Deriving Efficient Parallel Implementations of Algorithms Operating on General Sparse Matrices using Automatic Program Transformation*

Stephen Fitzpatrick[1], T. J. Harmer[1] and J. M. Boyle[2]

[1] Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, United Kingdom
[2] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne IL 60439, USA

email: s_fitzpatrick@cs.qub.ac.uk, harmer@cs.qub.ac.uk, boyle@mcs.anl.gov

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 a conjugate gradient algorithm.

Keywords: Program Transformation and Program Derivation, Automatic Parallelization and Mapping, Sparse Matrices, Functional Programming.

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 implementer will often sacrifice clarity in order 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

* 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

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 machine architecture to be used by the implementation and the characteristics of data on which the implementation operates. Often this may involve the development of a family of related implementations with each implementation tailored for a particular 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 but each implementation 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 [4, 6, 7]. 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, equalled 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, we demonstrate how an implementation can be derived for efficient execution on parallel computers from an abstract specification of an algorithm which does not refer to the sparsity of the data being manipulated. This approach is in contrast to that followed traditionally—in either a functional language context or imperative language context—where an implementation is constructed with a particular sparse representation in mind [13, 14, 9] As an example of our approach we consider general sparse matrices—matrices in which there is no pattern to the sparsity,but in which each row has (nearly) the same number of significant elements.

2 Functional Specifications

The language we use for algorithm specifications is a small subset of the Standard ML (SML) language [10]—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.

2.1 Example: Conjugate Gradient

Figure 1 is an example of the functional specification of a moderately complex algorithm: a conjugate gradient algorithm for the solution of systems of simultaneous equations — that is, the algorithm calculates a vector x (of size n) which satisfies $Ax = b$, where A is a matrix (of size $n \times n$) and b is a vector (of size n).

```
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:real vector = constant(shape(b),0.0);
  val r0:real vector = b;
  val p0:real vector = r0;
  val q0:real vector = 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
```

Fig. 1. Functional specification of conjugate gradient

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 defines an iterative process: 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 (`cg_iterate`); an initial value with which to begin the iteration (`cgstate(x0,r0,p0,q0,0)`); and a Boolean function that specifies when the iteration should stop (`is_accurate_solution`).

– The function `cg_iteration` takes an argument of type `cgstate` and returns a value of type `cgstate`. The type `cgstate` 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 indicate integer addition, real addition and vector addition. Such overloading is in accord with normal mathematical notation.

The library definitions we use are defined in a similar manner.

```
fun times(U:real vector,V:real vector):real vector
 = generate(size(U),fn(i:int)=>U@[i]*V@[i])
fun sum(V:real vector):real
 = reduce(+,0.0,size(V),fn(i:int)=>V@[i])
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))
```

Fig. 2. Functional specification of numerical mathematical primitives

These definitions use three basic array functions:

`generate(S:shape,g:index→α)`
defines an α-array over the index set `S`. The value of element i is the value of the *generating function* `g` applied to i. A generating function may be defined in SML using a *function prototype*: for example, `fn(i)=>E` defines a function with formal parameter `i`, and 'body' `E`.

`element(A:α array,i:index)`
returns the value of the element at the location specified by the `index i`.

`reduce(r:(α×α→α),r0:α,S:shape,g:(index→α))`
combines the elements of an array (defined as the combination of the `shape S` and the generating function `g`) into a single value using the binary *reducing function* `r`.

Space does not permit us to discuss our specification language further and the interested reader is referred to [6, 7].

We believe that functional specifications should be high-level: they should express operations in the most natural way, with no consideration being 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, while still maintaining a direct connection between that implementation and the specification used for rapid prototyping.

3 The Data Sets

In this paper, we show the steps required to develop an efficient implementation of the conjugate gradient algorithm 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[9], as illustrated in Fig. 3.

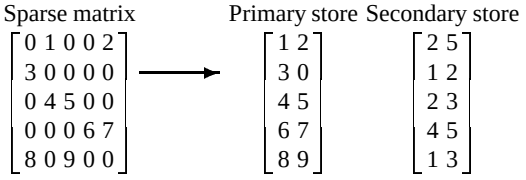


Fig. 3. Storage for a general sparse matrix

Such a matrix of order n, with w non-zeros per row, can be stored in two $n \times w$ matrices: the primary store A_p stores the values of the elements; the secondary store A_s stores the column indices for the elements in the primary store. Location $[i, j']$ in the primary store contains the value of the $[i, j]$ element of the sparse matrix, where $j \equiv A_s[i, j']$; that is, the following identity holds:

$$A_p[i, j'] \equiv A[i, [A_s[i, j']]] \ .$$

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 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. Program transformation is performed using TAMPR *transformation rules*. A transformation rule is a rewrite rule consisting of a pattern and a

replacement, both defined using a wide spectrum grammar [1, 5]. When a transformation rule is applied to a program, sections of the program that are matched by the pattern are changed into the replacement.

