A Family of Data-Parallel Derivations *

M Clint[1], Stephen Fitzpatrick[1], T J Harmer[1], P L Kilpatrick[1] and J M Boyle[2]

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

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, inevitably, achieving disappointing execution performance (when compared to the architecture's theoretical execution performance); or
- writing the algorithm implementation in an architecture-dependent way and thereby 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 increased 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 (a subset of the SML programming language[8]).

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, we intend our functional specifications to be a clear statement of the algorithm that captures its essence without consideration

* 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

for execution efficiency. Such a specification executes prohibitively slowly using available functional language compilers (if such a compiler is available for the architecture at all), 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, normally expressed in some dialect of Fortran, from the specification. Many imperative implementations can be derived from a single specification; each implementation is tailored for the hardware to be used and the data to be manipulated.

The derivation of an implementation from a specification is performed by automatically applying *program transformations*. A 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 transformation being 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 [6].

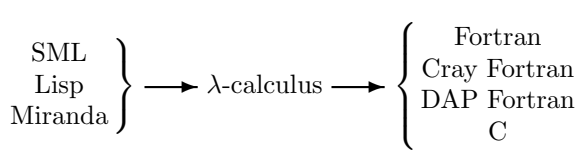
2 A Family of Transformational Derivations

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 is first 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 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). 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 1 is a (somewhat simplified) illustration of the relationships among various intermediate forms created by such sub-derivations.

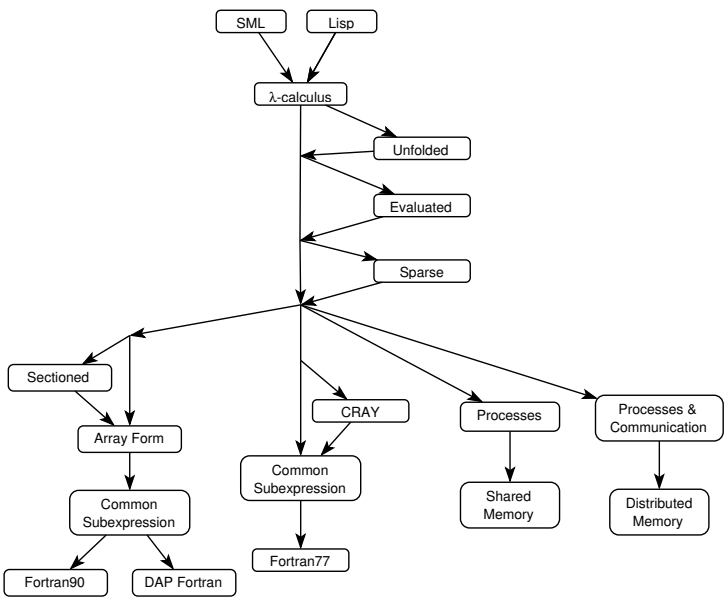


Fig. 1. A family of derivations

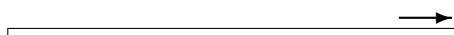
3 Example

The following figures illustrate some of the forms that may be derived from a specification during the implementation process. We use matrix-vector multiplication as an example.

- The Unfolded form is a simple, function form obtained by removing the 'syntactic sugar' from the SML form and by unfolding definitions.
- Details of the implementation of the Unfolded form in sequential Fortran are given in [2]. For this example, the sequential Fortran requires no further modification for vectorization by the CRAY compiler.
- See [6] for the derivation of the DAP Fortran form.
- The Sparse forms (for a tridiagonal matrix) are produced by first applying a sub-derivation that optimizes calculations involving a sparse matrix, and then applying the sequential, CRAY or DAP sub-derivations.

```
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
   = reduce(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))
```

SML specification of matrix-vector multiplication



```
generate ([n]) (λi·reduce ([n])
  (λj·real.times (element (A) ([i,j])) (element (V) ([j])))
  (real.plus) (0.0)
)
```

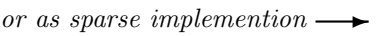
Unfolded form


```
DO i=1,n,1
 AV(i)=0.0
 DO j=1,n,1
  AV(i)=AV(i)+A(i,j)*V(j)
 ENDDO
ENDDO
```

Sequential/CRAY implementation

```
AV=sumc(A*matr(V,n))
```

DAP implementation

or as sparse implemention 

```
AV(1)=A(1,2)*V(1)+A(1,3)*V(2)
DO i=2,n-1
  AV(i)=A(i,1)*V(i-1)
     +A(i,2)*V(i)+A(i,3)*V(i+1)
ENDDO
AV(n)=A(n,1)*V(n-1)+A(n,2)*V(n)
```

Sparse sequential/CRAY implementation

```
AV=A1*shrp(V)+A2*V+A3*shlp(V)
```

Sparse DAP implementation

4 Results

The technique of deriving imperative implementations from functional specifications has proven surprisingly effective in practice. In [5] we discuss a variation on the basic derivation that targets the CRAY vector architecture; the implementation derived from a functional specification for a hyperbolic PDE solver runs slightly faster than its handwritten counterpart. Similarly, in [9] the implementation of an eigenvector algorithm derived from the functional specification

achieves parity with a hand-crafted version for the AMT DAP 510. While achieving high performance in the derived code for one example might be an accident, we have now done so on enough examples in widely varying problem areas to demonstrate that our approach has general validity.

5 Conclusion

We have outlined 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 to which further sub-derivations may be added 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, 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 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. A Practical Functional Program for the Cray X-MP, Journal of Functional Programming, 2(1), Pages 81-126, January 1992.
6. 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.
7. 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.
8. Functional Programming using Standard ML, Wilstöm, A, Prentice Hall, London 1987.
9. 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.
10. The Calculi of Lambda-Conversion, A. Church, Annals of Mathematics Studies, No. 6, Princeton University Press.
11. American National Standard Fortran, X3.9 — 1978 (FORTRAN 77) American National Standards Institute, 1430 Broadway, New York, NY 10018, U.S.A.
12. DAP Series: FORTRAN-PLUS enhanced, man102.01.

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