A supercomputer implementation of a functional data parallel language

Brad Alexander, Dean Engelhardt and Andrew Wendelborn*
University of Adelaide, South Australia

August 19, 1994

Abstract

The purpose of the Adl project is to demonstrate the efficient implementation of data parallel functional programming, firstly on the TMC CM-5 but ultimately on other parallel machines. We have designed a small polymorphic non-recursive language (Adl), which emphasizes high-level operations (second-order combinators) on aggregate structures. The Adl project incorporates the formal description of Adl semantics, translation and optimization, and the design of an abstract data-parallel machine which describes not only the CM-5 but also other distributed memory multicomputers and hence encourages architecture-independent code generation. An executable natural semantic description of translation to the Bird-Meertens Formalism (BMF) has been completed; similar techniques are being used with an optimizer. We also describe a thread-based compilation scheme to the abstract machine, and a CM-5 implementation of the abstract machine.

1 Introduction

Exploitation of parallelism in applications is a very attractive concept, but there is a noticeable reluctance to adopt parallel programming in practice. Programming parallel machines is complex, difficult and, despite many years of research, still comparatively poorly understood. Parallel software is often highly machine dependent, idiosyncratic, non-portable even across configuration changes to one machine, expensive and extremely difficult to debug.

Consequently, there is considerable research into general purpose parallel computing which aims for efficient, architecture independent parallel programming, an effective software development methodology to master the inherent complexity of the task and provide some notion of correctness, and to hide as much as possible details of program partitioning and communication. The advantages of higher-order functional programming in terms of abstraction and expressiveness are well known, but efficient implementations of such languages in their full generality is still difficult, especially for parallel implementations.

At the University of Adelaide, we have been investigating data parallel functional programming since 1992, with effective use of the SACPC CM-5 as a specific target. The project incorporates design and investigation of a small data parallel functional programming language named Adl, the formal description of its semantics, translation and optimization. To support this research we have designed an abstract data-parallel machine which describes not only

*Authors' Address: Department of Computer Science, University of Adelaide, Adelaide 5005, Australia. E-mail: {brad,dean,andrew}@cs.adelaide.edu.au

our immediate target architecture (the CM-5) but also other distributed memory multicomputers and hence encourages architecture-independent code generation.

The language Adl [1] is a one in which computation is expressed primarily through high-level bulk-data operations over data aggregates. These second-order combinators replace the role of recursion. The primary goal of our research is to implement this language by first translating it directly to a formalism called BMF (Bird-Meertens Formalism), utilising attendant equational theory to describe optimisation transformations [2], and finally producing code for our abstract machine from this optimised BMF form.

The abstract machine model we have defined is a distributed memory SPMD machine whose nodal mode of execution is based on a non-preemptive multi-threading scheme. Inter-node communications are asynchronous messages which alter the state of the receiving node (active messages). The multi-threading design emerged from our earlier experiments in low-level CM-5 programming [14], and our observation that multi-threading is necessary to support nested data parallelism on nodal SPMD machines such as the CM-5 in a manner that is independent of data partitioning.

We have shown in earlier work [15] that this abstract machine model may be utilised in a fashion which supports a very general model of data parallelism. Guy Blelloch [7, 6] has taken an alternative approach to achieve similar aims. In particular both implementations can support nested data parallelism (though ours does this in a more direct manner).

Our abstract machine model also shares several features with the TAM (Threaded Abstract Machine) designed and implemented at Berkeley [12, 29]. The TAM implementation has been as the basis for an Id compiler on the CM-5, a system which we plan to use for comparative evaluation of our work. The principal difference between our abstract machine and TAM is that ours offers a somewhat more general model of inter-thread communication, and defines flat, rather than multi-level, spaces of threads and store. The intended mode of use of the two machines also differs: TAM is a tool aimed at control parallel specification while our machine is intended for data parallel specification.

The extraction of parallelism from higher-order constructs over aggregate data structures is a common theme in the literature. Of particular interest is the recent work on Skeletons [9, 8, 13]. Skeletons are parallel code templates, usually based on well established higher order constructs, distilled from the code of existing functional programs. Once extracted these templates can then be manipulated to the desired form using well known identities and mapped to the target machine. The Skeletons work differs from ours in the use of established source languages (such as ML) and the piecemeal approach to detecting and translating appropriate program segments.

The Categorical Data Type (CDT) approach is also related. CDTs are mathematical objects, with foundations in category theory [30]. Consequently, an equational transformation system exists for each CDT, providing a software development methodology based on the refinement of high-level prototypes to forms known to be more efficient. This approach can also give much more confidence in program correctness, when compared to current techniques which rely on low-level debugging. CDT-based programming emphasizes the use of structure-preserving homomorphisms as the basic operations for programming. The fundamental Adl combinators *map, reduce* and *scan* are homomorphisms; in fact, Adl is translated to a concatenate-list CDT expressed in BMF notation.

