Program Transformations for Automatically Tailoring Algorithms to Sparse Matrices[1]

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

boyle@mcs.anl.gov, {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.

February 1994

Technical Report 1994/Feb-SF.TJH.JMB, Computer Science, QUB, February 1994
<URL:http://www.cs.qub.ac.uk/pub/TechReports/1994/1994/Feb-SF.TJH.JMB/>
<URL:ftp://ftp.cs.qub.ac.uk/pub/TechReports/1994/1994/Feb-SF.TJH.JMB.ps.gz>

Abstract

We show how efficient implementations can be derived from high-level functional specifications of numerical algorithms using automatic program transformation. We emphasize the automatic tailoring of implementations for manipulation of sparse data sets. Execution times are reported for matrix-vector multiplication and a conjugate gradient algorithm.

Keywords: Automatic Parallelization and Mapping, Sparse matrices, Functional programming, Program Transformation and Program Derivation.

1 Introduction

Developing an implementation of a numerical mathematical algorithm is a difficult task—from a *simple textbook specification* of an algorithm a programmer must create an *implementation efficient in execution*. The textbook specifier emphasizes clarity and ignores the detail of how an algorithm may be executed efficiently; the implementor sacrifices clarity and strives to organize the computation to achieve efficient execution. For many numerical mathematical algorithms an efficient implementation is essential because of their computational requirements.

A programmer will attempt to exploit characteristics of a particular machine architecture to achieve high execution performance. For example, loops will be designed to ensure that vectorization is possible and thus that the implementation can take advantage of available vector hardware. Similarly, if a machine can perform more than one task simultaneously then large computations may be partitioned into a number of parallel tasks.

A programmer will attempt to exploit the characteristics of the data being operated on by the algorithm to improve execution performance. For example, if a matrix is symmetric then this symmetry may be used in some computations to reduce the number of arithmetic operations performed. If a matrix is sparse then computations can be concentrated on the significant matrix elements. In addition, the characteristics of the data can be used to reduce the machine storage requirements of an implementation—only half the elements of a symmetric matrix, or only the significant elements of a sparse matrix, may be recorded.

The programmer must, then, develop an efficient implementation taking into consideration the particular computations to be performed, the type of machine architecture to be used by the implementation and the characteristics of data on which the implementation operates. Often this may involve the development

[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 Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38

A simplified version of this report was presented at CONPAR '94 -VAPP VI: see 'Parallel Processing: CONPAR 94-VAPP VI' pp 148-159, Springer-Verlag, Lecture Notes in Computer Science 854, September 1994; Bruno Buchberger and Jens Volkert (Ed.)

of a family of related implementations with each implementation tailored for a particular programming execution model, characteristics of data, or both. An implementation of a programming model may be refined further given the particular (and often peculiar) characteristics of some computers.

Such a family of implementations is derived from the single textbook algorithm specification and, yet, each implementations is different. Is it possible to automate the derivation of implementations from the abstract textbook algorithm specifications? We have demonstrated that automatic program transformation makes it possible to derive highly specialized, efficient implementations from high-level, abstract functional specifications [7], [8], [4]. The starting point for transformation is a high-level specification of an algorithm—a specification in the sense that it describes *what* should be computed without unnecessary detail about *how* the computation should be performed. The implementations for parallel execution that we derived have, in most cases, equaled the performance of equivalent hand-crafted implementations.

In this paper we consider how to obtain efficient implementations of algorithm specifications tailored for sparse data and vector parallel machines. That is, from an abstract specification of an algorithm which does not refer to the sparsity of the data being operated on we demonstrate how an efficient implementation can be derived for efficient execution on parallel computers. As an example we consider general sparse matrices—matrices in which there is no pattern to the sparsity,but each row has (nearly) the same number of significant elements. We also compare this implementation with implementations operating on regular, banded matrices—further details of this transformational derivation can be found in [9].

2 Functional Specifications

The language we use for algorithm definitions is a small subset of the Standard ML (SML) language [12]—a pure functional programming language. The important features of this language are:

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

We use a library of vector and matrix operations that commonly occur in numerical and 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 dimensionality (in definitions, vectors and matrices are considered to be particular forms of arrays) and returns the value of the element stored at the location specified by the `index`. For convenience, various alternative notations can be used for `element`. For example, the ith 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`

The `generate` function allows an array to be defined by specifying its dimensions and the values of its elements. The set of indices over which an array is defined is specified as a `shape`. The values of the elements are defined by a *generating function* which takes an `index` as argument and gives the value of the element at the location specified by the `index`.

