited to implementation for other array processors, or for conversion to other array forms such as the emerging parallel Fortran standards (Fortran90, High Performance Fortran);
- the functional form has simple semantics and so is amenable to further manipulation (such as common sub-expression elimination).

The array form is shown in figure 7. This form is considerably different from the previous forms, so we will briefly explain the functions used:

- the application of `reduce.rows` forms the sum of the elements along the rows of its first argument (a matrix), to produce a vector;
- `times.array` performs the elementwise multiplication of two matrices;
- the first matrix is `A`;
- the second matrix is formed by the application of `expand.rows` which duplicates the vector `V` so that each row of the matrix is the same as `V`.

```
fun init
   = λn·
     λA·λV·
       reduce.rows (times.array (A) (expand.rows (V) (n)))
         (real.plus) (expand (0.0) ([n]))
     (read_vector([n]))
     (read_matrix([n,n]))
```

Figure 7: MVM: array form

DAP Fortran

The AMT DAP 510 is an array processor with a 32 by 32 grid of single-bit processing elements. The language used to program the machine is Fortran Plus Enhanced (FPE) [11], which uses the grid to support data parallel operations on vectors and matrices of any size. Many of these operations are supported as functions, so the conversion of *some* of the array form functions is straightforward. The FPE version of MVM is shown in figure 8.

```
program init
parameter(n)
real A(*n,*n)
real V(*n)
real init(*n)

read_matrix(A)
read_vector(V)
init = sumc(A*matr(V,n))
end
```

Figure 8: MVM: DAP form

4.4 Sparse Implementations

A sparse matrix is one in which most of its elements are zero. A program developer can take advantage of sparsity to reduce the time and space complexity of an implementation by storing only the non-zero elements and by restricting certain operations to only non-zero elements.

For the MVM example, we assume that the matrix A is tridiagonal (figure 9); that is, all elements outside the tridiagonal region (the region for which the row index and the column index differ by not more than 1) are zero. The number of elements that must be stored is thus reduced from n^2 to $3n - 2$. In addition, when forming the inner product of a row of A with the vector V, only the non-zero elements in the row need be considered (as the rest contribute nothing to the sum); thus, matrix-vector multiplication requires $3n - 2$ multiplications and $2n - 2$ additions when the matrix is tridiagonal, compared with n^2 multiplications and $n(n - 1)$ additions when the matrix is dense.

$$\begin{bmatrix} 1,1 & 1,2 & 0 & 0 & 0 \\ 2,1 & 2,2 & 2,3 & 0 & 0 \\ 0 & 3,2 & 3,3 & 3,4 & 0 \\ 0 & 0 & 4,3 & 4,4 & 4,5 \\ 0 & 0 & 0 & 5,4 & 5,5 \end{bmatrix} \longrightarrow \begin{bmatrix} - & 1,1 & 1,2 \\ 2,1 & 2,2 & 2,3 \\ 3,2 & 3,3 & 3,4 \\ 4,3 & 4,4 & 4,5 \\ 5,4 & 5,5 & - \end{bmatrix}$$

Figure 9: A tridiagonal matrix and its storage scheme

The non-zero elements of A are stored in an $n \times 3$ matrix, as shown in figure 9. The diagonal elements are stored in column 2; the elements below the diagonal in column 1, and those above in column 3. Notice that there are positions in the storage matrix that do not correspond to positions in A; these positions must be excluded from some calculations. Other storage schemes are possible, and could be employed by other sub-derivations.

The sparse version of MVM is shown in figure 10. The conditional expressions are required for the first and last elements of the product, as the first and last rows of A have only 2 elements, while the other rows have 3.

4.4.1 Implementations of Sparse Form

The appropriate sub-derivations are applied to the sparse form to produce the sequential/CRAY form (figure 11) and DAP form (figure 12). Note that in the DAP implementation, the matrix A is stored as a set of three vectors (`A1`, `A2` and `A3`), rather than as a matrix of width 3. The latter would not make good use of the DAP processor grid, as only 3 of the 32 columns of processors would be used; the former uses the entire grid, as the DAP hardware can use the processors across a row of the grid to store and manipulate successive bits of a single datum. The conversion into a set of three vectors was performed by an additional sub-derivation.

```
fun init
   = λn·
     λA·λV·
       generate([n],λi·
         (if (i=1)
           then 0.0
           else times(element(A′,[i,1]),element(V,[i-1]))
         )
       + times(element(A′,[i,2]),element(V,[i]))
       + (if (i=n)
           then 0.0
           else times(element(A′,[i,3]),element(V,[i+1]))
          )
       )
     (read_vector([n]))
     (read_matrix([n,n]))
```