CDTs encapsulate both control flow and data representation; a CDT describes not only a data type and its operations (as does an abstract data type, one of the basic tools of modern software engineering), but also embodies the computation, communication and processor mapping patterns necessary to implement the CDT on a parallel machine.

Finally our work will draw upon (and perhaps contribute to) the large body of literature describing various BMF theories on CDTs and the identities associated with them[5, 20]. Many of these identities will be employed in the construction of the data movement optimiser and in specifying a parallelisation algorithm for the BMF code partitioner [25, 27].

The next section is a short outline of the Adl language and its most significant features. Section 3 describes these features using examples. Section 4 examines the BMF component of the project, including a brief survey of existing work. Section 5 describes the abstract machine. Work on a closely related alternate strategy for targeting the abstract machine is outlined in section 6. Finally, section 7 summarises and concludes the paper.

2 The Adl language

Adl is a small functional language designed for specifying highly parallel solutions to scientific programming applications. Adl is designed along minimalist lines so as not to distract its implementors with peripheral issues.

2.1 Language Features

Some of the more noteworthy features of Adl are:

- Strict semantics: All user-defined and most built-in functions have strict evaluation semantics.
- No interactive I/O: all input must be available at the time of program invocation.
- All functions have one argument: If a function requires more than one input value these must be enclosed in a tuple.
- Simple pattern matching: Based on structure not content. The sole purpose of patterns in Adl is to project values from functions' input tuples.
- Syntactic polymorphism: Universal quantification is allowed on the input types of all functions except the function accepting input at run-time. Types declared for this function are propagated through the program at compile-time. The type checker instantiates all functions to the concrete types they are called with.
- Lack of recursion: Adl disallows user-defined recursion. Operations over aggregate data structures and unbounded repetition are handled by a few higher-order combinators.

Of these features, probably the most unconventional is the lack of recursion. This feature, enforced by simple scoping rules, exists primarily, to simplify analysis at later stages of compilation. Other reasons are to promote Adl's built-in combinators to a central role in programming and to all allow a direct translation from Adl to BMF. Programming experience so far suggests the lack of recursion does not impose an inordinate burden on the Adl programmer. In most cases it seems of little consequence.

2.2 Types and Constructs

Adl has a small number of primitive types, two type constructors and a few primitive operators. The primitive scalar types are integer, boolean and real. A character type will be added in future when the need arises.

The type constructors are tuples and vectors. Tuples are simply devices to group values of possibly differing types, they are always expressed as literals. Vectors are monolithic, one-dimensional arrays. Vectors can only be created using the primitive `iota` operation or by specifying them as literals. There is no operator to append or cons vectors.

There are three flavours of higher-order functions over vectors. These are `map`, `reduce` and `scan`. Both `reduce` and `scan` have variants to cater for non-empty vectors and sequential (i.e. directed) operation. These three constructs, combined with the scalar operators, can be used to solve an enormous range of problems.

Adl also provides an `if` and unbounded iteration through `while`. In addition to these there are a number of built-in binary and unary operators. Perhaps of most note are vector index: `!` (an infix binary operator) and vector length: `#`.

3 Examples of Adl in use

The nature of Adl source code is perhaps best illustrated by example. One of the simplest bulk data operations likely to be encountered in numerical applications is matrix addition. A possible implementation is shown in figure 1. Adl's syntactic polymorphism has allowed us to avoid the use of type declarations except for those of the input variables `a` and `b`. The term `vof vof real` indicates the input matrices are represented as nested vectors of real numbers.

Both instances of `iota` in figure 1 are used to generate indices. The outer one to emulate matrix rows, the inner one for columns. The term `iota` n produces the vector: `[0,` $\ldots n-1$`]`. Adl vectors are always indexed from zero so `iota (# a)` generates all of the indices of `a` and `iota (# a!x)` makes all the indices of `a!x`. These indices are used to access both `a` and `b` so there is the implicit assumption they are the same shape.

```
ma (a:vof vof real,
    b:vof vof real) :=
  let
    f x :=
      let
        g y := a!x!y + b!x!y
      in
        map (g,iota (# a!x))
      endlet
  in
    map (f, iota (# a))
  endlet
```

Figure 1: Adl program for matrix addition.

Figure 2 contains a more complex example. `av_points` finds an "average" point of a vector of cartesian pairs. The first element of the result is the average of all the first elements in `parr`. The second result component is derived in an identical fashion from the second elements.

One noteworthy feature is the use of syntactic polymorphism. In Adl a function like `plus` inherits its overloading from the operators it contains (in this case "+") unless it is declared with concrete types[1]. The functions `first` and `second` are also polymorphic (though in a more conventional sense).

