Yale University
Department of Computer Science

Efficient Compilation of Array Expressions for the Connection Machine

Luis F. Ortiz and Ron Y. Pinter

YALEU/DCS/TR-720
July 1989

Efficient Compilation of Array Expressions for the Connection Machine[1]

Luis F. Ortiz

Ron Y. Pinter[2]

Department of Computer Science
Yale University
New Haven, CT 06520

July 1989

Abstract

The programming of SIMD machines which strongly support data parallelism, such as the Connection Machine, presents new challenges for efficient code generation. The wide range of data types and possible data layouts as well as the variety of operators for communication, data movement, and arithmetic require major changes in the program whenever small changes in the representation of information are called for. We present a language for parallel processing of arrays that enables the programmer to abstract the algorithmic code away from the underlying data layout, and yet still to achieve nearly optimal machine performance. A prototype compiler for this language, *ALP* (an Array Language for Parallelism), has been implemented for TMC's CM-2 and it generates code compatible with *Lisp Release 5.1.

Keywords: compilation, parallel programming

[1]Work supported in part by ONR grant number N00014-89-J-1906.
[2]On sabbatical leave from the IBM Israel Scientific Center.

1 Introduction

Programming a massively data-parallel machine is not easy. There are many issues the user needs to worry about, from the relatively high level of algorithms and data structures, to the details of inter-process synchronization and network routing directives. Various languages or language extensions allow the programmer to address these concerns explicitly, but their efficiency tends to rely on full and detailed specifications, resulting in long, tedious, and error-prone programming.

We claim that with an appropriate programming model — that provides the proper abstractions for data layout and data parallel operations — efficient compilation of a large family of high-level programs can take place. Such a model requires concentration on the key decision parameters, such as data layout and its distribution in the machine, a very terse notation, and a natural way for putting expressions together. The compiler can then fill in the details, even make some routine optimizations (using well-studied algorithms, as found in [1]), and generate the voluminous programs that so far have been hand-coded.

In this paper, we present *ALP*, a rather simple array language for data parallelism, which demonstrates this concept for data parallel programming of the Connection Machine. *ALP* supports a programming model that is very close to the idealized view presented originally by Hillis [3], in which *xectors* are arrays whose elements can be distributed across the machine. Xectors can be combined into expressions and operated on with parallel extensions of scalar operators, indexing, and summary operators (such as reduce and scan which are implemented as parallel-prefix [2, 4] computations).

An idealized xector can be implemented in a number of different ways, and the particular choice may effect the efficiency a great deal. Therefore, *ALP* provides the mechanism of *declarations* which allow the user to specify (quite painlessly) the exact layout of each object in the machine. This separation of the layout data from the algorithmic portions of the program (*i.e.* the expressions) liberates the user from the need to change any of the program body when experimenting with different data layouts, which otherwise is a rather laborious process.

An indexing operation in *ALP* has always the same semantics but may be instantiated in radically different ways depending on the layout specifications. Therefore, we find it is a particularly useful abstraction for dealing with various aspects of intra- and inter-process data movement, allowing the user to write the similar expressions for sends and receives as well as local data references.

We start with a summary of the machine's architecture and its programming model in Section 2. Next, in Section 3, we present (a somewhat partial version of) *ALP*, followed by a description of its compiler in Section 4.. Section 5 briefly surveys some extensions to the language and how they are handled, before we make some concluding remarks (in Section 6) on the contributions we have made.

2 Machine Architecture and the Programming Model

The Connection Machine [3, 6] is a hypercube-connected network of clusters of 32 processors, each of which is connected to 64K-by-32-bit words of random access memory, a 32-bit floating point processor (FPP), and an FPP interface chip. Fully configured machines have 64K processors, but smaller configurations (32K, 16K, and 8K) are also available. Message-passing between arbitrary processors is handled by a router, which is implemented almost completely in hardware. The host processor provides all the scalar processing capability and sends instructions to the CM via a host interface unit, which passes the instruction stream to a microcontroller that actually decodes the instructions and manages the CM processors.

