LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Dataflow Approach to General-purpose Parallel Computing

Computation Structures Group Memo 302
July 7, 1989

Arvind
Rishiyur S. Nikhil

Prepared for the proceedings commemorating the
25th Anniversary of Project MAC.

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 N00014-84-K-0099.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Dataflow Approach to General-purpose Parallel Computing

Arvind
Rishiyur S. Nikhil

July 7, 1989

1 Introduction

Our dataflow group at MIT has two goals related to parallel processing:

- To raise the level of parallel *programming* by designing and implementing an expressive, powerful programming language in which the natural description of an algorithm has abundant parallelism, without having to specify partitioning, mapping and scheduling.
- To design architectures that are more suitable to the requirements of parallelism than the traditional von Neumann design that has served so well for uniprocessors in the past.

While each goal is independently worthwhile, they achieve an exciting symbiosis when pursued together.

Our approach is based on the research programming language Id, its compilation to Tagged Token dataflow graphs— a parallel machine code— and architectures for direct execution of dataflow graphs. Our current architectural focus is on an abstract model called the Explicit Token Store model and its concrete implementation in the Monsoon dataflow machine. In this article we provide an overview of our approach.

2 Functions and Reduction

We are all familiar with the following simple notion of computation. Given an expression like this:

```
(2 + 3) + 4
```

we can *reduce* it (or *rewrite* it) to the expression

```
5 + 4
```

which can be rewritten further to the expression

```
9
```

No further reductions can take place, so we call this the "result" of the computation.

This simple principle is the basis of computation in functional languages such as Id. In addition to the fixed repertoire of rewrite rules for primitive operators such as "+", the programmer may specify additional rewrite rules through *function definitions*.

We begin with a simple example. It doesn't do much— it simply takes two arguments, adds them up, and returns the result.

```
def plus x y = x + y ;
```

The identifiers `x` and `y` are called *formal parameters* or *arguments* because they are just dummy names that stand for actual values, or *actual parameters* that will be supplied when this function is used.

The function can be read as a rewrite rule, *i.e.,* whenever we see an expression that matches the left-hand side:

```
plus e1 e2
```

where e_1 and e_2 are any expressions, we can always reduce it as specified by the right-hand side, to

```
e1 + e2
```

Suppose we were given the expression "`plus 2 3`". We can perform the rewrite as follows:

```
plus 2 3 ⟹ 2 + 3 ⟹ 5
```

A more complicated example:

```
      plus (2 * 3) (plus 2 3)
⟹    plus 6 (2 + 3)
⟹    plus 6 5
⟹    6 + 5
⟹    11
```

In general, whenever we see any expression that matches the left-hand side of a function definition, we can always replace it by the right-hand side, substituting formal parameters by actual parameters. By repeatedly performing such rewrites, we reduce the program expression to its result, *i.e.,* an expression that cannot be rewritten any further.

3 Parallelism and determinacy

At each point in the computation there may be many rewrite rules that are applicable, *i.e.,* many sub-expressions that are reducible. In our example above, we could have chosen to perform the rewrites in the following order instead:

```
      plus (2 * 3) (plus 2 3)
⟹    (2 * 3) + (plus 2 3)
⟹    6 + (2 + 3)
⟹    6 + 5
⟹    11
```

Happily, the result is still 11, as before. In fact, this is no accident— it is a very deep property of functional languages called the Church-Rosser property. It arises from the fact that, as in mathematics, expressions do not have any side-effects— each expression uniquely stands for some value, and can be replaced by any other expression denoting the same value without changing the meaning of the program. Reductions in a functional language are thus confluent, or determinate, *i.e.,* no matter what order we choose to perform the reductions, the resulting value for all terminating reduction sequences is always the same. This has major implications on parallel computation:

- We are free to perform as many rewrites as we wish in parallel.
- We have great flexibility in scheduling the computations on a real machine, because the order in which we choose to do the reductions can affect termination but not the result value.

Determinacy is an invaluable property for parallel programs, because it guarantees repeatability of results. It is extremely difficult to debug a program if different runs of the same program on the same inputs can produce different answers.

4 Higher order functions

Consider the following expression:

```
{  f = plus 1
 In
   f 3 }
```

which says: let `f` name the value of the expression "`plus 1`"; using this name, compute the value of the expression "`f 3`".

Is the expression "`plus 1`" meaningful? Is there an argument missing? No! We treat "`plus 1`" as an expression whose value is itself a function of one argument, *i.e.,* `f` represents a function that adds one to its argument and returns its value.

Intuitively, the definitions in a block can themselves be regarded as rewrite rules that are used in reducing the "return expression" of the block (*i.e.,* following the `In` keyword) to a value:

```
      {f = plus 1 In f 3 }
⟹    {f = plus 1 In (plus 1) 3 }        using the definition of f
```

where "`(plus 1) 3`" is a fully parenthesized version of "`plus 1 3`". By convention, application associates to the left, so we can drop the parentheses. Continuing,

```
      {f = plus 1 In plus 1 3 }
⟹    {f = plus 1 In 1 + 3 }             using the definition of plus
⟹    {f = plus 1 In 4 }
⟹    4
```

The last step was performed using a rewrite rule for blocks which states that when the return-expression has been reduced to a value, the block may be replaced by that value.

This notation, whereby "`plus` e_1" can itself be treated as a function, is a very clever and powerful notation found in functional languages, and is called "currying" after Haskell B. Curry, a famous logician who invented it earlier in this century. We will make much use of it in later examples.

Now, let us look at a really fascinating program.

```
def twice f x = f (f x) ;
```

In words: `twice` takes a function `f` and an argument `x`, and applies `f` to `x` twice. For example, the expression:

```
twice sqr 4
```

should apply the squaring function `sqr` twice to 4, *i.e.*, `sqr (sqr 4)`, producing 256. What about the following expression?

```
twice (plus 3) 4
```

Recall that "`plus 3`" represents a function of one argument that adds 3 to its argument. Thus, when applied twice to 4, we will add 6 to it, giving 10. Let us watch the reduction process:

```
    twice (plus 3) 4
⟹  (plus 3) ((plus 3) 4)      using the definition of twice
⟹  (plus 3) (3 + 4)           using the definition of plus
⟹  3 + (3 + 4)                using the definition of plus
⟹  10
```

Notice again the role of parentheses. The expression `(sqr sqr) 4` would indicate the application of the `sqr` function to itself, followed by the application of the result to 4. As one would expect, this is a meaningless expression— it does not make sense to apply `sqr` to itself. More formally, we say that it is a *data type error*, because `sqr` expects an argument of type integer, but is being given an argument that is a function instead.

As an interesting exercise, the reader may wish to determine what the following expression reduces to:

```
twice twice sqr 2
```

Hint: the answer is the address space of the venerable old 8080 and 6502 microprocessors!

5 Data structures

The most basic kind of data structure in Id is a *tuple*, which is just an aggregation of some component values. For example, the expression:

```
(2,3)
```

represents a 2-tuple (or pair), which is a data structure whose first component is the number 2 and whose second component is the number 3.