Figure 10: MVM: tridiagonal form

```
      program init
      real A(n,n)
      real V(n)
      real init(n)

      init(1) = A(1,2)*V(1)+A(1,3)*V(2)
      do 100 i=2,n-1
      init(i) = A(i,1)*V(i-1)+A(i,2)*V(i)+A(i,3)*V(i+1)
100   continue
      init(n) = A(n,1)*V(n-1)+A(n,2)*V(n)
      end
```

Figure 11: MVM: Sequential/CRAY tridiagonal form

```
program init
real A1(*n),A2(*n),A3(*n)
real V(*n)
real init(*n)

init = A1*shrp(V)+A2*v+A3*shlp(V)
end
```

Figure 12: MVM: DAP tridiagonal form

4.5 Comments

The emphasis in the derivational approach to implementing algorithms is that a programmer should develop methods for implementing algorithms and encode these methods (as transformations/derivations), rather than apply these methods to produce a single implementation of a single algorithm. Several advantages accrue from this approach:

Reusability
The results of a programmer's *travails* (derivations and transformations) are reusable: they can be applied to multiple specifications to produce multiple implementations of each.

Extensibility
A programmer's work may also be readily extended: if a new implementation technique is required – for example, to use a pattern of sparsity or a computational architecture not already catered for – a programmer need develop a sub-derivation that supports only that new technique; other sub-derivations should not need to be altered.

In addition, almost all of the sub-derivations manipulate functional forms (that is, they manipulate pure, referentially transparent expressions). Functional forms have simple semantics that facilitate much of the manipulation.

Further, even when a new sub-derivation has to be written, some of the transformations from other derivations may be reused. For example, the DAP has a sequential execution model and shares many transformations with the sequential Fortran sub-derivation.

Transferability
Once a sub-derivation has been written, it requires no expertise to use. Thus, programmers may take advantage of another's experience in a particular implementation technique.

Correctness
Establishing the correctness of an implementation may be performed by establishing that each transformation set preserves the meaning of a program. As a typical transformation is simple, it is usually obvious that such is the case. Further, as transformations are formal, formal proof techniques may be applied to show the correctness of more complex transformations.

In contrast, when a programmer manually implements a specification of an algorithm, the product is a single implementation (for a single computer system) of that single specification. In addition, for a complex algorithm or for complex implementation techniques, the likelihood is that the programmer has made mistakes, due either to the complexity of the implementation procedure or to carelessness.

Further, the product's usefulness is limited to execution. The product itself, for example, does not accommodate analysis of the effect of particular implementation techniques, nor can it reasonably act as a basis for other implementations: if the specification is changed significantly, or an implementation is required for a different algorithm, or a new implementation technique is to be used, or an implementation is required for a new architecture, it is probably more cost effective to write a fresh implementation (with fresh mistakes) than to modify the existing implementation.

4.6 Results

For the MVM example, the differences between the derived implementations and implementations that a programmer would produce are trivial. The similarity of the derived and hand-crafted implementations is a vindication of the efficacy of the derivations, though it does mean that our normal practice of assessing derivations by comparing execution times would not be informative.

However, to illustrate the importance of tailoring implementations and the efficacy of transformations in doing so, we compare here the execution performance of the sparse and dense implementations of MVM. In addition, we can make a further comparison for CRAYs as there is a standard BLAS routine for multiplying a vector by a banded matrix (a tridiagonal matrix being a banded matrix in which the band has width 3); the CRAY routine is more general than our derived routine, but is also somewhat slower.

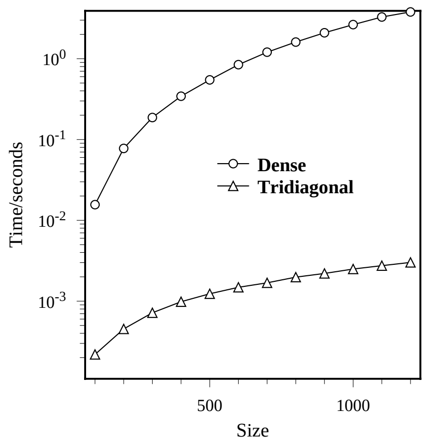

Figure 13: Times for sequential dense and tridiagonal implementations

Figure 13 shows the execution times for various matrix sizes for the dense and tridiagonal implementations on a sequential computer (A NeXT workstation). The tridiagonal implementation is considerably faster than the dense implementation.