The right hand sides of the definitions of `avx` and `avy` are almost identical. They differ only in that one calls `first` and the other calls `second`. This may seem to be verbose compared to the alternative of passing in `first` and `second` as arguments into a higher order function. However the simplicity af-

[1] Though this feature is convenient in practical terms it is not at all pleasing from a theoretical perspective. A user should really be able to specify all the languages types within the framework of the language. This is currently impossible in Adl; a point we hope to remedy.

```
av_points(parr: vof (real,real)) :=
  let
    plus (x,y) := x + y;
    first (x,y) := x;
    second(x,y) := y;
    len := # parr;
    avx :=
      reduce(plus,0.0,map(first,parr))
      / len;
    avy :=
      reduce(plus,0.0,map(second,parr))
      / len
   in
      (avx,avy)
   endlet
```

Figure 2: Adl program for finding an average point.

forded to the language implementor by not treating functions as first class objects more than makes up for the slight increase in space needed to express some algorithms. Some types of compile time analysis are much less likely to be useful in the presence of higher order functions.

4 Using BMF as an intermediate form

It has long been recognised that the use of simple higher-order constructs over aggregate data types is an elegant means of expressing some types of computation. There are a number of languages (referred to by Blelloch [6] in his thesis as "collection-oriented languages") built around this theme. Examples include APL [21], Backus' FP [4], *Lisp [19], and CM-Fortran [31] (among others). Most of these allow the programmer to define large-scale operations over array-like data structures using a small set of built-in combinators such as *map*, *reduce* and *scan*.

BMF or Bird-Meertens Formalism [5] is a framework which encapsulates these operations for a variety of data types. If a categorical data type adheres to a certain set of constraints (i.e. it is a Hagino type [17]) a useful BMF theory can be created from it. BMF theories exist for lists, sets, bags, trees and multi-dimensional arrays [23, 18].

A BMF theory of a data type contains a number of primitive aggregate operations e.g. *map*, *reduce*, *scan* and *filter* and several identities relating them. The identities are complete in the sense that any BMF term (or program) can be converted to all its equivalent forms using only these identities, at least at the level of the aggregate operations. BMF programs tend to consist of a composed series of these aggregate operations. In this aspect they bear a superficial resemblance to FP programs. It has been shown these identities can be used as tools to produce non-obvious code optimisations [5, 11, 10]. All of this follows automatically from a few properties of the type.

4.1 BMF concatenate lists

Adl's primary aggregate data type is the vector. The BMF theory most closely corresponding to Adl vectors is the theory of concatenate lists. Concatenate lists are the best established categorical data type within BMF. They are constructed from elements of a base type using the following three identities:

$$\begin{array}{lcl} [\,] & = & K[\,] \\ [\cdot]x & = & [x] \\ [xs] +\!\!+ [ys] & = & [xs, ys] \end{array}$$

Where $[\,]$ is a constant value, $[\cdot]$ is a make singleton function and $+\!\!+$ is a concatenate function to join lists together. The associativity of $+\!\!+$ means that a concatenate list of length three or greater can be decomposed and reassembled in a number of ways without changing its meaning. This flexibility is carried over to operations on concatenate lists. For example a function to generate the sum of squares of a concatenate list can be defined:

$$\begin{array}{lcl} sumsq\ [\,] & = & 0 \\ sumsq\ [x] & = & sqr(x) \\ sumsq\ [xs]{+\!\!+}[ys] & = & sumsq([xs]+[ys]) \end{array}$$

Transformations such as this are called list-homomorphisms since they change the list with operators that adhere to the structure of the list. An extremely wide range of operations can be classed as homomorphisms including *map*, *reduce*, *scan*, *filter*, a large range of arithmetic operations, and sorting. One useful theoretical result is that any homomorphism can be normalised to a simple canonical form in BMF:

$$\oplus/_{id} \cdot f*$$

Where $\oplus$ is any associative operator on the base-type (e.g $+$), the subscript id is the identity element of this operator (e.g. 0), and f is any unary function on the base-type (e.g. sqr). The $/$ and $*$ symbols stand for *reduce* and *map* respectively. It has been noted by Skillicorn [28, 27] that the inherent parallelism of *map* and *reduce* and the general usefulness of homomorphisms help make BMF an excellent model for parallelism.

The strategy of a programmer would be to specify a program as a sequence of composed functions in BMF in which the parallel operations are embedded as homomorphisms. The compiler can then optimise the code through a series of algebraic manipulations. After optimisation the code can be partitioned using similar techniques (see [28, 25, 23]). Finally the parallel operations such as *map*, *scan* and *reduce* can be translated to simple pre-defined code templates. The beauty of this approach is that all but the last step of compilation is portable. Only the templates need to be created for each target architecture. Skillicorn has also shown [26] that BMF code is also a portable model for parallel performance. Good performance models are necessary for the construction of an optimiser.

4.2 Mapping Adl to BMF

Most of the literature on using BMF assumes that the starting point is an unoptimised BMF program and the target is the base machine. Given the radical differences between BMF and a conventional programming languages we feel that this is unlikely to be a reflection of reality. A more plausible role for BMF is as an intermediate form between a conventional programming language and a target machine. Some of the work on using function Skeletons (see introduction) takes this approach. Skeletons are extracted automatically from suitable code in existing languages (such as ML). Once extracted Skeleton code can be manipulated according to BMF identities.