The tuple notation can be used in any context. For example,

```
(1+1,plus 1 2)
```

and

```
{ f = plus 1
 In
  (f 1,f 2) }
```

are also expressions that evaluate to 2-tuples containing the numbers 2 and 3.

Once a tuple is constructed, how do we gain access to its components? We use *pattern matching*. For example, suppose we want to define a function that takes a 2-tuple as argument and returns the sum of its components. This is how we would write it:

```
def add (x,y) = x + y ;
```

i.e., the formal parameter `(x,y)` is regarded as a pattern that is matched against the actual 2-tuple supplied as an argument, and the effect of this is that `x` names the first component and `y` names the second.

Again, let us observe the reduction process:

```
      add (2,3)
⟹    2 + 3
⟹    5
```

What is the difference between the functions `add` and `plus`? What happens if we said this?

```
add 2 3
```

Remember that, fully parenthesized, this really stands for:

```
(add 2) 3
```

Thus, `add` is being applied to a number instead of a 2-tuple. This is another example of a data type error. Similarly, consider:

```
plus (2,3)
```

(The parentheses are necessary here to override the default that application binds more tightly than the tupling comma.) Here, `plus` expects a number but is being given a 2-tuple, and so, this is also a type error. Here is a legal expression:

```
      add (2 * 3,(plus 4 5))
⟹    add (2 * 3,4 + 5)          using the definition of plus
⟹    (2 * 3) + (4 + 5)          using the definition of add
⟹    ...
⟹    15
```

Tuples are "first class values", *i.e.,* they can be nested within other tuples, returned as results from functions, *etc.* For example:

```
def grid n = (1,n),(1,n) ;
```

is a function that takes a number n and returns a 2-tuple, each of whose components itself contains a 2-tuple containing 1 and n. The value of the expression "`grid 10`" can be visualized as shown in Figure 1.

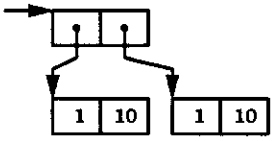

Figure 1: Value of "`grid 10`" (= `((1,10),(1,10))`)

6 Arrays

Tuples are "small" data structures whose components are specified and selected by textual position. In contrast, arrays and matrices are "large" data structures whose components are specified and selected by an *index*, which is a numeric name. For example, if `A` is an array, then `A[5]` and `A[2+3]` are expressions that select the component of `A` at index 5.

In principle, an array selection like `A[5]` and a function application like `(f 5)` are very similar. However, there are two main differences:

- An array is defined only on a finite, contiguous domain of numbers. We say that an array `A` has *index bounds* l and u, indicating that it is always an error to evaluate `A[`j`]` where $j < l$ or $u < j$. The index bounds of an array are specified in the program at array-construction time.
- Pragmatically, arrays are implemented so that selection of a component always takes a fixed amount of time, independent of the size of the array or the value of the index.

This intuitive connection between arrays and functions is used to specify the construction of an array. The expression:

```
make_array (1,10) f
```

produces an array (call it X) with index bounds 1 to 10, and with components such that $X[j] = f\ j$, for $1 \leq j \leq 10$.

For example, the expression:

```
make_array (1,10) (plus 10)
```

is an array (call it X) such that $X[j] =$ `(plus 10 j)` within its index bounds, as depicted in Figure 2.

This idea is readily generalized to multidimensional arrays:

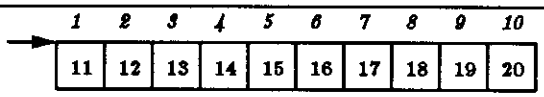

Figure 2: Value of: `make_array (1,10) (plus 10)`

```
make_matrix ((1,n),(1,n)) g
```

is a 2-dimensional array (call it Y) such that $Y[i,j] = g(i,j)$ for $1 \leq i \leq n$ and $1 \leq j \leq n$. In general, a k-dimensional array is specified using a function that takes a k-tuple argument.

For example, the expression:

```
make_matrix (grid 10) add
```

is an array (call it Y) with index bounds `((1,10),(1,10))`, within which $Y[i,j]$ has the value $i + j$, as shown in Figure 3.

	1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10	11
2	3	4	5	6	7	8	9	10	11	12
3	4	5	6	7	8	9	10	11	12	13
4	5	6	7	8	9	10	11	12	13	14
5	6	7	8	9	10	11	12	13	14	15
6	7	8	9	10	11	12	13	14	15	16
7	8	9	10	11	12	13	14	15	16	17
8	9	10	11	12	13	14	15	16	17	18
9	10	11	12	13	14	15	16	17	18	19
10	11	12	13	14	15	16	17	18	19	20

Figure 3: Value of: `make_matrix (grid 10) add`

Both examples are *declarative* specifications of arrays, *i.e.*, there is no implication of any particular order in which to compute the components. In principle, they could all be computed in parallel. The subtlety of this issue will become much more apparent in our next example.

6.1 Example: A Wavefront Computation

Suppose we want to construct an $n \times n$ matrix X as follows:

- Cells along the left and top borders contain 1, *i.e.*, $X[1,j] = X[i,1] = 1$.
- All other cells contain the sum of their neighbors to the left and to the top, *i.e.*, $X[i,j] = X[i-1,j] + X[i,j-1]$.

This can be expressed in Id as follows:

```
X = make_matrix (grid n) f ;

def f (i,j) = if (i == 1) then 1
              else ( if (j == 1) then 1
                     else X[i-1,j] + X[i,j-1]) ;
```

Some syntax explanations: in a conditional expression "*if* e_1 *then* e_2 *else* e_3", e_1 is evaluated first to a boolean value, *i.e.*, `true` or `false`; depending on this value, either e_2 or e_3 (*but not both!*) is evaluated and returned as the result. Equality of two values is tested using the infix operator "==".

There is something quite unusual going on here— in the first line, we are defining `X` using `f`, whereas in the second line we are defining `f` using `X`. The reason it makes sense is the same reason that recursive function definitions make sense— it is inductive, *i.e.*, no array component is defined in terms of itself. There is a base case where components are defined independently— when $i = 1$ or $j = 1$. Ultimately, every other component is defined in terms of these base components. Thus, the definition is perfectly meaningful.

Using our currying notation, the reader may be interested to note that the program could have equivalently been written thus:

```
X = make_matrix (grid n) (f X) ;

def f X (i,j) = if (i == 1) then 1
                else ( if (j == 1) then 1
                       else X[i-1,j] + X[i,j-1]) ;
```

Let us take a moment to examine the potential parallelism in the wavefront program. Imagine that `make_matrix` initiates n^2 computations, one to fill each component of the array. Most of these computations must *suspend* because they try to read neighboring components that are still empty. However, all computations for components in the top and left border can complete immediately, since they do not depend on anything else. When the border components at (1,2) and (2,1) have been filled, the computation for component (2,2) can proceed. As soon as it has completed, the computations for components (2,3) and (3,2) can proceed, and so on. Figure 4 shows a snapshot of the array during this process. The left and top borders, and some components on the top left have been computed. The shaded squares show the next components that can be computed, because all the components that they depend upon have already been computed. Thus, computation can proceed along a diagonal "wavefront" that sweeps across the matrix filling in components from the top left to bottom right. Parallelism grows until we reach the maximum diagonal, and then shrinks as we approach the bottom-right corner.