The Connection Machine is used as a peripheral processor to a host machine — such as a Symbolics 36xx series machine, a Sun 4/280, or a VAX with a BI bus interface — which supports the various host languages [6], such as Lisp-, C- , and Fortran-PARIS (the assembly language for the Connection Machine), C* (a C++ influenced extension to C), *Lisp (an extension of Common Lisp), and *Fortran (providing some of the proposed array extensions for Fortran 8X)[5].

One possible programming model, as promoted by Hillis [3] and to a large degree supported by the above languages, is that in which data entities are organized as xectors; each such value is best thought of as a (one dimensional) vector whose elements reside one per processor, with the processor id being the index of the element. Instructions are either scalar, *i.e.* they apply to every element independently (*e.g.* add one to every entry, subtract the ith element of a from the corresponding element of b), or they apply to the xector as one entity, *e.g.* perform a scan or a reduction operation on a.

In addition, xector elements can be copied (or sent) to other xectors. Sometimes, the elements are to be arranged in the same order, and then the copy operations can be performed locally in each processor (that is — element by element, in parallel), but often a different ordering of elements is desired, *e.g.* each processor wishes to send its value to the processor whose id is 1 more than its own. These permutation operations can be expressed very tersely using indexing of xectors, which may be implemented as a whole range of operators from send and receive to purely local memory references.

The CM also supports virtual processors sets, where each physical processor emulates N virtual processors by iterating over the data element replicated N times in storage, permitting a linear tradeoff between number of processors, speed, and memory space.

In addition, the CM supports contextualization of CM operations, which permits this SIMD machine to store the results of one operation conditionally, depending on the result of a previous operation. Otherwise, constructs such as `if-then-else` would not be possible.

3 The *ALP* Language

The *ALP* programming language is currently implemented as fragments that include array declarations and array expressions (in an augmented Algol-like notation) embedded in a Lisp environment. Syntactically, *ALP* program sections are delineated by the separators #{ and #}. This embedding provides a rich carrier language, which features all the control and procedural structures necessary for general purpose programming. The Lisp development environment also provides a wide set of debugging tools that greatly reduce the time needed to produce correct programs (and prototype the compiler itself). An example of *ALP* code follows:

```
(defun foo ()
    #{
        declare layout default (1 1024)
        declare variable xector, float x,y(4);
        declare variable host, integer v(4),z;
        declare variable xector,integer(12) w;

        x := 1 + (x/z);
        p *= (+/ x)(v;
        x(w) = y[w + 2];
        }#
)
```

3.1 Layaout Declaration

A data type declaration consists of two parts: the virtual (or type specification) information, which describes the logical structure of each data element, and physical (or layout) information, which describes the way each data element is distributed among the active processors. The type specification is similar to its counterparts in other languages (Fortran, Pascal, C, etc.), and includes the base type and the array's dimensions[1]. The layout portion is optional, and gives directives on how the objects are realized in the processor space. The user need only supply the keywords `host` or `xector` and, in the latter case, optionally supply the number of processors used to realize each single copy of the object, (again, optionally) the total number of processors used to keep all copies of the object[2]. This form allows the specification of replicated layouts, but describing the simple extremes (such as a `xector` with one element per processor, no replication) is made easy.

In the above example, x is a `xector` with one element per processor, y is fully replicated at each processor over 1024. The layout named `default` has a special meaning for the compiler, as the layout all objects are put into, if no other layout is explicitly mentioned

[1] For the sake of presentation, in this section we deal only with one-dimensional structures; generalizations to multidimensional objects will be provided in Section 5.

[2] The number of copies is simply the latter number divided by the former; we chose to specify the total number of processors for "cultural" reasons.

in the variable's definition. If no `default` layout has been defined, then it will be deduced from the program context.