Our work differs in two ways. Firstly there is no need to detect candidate code for conversion in the source language. In Adl *map*, *reduce* and *scan* are explicit. Secondly, Adl's simple structure, most notably its lack of recursion, allows us to define a complete translation from Adl to BMF. This translation is possible for only segments of code in the Skeleton projects.

4.2.1 The Adl to BMF translator

The action of the translator is perhaps best illustrated by example. Consider the Adl code to sum the absolute value of a vector of real numbers the body of such a program might look like:

```
reduce(plus, 0.0, map(sqrt, map(sqr, a)))
```

Where `a` is the input value and we will assume `plus` is defined as before and that `sqr` and `sqrt` are defined as primitive functions. The Adl to BMF translator will evaluate this code to the BMF expression[2]:

$$+/_{0.0} \cdot (\mathsf{sqrt} \cdot \pi_2) * \cdot \mathsf{distl} \cdot (\mathsf{id}, (\mathsf{sqr} \cdot \pi_2) * \cdot \mathsf{distl} \cdot (\mathsf{id}, \mathsf{id})^\circ)^\circ$$

[2] This is a slightly abridged version of the target code. The real translator would use a slightly more complex version of $+/_{0.0}$. For this example the semantics of the simpler version are the same.

The most fundamental difference between this BMF code and its Adl source is the lack of variable names, all data movement must be explicit. Most of the work needed for translation is a direct consequence of this requirement. In the example above the only variable reference to eliminate is to the input vector: a. This reference is replaced with a function: $(\mathsf{id}, \mathsf{id})^{\circ}$ which creates two copies of it to produce the pair: (a, a). Next the first copy of a is distributed across the second to form the list $[(a, a_0), \ldots, (a, a_n)]$. This is done just in case the function being mapped over this array needs to access any part of a besides the element in its formal parameter. In this case the function being mapped is $\mathsf{sqr} \cdot \pi_2$ where π_2 is a selector function for the second element of a pair. This implies that only the second component of the environment of each function instance gets used (i.e. the a_i component of each tuple).

The result of $(\mathsf{sqr} \cdot \pi_2)*$ is the array $[a_0^2, \ldots, a_n^2]$ The next part of the code performs a similar operation using *sqrt*. Again a is distributed across an array (this time the array of squares of a). Again the function inside the *map* extracts the second element of the input tuple and applies an operation to it. The last stage of the target program is the reduction $+/_{0.0}$ which executes a parallel sum over the absolute values of a.

This translation is extremely conservative. It works under the assumption that any function being mapped over a list may refer not only to its parameter but any variable within scope. Under this assumption the translator ensures that these variables are visible by using the distl function to distribute copies of the entire scope over the list of input values. Clearly this is inefficient in most cases, the next stage of the compiler, the data movement optimiser goes some way to remedying this and other sources of inefficiency.

4.2.2 The Data Movement Optimiser

Data movement optimisation is the specialisation of earlier parts of a BMF program with respect to the data needs of later stages. In the case of our previous example this means eliminating unnecessary copies of the input values. The code resulting from this process would be:

$$+/_{0.0} \cdot sqrt * \cdot sqr*$$

This task in itself is not particularly difficult. You only have to follow the data dependencies of the program from its result and eliminate the passing of those values not needed.

Unfortunately, not passing these unused values will not eliminate waste in its entirety. For many applications involving the manipulation of large, perhaps nested, vectors the major source of waste is going to be the passing of whole vectors when only a small part would suffice. The cause of this problem lies in traditional methods of vector access namely, indexing operations.

When the Adl to BMF translator encounters a function which may access a vector element a copy of the entire vector is passed on. If the function occurs inside a *map* then many copies of the vector are created and passed on. No attention is paid to the fact that each instance of the function probably accesses only a small part of the vector and this part probably varies across instances in a regular fashion. The main role of the data movement optimiser is to detect these patterns of access and feed each instance of a function invocation only the part of the array it needs. This feeding process is carried out using bulk data movement operators such as transpose (for swapping dimensions in nested vectors) and project (for projecting values from a vector given an index list).

The potential variety of communication patterns and the possible combinations of bulk data movement operators is almost infinite. A useful data movement optimiser will have to cater for a subset of these that covers most common applications. Our recent findings suggest that many complicated patterns of communication can be constructed by combining a few simple building blocks. This is a potentially rich field of research.

4.2.3 Code partitioning

One of the last steps in the process of mapping Adl to a target architecture is splitting the resulting code into the appropriate number of sequential components and defining the correct parallel constructs to communicate between these parts. This stage of processing, code partitioning, can be carried out whilst the program is still in its BMF form. The techniques used come from Roe [25].