This behaviour is illustrated in the *parallelism profile* in Figure 5. The profile is generated automatically by GITA, a tool that can show the maximum parallelism in the dataflow graphs produced by our Id compiler. As we shall see, a dataflow graph is a partial order on instructions, and GITA plots the maximum number of instructions that can execute at each time step, assuming each instruction takes one time step.

We had stated earlier that `make_matrix` does not by itself imply any particular ordering on the computations for the array components. It is clear from the wavefront program, however,

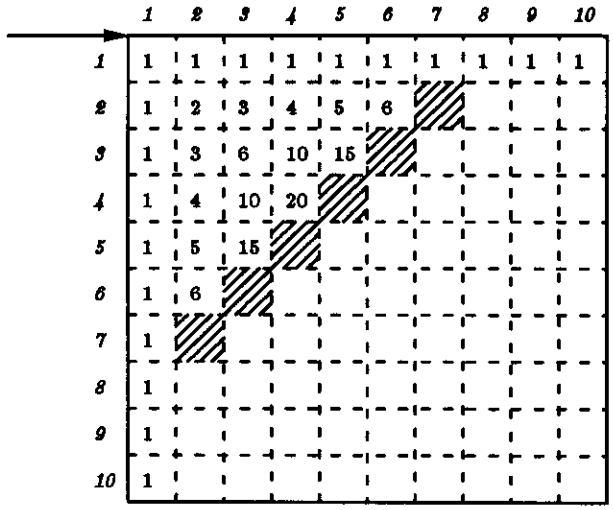

Figure 4: Wavefront computation

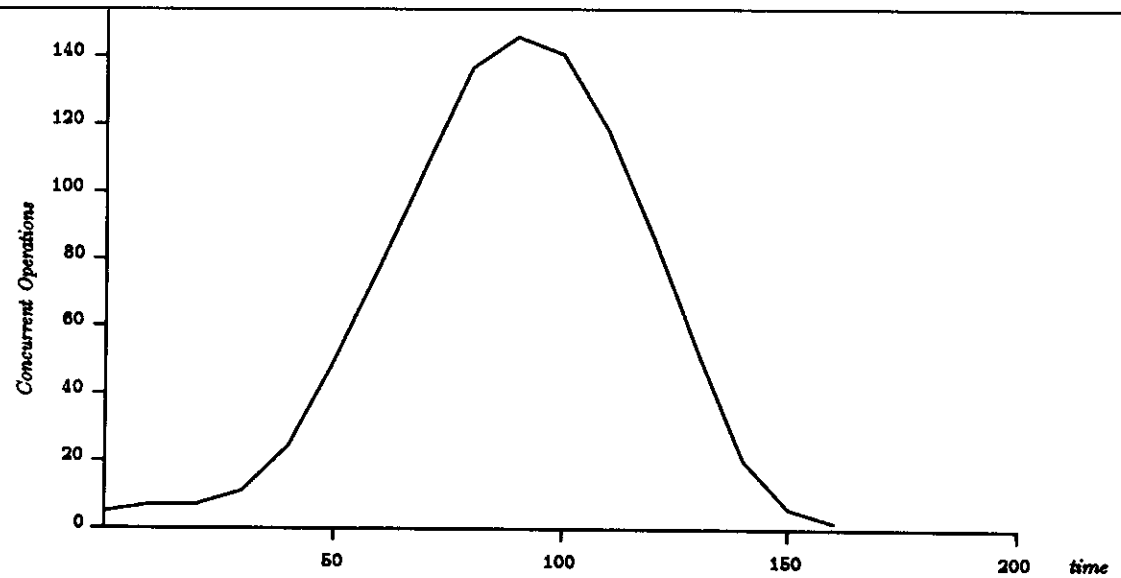

Figure 5: Parallelism profile for wavefront program

that some ordering may be implied by the *data dependencies* in a particular program. In functional languages like Id, *all* the ordering is based on data dependences; it can change from program to program and, indeed, on the particular input to a given program.

How can we implement this notion of dynamically adjusting the order of computations to accommodate the data dependencies that may arise in a particular run of a program? This is the rationale behind dataflow graphs, which constitute a parallel machine language, and dataflow architectures. Before we look at them, however, let us briefly examine conventional programming and machine languages.

7 Parallelism in conventional languages

In imperative languages like FORTRAN, a total, sequential order on computations is specified by the language definition. The compiler must, therefore, work very hard to analyze a particular program to discover where this total ordering is too conservative. If it can detect such situations, the compiler can then generate parallel code.

Let us take a moment to see how the wavefront program may be expressed in FORTRAN.

```
      DIMENSION X(10,10)

C  Initialize boundaries
      ...
C  Fill in the middle

      DO 200 i = 2,10
          DO 100 j = 2,10
              X(i,j) = X(i-1,j) + X(i,j-1)          Loop body
 100      CONTINUE
200   CONTINUE
```

For those unfamiliar with FORTRAN: the `DIMENSION` statement declares `X` to be an 10×10 array; as is usual in FORTRAN, we assume that each array index ranges from 1 to 10; comments are on lines beginning with "`C`"; the ellipses represent the code to set up the boundaries; finally, we have a doubly nested loop; for each i and each j in the range 2 to 10, the statement in the loop body specifies the computation of the (i, j)'th array element.

To run this code, we compile it (translate it) to a *machine language* representing instructions for a computer. A typical computer has two major components, a *processor* and a *memory*. The memory is a linear sequence of cells and, usually, the components of program arrays like `X` reside in a "flattened" form in the memory, for example, as shown in Figure 6.[1] Thus, if we let X_0 stand for the address of the first element of `X` then, to access element `X(2,3)`, the processor must fetch the value from memory location $X_0 + 21$.

The processor also has some local memory called *registers*. Typically, they will contain the values i, j, X_0, and other intermediate (temporary) results of the computation. In order to perform any arithmetic operation, the processor must copy the input data from the memory

[1]The figure shows what is commonly called the column-major representation, which is only one out of many possible representations of a matrix in a linear memory.

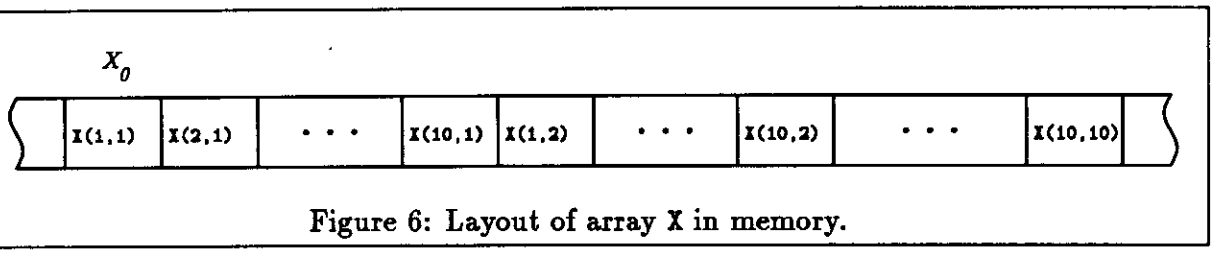

Figure 6: Layout of array X in memory.

into its registers, perform the operation locally, and then store the results back into memory. To copy data from memory location a into a register r, the processor must execute an instruction "`r := FETCH` r_a" where r_a is a register, containing the address a. To copy data back to memory, the processor must execute a "`STORE r` r_a" instruction.

