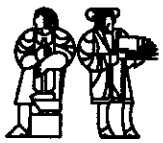
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Dataflow Programming Languages

Computation Structures Group Memo 333
March 18, 1991

Rishiyur S. Nikhil

To appear in *Proc. 13th IMACS World Congress on Computation and Applied Mathematics, Trinity College, Dublin, Ireland, July 22-26, 1991.*

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Dataflow Programming Languages

Rishiyur S. Nikhil
Laboratory for Computer Science
Massachusetts Institute of Technology

545 Technolology Square, Cambridge, MA 02139, USA

Introduction

Dataflow languages are attractive for parallel computation because of their expressive power—their high-level, machine-independent, implicitly parallel notation— and because of their fine-grain parallelism, which seems essential for effective, scalable utilization of parallel machines.

Dataflow languages are no longer tightly coupled with dataflow architectures. Today, we understand the compiling issues for dataflow languages well enough that we can recommend these languages to programmers without having to rely on architectural support.

There are several dataflow languages described in the literature, including Id [7], Sisal [6], and Val [1]; due to space limitations, we will base our discussions here on Id.

Expressive Power of Dataflow Languages

"Expressive power" does not have a precise definition; one can make reasoned arguments about why one language is more expressive than another, but ultimately the judgment is subjective. Theoretically, FORTRAN and assembly language have the same computational power, but almost everyone would agree that FORTRAN is more expressive, because it is architecture-independent and because it requires less detailed specification of problem solutions and is "closer" to the way human programmers think about problems.

Dataflow languages are based on *functional programming languages.* Unlike most programming languages that have evolved upwards from machine languages, functional languages are based on the λ-calculus, a mathematical theory of functions that predates digital computers [3].

Functional notation:

An aspect of the expressive power of functional languages is that one can manipulate complex objects (entire data structures, and even functions themselves) as ordinary values. Consider this function:

```
def map2 f A B = ...  details omitted ...
```

taking 3 arguments: a function `f` and two arrays `A` and `B`, and returning as its result a new array (call it `C`) where

```
C[j] = f A[j] B[j]
```

i.e., it applies `f` to each pair of components of `A` and `B` and returns an array containing the results. (In Id and other functional languages, it is common to use a "curried" notation, omitting parentheses, commas, semicolons *etc.* around the arguments of a function.)

Now, we can specialize `map2` to define a vector sum function:

Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

```
def vsum A B = map2 (+) A B ;
```

Here, the function "+" is supplied as the `f` parameter to `map2`, so that `vsum` adds the vectors `A` and `B` pointwise, and returns a vector containing the sums. Further, since `vsum` is itself an addition function, we can perform vector addition on vectors whose components are themselves vectors:

```
def vvsum AA BB = map2 vsum AA BB ;
```

Another useful higher order function is:

```
def foldl f z A = ... details omitted ...
```

which computes the "`f`-reduction" of array `A`, where `z` is the zero of the function `f`, *i.e.,* it computes:

```
f ...(f (f z A[1]) A[2]) A[n]
```

Using `foldl`, one can express the inner-product of two vectors:

```
def ip A B = foldl (+) 0 (map2 (*) A B) ;
```

Here, `map2` multiplies `A` and `B` pointwise, and `foldl` computes the sum of the resulting vector of products.

The ability to manipulate entire complex objects and functions themselves as values relieves the programmer of the tedium of descending down to explicit loops and assignments at every stage— such details can be hidden in a few generally useful functions such as `map2` and `foldl` [2]. Such higher order functions are the "power tools" of functional programming, in the sense that they amplify the programmer's effort.

Implicit parallelism:

Another aspect of expressive power is implicit parallelism. In most parallel languages, the user must carefully orchestrate parallelism: explicitly partition the program into parallel processes, specify the placement of data, specify the placement and scheduling of processes, *etc.* Often, these details are architecture specific, and sometimes even configuration specific, resulting in fragile, non-portable code.

In functional languages (and in all our examples above) we do not have to specify what can be done in parallel. In principle, everything can be performed in parallel, limited only by data dependencies. For example, the implementation of `map2` may perform all the applications of `f` in parallel. Thus, functional languages do not increase the complexity of the programmer's task when moving from a sequential to a parallel environment.

M-structures: implicitly synchronized state:

In pure functional languages, there is no concept of "state," *i.e.,* structures whose contents change during program execution. There are no assignment statements— programs are functions that compute new values from old. While many algorithms can be expressed clearly in this style, there are some problems.

One difficult area is: algorithms making effective use of non-determinism. Functional languages have the Church-Rosser property [3] which guarantees that the answer produced by a functional program does not depend on the particular parallel execution structure chosen by an implementation. This is usually a major advantage; unlike other parallel languages, functional programs are not subject to *race conditions* leading to non-deterministic and irreproducible behavior (which complicates debugging).

However, consider the problem of traversing a directed graph to count the number of nodes reachable from a given root node. The following strategy seems simple: starting at the root node, fan out in parallel on all outgoing edges from each visited node. To avoid repeated traversals of shared subgraphs (and cycles), the first traversal to reach a node marks it "visited," so that subsequent traversals reaching the node via other paths will observe the mark and retreat. Non-determinism is exploited because, of all the incoming traversals arriving at a shared node, we do not care which one arrives first to mark the node and traverse its successors. There is no way to express this idea in a functional language— one must pick a deterministic order to traverse the nodes. This not only clutters up the program significantly, but also sequentializes it.

