Deriving distributed MIMD implementations from functional specifications

J.P. Wray[a], S. Fitzpatrick, M. Clint, P.L. Kilpatrick

[a]Department of Computer Science,The Queen's University of Belfast,
Belfast BT7 1NN, Northern Ireland, UK

The construction of reliable and efficient scientific software for execution on novel computer architectures is a demanding and error-prone task. By automatically generating efficient parallel implementations of functional specifications using program transformation, many of these problems may be overcome. The basic method is discussed, with particular reference to its extension to generate implementations for execution on distributed memory MIMD machines.

1. INTRODUCTION

Many commonly occurring problems in science and engineering involve the application of a single operation to each element of a very large vector or matrix. Such problems are eminently suitable for fast solution on both SIMD and MIMD parallel computers. However, the construction of programs in the high level languages currently available for these machines suffers from several drawbacks:

- constructing reliable parallel programs is a very demanding task—for example, the programmer must consider non-determinism and deadlock;
- parallel programming languages are not, in general, portable; consequently porting an existing program to a new machine may necessitate that the program be completely rewritten.
- parallel programs are often hard to read and are difficult to prove correct.

By automatically generating multiple realisations of a program from a single machine-independent specification these disadvantages can be eliminated without any sacrifice in efficiency.

2. FUNCTIONAL SPECIFICATIONS

In the work described in this paper, machine-independent functional specifications are expressed in a small subset of standard ML [6]. A library of numerical mathematical primitives is provided for the most common matrix operations. For scientific applications ML specifications are easy to write and understand because they closely resemble the original mathematical descriptions[5, 3].

In order to illustrate the functional specification style, we outline the functional specification of a significant problem in applied mathematics:

2.1. Specification of the multigrid algorithm for Poisson's problem

We require to solve the equation

$Lu = f$

where L is an approximation, on a given step length, to the Poisson operator, u is a grid over which the equation is to be solved (each element of u is a function of x and y), and f is a given $n \times n$ grid (each element of which is a function of x and y). The problem is assumed to have Dirichilet boundary conditions, ie. the values of u are known on all boundaries. The solution is given by

`solve(L,Jc,f,bv,epsilon)`

where `L` is the operator, `Jc` is the Jacobian of the operator, `f` is the given grid, `bv` is the boundary value function, `epsilon` is a measure of the required accuracy, and `solve` is defined as

```
fun solve(L:grid*int*int->real, Jc:jacobian, f:grid,
                bv:int*int->real, epsilon:real) =
        let
                val n = size(f);
                val h = 1.0/real(n-1);
                val u0 = create(n, bv);
                val hsqf = scale(f, h*h);
                val epsilon0 = epsilon*residual(L, hsqf, u0)
        in
                iterate(L, Jc, u0, hsqf, epsilon0)
        end;
```

The functions `size`, `create`, and `scale` are library functions which return the dimension of a grid, generate a grid from an index function, and scale each element of a grid, respectively.

The function `iterate` repeatedly applies a function `mg` (multigrid) until an acceptably accurate approximation to the result is obtained:

```
fun iterate(L:grid*int*int->real, Jc:jacobian, ui:grid,
                f:grid, epsilon:real) =
        let
                val res = residual(L, f, ui);
        in
                if res < epsilon
                then ui
                else iterate(L, Jc, mg(L, Jc, ui, f), f, epsilon)
        end;
```

The function `residual` returns the error in an approximation to the result.

A single application of the multigrid method is defined as follows:

```
fun mg(L:grid*int*int->real, Jc:jacobian, u:grid, f:grid):grid =
        let
                val relaxu = relax(L, Jc, u, f);
                val csize = (Grid.size(u)+1) div 2
        in
                if size(u) = 3
                then relaxu
                else relax(L, Jc,
                          interpolate( mg(L, Jc, constant(csize, 0.0),
                                       restrict(defect(L, f, relaxu))),
                                     relaxu),
                          f)
        end;
```

The functions `relax, restrict,` etc, are all specified in the same style. This top-down derivation continues until all functions are expressed in terms of primitive functions or previously defined functions.

The important thing to note about this style of specification is its transparency—the functions specify the computation to be carried out, without adding any implementation details. Furthermore, the specification is not biased towards a particular implementation language or architecture. As well as being amenable to formal analysis, the specification can be executed for testing purposes. However, direct execution of the specification is extremely inefficient in comparison with an imperative implementation. We outline in the next section how an efficient imperative implementation may be generated from the functional specification.