In order to illustrate exactly how layout declarations work, take the layouts declared in Figure 1.

```
declare layout alpha (1 8)
declare layout beta  (2 8)
declare layout gamma (3 8)
declare layout delta (* 8)
```

Figure 1: Layout definitions for a `vector(3)`.

The effective layout for a `vector` of length 3 in each of these layouts is as illustrated in Figure 2. Obviously, layout `beta` is not optimal, in terms of storage space, for this variable, but it may result in savings of communication times, if it is used often in computation with another object in the same geometry.

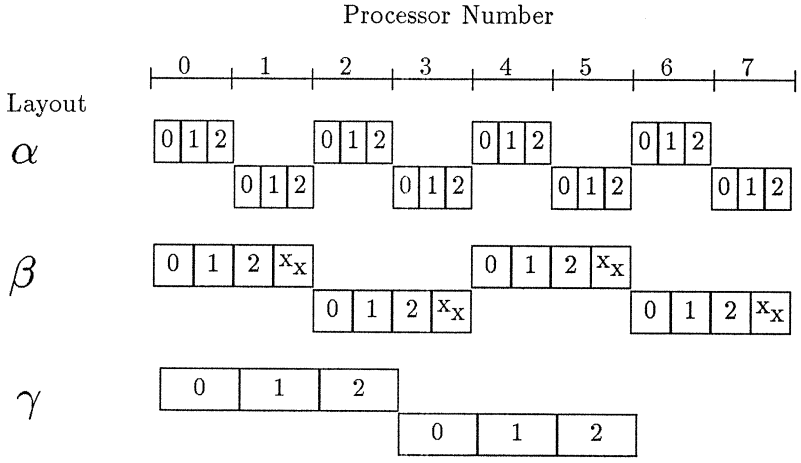

Figure 2: Layout options for a `vector(3)`.

3.2 Type Declaration

The declaration of data types is done in two steps. The first is the definition of a super-type by composing basic data types, properties, layout type, and/or pre-existing data types. The basic data types are `signed`, `unsigned`, `float`, `array`, `vector`, and `number`. Obviously, some of these types are mutually exclusive, such as `float` and `signed`. In addition, there are a few property types available, such as `replicated`, `shared`, `slicewise`, which provide directives to the compiler as to possible implementation methods. The `array` or `vector` types can also be followed by a comma-delimited list to specify the size of the array to be created, as in Fortran.

```
declare type    my-xfloat       xector,float,number;
declare type    my-ffloat       fe,float,number;
declare type    my-arrayf       xector,float,array(3,3);
```

Finally, the actual declaration of variables is done by naming a type followed by a comma-separated list of all the variables to be associated with that type. Each variable name can be followed by an array declaration list similar to the one used in the type specification. For example:

```
declare variable        my-arrayf alpha;
declare variable        my-xfloat x,y,beta(3);
declare variable        my-ffloat z,gamma(4,4);
```

`Alpha` is a variable declared to be a `xector` of 3x3 arrays, `beta` is a `xector` of vectors of length 3, and `gamma` is a `host array` of size 4x4. If the type or layout information used to declare a variable is incomplete, the compiler does not consider it to be an error, and attempts to infer missing information from the program itself.

3.3 Operators

ALP array references come in two formats: parentheses delimited array references are used to express inter-processor data movement while square bracket array references express intra-processor data movement. They are true postfix operators, so they can be applied to expressions as well as variables and can be composed with the other operators that *ALP* provides. In the previous example, a reference to `y(1)` would be to the `array` in processor one, while a reference to `y[1]` would be a `xector float` whose contents in each processor would be the 1-th element of the array local to each processor.

The standard diadic `+, -, *, /, <, <=, ==, =>, >, &, |, ^,` ↑, ↓ and monadic `-, ~` as well as the scan operators `+/, -/, */,` ≡`/,` ↑`/, +\\, -\\, *\\,` ≡`\\,` ↑`\\` in both monadic and dyadic forms are also provided along with two forms of the assignment operator `*=` and `:=`, which are the unconditional and conditional assignment operators respectively. The first left argument to the dyadic scan operations is a boolean operand which when true defines the start of an scanning segment[2].

