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

Stephen Fitzpatrick
Terence J. Harmer
Department of Computer Science
The Queen's University of Belfast
Belfast, Northern Ireland, UK

James M. Boyle
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, USA

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
Clarity V Efficiency
High-level, architecture-independent programs
- Easier to construct
- Easier to understand
- Portable

Efficient programs
- Tailored to particular machine:
  - non-portable
- Awash with details
- Difficult to construct
- Difficult to understand
Example: Transpose of a Matrix

Definition: transpose $A^T$ of $m \times n$ matrix $A$ is an $n \times m$ matrix such that

$$\forall i, j : A^T[i,j] \equiv A[j,i]$$

High-level implementation

```plaintext
function transpose(A,m,n) = generate([n,m],fn(i,j)=>A[j,i])
```

Efficient sequential implementation for square matrix ($m=n$)

```plaintext
SUBROUTINE transpose(A,n)
  DO i=1,n
    DO j=i+1,n
      t := A[i,j]
      A[j,i] := t
    END
  END
END
```
Our resolution
Programmer constructs specification; implementation *automatically* derived.

**Specification language**
Functional programming language
- Mathematically based
- Simple semantics: easily understood
- Useful mathematical properties
- Executable prototypes

**Implementation language**
Whatever required by implementation environment; usually version of Fortran or C.
- Efficient
- Good vendor support
- More convenient than machine language

**Derivation by program transformation**
Program Transformations

Program rewrite rules:

\[ \text{pattern} \rightarrow \text{replacement} \]

All occurrences of \textit{pattern} in program changed to \textit{replacement}.

- Achieves a small, local change
- Based on formal properties
  - Clearly preserves meaning of program
- Formally defined in wide spectrum grammar
- Formal proof possible

Derivations

Sequences of transformations

- Complete metamorphosis through many applications of many transformations
- Automatically applied by TAMPR system
Family of Derivations

Derivation performed in steps

- *Sub-derivations*
- *Intermediate forms* between specification and implementation languages

For example:

\[ \text{SML} \rightarrow \lambda\text{-calculus} \rightarrow \text{Fortran77} \]

Same intermediate form for:

- other specification languages
- other architectures/implemention languages

Combinations have included:

\[
\begin{align*}
\text{SML, Lisp, Miranda} & \quad \rightarrow \quad \lambda\text{-calculus} \\
& \quad \rightarrow \quad \text{Fortran, CRAY Fortran, DAP Fortran, C}
\end{align*}
\]
Other sub-derivations/intermediate forms for:
- Optimization e.g.
  - function unfolding
  - common subexpression elimination
- Tailoring for particular forms of data
  e.g. sparse matrices
Sparse Matrices

We consider a matrix which has a fixed number of non-zero elements per row:

\[
\begin{bmatrix}
0 & 1 & 0 & 0 & 2 \\
3 & 0 & 0 & 0 & 0 \\
0 & 4 & 5 & 0 & 0 \\
0 & 0 & 0 & 6 & 7 \\
8 & 0 & 9 & 0 & 0
\end{bmatrix}
\rightarrow
\begin{bmatrix}
1 & 2 \\
3 & 0 \\
4 & 5 \\
6 & 7 \\
8 & 9
\end{bmatrix}
\rightarrow
\begin{bmatrix}
2 & 5 \\
1 & 2 \\
2 & 3 \\
4 & 5 \\
1 & 3
\end{bmatrix}
\]

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

- 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.
  - An illustration where tailoring for a compressed data representation and a parallel computer is performed.
  - Other representations are possible by substituting the mapping phase of the transformations (later).
Example
Matrix-vector multiplication

\[
\begin{bmatrix}
\ldots \\
\ldots \\
1 & 2 & 3 & 4 \\
\ldots \\
\ldots \\
\ldots \\
\end{bmatrix}
\times
\begin{bmatrix}
a \\
b \\
c \\
d \\
\end{bmatrix}
= 
\begin{bmatrix}
\ldots \\
1a + 2b + 3c + 4d \\
\ldots \\
\ldots \\
\end{bmatrix}
\]

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

SML specification

Data parallel functions
- `generate` defines vector/matrix
- `reduce` combines elements of vector/matrix into single value
Derivation Stages

1. Abstract Functional Specification

fun times: real vector × real vector → real vector
   = λ.U,V.generate(size(U),
     λ.i.real.times(element(U,i),element(V,i)))

fun sum: real vector → real
   = λ.V.reduce(+,0.0,size(V),λ.i.element(V,i))

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

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

2. Unfolding and Static Evaluation

fun mvmult = generate(n,
   λ.i.reduce(+,0.0,n,
     λ.j.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.
Derivation Stages - Continued

3. Sparse Specialization

Phase 1: annotation

Explicitly distinguish non-zero elements from zero elements.