3. DERIVATION OF IMPERATIVE CODE

An efficient imperative implementation (expressed in Fortran or C, say) of the specification may be obtained by the automatic application to the specification of a sequence of transformations. The transformations are applied using the TAMPR transformation system[4]. Each transformation is defined by a syntactic pattern replacement rule. For example, a transformation which applies the distributive law might be expressed as follows:

```
Transform:
 (<factor>"1" <addop>"1" <factor>"2") <multop>"1" <factor>"3"
  ==>
  <factor>"1" <mult op>"1" <factor>"3" <add op>"1"
  <factor>"2" <mult op>"1" <factor>"3"
```

Note that `<factor>`, `<addop>` and `<multop>` are non-terminals of the grammar being transformed and not entities defined by the transformation tool. This rule would, for example, transform any expression of the form `(A+B)*C` (where `A,B,` and `C` belong to the syntax class `<factor>`) into the form `A*C + B*C`.

A single application of a transformation effects a small change to the specification; consequently it is not difficult to prove that correctness is preserved. The cumulative

effect of the application of many transformations can produce an efficiently executable Fortran or C version of the original specification. The main steps that are common to all derivations are as follows:

1. The specification is prepared in order to simplify later transformations. For example, bound variables are renamed in order to eliminate the possibility of name clashes, and multiple-variable lambda expressions are converted to nests of single variable lambda expressions. At this stage the transformed specification is in a standard functional form. Employing a standard form allows alternative functional specification languages (eg. LISP, Miranda) to be used as the starting point: transforming any functional language into standard form is straightforward.

2. The specification is simplified using function unfolding and folding.

3. The prepared specification is manipulated into a form in which all function applications have simple arguments (either variables or constants). The transformations which effect this change are based on algebraic identities of the λ-calculus.

4. Transformations based on the distributive laws change assignments involving complicated functional expressions into sequences of assignments involving only simple expressions.

5. Function evaluation is implemented using a mechanism suitable for the target architecture; for example, a simple push-pop stack.

6. The implementation is completed by inserting imperative equivalents for certain functional primitives.

This is a somewhat simplified description: the complete process is described in more detail in [4].

It is important to note that this approach is not a monolithic process such as compilation. The user can inspect the transformed specification at any stage of the process, and is free to modify the process by inserting additional transformations or modifying/deleting existing ones. Therefore the entire process can be tailored to the user's specific requirements. In particular, by inserting a few architecture-specific transformations at well-chosen points in a derivation, an appropriate new implementation of the original specification can be quickly generated. The method has already been successfully applied to generate efficient implementations for sequential, shared-memory multiprocessor, vector, and distributed array processor machines[4, 1, 3, 2]. In all cases the performance of the automatically generated code matched that of hand-crafted versions.

In the remainder of this paper the extension of the method to allow the generation of code for distributed memory MIMD machines is described.

4. TRANSFORMATIONS FOR MIMD ARCHITECTURES

4.1. Implementation Strategy

All data parallel vector and matrix functions in a specification are (automatically) unfolded and expressed in terms of a primitive function, `generate`. This function takes

two arguments: a shape expression (which defines the structure of the result) and a generating function (which defines the computation to be carried out for each element of the structure). For example, the application

$$\texttt{generate}([\texttt{1} : \texttt{n}, \texttt{1} : \texttt{n}], \texttt{fun}(\texttt{i}, \texttt{j}) => \texttt{a}(\texttt{i}, \texttt{j}) + \texttt{b}(\texttt{i}, \texttt{j}))$$

returns the matrix sum of `a` and `b`. On a sequential machine, the result of this expression is calculated by evaluating the expression `a(i,j) + b(i,j)` for each element of the matrix. The result can be efficiently evaluated on a distributed memory machine if the matrices `a` and `b` are partitioned over the available processing elements so that, for each `i` and `j`, elements `a(i,j)` and `b(i,j)` are stored on the same processing element; then the individual portions of the result can be evaluated in parallel.