Transformations are often constructed so that two or more rules are combined to achieve some change. Such combinations of rules are grouped in a transformation set: each rule in the set is applied exhaustively until none of the patterns matches any section of the program.

A transformation set achieves some well-defined change in a program. A practical program transformation may require many such changes, so sequences of transformation sets — or *derivations* — are constructed. In practice, it is useful to construct derivations from *sub-derivations*, which in turn are constructed from transformation sets, or even from other sub-derivations.

For example, the transformation from SML to Fortran is performed by an SML-Fortran derivation, consisting of an SML–λ-Calculus[3] sub-derivation and a λ-Calculus–Fortran sub-derivation. Both of these sub-derivations are sequences of transformation sets. The λ-calculus is an example of an *intermediate form* created by a sub-derivation.

There are advantages to dividing a derivation into sub-derivations beyond the standard divide-and-conquer simplification of the task of implementing a specification: 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 implementation in C or an array processor implementation (for example, for the AMT DAP [7]). Similarly, the λ-Calculus–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 sparsity. Figure 4 is a (somewhat simplified) illustration of the relationships among various intermediate forms created by such sub-derivations.

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 An Example: Matrix-Vector Multiplication

In this section we illustrate the stages in deriving implementations of the functional specification of matrix-vector multiplication defined in Fig. 2—the conjugate gradi-

[3] The λ-calculus is a standard mathematical formalism for expressing computation.

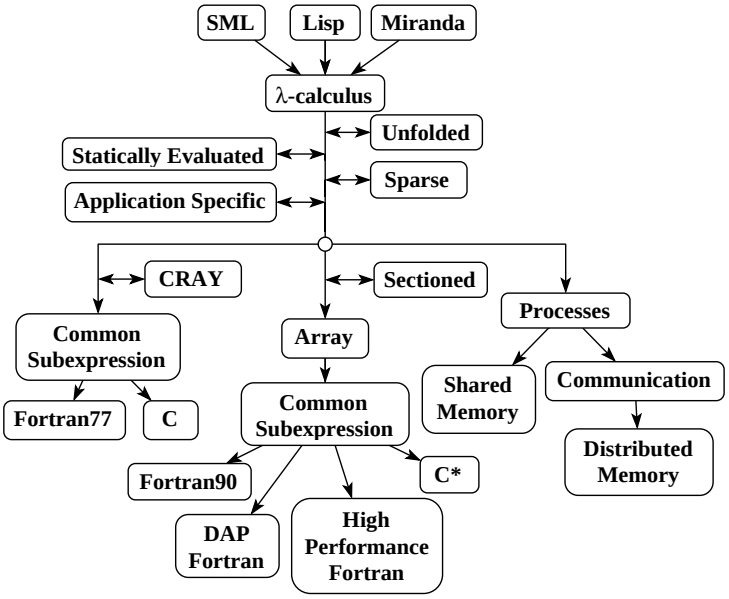


Fig. 4. A family of derivations

ent example is too large for presentation purposes, although later we present timing information for a sparse version derived from our conjugate gradient specification.

For this simple specification we must decide upon the type of implementation to be produced and thus select the transformation sets that should be applied (see Fig. 4). It is important to realize that the transformations are performed automatically (after the user has selected a particular derivation path) and that for most users the transformation process is like a sophisticated programming language compiler.

For an implementation in Fortran and with sparse data representation the stages are:

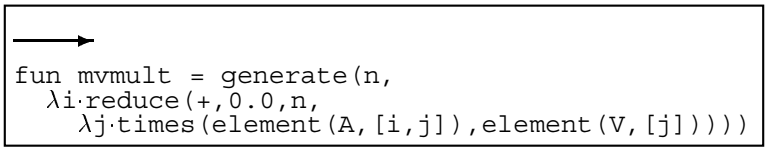
1. Conversion to Abstract Functional Specification Language (the λ-Calculus)
The SML special forms are removed from the program text. A derivation is now independent of the starting language—SML, Lisp, Miranda or another functional programming language.

```
fun times:real vector × real vector → real vector
 = λU,V·generate(size(U),
    λi·real.times(element(U,i),element(V,i)))
fun sum:real vector → real
 = λV·reduce(+,0.0,size(V),λi·element(V,i))
fun innerproduct:real vector × real vector → real
 = λU,V·sum(times(U,V))
fun mvmult:real matrix × real vector → real vector
 = λM,V·generate(size(M,0),λi·innerproduct(row(M,i),V))
```