Why do we not have an instuction "`r := FETCH a`"? That is, why does `FETCH` name a register r_a containing the address a instead of naming the address a directly? The reason is that a itself has to be *computed* at run time.[2] In general, to access `X(i,j)`, the processor needs to compute the address $X_0 + (i-1) + 10(j-1)$.

We can now see, in outline, the machine code corresponding to the body of the wavefront loop. We assume that `l1`, `r1`, `l2`, `r2`, `l3`, and `r3` are registers in the processor.

```
l1 :=  compute address of X(i-1,j)
r1 := FETCH l1

l2 :=  compute address of X(i,j-1)
r2 := FETCH l2

r3 := r1 + r2

l3 :=  compute address of X(i,j)
STORE r3 l3
```

In a typical processor, these instructions are executed in sequence, one at a time, from top to bottom, *i.e.,* control flows sequentially from one instruction to the next. By examining it in a little more detail, we see that, in fact, the following rearrangement of the code would also be correct.

```
l1 :=  compute address of X(i-1,j)
l2 :=  compute address of X(i,j-1)
l3 :=  compute address of X(i,j)

r1 := FETCH l1
r2 := FETCH l2

r3 := r1 + r2
```

[2] An historical aside: Originally, `FETCH` instructions in von Neumann machines *did* directly name an address. The only way to access locations whose addresses were computed dynamically was, therefore, to modify the instruction itself with new addresses at run time. This is not considered acceptable anymore, both because such programs are extremely opaque as well as because it complicates the design of the high-speed processors. Ironically, it was one of von Neumann's remarkable observations that, by allowing modification of instructions one could build a universal computing machine!

```
STORE r3 I3
```

Of course, several other orderings are possible. In general, instead of specifying a total order on instructions, we would like to specify only a partial order, as shown in Figure 7. This

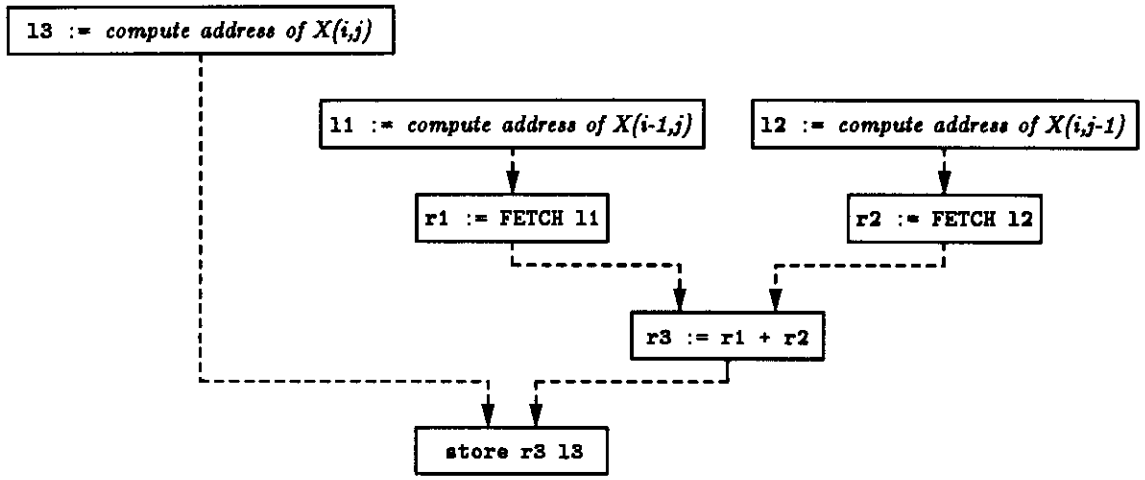

Figure 7: Partial order of instructions for loop body.

control flow graph specifies only the *necessary* constraints on the ordering. It illustrates clearly that all three address calculations can be performed in parallel and that both the FETCHes could be done in parallel.

This kind of reordering of instructions for higher performance was pioneered by Seymour Cray in the 1960's— his design of the CDC6600 was a breakthrough in that regard. It remains the key idea behind compilers and architectures for pipelined machines, RISC processors, VLIW machines, shared memory multiprocessors, *etc.*

8 Dataflow graphs

To achieve the full parallelism of the wavefront program, however, we need to go much further. We need to specify that all the address computations in *all* the iterations can proceed concurrently, not just the three within a single iteration. Much more tricky is this: we need to specify that the FETCHes in one iteration can proceed as soon as the corresponding STOREs in some previous iterations have completed. It is not easy to generalize control flow graphs to express these ideas.

It was Jack Dennis of MIT who, around 1970, developed the idea of *Dataflow Graphs* as a suitable formalism for expressing parallel computations. Dataflow graphs constitute a parallel machine language that is suitable both as a target for compilers for high-level parallel languages and as a language that can be directly executed by parallel hardware.

There are many variations on dataflow graphs in the literature. The simplest are direct expressions of an underlying hardware organization, and are often used for signal-processing

applications. Dataflow ideas are also used in many of today's supercomputers in expression evaluation and vector "chaining". For more general computations, however, one usually talks of machines that *interpret* a dataflow graph in the same sense that a von Neumann machine interprets a machine language. Static dataflow graphs allow control structures such as loops and conditionals, whereas dynamic dataflow graphs permit arbitrary recursion and data structures. In this article we concentrate on dynamic dataflow graphs that are also called "Tagged-token Dataflow Graphs".

Each node in a DFG represents an *instruction* and each edge in a DFG specifies a *data dependency* between two instructions, *i.e.*, an edge from instruction I_1 to instruction I_2 specifies that the output value produced by instruction I_1 is an input value for instruction I_2.

The dataflow graph for the loop body of the wavefront program is shown in Figure 8. Recall

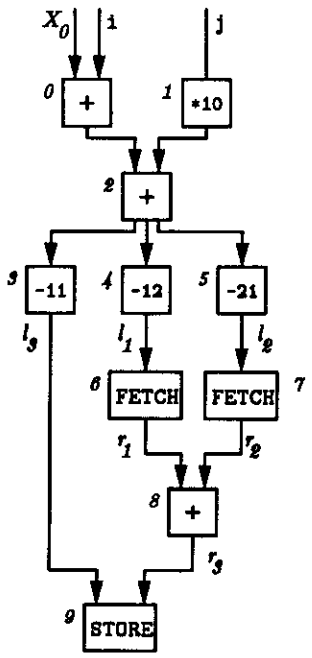

Figure 8: Dataflow graph for wavefront loop body.

that the address of `X(i,j)` is given by the expression $X_0 + (i-1) + n(j-1)$. With $n = 10$ and a little algebraic manipulation, we can see that the addresses of `X(i,j)`, `X(i-1,j)` and `X(i,j-1)` are given by

$$X_0 + i + 10j - 11\ ,$$
$$X_0 + i + 10j - 12 \text{ and}$$
$$X_0 + i + 10j - 21$$

respectively. Instructions 1 through 3 compute the common expression $X_0 + i + 10j$; the rest of the graph is self-evident. Notice that all the "register variables" of the previous section

are depicted here as labels on edges in the graph. We have also labelled each instruction with a number; initially, we will just use them for reference but later we will interpret them as addresses in an instruction memory.