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 can be defined as

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

where the function `transposed_shape` returns the shape of its array argument with the rows and columns interchanged (this function is defined as part of the `shape` abstract data type (ADT)).

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

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 $V[1] + V[2] + \ldots$. (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 element-wise 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 general matrix and vector operations:

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

fun mvmult(M:real matrix, V:real vector):real vector
  = generate(size(M,0),fn(i:int)=>innerproduct(row(M,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. The problem with the rapid prototyping approach is that it leads to *very* leisurely execution. The theme of this paper is to show how an efficient implementation can be obtained to speed up execution, while still maintaining a direct connection between that implementation and the specification used in rapid prototyping.

2.1 Conjugate Gradient

Figure 1 is an example of the functional specification of a more complex algorithm: a conjugate gradient algorithm for the solution of systems of simultaneous equations: that is, to calculate a vector x which satisfies $Ax = b$, where A is a matrix and b is a vector. Most of the specification should be readily understood by anyone with a knowledge of conjugate gradient algorithms. Some of the significant features of the specification are:

- The algorithm is an iterative algorithm: it repeatedly applies a function until a satisfactory solution is obtained. The function `iterate` is used to define this repetition. The arguments to `iterate` are, in order: the function to be repeatedly applied; an initial value with which to begin the iteration; and a Boolean function that specifies when the iteration should stop.
- The function to be repeatedly applied is `cg_iteration` (defined locally in the `cg_construct` function). This iteration function takes an argument of type `cgstate`, which is the Cartesian product of four real vectors and an integer.

- The operator * is overloaded in the specification to indicate both matrix-vector multiplication and the multiplication of a vector by a scalar. Similarly, the operator + is overloaded to mean: integer addition, real addition and vector addition. Such overloading is in accord with normal mathematical notation.

```
val epsilon:real = 1.0E-14;
type cgstate
  = real vector * real vector * real vector * real vector * int;

fun cg_construct(A:real matrix, b:real vector):cgstate =
let
  val x0 = constant(shape(b), 0.0);
  val r0 = b;
  val p0 = r0;
  val q0 = A*p0;

  fun is_accurate_solution((x,r,p,q,cnt):cgstate):bool =
       innerproduct(r,r)<epsilon;

  fun cg_iteration((x,r,p,q,cnt):cgstate):cgstate =
      let
    val rr:real = innerproduct(r,r);
    val cnt':int = cnt+1;
    val alpha:real = rr/innerproduct(q,q);
    val x':real vector = x+p*alpha;
    val r':real vector = r-transpose(A)*q*alpha;
    val beta:real = innerproduct(r',r')/rr;
    val p':real vector = r'+p*beta;
    val q':real vector = A*r'+q*beta
      in
    cgstate(x',r',p',q',cnt'),
      end

in
     iterate(cg_iteration,
       cgstate(x0, r0, p0, q0, 0),
       is_accurate_solution)
end;
```

Figure 1: Functional specification of conjugate gradient

3 The Data Sets

In this paper, we tailor the conjugate gradient algorithm for execution when the matrix A is known to be sparse. There are many types of sparsity: here we consider a matrix which has a fixed number of non-zero elements per row, as illustrated in figure 2 [11]. Such a matrix of order n, with w non-zeros per row, can be stored in two $n \times w$ matrices: one — the primary store Ap — stores the values of the elements; the other — the secondary store As — stores the column indices for the elements in the primary store. Location $[i, j']$ in the primary matrix stores the value of the $[i, j]$ element of the sparse matrix, where $j \equiv As[i, j']$; that is, the following identity holds: $Ap[i, j'] \equiv A[i, [As[i, j']]$.

If the number of non-zero elements in a given row is less than w, then sufficient zeros can be stored in the primary matrix to fill the row. This form of sparsity is efficient in storage if the number of non-zeros averaged over the rows is not much less than the maximum number of non-zeros.