Introducing non-determinism into a functional language is not very different from introducing state— constructs for one can be used to simulate the other. Not surprisingly, functional languages also have difficulty in dealing with input-output, which seems quintessentially linked to the notion of state.

In Id, the functional core is extended with *M-structures* which are updatable data structures. Unlike other languages where updates are protected by separate synchronization primitives (*e.g.*, semaphores), accesses to M-structures in Id are combined with implicit synchronization. For example, a component of an M-structure array (an "`M_array`") can be incremented using the statement:

```
A![j] = A![j] + 1
```

The expression `A![j]` on the right hand side not only reads, but simultaneously locks the location. The assignment "`A![j] = ...`" not only writes the value back, but simultaneously unlocks the location. This guaratees that the increment is *atomic*, *i.e.*, even if several computations execute this statement concurrently, the increments will be done properly. Coupling accesses with synchronization in this manner leads to clear and concise programs [4].

Implementations and performance

Dataflow implementations have become increasingly sophisticated in recent years.

Compilers for dataflow languages do not have the problem of detecting parallelism; rather, they often have the opposite problem of trying to contain excess parallelism. Without optimization, dataflow programs can be quite voracious in their resource demands. For example, a direct interpretation of the `map2` function would allocate a new vector for each result that it computes. In general, this storage must be allocated dynamically, and it is necessary to have a garbage collector to recycle memory occupied by structures no longer in use. A related problem is computational cost. If we have a "point" represented as a vector of 3 values (x, y, z), and we wish to displace it one unit along the x-axis, functional semantics demands that we produce a new vector containing $(x+1, y, z)$. Notice that we have to copy y and z. These overheads can be quite high. Thus, reuse of storage is essential for efficient implementations.

For Sisal, which is a pure functional language, researchers have developed program analysis techniques that allow the implementation to reuse storage heavily [9]. Sisal researchers report that on the Livermore loops, they are now able to compile code for Cray supercomputers that is competitive with FORTRAN codes.

The Id compiler [10] is unable to use the Sisal analysis techniques directly because of higher-order functions, non-strictness and recursive data structures. However, it is a highly optimizing compiler that uses lifetime analysis, loop bounding and several other novel optimization techniques. The Id

compiler is currently targeted to the Monsoon dataflow machine that is being built as a collaboration between MIT and Motorola, Inc. At time of this writing (March 1991), complete uniprocessors have just become operational, and 8-processor Monsoon machines are expected in April 1991. We have run only a few small benchmarks (such as matrix multiplication), but it appears that Id on Monsoon can be competitive with C on a modern workstation.

Fine grain parallelism seems essential if we are to simultaneously achieve the goals of general-purpose parallel programming and scalable performance. As processors become faster and machines become more parallel, memory latencies are becoming relatively larger. A general way to address this is rapid, fine-grain thread switching, *i.e.*, each processor needs a large pool of threads amongst which it can be efficiently multiplexed so that it can always do useful work while some threads are waiting (for long-latency memory requests, for synchronization, for results of procedures, *etc.*).

Dataflow languages and their compilers offer a systematic, complete approach towards this computational model. New compilers are under construction at MIT and Berkeley that use dataflow principles to compile Id even for non-dataflow machines [8, 5]; these compilers should enable Id to become widely available.

Conclusion

There is an interesting parallel between parallel programming today and sequential programming in the early 1950's. At that time, programs were written in assembly language, and programmers had to be acutely aware of the architectural details of their machines. There was widespread skepticism that high-level, machine-independent programming was possible with acceptable efficiency. John Backus and his group changed all that, with their outstanding FORTRAN implementation. Today, parallel programmers have to be acutely aware of the architectures of their parallel machines, and there seems to be widespread skepticism that high level, machine independent parallel programming is possible with acceptable efficiency. Our hope is that dataflow languages like Id can do for parallel programming what FORTRAN did for sequential programming. The prospects look bright.

References

[1] W. B. Ackerman and J. B. Dennis. VAL– A Value-oriented Algorithmic Language: Preliminary Reference Manual. Technical Report TR-218, MIT LCS, 545 Tech. Sq., Cambridge, MA 02139, June 1979.

[2] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *CACM*, 21(8):613–641, August 1978.

[3] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North Holland, Amsterdam, 1981.

[4] P. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. Technical report, MIT LCS, 545 Tech. Sq., Cambridge, MA 02139, March 1991. Submitted for publication.

[5] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Threaded Abstract Machine. In *4th*

Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1991.

[6] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, P. Hohensee, and I. Dobes. SISAL Reference Manual. Technical report, Lawrence Livermore Natl. Lab., 1984.

[7] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-1, MIT LCS, 545 Tech. Sq., Cambridge, MA 02139, USA, July 1990.

[8] R. S. Nikhil. The Parallel Programming Language Id and its Compilation for Parallel Machines. In *Proc. Wkshp on Massive Parallelism, Amalfi, Italy, October 1989.* Academic Press, 1990 (to appear).

[9] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, Lawrence Livermore Natl. Lab., 1987.

[10] K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. Technical Report LCS TR-370, MIT LCS, 545 Tech. Sq., Cambridge MA 02139, August 1986.