The Bird-Meertens Formalism as a Parallel Model

D.B. Skillicorn

May 1992
External Technical Report
ISSN-0836-0227-
92-332

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared August 20, 1992

Abstract

The expense of developing and maintaining software is the major obstacle to the routine use of parallel computation. Architecture independent programming offers a way of avoiding the problem, but the requirements for a model of parallel computation that will permit it are demanding. The Bird-Meertens formalism is an approach to developing and executing data-parallel programs; it encourages software development by equational transformation; it can be implemented efficiently across a wide range of architecture families; and it can be equipped with a realistic cost calculus, so that trade-offs in software design can be explored before implementation. It makes an ideal model of parallel computation.

Keywords: General purpose parallel computing, models of parallel computation, architecture independent programming, categorical data type, program transformation, code generation.

1 Properties of Models of Parallel Computation

Parallel computation is still the domain of researchers and those desperate for performance, rather than the normal way of doing things. The reason surely does not lie with parallel hardware, whose performance follows a curve of the same slope as that of sequential hardware, and whose cost per cycle is typically much lower than a high performance uniprocessor. The difficulty lies with parallel software: how to write it, how to translate it to something that can be efficiently executed, and how to port it to new platforms as they are developed. Our inability to solve any of these problems well has led some to talk of a "parallel software crisis."

It is not really surprising that industry has failed to use parallelism enthusiastically. Although the average performance of parallel architectures has steadily improved, the family of architectures that has the best performance at a given time has changed frequently. There is little sign that a candidate for "best architecture over the long term" can be agreed upon. Indeed, the development of optical devices for communication and, more slowly, for computation shows every sign of confusing the picture even more. There has therefore never been a plausible architecture family that a company could select with confidence in its medium term future.

Around each family of architectures has grown up a collection of computation models, languages, compilation techniques, and idioms; and a group of programmers proficient in their use. This family-specific knowledge means that neither software, nor programmers, can easily migrate from one architecture family to another. Indeed, it is not unknown for software to fail when moved from one machine to a larger one from the same family. Since software lifetimes are normally measured in decades, while machines change much more frequently, a huge investment is needed to keep parallel software running. This situation is made much worse for companies who already have large amounts of "dusty deck" software, since it is never clear which computation model they should migrate it to. Unsurprisingly, they decide to wait and see.

A good solution to the parallel software development problem is important to the research community. If a good one is not found, the first workable one is likely to be adopted and that, in the long run, will be an expensive error.

Let us consider what a solution might look like. There are two identifiable groups who want different things: programmers and implementers. The interface between them is a model of parallel computation. Programmers benefit from high levels of abstraction while implementers prefer a lower level. Selecting an appropriate model therefore requires a careful choice of the level of abstraction that maximises the benefits to both. The requirements of each group are so strong that, until recently, there was little belief that a useful model could be found. Fundamental results of the last five years have changed this view.

Let us consider what properties a model should have from the point of view of each group. Programmers want:

- A methodology for software development that is independent of any particular architecture family. This includes both a programming language that is architecture independent and an approach for developing software. Current practice is to carry over into the parallel domain what we do to develop sequential software; it seems improbable that this will scale and, if correctness is a concern, as it surely must be, the "build then verify" sequence seems impossibly difficult for parallel systems.

- A way of measuring software's cost of execution that is architecture independent but accurate. Without this it is not really possible to design software, since the trade-offs between different

implementations can only be done by guesswork.

- An intellectually manageable model of what is happening when the software executes. Because, in the long run, parallelism will be massive, the model must reduce or eliminate the cognitive burden of decomposition or partitioning into threads or processes, the explicit description of communication, and the specification of synchronisation between threads. The model must therefore be independent not only of the architecture family but also of the number of processors to be used and of the interconnection topology.

- A migration path that specifies how to alter a program when the underlying target architecture is changed. For models that are sufficiently architecture independent, this is not necessary; even for models that lack architecture independence, it is an advantage to make the migration path the responsibility of the implementer. For models that are not architecture independent and do not provide a migration path, each new platform requires a complete redesign of software.

Implementers want the following properties of a model:

- It must be able to be "efficiently" implemented on a full range of current and foreseeable architectures. This must include at least some members of the following architecture families: sequential, vector processors, shared memory MIMD machines, hypercube-based distributed memory MIMD, distributed memory MIMD with constant valence interconnection topologies, and possibly SIMD machines.