The simplest way to visualize the execution of the DFG is as follows. Think of the edges as tubes. Imagine that we drop one token into each of the three input edges of the DFG, the value the starting address of array `x` in memory, and the values of `i` and `j` respectively. We now repeatedly apply the following simple "firing rule":

> Whenever an instruction has tokens on all its input edges, remove the tokens, compute the result value according to the instruction, and produce tokens on all its output edges carrying this value.

Initially, both instructions 0 and 1 are ready to fire. The former adds the values of X_0 and `i`, and the latter multiplies the value of `j` by 10. A constant like 10 can be regarded as a "literal" that is part of the instruction itself. Now, instruction 2 can fire, and it puts its value onto tokens on each of its three output edges. Then, instructions 3, 4 and 5 can fire concurrently, completing the address computations. Instructions 6 and 7 can then fire concurrently, followed by instruction 8 and, finally, instruction 9. It is this visualization of tokens "flowing" through the graph that gives rise to the name "dataflow".

Note that at each instant, there may be more than one instruction ready to fire. This is in sharp contrast to conventional machine languages where, at each instant, there is exactly one instruction that is ready to be executed. This is why we say that DFGs specify only a *partial order* on instructions.

We do not have to fire all ready instructions at each instant; indeed, in any real machine, we will not usually have the resources to fire all of them at once. Fortunately, DFGs are determinate, *i.e.,* we can safely fire any subset of ready instructions for which we have machine resources available.[3]

9 The Explicit Token Store processor model

Now that we have the intuition behind dataflow graphs under our belts, it is time to be less abstract and to work out some details. How is a dataflow graph represented? How is a token represented? What does it mean for a token to be placed on an edge? How do we detect that an instruction is ready to fire? What happens when an instruction is fired? To explain all this, we use a processor model developed by G. P. Papadopoulos called the Explicit Token Store (ETS) model. The ETS model is suitable for direct hardware implementation and is shown in Figure 9.

Without loss of generality, we take the following position: each instruction has exactly one or two input edges, *i.e.,* it is either monadic or dyadic. When a token arrives for a monadic instruction, the instruction can be fired immediately. On the other hand, when the first token

[3]However, differenct choices of the instructions to fire may entail different machine resource requirements, and this is always a concern in a rea. machine.

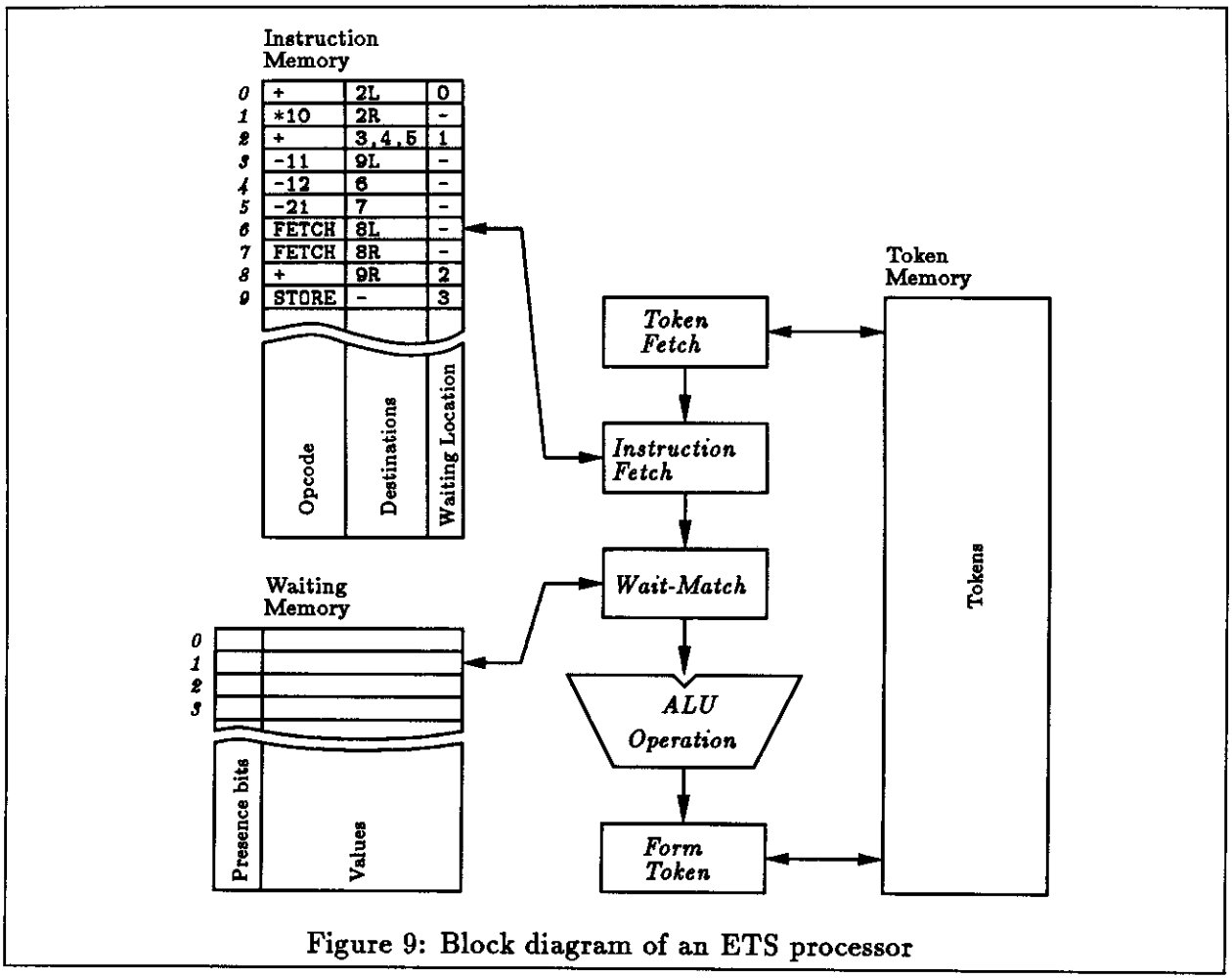

Figure 9: Block diagram of an ETS processor

arrives for a dyadic instruction, whether on the left or the right edge, it must wait until its partner arrives. Thus, we need to reserve a storage location for each dyadic instruction where its first input token will wait. This motivates our representation of dataflow graphs, as shown in the *Instruction Memory* part of Figure 9.

Our previous instruction numbers can now be seen as addresses in instruction memory. Each instruction has three fields: an *Opcode*, *Destinations* and *Waiting Location*. The opcode is an operation, such as + or * or *2. The destinations encode the edges of the dataflow graph. For example, the destination of instruction 3 is 9L (instruction 9, left-hand port). The L and R port designations are present only for binary destinations. The waiting location `r` is an address in *Waiting Memory* for dyadic instructions. For example, the waiting location for instruction 8 is location 3 in waiting memory.