This is an example of a particular form of sparsity—one that was useful in our work—and is intended as an illustration of how transformation can simultaneously tailor an implementation to use both this form of compressed data representation and a parallel computer. Other representations are possible, of course, and it is the strength of the program transformation approach that it is a relatively straightforward procedure

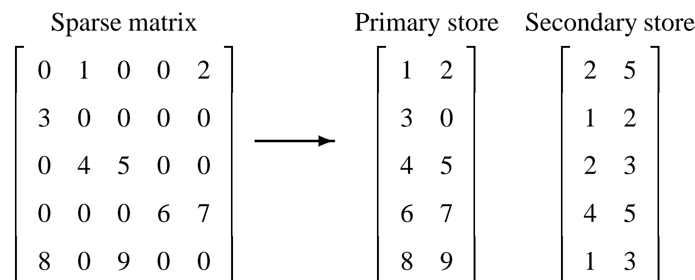


Figure 2: Storage for a General Sparse Matrix

to change the type of implementation generated by transformational derivations. In our derivation a new implementation could be generated by replacing the *mapping* transformation phase (outlined below).

4 Program Transformation

Because the direct execution of functional specifications for rapid prototyping is slow, we do not intend functional definitions to be used as practical implementations of algorithms; rather, they are the source from which we derive efficient implementations by program transformation. A *transformation rule* is a rewrite rule consisting of a pattern and a replacement, both defined using a wide spectrum grammar [1] [6]. For example,

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

is a simple transformation rule. The <ident>s are non-terminals (corresponding to 'identifiers') in the grammar being transformed. The same label ("1") on the non-terminals requires the non-terminals to match the same identifier; in the replacement <ident>"1" refers to whatever identifier was matched in the pattern. Thus, this transformation rule would convert the expression x+x into 2*x, but it would not convert x+y (because the identifiers are different), nor would it convert 1+1 (because 1 is not an <ident>). The transformation system exhaustively scans a program for instances of the pattern, replacing each such instance with the corresponding replacement.

Usually, several transformation rules are designed to work together. For example, simplification of expressions can be performed using the many algebraic properties of data types, such as zero being the identity for addition, one being the identity for multiplication, etc. It is impossible to achieve full simplification of expressions by applying these laws in sequence (i.e. by exhaustively applying one law, then exhaustively applying another law, and so on) since the application of one may give rise to an expression which can be simplified by a transformation rule that has already been applied. Thus, the rules are grouped to create a transformation set and the entire set applied exhaustively; i.e. until no transformation in the set can be applied.

A derivation is constructed by forming a sequence of such transformation sets. The derivation of an implementation is thus not a single, monolithic process that transforms a functional specification to an imperative implementation; rather, it is a process that molds an implementation from the definition through many intermediate forms.

Unlike a conventional programming-language compiler, a transformational derivation can be customized for a particular target architecture or algorithm. Steps in the derivation may be added or removed depending on the particular requirements of the algorithm or of the implementation. A transformation system can thus be thought of as providing an easy way to define a diverse family of compilers.

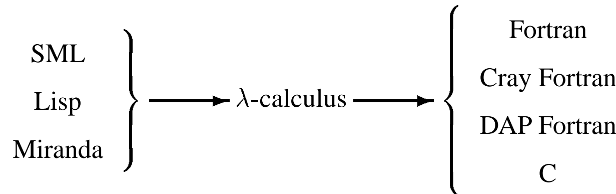
5 A Family of Transformational Derivations

The implementation of a functional specification (by transformation) 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 can first be converted into the λ-calculus and then into Fortran:

SML ⟶ λ-calculus ⟶ Fortran.

Each of these transitions may also be sub-divided into further intermediate forms, to whatever degree 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 language, and so may be combined with another sub-derivation to create, say, an array processor implementation (for example, for the AMT DAP [8]). 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.



Further, sub-derivations can be added to optimize implementations in various ways 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 3 is a (somewhat simplified) illustration of the relationships among various intermediate forms created by such sub-derivations.