The first step of the method is to define a combinator to split a vector argument evenly into a vector of p sub-vectors, where p is the number of processors on the machine. This combinator can be referred to as *spread* or $split_p$. The next step is to compose it with its complement: $+\!\!+/$ or *gather* so we have the identity function:

$$+\!\!+/ \cdot split_p = \mathsf{id}$$

This identity can be inserted into any BMF program without changing its semantics. If we start by putting this construct at the front of our example program and move move the $+\!\!+/$ operator backwards via a series of algebraic transformations we get:

$$\begin{array}{ll}
+/_{0.0} \cdot sqrt * \cdot sqr * +\!\!+/ \cdot split_p & = \\
+/_{0.0} \cdot sqrt * \cdot +\!\!+/ \cdot sqr ** \cdot split_p & = \\
+/_{0.0} \cdot +\!\!+/ \cdot sqrt ** \cdot sqr ** \cdot split_p & = \\
+/_{0.0} \cdot (+/_{0.0}) * \cdot sqrt ** \cdot sqr ** \cdot split_p &
\end{array}$$

At the architecture level $split_p$ can be viewed as an operator for distributing input data nodes of the target machine and $+\!\!+/$ as an operator to coalesce data back onto one node. Interpreting these operators in this way makes it easier to rationalise the tranforms in this example. The first line shows the program prior to partitioning. $split_p$ distributes the input data from an source node across the machine and then immediately gathers it back onto the source node using $+\!\!+/$. The rest of the code is then executed sequentially on this node.

In the second line data coalescence is delayed until after *sqr* has been applied to all input values. Since $split_p$ yields a nested vector *sqr* must be applied to its elements using nested *maps*. The inner *map* can occur sequentially on each node of our target machine. The outer map is executed in parallel. The third line shows the program further transformed so data coalescence is delayed until after $sqrt\,**$.

The fourth line shows the entire program after it has been parallelised. The $+\!\!+/$ has been safely removed by making use of the BMF identity:

$$\oplus/ \cdot +\!\!+/ = \oplus/ \cdot (\oplus/)*$$

Part of the beauty of using BMF as an intermediate form is that individual transforms such as the ones in this example are both intuitive (once one is used to the notation) and, more importantly, easy to implement as a series of rewrites using identities like the one above. The extent to which it is possible to partition complex programs in this way and the types of partioning schemes that can be used is subject for further research.

4.2.4 Code generator

The final phase of BMF stream of the project is the targeting of the partitioned BMF code to the underlying architecture. In our case the underlying architecture will be our thread based abstract machine (see section 5). Using this approach the code generator will only have to be built once to support a variety of machines.

An instance of the abstract machine i.e. its code libraries and data structures must be created for every target machine but the work involved is trivial compared to the task of retargeting a code generator.

The code generator must create an identical C program for each node on the abstract machine. Calls for communication and thread control are made via the abstract machines instruction set. Each combinator in BMF will map to C code constructed from a standard framework. The distinct roles played by each node in the execution of a parallel combinator are resolved by reference to node numbers and the location of relevant data. Much of the code produced by this stream of the project will be for the mass movement of data through combinators such as transpose and project

4.3 Work to date: BMF stream

At the time of writing a formal description of the Adl to BMF translation process is complete [2]. The description consists of an annotated system of inference rules in natural semantics. Such a system should be suitable (with little modification) for execution on INRIA's Centaur, language development system [22]. This allows the rules of the translation to be checked against Adl interpreters already implemented using Centaur.

A C version of the translator is planned for the near future. If past experience is repeated then the task of mapping from the formal description to C code should be straightforward.

There is also a near complete formal description of a data movement optimiser. This optimiser is capable of eliminating excess data movement caused by vector indexing as well as the unnecessary passing of values. The rules formulated so far should be sufficient to produce reasonably efficient versions of a wide range of programs.

The code partitioner and code generator are still in the planning stage. A working implementation of the abstract machine already exists for the CM-5 (described in section 6.1). Code from both streams of the project should be tested on this in the near future.

5 A Threaded Abstract Machine

Our efforts in compiling Adl through the intermediate of BMF rely on a particular execution model in terms of which object code may be framed. We have shown in earlier work [15] that a natural expression of parametric nested data parallelism may be made in terms of a multi-threaded model of execution. Hence we choose to define a threaded abstract machine to which our compilation may be targeted.

The machine we have designed is a distributed memory multiprocessor SPMD machine whose nodal model execution is based on a paradigm of self-scheduling threads. This paradigm is similar to others proposed in recent literature in both the parallel hardware community [3, 24] and the parallel software community [12].

At the top-most level of abstraction we define our machine in terms of an abstract machine "program" P and a set of "processing nodes" n_i. An abstract machine program is defined a collection of disjoint "threads" each made up of a sequence of abstract machine instructions. Threads are analogous to traditional procedural units of activation; the instructions in the block are executed in sequence until a distinguished final instruction is reached.