  By "efficiently" implemented, we mean that the time-processor product for the implementation should not be asymptotically worse than the time-processor product of the best known abstract (that is, PRAM) implementation of the same program. In other words, the implementation must preserve the work of the abstract program. This is the strongest requirement we can make – in general real implementations cannot match just the time of the best abstract program because real communication takes longer than unit time.

- A methodology for architecture-dependent transformation, code generation, and choice of number of processors (all of the decisions that have been kept from the programmer) that produces these efficient implementations.

- A migration path that specifies how to alter a program to fit a new architecture. If the model is sufficiently powerful, this may just be recompilation and the use of a new set of architecture-dependent transformations.

The difficulty with finding such a model is the apparent intractability of the mapping problem. In general, a parallel program has some reference structure that can be represented as a graph and must be mapped to an architecture with some interconnection topology. This mapping is known to be NP hard. It has come to be realised that plausible solutions that avoid the full generality of the mapping problem exist. For example, *uniform architectures* dispense with locality in exchange for bounded delivery times on all communications, by using memory hashing [?] or randomized routing [?]. Thus the topology of the target architecture becomes irrelevant to mapping since the target appears fully connected with some bounded transit time. Another kind of solution involves restricting the full generality of programs that can be written to certain primitives with known computation or communication patterns. Experiments with this idea include adding operations such as *scan* [?], *multiprefix* [?], skeletons [?], P^3L [?, ?], *paralations*

[?], and the *scan vector model* [?]. In these approaches, the complexity of the mapping problem is avoided by reducing the topological structure of the program.

2 Some Proposed Models

Let us consider some proposed models to see how well they satisfy these criteria. We will examine the PRAM model, Valiant's *Bulk Synchronous Parallel* (BSP) model [?, ?], Linda [?, ?], and higher order functional programming [?]. This is only a small selection of the models of parallel computation that have been suggested – a survey covering many more can be found in [?].

The PRAM Model does not meet the needs of the programmer very well. It requires a complete description of the partitioning of code among processors and of the way in which memory is arranged to provide communication. The difficulty of writing computations using the PRAM model can be clearly seen by two facts: almost all PRAM computations are actually SIMD, although the model doesn't require it; and a paper describing a single computation in the PRAM model is still considered publishable. Cost measures for the PRAM model exist. Valiant has shown [?] that, for implementations on uniform architectures (those that use randomized techniques to get bounded memory access times), PRAM costs can be realized on shared memory and hypercube machines. Implementations on distributed memory machines with constant valence interconnections have an unavoidable overhead of order logarithmic in the number of processors. These implementations require the use of parallel slackness, an excess of virtual over physical parallelism. Thus the PRAM model is architecture independent in the sense that a PRAM computation can be automatically implemented over a range of architectures.

From the implementer's point of view, the unavoidable overhead in implementation on some architectures means that efficient implementation, in our sense, is not possible. This is especially unfortunate since the family that causes the problem is the most easily scaled. Apart from this, the PRAM model is easy to implement because it is low level; the choice of number of processors is dictated by parallel slackness considerations. However, it fails to provide a migration path in the sense that there is general method for changing a program, either before or during compilation, to take advantage of some architectural feature. For example, parsing context free languages can be done in time $O(n)$ on $O(n^2)$ processors or in time $O(\log n)$ on $O(n^6)$ processors, but the two algorithms were discovered independently and there is no obvious way to transform one into the other.

The problem with efficient implementation of the PRAM model on distributed memory systems with constant valence topologies is that the amount of communication generated at each step can easily overrun the bandwidth provided by such a topology. Valiant therefore suggested the *BSP model* [?, ?], a version of the PRAM in which communication between threads is restricted in frequency. The actual restriction is based on parameters of the architecture: l, the minimum time between global communication operations, and g the ratio between processor speed and network delivery speed. Thus, given a computation, its performance on a machine for which these characteristics are known can be computed. Like the PRAM model, the BSP model requires programmers to be aware of partitioning and placement, and access to memory. In fact, it is arguably more complex since programmers must check that global operations occur sufficiently infrequently. Migration is harder than for the PRAM, since changing the number of processors in an implementation will change l and may require recasting the whole computation to alter the frequency of global operations.

For the implementer, the BSP model is similar to the PRAM model, except if changing granularity (that is, frequency of global operations) is done at compilation, which may the only way to make it

practical.