6 An Example: Matrix-Vector Multiplication

In this section we illustrate the stages in deriving implementations of the functional specification of matrix-vector multiplication—the conjugate gradient example is too large for presentation purposes, although later we present timing information for a sparse version derived from our conjugate gradient specification.

```
fun sum(V: real vector): real
    reduce(size(V), V, +, 0.0)

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

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

For this simple specification we must decide the type of implementation to be produced and thus select the transformation sets that should be applied (see figure 3). It is important to realize that the transformations are performed automatically (after the user has selected a particular derivation path) and for most users the transformation process is like a sophisticated programming language compiler.

Conversion to Abstract Functional Specification Language - λ-Calculus

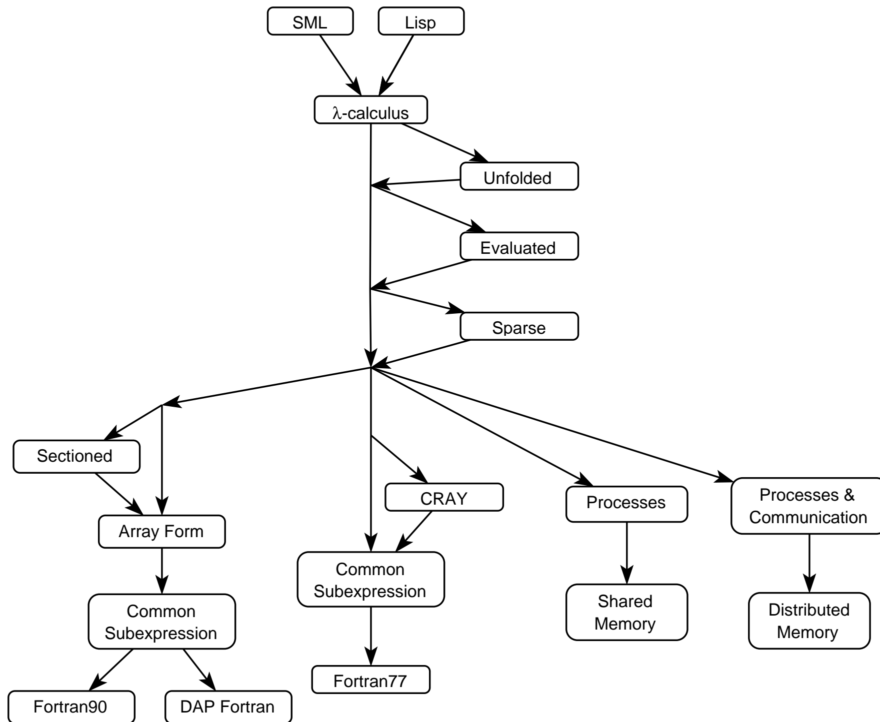


Figure 3: A family of derivations

```
fun sum: real
    λ V: real vector.
         reduce(size(V),V,+,0.0)

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

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

The SML special forms are removed from the program text. A derivation is now independent of the starting language—SML, Lisp, Miranda or another functional programming language.

Unfolding and Static Evaluation

Much of the structure of a functional specification is present for human reading. For example, separate function definitions divide the definition into easily understood sections. However, if this structure remains in the final implementation, it increases the execution cost of the program.

These potential inefficiencies can be removed using the techniques of function unfolding (in which an application of a function is replaced with the body of the function's definition) and static evaluation (in which algebraic properties of data types are used to simplify expressions). The result of applying these sub-derivations to the expression for matrix-vector multiplication is:

```
fun mvmult: real vector =
   generate(n, λ i.
                  reduce(n,
                         λ j.
                            times(element(A,[i,j]),element(V,[j])),
                         +,0.0))
```

where we assume that the sizes of A and V have been defined in terms of some parameter `n`.

Sparse Specialization

The specification before this stage assumes that the matrix is dense. A sparse specialization must identify data that is sparse, optimize the computation to take advantage of the sparsity and map dense arrays to compressed array representations.

A sparse specialization derivation is subdivided into three phases.

Phase 1

The functional specification is annotated to indicate that particular data is sparse, using a definition supplied by the specifier of the particular sparsity in the data. For example, the specifier may indicate that certain input matrices are tridiagonal or, as in this worked example, that a matrix has a fixed number of elements in each row, but those elements do not have a simple pattern of occurrence. (The interested reader is referred to [9] for details of deriving efficient implementations for banded matrix types.)