Each node of our machine holds its own distinct copy of the abstract machine program P, and executes threads from that program in a manner that is independent of the state of any other node. That is, our abstract machine is SPMD. Nodes additionally maintain a computational state that is defined as a tuple incorporating a host id, a set of thread activations, a set of result values (which are pairs *(key, value)*), the node's memory pool, and a currently executing activation.

During the course of thread execution, nodes may engage in communication operations with other nodes. All such operations within our model are framed in terms of asynchronous messages which alter the state of the receiving node in a manner that is invisible to the receiving node's executing activation. Most available distributed memory architectures support this style of communication and hence are described by our model. In this sense our model is architecture-independent.

The abstract machine supports a single form of data aggregate, the one-dimensional vector. More complex data structures may be constructed as a nesting of such vectors. Every such vector is by definition a partitioned aggregate, that is, its component elements are spread among the various nodes of the abstract machine. These elements are stored in the nodal memory pool. The fashion in which this partitioning is performed is encapsulated within a *partitioning function* which is associated with the vector.

5.1 Nodal Model of Execution

In traditional SPMD machines, the flow of control between code units is deterministic and fully specified by the instructions within the code. A code unit becomes active on node n_i, after which time instructions from that unit are executed until such time as the unit terminates or it enters a request that a new code unit be activated. In the latter case, the currently executing code unit's execution is suspended, and the execution of the new activation commences. Once the activation it entered has terminated, the original code unit is resumed.

In our abstract machine the model of execution is significantly different. A thread which has become active continues its execution until such time as it explicitly requests suspension or termination. In the course of its execution an activation may request the activation of other threads, however such activations do not occur immediately. Rather, such a request causes an entry to be added to the node's "task pool", which maintains records of all activations, both active, suspended and awaiting execution. Whenever the node becomes idle, through either the termination or voluntary suspension of the presently executing activation, an activation is chosen non-deterministically from eligible activations in this pool.

Possible activation states are "pending", "active" and "suspended". A pending activation is one that has been entered by another activation but has to date not been scheduled for execution. There can be at most one activation with state "active" within a task pool. A "suspended" activation is one which voluntarily suspended its execution. Upon suspension, an activation specifies a "key" (or identifier) which denotes the conditions under which its execution may continue.

Each node maintains a "result pool", a set of tuples, the first element of which is a key, the remaining elements arbitrary values. Its purpose is as a medium of communication and synchronisation between activations. An activation may add a result tuple to either the pool of the node on which it is presently executing, or to another node's pool through a communication operation.

The result pool of a node plays an important role in the scheduling of activations. Activations which have voluntarily suspended their execution may only be rescheduled when one or more results with the same key as that which on which they suspended exist within the node's pool. The semantics of resumption are that exactly one result which bears the appropriate key is removed from the pool and returned as the result of the call to the suspension function when the code stream in the resuming activation continues.

As can be seen above, an important factor in the scheduling and communication functions of a node are the identifiers known as keys, defined to be elements of an infinite "key-space". It is the responsibility of the abstract machine program to institute a protocol of key management which ensures that results intended for receipt by a particular suspended activation are never intercepted by another.

5.2 Abstract Machine Operations

In addition to traditional algorithmic control constructs (for, while, if) and the standard imperative operation of assignment, the abstract machine has a number of special operations that deal with threads and communications. Instructions are provided to enter an activation, to activate a thread on another node, to deposit values in either the host result pool or that of another node, to suspend and terminate activations, to allocate storage, and to generate unique identifiers.

In terms of this instruction set it is possible to define partitioning-independent nested data parallel operations. At the most abstract level such operations are modelled as a set of thread activations, one for each element of the source vector. Each such activation runs on the node owning the corresponding vector element ("Owner Computes") and accepts the element as input.

We have expressed a number of important data parallel primitives in such a fashion: thread synchronisation, distributed vector allocation, and Adl's second-order *map*, *reduce*, and *scan* [15]. The threaded nature of the abstract machine permits such descriptions to take advantage of latency-hiding.

6 Compiling Adl to a Multi-Threaded Abstract Machine

In the previous section we have outlined the usefulness of a threaded execution model in providing dynamic support for executing nested data parallel operations. As an alternate implementation strategy to that offered by our work with BMF, we have investigated compilation of Adl directly into thread code executable on a concrete instance of our abstract machine. This section describes the construction of such a machine on the Thinking Machines CM-5 and discusses some of the more important issues relating to the compilation process. A more detailed evaluation of this work may be found in [16].

6.1 AMAM: A Multi-Threaded Abstract Machine on the CM-5

The first stage of our work in compiling Adl directly to a threaded description was to construct an underlying threading system on the CM-5 to which our compiler's object code would be targeted. We chose to produce a system which very closely mirrors the abstract machine described earlier. In our previous report [15] we provided operational specifications for a number of Adl's nestable data parallel in terms of this abstract machine. These operational specifications easily map to our CM-5 threading system and form a basis for our compiler's system library (see below).