4 The Compiler

Currently the compiler is implemented in four passes:

1. A lexical analysis phase, where the character stream read by the Lisp reader system is converted into a stream of tokens.
2. A parse phase, where the token stream is parsed and converted into an AST (Abstract Syntax Tree) representing the structure of the program.

3. A resolution phase, where types for all expressions are resolved, and operators selected.

4. A code generation phase, where output code is generated from the AST.

4.1 Lexical Analysis

The lexical analyzer uses a graphical representation of a DFA constructed by converting NDFA into a DFA by state combination and extension. State transitions are sped up by compressing all output transitions from a given state into a table indexed by the character received.

4.2 AST Definition

Conceptually, each node in the AST is a 4-tuple in the following format: `<operator, children, typeset, opset>`. The `operator` is the generic operator being performed at this node in the AST; the `children` is where a list of all the inferior nodes of this node in the AST is recorded; the `typeset` is the set of possible types for this node; and the `opset` is the set of possible specific operators for this node.

4.3 AST Generation

The AST is generated by a standard recursive descent for LR grammars. The original source lines are recorded into the AST to make debugging of syntax errors easier.

4.4 Data Typing

The data type information is represented internally as a triple encoding both virtual and physical attributes, namely the element type, the virtual shape of the array object, and the layout (frame) description. For variables, this information is derived straightforwardly from the declaration; for intermediate values, it is computed by the type arbitration mechanism described below.

In practice, the triple specifications are slightly more complicated, since they include information specifying the size of the element type, as well as various kinds of wildcarding mechanisms to permit the use of types that are not fully specified. For example, `<(integer *), (4 4), (* @)>` could be a type that specifies an integer of any size, the `*` matching any one element, and a computational frame of one or more dimensions, the `@` matching zero or more elements.

4.5 Operator Definitions

Operators in *ALP* are associated with the underlying instructions that the machine supports via generic and specific operators. Generic operators associate a given *ALP* operator with the set of machine instructions that perform that logical operation. Associated with every generic operator is a type propagation function that allows the generation and propagation of type information via that operator.

Specific operations are associated with 5-tuple in the following format: `<input-typeset, output-type, code-f, cost-f, temp-f>`. The `input-typeset` is a list specifying the acceptable input type for each argument to the specific function, each type being either fully instantiated or possibly wildcarded, since some operators can work on a variety of input types. The `output-type` is the type of the output generated, given inputs conforming to the input type restrictions. The `code-f` is the function used to generate code, given an AST as input. The `cost-f` is the function that can give cost estimates for a given specific operator with respect to time, data space, code space, and desirability. The `temp-f` is a function that knows how to allocate temporary variables for the operation.

4.6 Type Resolution

There are five kinds of type arbitration in *ALP*: bottom-up arbitration, top-down arbitration, variable disambiguation, operator choosing, and type instantiation.

4.6.1 Bottom-up Type Arbitration

In bottom-up type arbitration, the operator set at the current AST node is reduced by restricting the operator set to the set of operators matching the types for the node's children. The node's typeset is then set to the union of the output types of the operator set. All this is done while traversing the tree in a bottom-up manner.

4.6.2 Top-down Type Arbitration

In top-down type arbitration, the operator set at the current AST is restricted to the subset of the operator set whose output types are a member of the node's `typeset`. The node's children's typesets are then set to the union of the input types of the operator set. All this is done while traversing the tree in a top-down manner.

4.6.3 Variable Disambiguation

Variable disambiguation seeks to transform an ambiguous type for a variable into a more specific type by collecting the superset of all the attempted uses and using a voting algorithm to select a more specific type. Normally this is done after at least one bottom-up and one top-down type arbitration pass. After a variable has been more fully resolved, a bottom-up

and a top-down type arbitration pass must be done in every subtree of the AST containing a reference to this variable, in order to instantiate the altered variable definition.