However, it is frequently necessary to evaluate expressions such as

$$\texttt{generate}([\texttt{1} : \texttt{n}, \texttt{1} : \texttt{n}], \texttt{fun}(\texttt{i}, \texttt{j}) => \texttt{F}(\texttt{a}(\texttt{i} + \texttt{1}, \texttt{j}), \texttt{b}(\texttt{i}, \texttt{j} - \texttt{1}))).$$

In this case, assuming the same partition of data structures as before, there is no guarantee that the elements `a(i+1,j)` and `b(i,j-1)` are local to the processor that is required to calculate the `(i,j)`th element of the result. Therefore, in general, each processor must obtain some non-local data, and transmit data to other processors, before it can evaluate its portion of the result.

It is convenient to adopt a vertical "striped" partition of data structures. For example, a structure of dimension (1:16, 1:16) could be partitioned over a four-element processor array in segments of dimension (1:16,1:4), (1:16,5:8), (1:16,9:12) and (1:16,13:16) respectively. The advantage of using stripes is that the north and south neighbours of each element will always be stored on the same processor. Thus, for example, the expression

$$\texttt{a}(\texttt{i}, \texttt{j}) + \texttt{b}(\texttt{i} + \texttt{1}, \texttt{j}) + \texttt{c}(\texttt{i} - \texttt{1}, \texttt{j})$$

could be evaluated locally on each processor without any inter-processor communication.

We assume that only nearest-neighbour data transfers are required. Many scientific applications involve only generating functions of this type, and those that do not can be re-expressed in this form.

Thus, evaluation of a `generate` involves, in general, a message-passing phase in which neighbouring processing elements exchange the values of their leftmost and rightmost columns, and a computation phase in which the individual segments of the result are calculated in parallel.

4.2. Automatic Generation of a MIMD Implementation

The initial "Fortranizing" transformations will ensure that all instances of data parallel primitives (`generate`) appear on the right hand side of assignment statements, *viz*

$$\texttt{g999} = \texttt{generate}([\texttt{1} : \texttt{n}, \texttt{1} : \texttt{n}], \texttt{fun}(\texttt{i}, \texttt{j}) => \texttt{F}(\texttt{a}(\texttt{i} + \texttt{1}, \texttt{j}), \texttt{b}(\texttt{i}, \texttt{j} - \texttt{1}))).$$

We now consider, in a little more detail, how a distributed MIMD implementation of this assignment is derived. For the sake of clarity, a syntax slightly different from that used in practice is employed.

The first step is to make explicit the data that will be required by each processing element. This is achieved using transformations which abstract each occurrence of a variable out of the body of the generating function. After the application of these transformations the assignment becomes

```
g999 = generateusing((a(i + 1, j), b(i, j − 1)), [1 : n, 1 : n],
                                  fun(i, j) => F(a(i + 1, j), b(i, j − 1))).
```

The first argument of `generateusing` defines a list of matrix elements that are required to evaluate the generating function for each element of the result. These elements may or may not be local to the processing element which requires them. The second and third arguments are the two arguments of the original `generate`.

A recursive transformation then generates the abstract message-passing and computation phases necessary to implement the assignment:

```
call obtainifnecessary(a(i + 1, j), i, j);
call obtainifnecessary(b(i, j − 1), i, j);
g999 = localgenerate([1 : n, 1 : n], fun(i, j) => F(a(i + 1, j), b(i, j − 1)))
```

Note that each of the stages described so far is common to *any* distributed MIMD implementation. Only at this point is it necessary to introduce language-specific transformations.

The next stage is to substitute the actual message passing code required for the two calls to `obtainifnecessary`. Two subroutines, `getfromeast` and `getfromwest`, pre-written in the target language are required for this. The subroutine `getfromeast` causes each processor to pass the leftmost column of its portion of a structure to its west neighbour, and obtain the corresponding values from its east neighbour; subroutine `getfromwest` is similar. The first call to `obtainifnecessary` above is transformed to the empty statement since, for each `i` and `j`, the element `a(i+1,j)` is local to the processor which is calculating the `(i,j)`th element of the result. The second call, however, necessitates data transfer from west to east and is therefore transformed to

```
call getfromwest(b, firstcolumn, lastcolumn);
```

(Note that `firstcolumn` and `lastcolumn` are local constants which define a processor's share of the matrix.)