Linda [?, ?, ?] is a model that provides an abstraction of a content-addressable shared memory. For the programmer, the overhead of managing partitioning is still present, but that of communication is reduced because of content addressability, and that of synchronisation disappears. Software development still requires a message passing idiom, and correctness is presumably difficult because of the inherent non-determinism of input from tuple space. The biggest weakness of Linda for software development is that it completely hides any idea of the cost of a computation from the programmer – it is not possible to assume anything about the response time of tuple space accesses.

For the implementer, a reasonably efficient implementation of tuple space is a challenge. For a distributed memory implementation on a constant valence interconnection, it is clear that a logarithmic order penalty is unavoidable. The semantics of *in* that requires destructive reads from tuple space makes this even harder.

Higher order functional programming, unlike these other models, is abstract and therefore better for the programmer. It comes with a methodology for software development (equational transformation). Nothing need be said at the program level about partitioning, communication or synchronisation. Cost measures (somewhat coarse ones in terms of function call counts) have been developed [?, ?, ?], and linear logic could be used to develop better ones.

This abstraction at the programmer level causes difficulties for the implementer. Partitioning, communication, and synchronisation must be inferred, at compile time or dynamically during execution. This has been done with some success for shared memory machines but has not been very successful for other architecture families.

Data parallel models are usually relatively easy to implement on parallel architectures because of the explicit and limited nature of their communication and computation operations. They are also attractive because they are single-threaded and thus do not require programmers to explicitly think about communication and synchronisation. However, their weakness in general lies in the area of software development; the choice of data parallel operations is usually *ad hoc* and the relationships between them hard to discover and exploit.

In the next section we introduce the Bird-Meertens formalism, a model of parallel computation that combines an abstract view for the programmer with straightforward and efficient implementability.

3 The Bird-Meertens Formalism

The Bird-Meertens formalism is an approach to software development and computation based on theories, that is categorical data types and their attendant operations. A theory is built from the constructors of a type using a categorical construction based on initiality and developed by Malcolm [?], Spivey [?], and others. Unlike abstract data types, categorical data types have operations, equations relating them, and a guarantee that all of the required operations and equations are present. This guarantee of completeness is the major advantage of categorical over abstract data types.

Beginning with some set of constructors, the categorical construction gives the following:

- a polymorphic datatype described by the constructors; that is there is a constructed type for each existing type;
- operations defined on the new datatype, including a generalized *map* operation and a generalized *reduction* operation; the set of operations is fixed and the communication and computation pattern

each embodies can be exploited both by the compiler and at run time;

- equations on the new operations and on the constructors defining how the new operations relate to each other and how to evaluate homomorphisms in terms of the constructors;
- a guarantee of the completeness of the set of equations in the sense that any formulation of a homomorphism on the type can be transformed into any other formulation by equational substitution;
- a guarantee that any function on an object of the constructed type that relies only on its type properties can be expressed in terms of the new operations;
- locality of reference for homomorphisms if it is present in the constructors.

BMF theories have been built for bags, *cons* lists [?], *cat* lists [?], trees [?], and arrays [?]. BMF programs are compositions of these (second order) operations on appropriate datatypes.

To illustrate we describe the BMF theory of lists. We use concatenation lists, since the theory of cons-lists is inherently sequential. Lists are built using three constructors:

$$\begin{aligned} [\cdot] &: \alpha \rightarrow \alpha* & [\cdot]a &= [a] \\ [\,] &: \mathrm{unit} \rightarrow \alpha* & [\,] &= K_{[\,]} \\ +\!\!+ &: \alpha* \times \alpha* \rightarrow \alpha* & [as] +\!\!+ [bs] &= [as, bs] \end{aligned}$$

The new operations on the type are:

- For any function $f : \alpha \rightarrow \beta$ a function $f* : \alpha* \rightarrow \beta*$;
- For any $M = (\alpha, \ \ , \mathrm{id})$ a monoid, a reduction $\ \ / : \alpha* \rightarrow M$

Both of these operations contain inherent parallelism and a fixed communication pattern; *map* is completely parallel and requires no communication, while *reduction* is evaluated in an obvious tree-like way and contains significant parallelism. Equations such as the following hold for these new operations:

$$\begin{aligned} f* \cdot [\cdot]_\alpha &= [\cdot]_\beta \cdot f \\ f* \cdot +\!\!+_\alpha &= +\!\!+_\beta \cdot (f*, f*) \\ f* \cdot []_\alpha &= []_\beta \\ \mathrm{id}_\alpha* &= \mathrm{id}_{\alpha*} \\ (g \cdot f)* &= g* \cdot f* \\ / \cdot []_\alpha &= \mathrm{id} \\ / \cdot [\cdot]_\alpha &= \mathrm{id}_\alpha \\ / \cdot +\!\!+ &= (\ / \ ) \ \ (\ / \ ) \\ h \cdot \ / &= \ / \cdot h* \ (h \text{ a homomorphism}) \\ h &= \ / \cdot (h \cdot [\cdot]_\alpha)* \ (h \text{ a homomorphism}) \end{aligned}$$