2. Unfolding and Static Evaluation

Much of the structure of a functional specification is provided to facilitate human understanding. For example, separate function definitions divide the definition into easily understood parts. However, if this structure persists in the final implementation, it increases the execution costs 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 = generate(n,
  λi·reduce(+,0.0,n,
    λj·times(element(A,[i,j]),element(V,[j]))))
```

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

3. Sparse Specialization

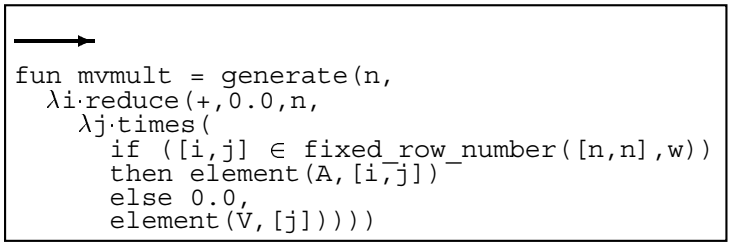
The specification at this stage assumes that the matrix is dense. A sparse specialization must optimize the computation to take advantage of the sparsity and map the sparse matrix from its standard, dense representation to its compressed representation.

A sparse specialization derivation is sub-divided into three phases.

Phase 1:annotation

The functional specification is annotated to indicate that particular data structures are sparse. For example, the specifier may indicate that certain input matrices are tridiagonal or, as in the example presented here, that a matrix has a fixed number of non-zero elements scattered along each row.

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 element — if the former, the conditional expression evaluates to zero; if the latter, it yields the value of the element.

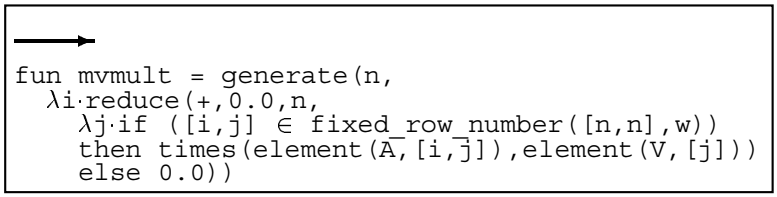


```
fun mvmult = generate(n,
  λi·reduce(+,0.0,n,
    λj·times(
      if ([i,j] ∈ fixed_row_number([n,n],w))
      then element(A,[i,j])
      else 0.0,
      element(V,[j]))))
```

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 the implementation is executed.

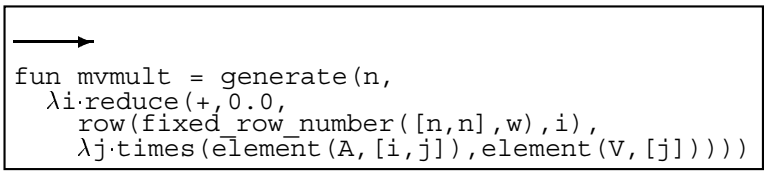
Phase 2: optimization

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



```
fun mvmult = generate(n,
  λi·reduce(+,0.0,n,
    λj·if ([i,j] ∈ fixed_row_number([n,n],w))
    then times(element(A,[i,j]),element(V,[j]))
    else 0.0))
```

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