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

**fixed_row_number** is the set of significant indices of the matrix.
Derivation Stages – Continued

3. Sparse Specialization

Phase 2: optimization

\[
\text{fun mvmult = generate(n,} \\
\quad \lambda . i . \text{reduce}(+, 0.0, n,} \\
\quad \quad \lambda . j . \text{if } ([i, j] \in \text{fixed_row_number}([n, n], w))} \\
\quad \quad \quad \text{then times(element}(A, [i, j]), \text{element}(V, [j]))} \\
\quad \quad \quad \text{else 0.0})
\]

\[
\text{fun mvmult = generate(n,} \\
\quad \lambda . i . \text{reduce}(+, 0.0,} \\
\quad \quad \text{row(fixed_row_number}([n, n], w), i),} \\
\quad \quad \lambda . j . \text{times(element}(A, [i, j]), \text{element}(V, [j])))
\]

Function row returns the set of indices of non-zero elements in a specified row.
3. Sparse Specialization

Phase 3: mapping

Provide a compact realization for sparse matrices.

\[ [i, j] \rightarrow [i, \text{locate}(\text{shape}, [i, j])] \]

and the inverse

\[ [i, j'] \rightarrow [i, \text{secondary}([i, j'])] \].

```haskell
fun mvmult = generate(n,
  \(i\).reduce(+, 0.0, w,
    \(j\).times(
      element(A: [i, j']),
      element(V, [secondary(A, [i, j'])]))))
```
Derivation – Continued

4. Imperative Implementation

```plaintext
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
```
Conjugate Gradient Definition

To solve $Ax=b$, where $A$ is a positive definite symmetric $n\times n$ matrix $x$:

Set an initial approximation vector $x_0$,
calculate the initial residual $r_0 = b - Ax_0$,
set the initial search direction $p_0 = r_0$,
then, for $i=0,1,\ldots$,
(a) calculate the coefficient $\alpha_i = p_i^T r_i / p_i^T A p_i$,
(b) set the new estimate $x_{i+1} = x_i + \alpha_i p_i$,
(c) evaluate the new residual $r_{i+1} = r_i - \alpha_i A p_i$,
(d) calculate the coefficient $\beta_i = -r_{i+1} A p_i / p_i^T A p_i$,
(e) determine the new direction $p_{i+1} = r_{i+1} + \beta_i p_i$,
continue until either $r_i$ or $p_i$ is zero.

from Modi, p152
Conjugate Gradient Specification

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
Conjugate Gradient - Derived

```fortran
integer n, w
parameter(n=SIZE, w=2*n/100)
real x(n), q(n), p(n), b(n)
integer cnt, k, As(n, w), i, j
real Ap(n, w), r(n), r1(n), alpha, atq(n), beta, g63, rr

continue
rr = 0.0
do 210 j = 1, n
   rr = rr + r(j) * r(j)
210 continue
if (sqrt(rr) .lt. 1E-14) then
   goto 500
else
   alpha = 0.0
   do 220 i = 1, n
      alpha = alpha + q(i) * q(i)
   220 continue
   alpha = rr / alpha
   do 230 i = 1, n
      atq(i) = 0.0
   230 continue
   do 240 i = 1, n
      do 240 k = 1, w
         atq(As(i, k)) = atq(As(i, k)) + Ap(i, k) * q(i)
   240 continue
   do 260 j = 1, n
      r1(j) = r(j) - atq(j) * alpha
   260 continue
   beta = 0.0
   do 270 j = 1, n
      beta = beta + r1(j) * r1(j)
   270 continue
   beta = beta / rr
   cnt = cnt + 1
   do 280 j = 1, n
      x(j) = x(j) + p(j) * alpha
   280 continue
   do 290 j = 1, n
      p(j) = r1(j) + p(j) * beta
   290 continue
   do 300 i = 1, n
      r(i) = r1(i)
   300 continue
   do 340 j = 1, n
      g63 = 0.0
      do 330 k = 1, w
         g63 = g63 + Ap(j, k) * r1(As(j, k))
      330 continue
      q(j) = g63 + q(j) * beta
   340 continue
   goto 200
endif
500 continue
end
```
Results

![Graph](image)

Conjugate Gradient
**Assessment**

Techniques have been applied to significant algorithms for sequential, vector, array and shared-memory architectures.

Comparing with independent, manually constructed implementations:
- Derived implementations similar.
- Execution performance equal or better.

Techniques are being extended for yet more complex algorithms, for distributed and shared memory parallel architectures and for further special data structures.
Conclusion-Summary

With derivational approach, programmer

- develops implementation techniques
- encodes techniques as derivations

Reusability
- Multiple specifications
- Multiple implementations of each
- Algorithm modified: modify specification and re-apply derivation

- it is possible to experiment with different implementations easily.

Extensibility
- New optimization technique
- or new architecture
- or new data representation: 'slot in' new sub-derivation

Transferability
- Sub-derivation requires no expertise to use
- One programmer may use another's work

Correctness
- Correctness of transformations implies correctness of implementation