Because concatenation is a list operation with locality, it follows that evaluating homomorphisms on lists will also be an operation with locality. Other operations that can be defined in terms of these basic operations are

- **prefix** (written $/\!/$), which given a list of values returns a list of prefixes of these values by applying an associative operator:

$$/\!/[a_1, a_2, \cdots, a_n] = [a_1, a_1 \quad a_2, \cdots, a_1 \quad a_2 \quad \cdots \quad a_n]$$

- **inits** which generates all of the initial segments of its argument list:

$$inits[a_1, a_2, \cdots, a_n] = [[a_1], [a_1, a_2], \cdots, [a_1, a_2, \cdots, a_n]]$$

- **zip** (written), which combines two lists of the same length by applying to the pairs with one element from the first list argument and the other from the second:

$$[a_1, a_2, \cdots, a_n] \quad [b_1, b_2, \cdots, b_n] = [a_1 \quad b_1, a_2 \quad b_2, \cdots, a_n \quad b_n]$$

A simple example of a homomorphism on lists is *sorting* since

$$\text{sort}(as +\!\!+ bs) = \text{sort}(as) \quad \text{sort}(bs)$$

where is the binary *merge* operation. Thus *sort* can be written, using the last equation of the list above, as

$$\text{sort} = \quad / \cdot ([\cdot])*$$

where $[\cdot]$ is the *make singleton* function and the occurrence of *sort* on the right hand side has been deleted because a singleton is necessarily already sorted. Many other examples of the use of the theory of lists can be found in [?, ?, ?, ?]; a particularly interesting example is the derivation of the Knuth-Morris-Pratt string matching algorithm [?]. Examples of operations on trees such as parenthesis matching and attribute grammar evaluation can be found in [?].

We now turn to evaluating the Bird-Meertens formalism by the criteria we suggested earlier.

Methodology. The BMF style views software development as the transformation of an initial solution to a given specification into an efficient one. Many initial solutions have the form of comprehensions. It is straightforward to check that the result of a particular comprehension meets a specification but it is typically an expensive solution computationally. Using equations from theories, such a program can be transformed into one that is more efficient. Since transformation is equational, correctness is necessarily preserved. The completeness of the equational system guarantees that the optimal solution is reachable, and that mistakes made during derivation are reversible. A program that computes a homomorphism has a normal form called a *catamorphism* into which it can be translated.

Cost Measures. The difficulties of implementation on distributed memory constant valence topology architectures can be avoided by ensuring that all communication occurs between near neighbours. The locality of operations on categorical data types reflects the locality of the constructors of the data type, that is, the extent to which building an object of the type involves putting together pieces at a small number of points. For example, concatenation involves joining lists at a single point; a constructor using a cartesian product does not. For all the types so far considered, locality does seem to be a natural property of constructors, so that the resulting BMF theories will exhibit considerable locality.

Since local communication can be implemented in constant time, the cost measures for BMF programs can be those of "free communication" models. Such cost measures only account for operations and their

arrangement. We have shown [?] that two appropriate measures are Blelloch's vc, the total number of operations performed by a computation, and sc, the length of the computation's critical path [?]. From these, an implementation on p processors has a time complexity of

$$T_p = O(vc/p + sc)$$

by a simple extension of Brent's theorem [?]. This time complexity holds for implementations on a wide range of architectures, provided that BMF computations can be mapped onto their topologies while preserving near neighbour adjacencies. This can certainly be done for lists, and there seems to be no fundamental impediment for other, more complex, types.

Intellectual Manageability. BMF programs are single threaded and data parallel. They are thus conceptually simple to understand and reason about. In particular, decomposition occurs only in the data, and communication and synchronisation are hidden within the fixed set of second order functions.

Migration Path. The completeness result shows that any form of a computation can be transformed into any other when that is appropriate. However, the model is so architecture-independent that there is never a need to do this at the programmer level. It can be taken care of by the compiler, where all architecture-dependent transformation occurs.

From the implementer's perspective, we are concerned with how to make such systems execute efficiently.