```
fun mvmult = generate(n,
  λi·reduce(+,0.0,
    row(fixed_row_number([n,n],w),i),
    λj·times(element(A,[i,j]),element(V,[j]))))
```

where the function `row` returns the set of indices of non-zero elements in a specified row. This expression is more efficient in two ways when compared with the expression at the end of phase 1: there are no checks for zero and non-zero elements; and the reduction is over only the w non-zero elements in a row.

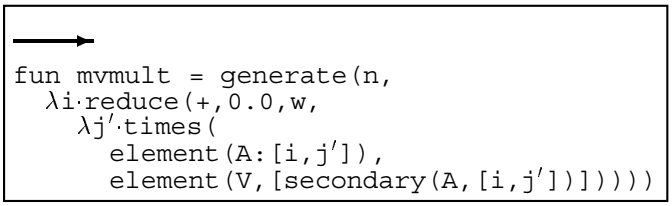
Phase 3: mapping

The sparse matrix is mapped onto primary and secondary stores, according to the rules:

$$[i,j] \rightarrow [i, locate(shape,[i,j])]$$

and the inverse

$$[i,j'] \rightarrow [i, secondary([i,j'])] \ .$$



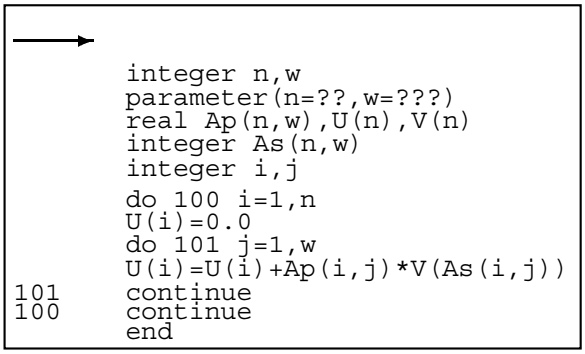
```
fun mvmult = generate(n,
  λi·reduce(+,0.0,w,
    λj'·times(
      element(A:[i,j']),
      element(V,[secondary(A,[i,j'])]))))
```

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 index set 1 to w rather than over the set of indices of the non-zero elements in a particular row.

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 idiosyncrasies of the application.

4. Imperative Implementation
This stage of the derivation tailors the specification for execution on a particular computer—the output language may be either Fortran or 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 [6].

Implementations for other computer architectures are possible, such as an implementation tailored for the AMT DAP 510 array processor and a partitioned MIMD solution [7, 8].

6 Results

Matrix-vector multiplication is a simple operation, and the derived code presented above is clearly well tailored for sparse matrices. The derived code for more complex examples such as the conjugate gradient algorithm is not so amenable to scrutiny (it is designed for compilation, not for reading). To show that the tailoring is still effective, we compare the execution times of the sparse implementation with the execution times of a dense implementation.

Figure 5 displays execution times for both the standard, dense version and the sparse version of the conjugate gradient algorithm on a sequential computer (a NeXT workstation) and on a CRAY vector machine. For the sparse version, each row contained 2% non-zeros in random positions.

7 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 implementation is part of a family of

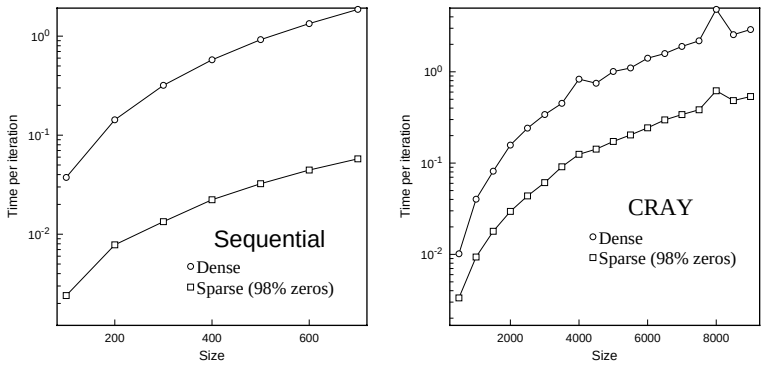


Fig. 5. Execution times for conjugate gradient

implementations, all derived from the same specification. Each implementation is tailored for the particular architecture on which the algorithm will be executed 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 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. A given 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, available collections of derivations and transformations provide 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 devoted to understanding the programming forms that execute well on the CRAY. Once this effort has been expended, of course, the derivation may be used with many specifications and without the user needing to understand the transformations. 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 lie between an abstract functional specification and some implementation model. These models can then be encoded as transformations. The identification of intermediate models facilitates the 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 Fig. 4).

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, IEEE Trans. Software Eng., 10 (5): 574-88 (Sept.) 1984.
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. *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.
6. *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.
7. *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.
8. *Deriving Distributed MIMD Implementations from Functional Specifications*, JM Boyle, M Clint, S Fitzpatrick and TJ Harmer, Proceedings of *ParCo '93* pp44-48, 7-10 September 1993, Grenoble, France.
9. *Computer program for solution of large sparse unsymmetric systems of linear equations*, JE Key, Int. J. Numer. meth. Eng. 6, pp497-509.
10. *Functional Programming using Standard ML*, A. Wilström, Prentice Hall, London 1987.
11. *Sparse Matrix Technology*, Sergio Pissanetsky, Academic Press, 1984.
12. *Parallel Algorithm Derivation and Program Transformation*, editors R Paige, J Reif and R Wachter, Kluwer Publishing, 1993.
13. *A Study of Sparse Matrix Representations for Solving Linear Systems in a Functional Language*, RL Wainwright and ME Sexton, Journal of Functional Programming, 2(1), pp61-72.
14. *Parallel Decomposition of Matrix Inversion using Quadtrees*, DS Wise, Proc. Int. Conf. on Parallel Processing, 1986, pp92-99.

This article was processed using the LaTeX macro package with LLNCS style