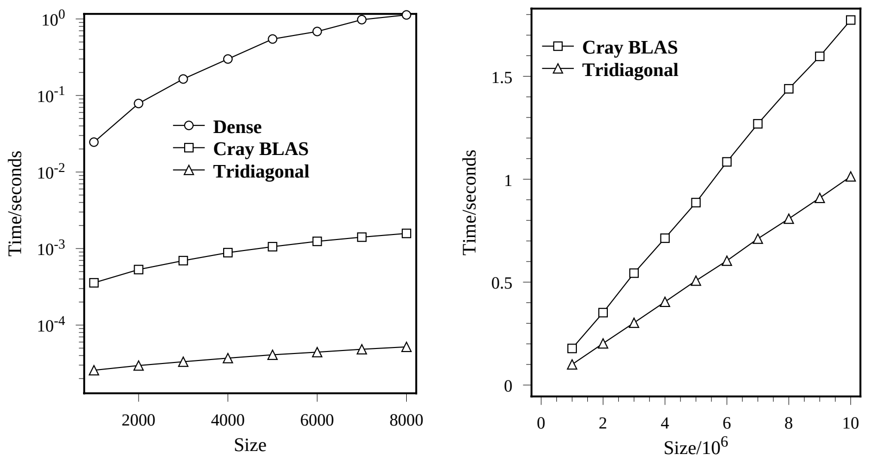

Figure 14: Times for CRAY dense and tridiagonal implementations, and CRAY BLAS routine

The graphs in figure 14 show the execution times for the dense and tridiagonal implementations on a 3 processor CRAY YMP with 128M words of (shared) memory. Each implementation was compiled for one processor only and with full vectorization. These graphs also show timings for the CRAY implementation of the BLAS SGBMV subroutine. This routine performs the assignment $\texttt{y:=}\alpha\texttt{Ax+}\beta\texttt{y}$ where $\texttt{A}$ is a sparse matrix with an arbitrary number of contiguous sub- and super-diagonals, $\texttt{x}$ and $\texttt{y}$ are dense vectors and α and β are scalars. With 1 super-diagonal and 1 sub-diagonal, and with $\alpha = 1$ and $\beta = 0$, this routine corresponds to tridiagonal multiplication.

The left graph shows timings for the range over which the dense implementation could be used. The right graph shows timings for the tridiagonal and BLAS implementations for much larger matrices (approaching the maximum size that could be used with both of these implementations).

The BLAS routine is approximately 1.75 times slower than our tridiagonal routine for large matrices. The comparison is somewhat unfair as the CRAY routine is a more general routine, although the comparison does demonstrate the advantage of specialization that is part of the transformational approach.

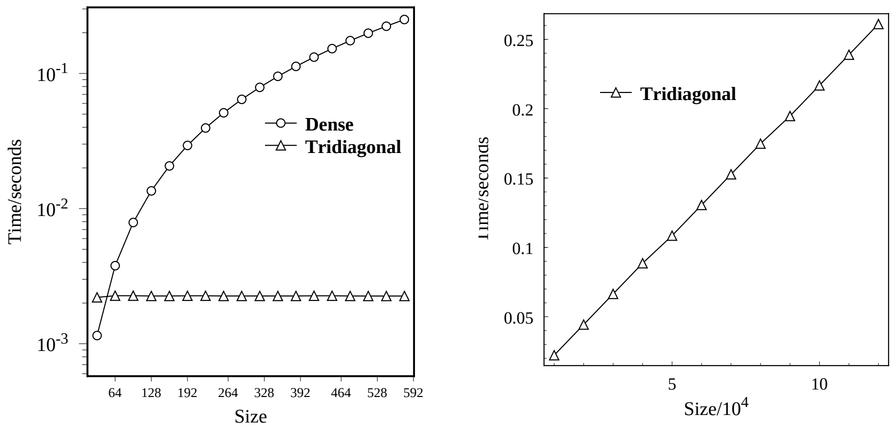

Figure 15: Times for DAP dense and tridiagonal implementations

Figure 15 shows timings for the DAP implementations. The left graph shows timings for the range over which the dense implementation could be used. The right graph shows the timings for the tridiagonal implementation over a wider range.