4.6.4 Operator Choosing

Often there are many ways of doing the same operation, and thus the `opset` at a node of the AST may contain more than one element. Operator choosing is the technique by which an operator is chosen by evaluating the `cost-f` of each specific operator. After making a choice, a top-down type arbitration pass is performed on the node.

4.6.5 Type Instantiation

This is a special case of variable disambiguation. Sometimes it is not possible to select a "better" type, and the voting algorithm fails. Associated with each type is a default type that can be used to disambiguate types even further. For example, the basic type `float` has an associated type of `single-float`, and `array` has an associated default type of `array(0)`. The default type is merged with the type of the variable, and then the process proceeds as for variable disambiguation.

4.7 Code Generation Phase

Code generation is divided into three subphases:

1. Temporary variable allocation
2. Temporary pooling
3. Code generation

4.7.1 Temporary Variable Allocation

Temporary variable allocation is done in two passes: one bottom-up pass and one top-down pass. The bottom-up pass is used to propagate operator source information up the AST, to facilitate sharing; the top-down pass propagates down operator destination information, to further facilitate sharing of temporaries.

4.7.2 Temporary Variable Pooling

Temporary variable pooling minimizes the number of temporary variables that must be created: all the temporaries needed by each assignment statement are pooled into the statement block above it in the AST. This process repeats at each higher level in the AST, until it is not possible to promote any shared variables. Note that a temporary is promoted

to the next higher level only if it can be shared at that level. This prevents temporary variables from being promoted too far from the statement block where they are live.

4.7.3 Code Generation

At this point, code generation is trivial. The code generation function need only read the temporary variable assignments, the current type, and the source information from its children, and then emit the instruction. The code generation function need not worry about valid destinations or mismatched output types.

5 Multidimensional Extensions

Shifting away from a unidimensional processor organization can be done with a few minor extensions to the syntax. Layout declarations can be extended to take lists of integers instead of simple integers. For example:

```
declare  layout  grid ((2 2) (512 512))
```

This defines a layout where each copy of the object defined in this layout is distributed in a rectangular array of four processors, with justification occurring towards the origin. Type declarations are otherwise not effected.

Array access can be extended by allowing comma-delimited array indexes. Index ranges, like the fortran 8x ':' operator and axis wildcarding could also be added, even in the unidimensional model.

The compiler instantiates multi-dimensional arrays using column major ordering. Once the array is embedded as above, it is treated as a one-dimensional object for all intents and purposes, including how the layout information is being interpreted. This can be modified and will the subject of further investigation.

6 Status and Conclusions

An experimental version of the *ALP* compiler has been constructed. We are in the process of evaluating and tuning the code generator, but initial results indicate that indeed hand code quality is being attained (for the scope of programs *ALP* is meant to cater for). The next big test for *ALP* is to convert the algorithms of the companion paper, "Efficient Breadth-First Expansion on the Connection Machine, or: Parallel Processing of L-Systems" (by Pinter and Pinter, YALEU/DCS/TR-719) and see how they get compiled; the results will be reported in a subsequent version of this paper.

All in all, we believe this tool could be extremely helpful in developing truly data parallel programming for CM-class machines. It is helpful not only in fast prototyping and evalu-

ation of design alternatives, but given the quality of the code generator it could well serve as the basis for a real compiler. Further extensions to the language (in the spirit of *APL*) are feasible and will be the subject of further investigation.

Acknowledgements. We would like to thank Joe Crawford for editorial comments on a previous draft of this paper.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] G. Blelloch. Scans as primitive parallel operations. In *International Conference on Parallel Processing*, pages 355–362, 1987.

[3] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[4] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *JACM*, 27(4):831–838, October 1980.

[5] M. Metcalf and J. Reid. *Fortran 8X Explained.* Oxford Science Publishers, 1987.

[6] Thinking Machines Corporation, Technical Report HA87-4. *Connection Machine Model CM-2 Technical Summary*, April 1987.