Transformations then use such annotations to make explicit the distinction between the zeros and non-zeros: each application of the `element` function is replaced with a conditional expression that checks whether the index refers to a zero or a non-zero: if the former, the conditional evaluates to zero; if the latter, to the value of the stored element.

```
fun mvmult: real vector =
generate(N,
      λ i.
        reduce([N],
              λ j.
                times( if ([i,j] ∈ fixed_row_number([n,n], w))
                       then element(A,[i,j]) else 0.0,
                       element(V,[j])),
              +, 0.0))
```

The function `fixed_row_number` is the set of significant indices of the matrix—which is of size $n \times n$ with w elements significant in each row. The derivation treats this function as an unknown: the set of indices for a particular matrix is not known until an implementation is complete and the derived program is to be compiled for execution.

Phase 2

The computation is optimized to take advantage of the sparse data being manipulated.

The conditional expression is distributed out of the application of the `times` function to give

$\longrightarrow$

```
fun mvmult: real vector =
generate(n, λ i.
  reduce(n, λ j.
          if ([i,j] ∈ fixed_row_number([n,n], w) )
          then times(element(A, [i,j]), element(V, [j])
          else 0.0,
          +, 0.0))
```

which can be further optimized by restricting the reduction to only the significant elements of a row of the sparse matrix:

$\longrightarrow$

```
fun mvmult: real vector =
   generate(n,
            λ i.
              reduce(row(fixed_row_number([n,n],w),i),
                       λ j.
                         times(element(A,[i,j]) element(V,[j])),
                       +, 0.0)).
```

This expression is optimized in two ways: there are no checks for zero and non-zero elements; the reduction is over only the w non-zero elements in a row (represented by the function `row`).

Phase 3

The sparse matrix is mapped onto primary and secondary stores.

The mapping on to the primary store is made according to the rules:

$$[i,j] -> [i, locate(shape, [i,j])]$$

and the inverse

$$[i,j'] -> [i, secondary([i,j])]$$

```
fun mvmult: real vector =
generate(n,
         λ i.
           reduce(w,
                  λ j'.
                    times(element(A:[i,j'])
                            element(V,[secondary(A, [i,j'])])),
                  +, 0.0))
```

The function `secondary` maps locations in the primary store onto locations in the sparse matrix (using the column indices stored in the secondary store). Note that the reduction is now over the range 1 to w.

This phase may be tailored to the particular needs of the user—thus this implementation is one of many possible implementation that could be generated. The strength of the transformation method is that a user can mix-and-match transformation sets to cater for the particular needs of the application.

Imperative Implementation

This phase of the derivation tailors the specification for execution on a particular computer—the output language may be either Fortran of C.

Sequential Implementation

The derivation is specialized to execute on a sequential computer.

```
                integer n, w
                parameter(n=??, w=???)
                real Ap(n,w), U(n), V(n)
                integer As(n,w)
                integer i,j

                do 100 i=1,n
                     U(i) = 0.0
                     do 101 j=1,w
                          U(i) = U(i)+Ap(i,j)*V(As(i,j))
       101           continue
       100      continue
                end
```

Additional transformations can be applied as part of the sequential derivation to derive an implementation suitable for execution on a vector computer, such as the Cray X-MP [7].

Implementations for other computer architectures are possible, such as an implementation tailored for the AMT DAP 510 computer and a partitioned MIMD solution. [8], [5], [10].

7 Results

Figure 4 show execution times for both the standard, dense version and the sparse version of matrix-vector multiplication on a sequential computer (a NeXT workstation to be precise) and on a CRAY vector computer. For the sparse version, 98% of the elements in each row of the matrix were zero; the non-zero elements were arranged in random positions.

The corresponding results for conjugate gradient are shown in figures 5.

The execution times for the sparse versions of the algorithms are considerably reduced compared with the dense versions; this is testimony to the efficacy of the tailoring process, and of the transformations. However, for our purposes, a more important evaluation criterion is that the derived implementations are not significantly different from what we would produce if we manually applied our skills as programmers.