Efficient Implementation. Bird-Meertens theories are not very demanding of architectures. They require only a subgraph in the architecture topology that matches the structure of the relevant datatype's constructors (path for lists, log depth spanning tree for trees, mesh for arrays). All communication is local so that the global communication structure is of little relevance.

The choice of number of processors to use can be made by using the cost measures outlined above. If a computation has costs vc and sc then a locality-based implementation can compute it on p processors in time $O(vc/p + sc)$. Therefore the best choice of p is the one that makes the two terms vc/p and sc most nearly equal. Using more processors than this doesn't provide further speed-up because of the critical path; using fewer misses potential speed-up.

The number of processors will not, in general, match the size of the data structures being used. Thus these structures have to be aggregated so that a piece can be allocated to each processor. It turns out that the best way to do this is to allocate contiguous pieces of the data structure to each processor rather than to use a round-robin allocation [?, ?]. Code generation for an operation over the whole structure now involves generating two kinds of instructions: one to take care of the (sequential) computation on the piece within each processor, and a second to take care of the global computation on the results from each processor. Thus each operation f, applied to a data structure of size n on p processors, decomposes into a sequential operation on a piece of size n/p executed in parallel on all p processors, followed by a parallel operation of size p to compute the final result.

This kind of code generation can be expressed equationally. For example, if we use subscripts to indicate the size of piece to which an operation applies and an overbar to indicate the sequential version of an operation, then a reduction applied to a list of n values, computed on p processors, satisfies the equation:

$$(\ /)_n = (\ /)_p \cdot (\overline{(\ /)}_{n/p})*_p$$

Given p, this equation describes precisely what code should be generated for the original instruction. We

can also compute the actual running cost on p processors from the equation by summing each of the parallel steps; in this case we get

$$t_p = \log p + n/p$$

For other operations, the implementation equation describing how to compute it in pieces can be quite complex. For example, computing a prefix operation, we find that the implementation equation is

$$/\!/_n = (\overline{(+x)*}_{n/p}) *_p \cdot \text{shift} \cdot \quad /\!/_p \cdot (\overline{\quad /\!/}_{n/p}) *_p$$

with cost

$$t_p = \frac{n}{p} + 1 + \log p + \frac{n}{p}$$

We have built a small compiler that generates code for a Transputer system [?] and shown how these ideas can be worked out in practice.

Architecture-Dependent Transformation. Such transformation is not necessary in the implementation of the Bird-Meertens approach, since all architectures with an appropriately rich topology look the same to the compiler until final code generation.

Migration Path. The completeness of the equation set for each datatype means that it is possible in principle to convert any form of a program satisfying a given specification into any other. Doing this to find a new algorithm requires human reasoning at present, although there is potential for automating part of the process. We have built a small transformation assistant that allows a programmer to select a subpart of a program, be shown all of the rules that apply, and select one. Replacement of the altered part is then done automatically. It is possible to add cost information to this kind of assisted transformation system although, of course, most developments do not simply reduce cost in a monotonic way. Low level transformation is not really required to provide a migration path because of the level of architecture independence of the model.

4 Conclusions

It is clear that data parallelism has much to commend it as the standard way to program massively parallel machines and still retain portability. Its two advantages are that it hides much of the complexity of managing massively parallel threads, and that it requires only a fixed set of global operations to be implemented on each potential target architecture. Of course, programming in the data parallel style is a restriction on how computations can be arranged and there will be those for whom this seems unattractive or cumbersome. The expressiveness question can only be resolved with sufficient experience in data parallel programming and the development of programming assistants (transformation systems, optimizing compilers) for data parallel software development.

The principal weakness of existing data parallel models is the *ad hoc* choice of the operations allowed on each data type. Choices are often made with expressiveness in mind, but no proof of completeness in the sense we have been discussing is usually possible. The main advantage of the Bird-Meertens approach is the guarantee of completeness that the CDT construction provides. Nevertheless there are substantial pay-offs in use and compilation because of the underlying regularity and elegance of the approach, as we have tried to illustrate.

The future of software for parallel computation depends on making it a mainstream technology and thus bringing the benefits of parallel hardware within reach of the average industrial user. Since portability

seems, at the moment, to be the biggest stumbling block to this goal, development of architecture independent models should be a priority of the research community. The Bird-Meertens formalism is a very attractive candidate.

Acknowledgements: The compiler was developed by Wentong Cai and the transformation tool by Scott Kirkwood. This work was supported by the Natural Sciences and Engineering Research Council of Canada.