For simplicity, we are assuming separate instruction and waiting memories, *i.e.*, instruction 2 and data location 2 are distinct. Further, both these memories are *local* to the processor, unlike the separate data memory that holds the array `X`.

There is one more memory in the processor, called the *Token Memory*, each location of which can contain a token. A token consists of two fields `<IP,v>`. IP (for "Instruction Pointer") is the address and port to which the token is destined, and `v` is the value that it carries. To execute our example graph, suppose that token memory contains our initial three tokens:

`<0L,`X_0`>`	*value of* X_0 *destined for left port of instruction 0*
`<0R,`i`>`	*value of* i *destined for right port of instruction 0*
`<1,`j`>`	*value of* j *destined for instruction 1*

The machine executes programs by repeatedly doing the following:

1. *Token Fetch*: remove a token `<IP,v>` from token memory.

2. *Instruction Fetch*: fetch the instruction `<op;d1,...,dN;r>` from `IP` in instruction memory.

3. *Wait-Match*: If `op` is monadic, execute the instruction (steps 4 and 5 below). If it is dyadic, examine location `r` in waiting memory to see if it is empty or full, *i.e.*, check if this token is the first or second to arrive for this instruction. If the location is empty, store the value `v` from this token there, mark it "full" and extract the next token (step 1). Otherwise, extract the value in the waiting location and execute the instruction (steps 4 and 5).

   To implement this, every waiting location `r` must have some additional bits called "presence bits" indicating whether it is full or empty.

4. *ALU Operation*: compute the output value by applying `op` to the input value(s).

5. *Form Token*: attach the result value to `d1`, ..., `dN`, producing N new tokens, and place them in the token memory. Go to step 1 to process another token.

In fact, as suggested by Figure 9, these steps are independent and can be pipelined quite easily for high performance.

There is a serious issue not yet addressed by the scheme just described. We can have many simultaneous invocations of the same dataflow graph (once for each iteration of the loop). So, we need a way to send multiple sets of tokens through the same dataflow graph and to have some way to avoid mixing up tokens from the two sets. Abstractly, if we think of one set being red and the other set being green, we should not fire an instruction using a red and a green token. Technically, we need to make the code *re-entrant.*

To handle this, we change our interpretation of the waiting-location addresses. With each invocation of a dataflow graph we associate a fresh chunk of storage in waiting memory called a *frame*. Each token now has three fields <IP,FP,v>. FP (Frame Pointer) is the starting address of the frame representing the invocation that this token belongs to. Now, instead of interpreting the waiting-location address r as an *absolute* address in waiting memory, we interpret it as an address *relative* to FP, *i.e.,* in the wait-match stage, we examine the location FP+r instead of the location r.

The step we have just taken is analogous to the step from FORTRAN to Algol which introduced stack frames, allowing recursion and re-entrancy. The only difference is that here we have a parallel runtime structure, *i.e.,* a tree of frames instead of a stack. Of course, we need a *linkage* mechanism by which we can dynamically allocate new frames, deallocate used frames, send arguments and return results; we omit these details for lack of space.

10 Data structures and I-structure memory

We have one more implementation issue to tackle. We have already captured the parallelism within one iteration of the wavefront loop. Further, if all iterations are initiated in parallel, then *all* the address computations can indeed proceed in parallel. However, we still need to express the idea that the FETCHes in one iteration must wait, if necessary, until the corresponding STOREs in other iterations have completed. To deal with this, we must refine our machine model a little. All that we have described so far was within a single processor. We now turn our attention to the interaction between the processor and the data memory that holds the array X.

In conventional machines, we are used to the following two transactions between the processor and memory. When the processor sends a <STORE,d1,1> message to memory, it stores the datum d1 in location 1. When the processor sends a <FETCH,1> message to memory, it extracts the datum d1 from location 1 and sends back the message <d1>. In a dataflow machine, there is an additional kind of transaction which we get by replacing a FETCH instruction by an I-FETCH instruction. This is shown in Figure 10 using an excerpt of our previous dataflow graph. When the I-fetch$_{8R}$ at address 7 is executed relative to a frame FP, the processor sends the following message to memory:

<I-FETCH,1;8R,FP>

The memory, in turn, reads the value d1 in location 1 and sends a message back to the processor containing <8R,FP,d1>. Note that this message has exactly the form of a token. In

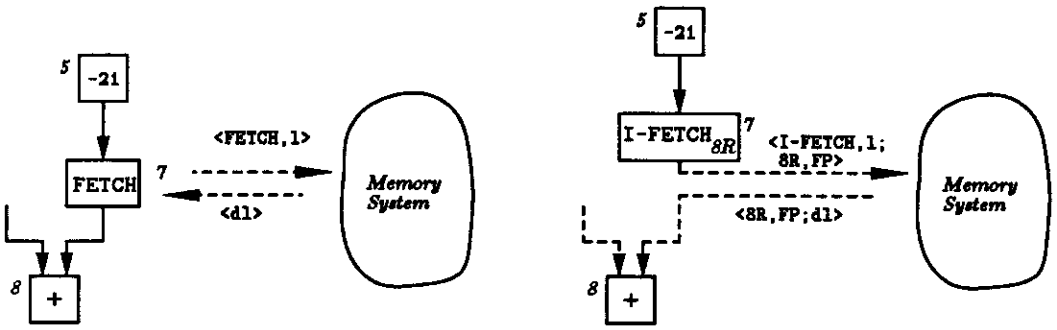

Figure 10: FETCH *vs.* I-FETCH instructions

fact, when it is received at the processor, it is processed exactly like any other token. Unlike a FETCH transaction, both the request and the response carry additional <IP,FP> information.

A complementary mechanism is found in the memory system, as shown in Figure 11. Such a memory is called an *I-structure memory*. Like frame memory, each I-structure memory

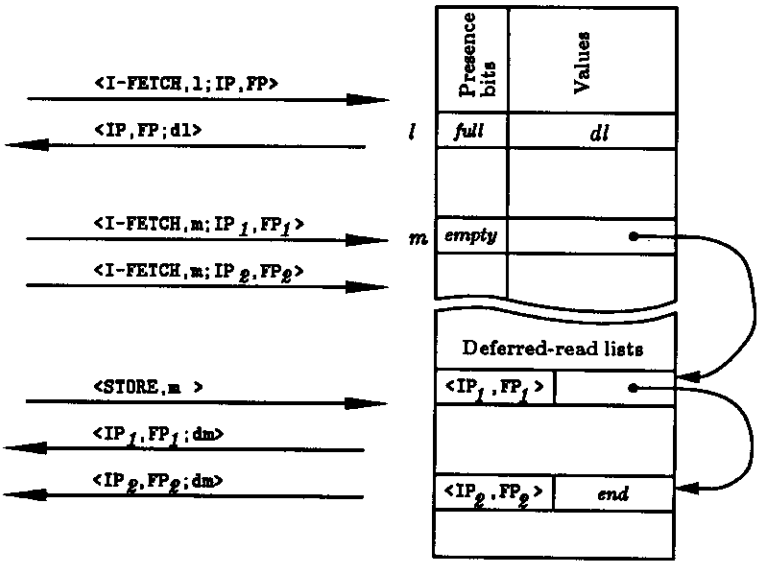

Figure 11: I-structure memory.

location has presence bits indicating whether it is full or empty. When the I-structure module receives <I-FETCH,m;IP,FP> messages for a location m that is still empty, it builds a "deferred-read" list locally, containing all the <IP,FP> pairs that accompanied such requests. The list is anchored at location m, *i.e.*, each location can have its own deferred list. Later, when it receives a <STORE,m,dm> message, it writes the value dm to location m, marks it "full", and also sends all the responses <IP,FP,dm> for the I-FETCHes deferred at that location.