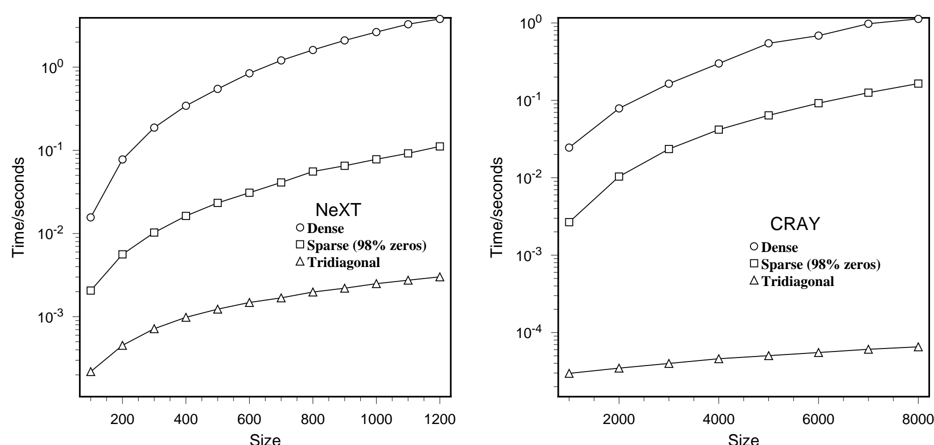


Figure 4: Execution times for matrix-vector multiplication

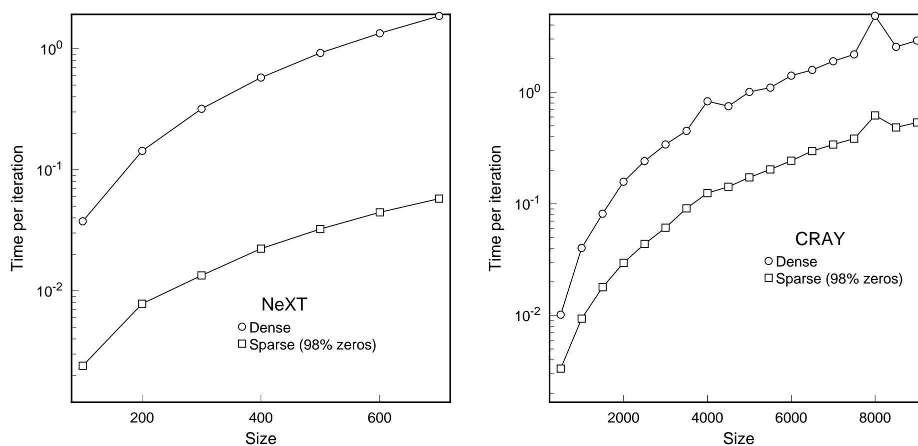


Figure 5: Execution times for conjugate gradient

8 Conclusion

We have outlined how a highly efficient implementation, tailored to a particular type of data set, may be automatically produced from a clear, high-level, implementation-independent algorithm specification. Such

an implementations is part of a family of implementations, all derived from the same specification. Each implementation is tailored for the particular hardware architecture that will execute the algorithm and tailored 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 specialized derivation is developed and a new implementation derived from the same initial specification.

The transformational 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 and forms of data *does* require specialized skills. However, existing derivations and transformations form a backbone to which further sub-derivations may be added with minimal programmer effort. The development of a derivation for a particular data form 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 revalidate each new 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, it produces models that are shared by related derivations.

For example, most of the transformations used by the sub-derivations that create implementations for the CRAY and AMT DAP architectures—which have distinctly different implementation models—are common to the two sub-derivations. Indeed, the transformations are also shared with the derivation for a shared-memory multiprocessor (see Figure 3).

The transformational approach is still in its infancy. Additional work is required in analysing additional algorithm specifications and understanding and encoding the optimizations applied by programmers.

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

[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] *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, Amsterdam

[7] *A Practical Functional Program for the Cray X-MP*, J.M. Boyle and T.J. Harmer, Journal of Functional Programming, 2(1), 1992, pp81-126.

[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.

[9] *Deriving Efficient Sparse Implementations of Numerical Mathematical Algorithms using Automatic Program Transformation*, J.M. Boyle, S Fitzpatrick and T.J. Harmer, QUB Technical Report 1994.

[10] *Deriving Distributed MIMD Implementations from Functional Specifications*, JM Boyle, M Clint, S Fitzpatrick and TJ Harmer, Proceedings of *ParCo '93*, 7-10 September 1993, Grenoble, France. pp44-48.

[11] *Computer program for solution of large sparse unsymmetric systems of linear equations*, JE Key, Int. J. Numer. meth. Eng. 6,497-509

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