Finally, the `localgenerate` is implemented using a nested loop, the bounds for the inner loop being defined by the constants `firstcolumn` and `lastcolumn`. Other transformations introduce the declarations for loop-counter variables. Thus we obtain:

```
integer i, j,
        a(1 : n, firstcolumn : lastcolum), b(1 : n, firstcolumn : lastcolum)
call getfromwest(b, firstcolumn, lastcolumn);
do i = 1, n
    do j = firstcolumn, lastcolumn
        g999(i, j) = F(a(i + 1, j), b(i, j − 1))
    end do
end do;
.....
```

5. A DISTRIBUTED IMPLEMENTATION OF MULTIGRID

The transformations described in the previous section provide a basis for the generation of a variety of distributed implementations of a functional specification. They have been applied to produce an implementation of the multigrid specification outlined in Section 2.1 for execution on a ring of transputers. The implementation language used is Meiko Fortran 77. In order to produce this implementation, it was necessary to write the skeleton code for setting up channel names, etc, and to code the subroutines `getfromeast` and `getfromwest` in terms of the message passing primitives of the language. Of course, this only needs to be done once, so implementations of other functional specifications may be generated without further coding by hand. The code for `getfromeast` is given below:

```
 SUBROUTINE getfromeast(firstcolumn,lastcolumn,columnsize,
     *                        dimension,slice,ename,wname,transport,
     *                        processid)


#include<csn/csn.inc>
#include<cs.inc>
#include<csn/names.inc>

       PARAMETER (dimens=16)
       integer westslaveid,status,firstcolumn,lastcolumn,
     *         dimension,transport,processid,columnsize,
     *         slice(dimens,0:*)
       character*15 ename, wname
       logical notonleft, notonright
       notonleft= firstcolumn.NE.1
       IF (notonleft) THEN
         status=csnlookupname(westslaveid,wname,.TRUE.)
         IF (status.NE.CSNOK) THEN
           CALL csabort('slave1: cannot look up '//wname,-1)
         ENDIF
         CALL csntx(transport ,0, westslaveid, slice(1,0),columnsize)
       ENDIF
       notonright= lastcolumn.NE.dimension
       IF (notonright) THEN
         CALL csnrx(transport,CSNNULLID,
     *              slice(1,lastcolumn-firstcolumn+2)
     *                ,columnsize)
              ENDIF
       RETURN

       END
```

Even if adequately commented, code of this nature is quite difficult to read, write or modify due to the low-level nature of the communication primitives it uses. However, since

calls to this routine are automatically generated by transformations, these problems are of no concern to the user, whose view of the implementation is provided by the functional specification.

As expected, the performance of the distributed imperative implementation far exceeds that achieved by executing the functional specification, although it is not yet as good as that of the best handwritten version. However, the automatically generated code can be inspected to determine what optimizations might be applied to improve efficiency, and these optimizations may then be applied automatically to the program using further transformations.

6. CONCLUSION

It has been demonstrated that it is possible to derive automatically an (admittedly primitive) imperative implementation of a functional specification for a distributed memory machine. This derivation is the skeleton for further development: in particular, by considering other topologies with more sophisticated data transfer requirements and hybrid architectures. The strength of the transformational approach is that such tailoring is a natural part of the software development process.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the work of Dr Jim Boyle in developing the TAMPR system, and Dr Terry Harmer for providing many helpful suggestions. This work is funded by the SERC under grant No. GR/G 57970.

REFERENCES

1. J.M. Boyle. Program adaptation and program transformation. In R. Ebert, J. Lueger, and L. Goecke, editors, *Practice in Software Adaptation and Maintenance*, pages 3–20. North-Holland Publishing Co., Amsterdam, 1980.
2. J.M. Boyle, M. Clint, S. Fitzpatrick, and T.J. Harmer. The construction of numerical mathematical software for the AMT DAP by program transformation. In L. Bougé, M. Cosnard, Y. Robert, and D. Trystam, editors, *Parallel Processing: CONPAR92-VAPP V (LNCS 634)*, pages 761–767. Springer-Verlag, Berlin, 1992.
3. J.M. Boyle and T.J. Harmer. A practical functional program for the Cray X-MP. *Journal of Functional Programming*, 2(1):81–126, 1992.
4. J.M. Boyle and M.N. Muralidharan. Program reusablility through program transformation. *IEEE Transactions on Software Engineering*, SE-10(5):574–588, 1984.
5. J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.
6. A. Wilstöm. *Functional Programming using Standard ML*. Prentice Hall, London, 1987.