4.7 Further Results

While matrix-vector multiplication is an important component of many numerical algorithms, it is still only a component. The derivations discussed in this paper have been applied to other, more complex, scientifically significant algorithms, producing efficient implementations. See, for example:

- [4] which reports on the derivation of a CRAY implementation of a solver for hyperbolic partial differential equations, the performance of which surpasses the performance of a hand-crafted implementation;
- [8] which reports on the near equivalence in performance of a derived DAP implementation of an eigenvalue algorithm with a hand-crafted implementation.

5 Conclusion

We have discussed a family of data-parallel derivations that produce highly efficient implementations from a single clear, implementation-independent algorithm specification. These implementations are tailored for the particular hardware architecture that will execute the algorithm and can be tailored further for the particular data being manipulated. The tailoring of the implementation for the architecture ensures that the best performance is extracted from the parallel architecture being used. When an implementation for another architecture is required a new derivation specialization is developed and a new implementation derived from the same initial specification.

The use of this approach does not require significant effort from the user. In our experience a competent mathematician can write functional specifications in a few hours. An existing derivation can be used in the same way a conventional compiler is used, without knowledge (or understanding) of the internal transformation process—the programmer provides a functional specification and receives as output a Fortran or C program which can be compiled and executed.

Developing a specialized derivation for a new architecture requires specialized skills. However, existing derivations and transformations form a backbone from which new derivations may be extruded with minimal programmer effort. The development of a derivation for a particular architecture requires, in practice, a few weeks. For example, the development of the CRAY derivation specialization took approximately two weeks and much of this effort was in understanding the programming forms that execute well on the CRAY. Of course once this effort has been expended, the derivation may be used with many specifications and without the user needing to understanding the transformations that have been written. In contrast, using a conventional approach, a programmer must reapply his skills for each new algorithm implementation and must validate the implementation produced.

The transformational approach is comparable *in purpose* to that of developing a programming language compiler, yet fundamentally different *in method*. In constructing a derivation we attempt to identify the many distinct language models that exist between an abstract functional specification and some implementation model. These models are subsequently encoded as transformations. The identification of intermediate models simplifies development of a derivation, but, more importantly, produces models that are shared by related derivations. So, for example, most of the transformations required to produce efficient code for the CRAY and the AMT DAP — distinctly different implementation models — are shared. Indeed, the transformations are also shared with the derivation for a shared-memory multiprocessor (see Figure 1).

The transformational approach is still in its infancy. Additional work is required in analysing additional algorithm specifications and understanding and encoding programmer optimizations. Our work is concentrated on the consideration of example algorithms and the many possible implementations of these algorithms.

References

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

[2] 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.

[3] Program reusability through program transformation, James M. Boyle and M. N. Muralidharan, 1984, IEEE Trans. Software Eng., 10 (5): 574-88 (Sept.).

[4] Functional specifications for mathematical computations, James M. Boyle, T. J. Harmer, Editor B. Moeller in Proc. IFIP TC2/WG2.1 Working Conf. on Construction Programs from Specifications, pp 761–767.

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

[6] Program adaption and program transformation, In R. Ebert, J. Lueger and L. Goecke (editors), Practice in Software Adaption and Maintenance, pp. 3–20, North-Holland Publishing Co., Amsterdam, 1980.

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

[8] The Construction of Numerical Mathematical Software for the AMT DAP by Program Transformation, J.M. Boyle, M. Clint, Stephen Fitzpatrick and T.J. Harmer, Proceedings of CONPAR 92-VAPP V, L Bouge, M. Cosnard, Y. Robert, D. Trystram (editors), Springer-Verlag, 1992.

[9] The Calculi of Lambda-Conversion, A. Church, Annals of Mathematics Studies, No. 6, Princeton University Press.

[10] American National Standard Fortran, X3.9 — 1978 (FORTRAN 77) American National Standards Institute, 1430 Broadway, New York, NY 10018, U.S.A.

[11] DAP Series: FORTRAN-PLUS enhanced, man102.01.