The CM-5 implementation of our abstract machine is constructed as a library of C routines which manage the per-node data structures defined by the machine. Also provided in the library are implementations of all abstract machine system calls. Those that embody node-to-node communication are constructed in terms of active message primitives from the CMAML layer of the CM-5 CMMD library [32].

The abstract machine library models threads simply as C procedures and schedules an activation for execution simply by calling the procedure which models it. Certain rules concerning the contents of the C counterpart of a thread ensure that this activation modelling is correct.

6.2 Compiler Overview

The approach we take in compiling Adl directly to our threaded abstract machine is significantly different from the transformational approach we use for our work with BMF. Our strategy for the direct system is to produce a library of parametric thread collections each of which implements one of Adl's parallel constructs. The compiler takes a structured Adl program and maps it into a series of threads, each of which refers to sections of this library whenever a complex operation is issued.

A number of interesting issues arise in the later parts of the compilation process, specifically those dealing with data partitioning, thread production and code generation. The most important of these are discussed briefly below.

6.3 Generating Threads

There is a radical difference between the structuring primitives offered in Adl and those offered in the target language (the multi-threaded abstract machine). The former supports arbitrarily nested functions and `let` expressions, while the latter forces a program to consist of a collection of non-nested threads. Clearly there are many ways that a thread description can be constructed to model the original program structure. A component of our compiler, the "Thread Colouriser" makes this partitioning.

Our partitioning scheme is a simple one, driven by program syntax and operating in two phases. The first of these phases considers issues of function modelling. The threading system provides a single code activation mechanism, namely the thread activation. All functions must be modelled in terms of this. Thus we initially map every function in a program, regardless of nesting, into its own thread.

The second phase of our thread colouring operation considers operations within a function. Certain Adl operators, particularly those involving operations across vectors, have the property of being "latency unbound." That is, it is impossible to determine *a priori* the exact execution cost of such operations, typically because implicit inter-node communication is required. It seems desirable to use the latency hiding properties of the multi-threading system to mask the possibly high latency of these operators. To do this we must require that the issuing of such an operation immediately be followed by a thread suspension operation. Thus the representation of a function may now be split into a number of threads: one for the function invocation and the remainder as continuations after thread suspension.

6.4 Modelling Scope

Adl is a language which incorporates the traditional notion of scoping. That is, a function nested within another may make reference to a value defined locally to the outer function. Under the threaded model presented by AMAM, however, there exists no such concept of block nesting or scope. Clearly we need to construct a mechanism to model scope in the AMAM threads representing the Adl program, otherwise reference to non-local values will be impossible.

The mechanism we choose to model Adl's scope is the traditional "Static Link". Into every thread's state object we add a space in which to store a pointer to the state object of its static parent, that is the block which immediately contains the function modelled by the thread in question. At function invocation, during the creation and setup of the callee thread's state, this slot is filled in with the appropriate pointer. The newly activated thread may then access values defined in its parent's state by means of following this pointer. Since this parent itself has a pointer to its parent and so on, the newly activated thread may therefore access any value in scope.

Note that multiple dynamic thread activations for a function f may exist at any one time, e.g. as part of a data parallel operation. Each such instance will have the same static parent, hence the same static link. It is clearly necessary that these multiple function instances contain some information which allows the appropriate instance to return a value to the proper invoking activation. In some systems this is achieved by the provision of a "Dynamic Link," that is a pointer to the activation's dynamic parent. In the Adl language system an alternate mechanism, one of logical keys, achieves an equivalent result.

6.5 Key Protocol Issues

Control and data flow in a thread program running under our abstract machine is specified in terms of logical keys. Thread activations that suspend awaiting a result value from another activation do so upon a particular key. When a result arrives with that key they may be scheduled again. It is clearly important in a thread translation of an Adl program that some form of key protocol be set in place to ensure that situations may never arrive where data and control is incorrectly routed.

Our current system defines a key to be a tuple of four integers. In the Adl compiler system we use this key tuple in a way which gives a different logical significance to each element of this tuple. We assign the first element of the tuple to hold an "operation ID," that is an identifier which statically identifies the particular operator instance that caused the suspension. The second element holds the dynamic identifier of the vector over which the operator is acting (if such exists), while the third holds the node number of the host who is expected to produce the result. The final value is used to disambiguate cases in which multiple distinct activations would bear the same value for the previous three properties. Together these four tuple elements form a key guaranteed to be unique in the dynamic lifetime of an Adl program.

7 Conclusions

We have described the current state of a project whereby we hope to demonstrate the viability of parallel functional programming on the CM-5 and other multicomputers. The language used, Adl, is designed to be both simple, so that a complete implementation to CM-5 level is feasible with modest resources, and also novel in its utilization of mathematically-based higher-order language defined combinators to supplant recursion and user-defined higher-order functions. Although our initial programming experience suggests that, with a little effort at understanding a slightly different approach, the language is powerful and expressive, we envisage further work in this important area.