It is this protocol that allows us to initiate all iterations of the wavefront loop in parallel, and allows an I-FETCH in one iteration automatically to wait for the corresponding STORE in

another. This protocol is, in fact, one of the key reasons that the dataflow approach is so attractive for parallel machines:

- Assuming that there are enough tokens in the processor's token memory, it can continue to process other tokens during the I-structure memory transaction, *i.e.,* the processor does not have to be idle during long memory reads. This tolerance of memory latency is becoming increasingly important in high-speed machines. For example, in the Cray 2, while the processor can initiate an arithmetic operation on every cycle, it takes 40 to 60 cycles to access a memory location. In large multiprocessors, memory latency can be much worse.
- The processor can issue many I-structure memory reads before it has received the response to the first, *i.e.,* the memory system can be *pipelined* for better performance.
- When a processor issues several I-structure requests, it is ready to receive the responses in a different order. This may happen either because the distances to different memory modules may vary in a parallel machine, or because some reads are deferred at the memory.

11 Which language should we program in?

A popular approach to parallel programming is to take a sequential, imperative language and extend it with new, parallel constructs. For example, in a FORTRAN program we can specify that all iterations of a loop are to be done in parallel by using the keyword DOALL in place of DO. Recognizing the wavefront parallelism in our example, we could express it explicitly as follows. We change the program so that we are no longer iterating on columns and rows, but on diagonals:

```
      DIMENSION X(n,n)

C     Initialize boundaries
      ...

C     Fill in the upper triangle

      DO 200 m = 4,n+1                  traverse diagonal sequentially

         DOALL 100 i = 2,m-2            compute cross-diagonal elements in parallel
            j = m - i
            X(i,j) = X(i-1,j) + X(i,j-1)
100      CONTINUE
         BARRIER

200   CONTINUE

C     Fill in the lower triangle
      ...
```

The BARRIER statement specifies that all the parallel computations on one cross-diagonal have completed before the program moves on to the next cross-diagonal.

There are several problems with this "explicit parallelism" approach. First, the parallel FORTRAN program has already lost the intuitive nature of the original sequential program, *i.e.,* the task of programming has been seriously complicated.

Second, it opens the door to new bugs due to non-determinacy. For example, if we accidentally omit the BARRIER statement, the program would continue to run, except that we may now read an array location in the m+1'st iteration too early, *i.e.,* before its value has been stored during the m'th iteration. Another example: in the DOALL loop, each iteration writes a value into j and reads it. Clearly, all iterations cannot share the same location for j. We have failed to declare that j must be a *private* variable, *i.e.,* a separate location for each iteration of the loop. These kinds of bugs are quite pernicious because they happen silently, *i.e.,* it may not be obvious from the output of the program that something went wrong. Even if we knew that something was wrong, tracking down such bugs is a nightmare because it is difficult to repeat the experiment— even the act of debugging can change the schedule of computations, thus obscuring the bug.

Another approach, pioneered by David Kuck of the University of Illinois, is to stay with a sequential language, leaving it up to the compiler to "parallelize" it automatically. Using sophisticated *dependency analysis*, the compiler attempts to discover the data dependencies in a program by identifying which parts of the program read and write to each data location. Once such dependencies are known, the compiler can reorder the instructions in the code to allow maximum parallelism while preserving these dependencies. Because all parallelism is introduced by the compiler using transformations that have been proved correct, this approach is not subject to the non-determinacy bugs mentioned earlier.

However, we have serious doubts whether much parallelism can actually be detected automatically using this approach. Dependency analysis is difficult for several reasons. The first is that the exact location read or written by a program fragment may not be statically predictable, because the index expression for an array can be very complex, *e.g.,* involving a function call or another indexed expression. Procedure calls, pointer variables, *etc.* are further factors complicating the analysis. Whenever the compiler is unable to detect the dependencies accurately, it must err on the conservative side; often, this means giving up much parallelism.

To illustrate these problems further, let us move to a slightly more complicated example called *successive overrelaxation* (SOR). Here, we compute a succession of matrices. In each matrix, the (i,j)'th element depends not only on some of its neighbors (as in the wavefront computation), but also on some elements of the *previous* matrix. Figure 12 depicts a stylized SOR in which the (i,j)'th element depends on two of its neighbors and on the (i,j)'th element of the previous matrix (in an actual SOR application, the recurrence may involve more terms). The Id code to express this computation is shown below:

```
def SOR kmax X = { for k <- 1  to kmax  do
                     next X = make_matrix (grid n) (f ( next X) X)
                   finally X} ;
```

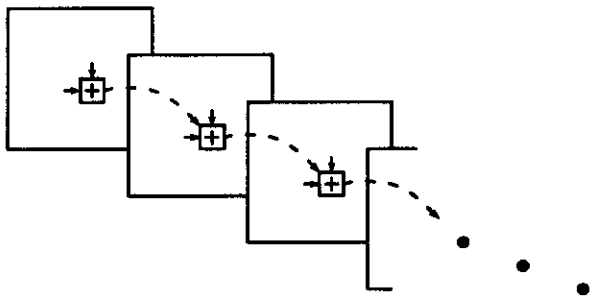

Figure 12: Successive overrelaxation

```
def f X oldX (i,j) = if (i == 1) or (j == 1) then 1
                     else X[i-1,j] + X[i,j-1] + oldX[i,j] ;
```

Some notation: `kmax` is the number of matrices to be computed. The body of the *for*-loop specifies the relationship between each matrix (`X`) and the next one (*next* `X`). The *finally* expression specifies that the last matrix `X` is the final result. The function `f` is similar to the previous one except that it also takes the previous matrix `oldX` as parameter.

Again, let us take a moment to analyze the parallelism of this program. All the borders of all the matrices can be computed in parallel. At some point, it will become possible to define the `[2,2]`'th element of the first matrix because its neighbors, on the border, are ready. Of course, this makes it possible to define the `[3,2]` and `[2,3]`'th elements of the first matrix, but it also enables the computation of the `[2,2]`'th element of the *second* matrix. This, in turn, makes it possible to compute elements of the first matrix at `[4,2]`, `[3,3]` and `[2,4]`, elements of the second matrix at `[3,2]` and `[2,3]`, and the element of the *third* matrix at `[2,2]`. Thus, where previously our wavefront was a diagonal line sweeping across a single matrix, it is now a diagonal *plane* sweeping across a collection of matrices. Figure 13 depicts two stages of the wavefront.