We are encouraged by progress in both the abstract machine design and its refinement to the target machine, and the use of the BMF formalism in describing semantics of both the language and its optimizations. When used with Centaur to generate both an interpreter and a programming environment, we have a powerful design tool. In the future, we plan to complete an initial implementation on the CM-5 and conduct performance evaluations (for example, by comparison with the CM-5 Id implementation), investigate an improved type system based on multiple variable type classes, and develop appropriate data partitioning schemes (for example, by evaluating different partitioning functions).

References

[1] Brad Alexander. Adl review. Technical Report 94-10, University of Adelaide, May 1994.

[2] Brad Alexander. Mapping Adl to the Bird-Meertens Formalism. Technical report, The University of Adelaide, Author's email address: `brad@cs.adelaide.edu.au`, 1994.

[3] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *1990 International Conference on Supercomputing*, June 1990.

[4] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613 – 641, August 1978.

[5] Richard S Bird. Lectures in constructive functional programming. In *Constructive Methods in Computing Science*, pages 151 – 216. NATO ASI Series, Vol F55, 1989.

[6] Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. The MIT press, Cambridge Massachusetts, 1990.

[7] Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstien, Marco Zahga, and Siddhartha Chatterjee. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[8] Tore A Bratvold. A Skeleton-Based Parallelising Compiler for ML. In *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–34, September 1993.

[9] David Busvine. Implementing recursive functions as processor farms. *Journal of Parallel Computation*, 19:1141–1153, 1993.

[10] Murray Cole. List homomorphic parallel algorithms for bracket matching. Technical Report CSR-29-93, The University of Edinburugh, Author's email address: `mic@dcs.ed.ac.uk`, 1993.

[11] Murray Cole. Parallel programming, list homomorphisms and the maximum segment sum problem. Technical Report CSR-25-93, The University of Edinburugh, Author's email address: `mic@dcs.ed.ac.uk`, 1993.

[12] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. TAM — a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, 1993.

[13] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *PARLE'93: Parallel Architectures and Languages Europe*. Springer Verlag, June 1993. LNCS 694.

[14] Dean Engelhardt and Andrew Wendelborn. Experiences with CDPEAC: Vectorising a Laplace iteration on the CM5. In Gopal Gupta, editor, *Seventeenth Annual Computer Science Conference, Part A*, pages 145–156, 1994.

[15] Dean Engelhardt and Andrew Wendelborn. Expressing nested data parallel operations through multithreading. Technical Report 94-16, University of Adelaide, August 1994.

[16] Dean Engelhardt and Andrew Wendelborn. Progress in implementing Adl on the CM-5 supercomputer. Technical report, University of Adelaide, August 1994.

[17] Tatsuya Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science: Edinburgh U.K.*, pages 140–157. Springer-Verlag, September 1987. Appears in, Lecture Notes in Computer Science Vol 283.

[18] Gaetan Hains and Lenore M. R. Mullin. Parallel functional programming with arrays. Listed for publication in the Computer Journal, 1993.

[19] W. Daniel Hillis. *The Connection Machine*. MIT Press, Cambridge Mass, 1985.

[20] Paul F. Hoogendijk and Roland C. Backhouse. Relational programming laws in the tree, list, bag, set hierarchy. *Science of Computer Programming*, 22:67–105, 1994.

[21] Kenneth E. Iverson. A dictionary of APL. *APL quote quad*, 18(1):5 – 40, September 1987.

[22] Ian Jacobs and Laurence Rideau. A centaur tutorial. Technical Report 141, INRIA, France, August 1992.

[23] Richard Miller. *A constructive theory of Multidimensional Arrays*. PhD thesis, Lady Margret Hall, Oxford, February 1993.

[24] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-Machine multicomputer: An architectural evaluation. *Computer Architecture News*, 21(2):224–235, May 1993.

[25] Paul Roe. Derivation of efficient data parallel programs. In *ACSC-17*, pages 621 – 628, January 1994.

[26] David B Skillicorn. Architecture independent parallel computation. *IEEE COMPUTER*, pages 38 – 49, December 1990.

[27] David B. Skillicorn. Parallelism and the Bird-Meertens formalism. Obtained via ftp from Queens university, Canada, April 1992.

[28] David B. Skillicorn. the Bird-Meertens formalism as a parallel model. In *Software for Parallel Computation*, June 1992.

[29] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of mechanisms for fine-grained parallel programs in the J-Machine and the CM-5. *Computer Architecture News*, 21(2):302–313, May 1993.

[30] Mike Spivey. A categorical approach to the theory of lists. In *Mathematics of Program Construction*, pages 400–408. Springer-Verlag, June 1989. Found in: Lecture Notes in Computer Science Vol 375.

[31] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual (version 2.1 Beta)*, December 1992.

[32] Thinking Machines Corporation, Cambridge, Massachusetts. *CMMD Reference Manual (version 2.0)*, August 1992.