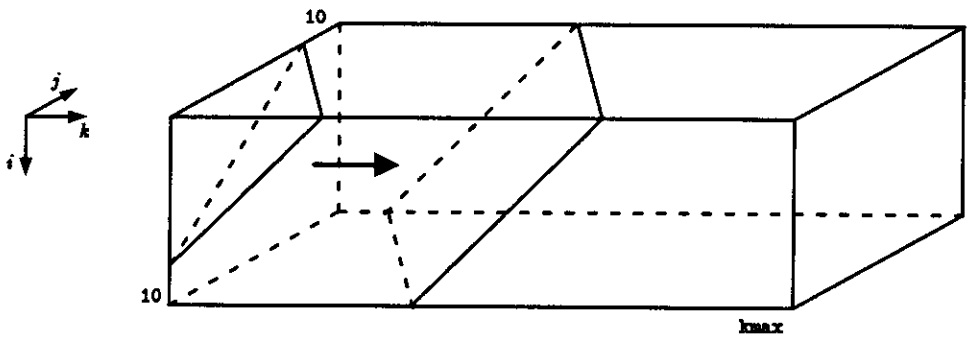

Figure 13: Wavefront parallelism in successive over-relaxation.

Figure 14 shows the parallelism profile for the SOR program with kmax = 10. It also shows the profile for a version of SOR in which we artificially hold back the $k+1$'st iteration until the k'th iteration completes, *i.e.*, simulating a BARRIER between the k iterations, as may happen in a FORTRAN program that is unable to exploit the parallelism between the k iterations. This graph is essentially like the concatenation of 10 copies of the wavefront profile of Figure 5. The example vividly demonstrates how the dynamic exploitation of parallelism in Id can drastically increase the parallelism (from about 150 to 1500 in this example) and reduce the critical path (from about 1700 down to about 250 in this example).

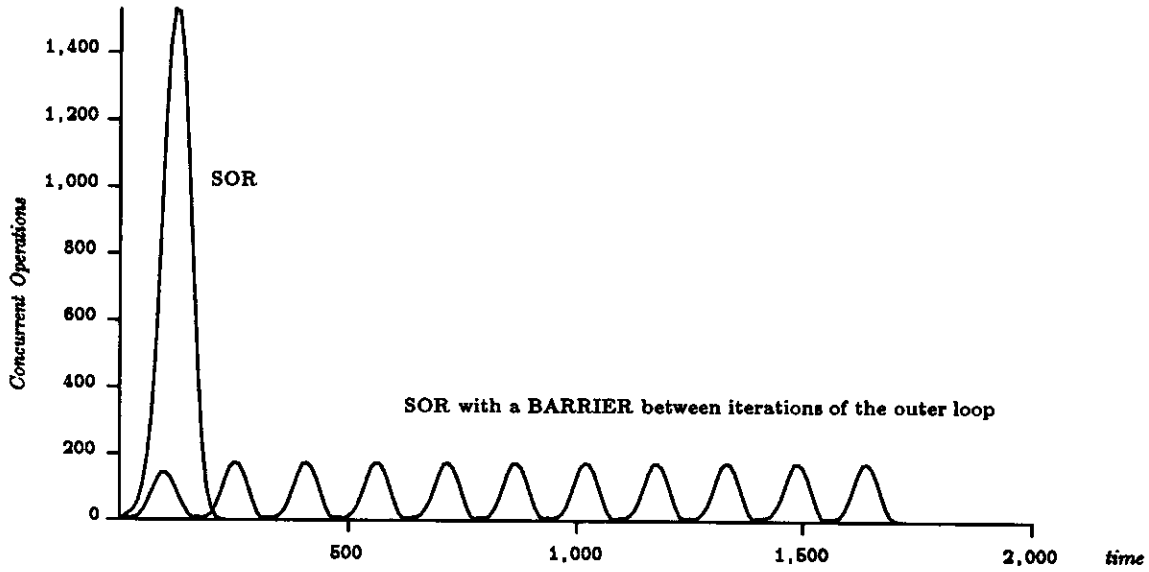

Figure 14: Parallelism profiles for successive over-relaxation, both with and without a barrier between relaxation steps.

Let us now look at SOR in FORTRAN. A typical sequential solution needs only a small modification to our original wavefront program:

```
      DO 300 k = 1,kmax
         DO 200 i = 2,10
            DO 100 j = 2,10
               X(i,j) = X(i-1,j) + X(i,j-1) + X(i,j)        Loop body
  100       CONTINUE
  200    CONTINUE
  300 CONTINUE
```

Here, despite the fact that all the X's in the loop body refer to the same storage area, the loop body actually computes the recurrence shown below.

$$X^k_{i,j} = X^k_{i-1,j} + X^k_{i,j-1} + X^{k-1}_{i,j}$$

This relies crucially on the sequential semantics of FORTRAN loops. For example, if we replaced "j-1" by "j+1" in the third term of the loop body, it also changes the k superscript in the corresponding recurrence term, *i.e.*, from $X^k_{i,j-1}$ to $X^{k-1}_{i,j+1}$. The programmer has

cleverly exploited this behavior in being able to perform the entire computation in place, *i.e.*, with a single matrix.

If the compiler is to match the parallelism found in the Id program, however, it must reconstruct the 3-dimensional abstract recurrence from the FORTRAN program, change the 2-dimensional matrix `X` into a 3-dimensional array and, finally, set up the parallel computations to fill up the array. There are major pitfalls at each stage.

First, the programmer's cleverness in reusing storage in the sequential program makes dependency analysis (to discover the 3-dimensional recurrence) much harder.

Second, if `kmax` is not known at compile time, it is difficult to convert the matrix into a 3-dimensional array because in FORTRAN, storage must be allocated statically. Even if `kmax` is known statically, the permanent allocation of all the storage for the entire 3-dimensional array is wasteful, since only a small part of it is used at a time (along the diagonal wavefront plane).

Finally, because of the lack of fine-grained parallelism enabled due to I-structures, the parallel program must have the following structure:

```
      DO 200 p = 1, number of wavefront planes
        DOALL 100  for each I,J,K  in wavefront plane p
          X(I,J,K) = X(I-1,J,K) + X(I,J-1,K) + X(I,J,K-1)
100     CONTINUE
        BARRIER
200   CONTINUE
```

It is actually very tricky to set up the iterations that we have abbreviated in the `DOALL` line because the boundary of the wavefront plane varies greatly. It may be a triangle, pentagon, trapezoid or rectangle depending on the position of plane (two of these planes are shown in Figure 13). The resulting program would be practically unreadable and is certainly not something that ought to be programmed manually.

Thus, in FORTRAN, clarity of expression and maximal parallelism are often contradictory requirements. In Id, on the other hand, the most natural expression of the program *is* the parallel version. The "3-dimensional" storage arises because each iteration of the Id loop dynamically allocates a new matrix. Unlike the FORTRAN version, all `kmax` versions of the matrix do not have to be permanently allocated. When a matrix in one k iteration is no longer in use, *i.e.*, all references from the $k+1$'st iteration have been completed, its storage can be reclaimed dynamically. The general mechanism for this is a continuous process known as "garbage collection". In fact, it is possible to be even more efficient— using a technique known as "loop bounding", it is possible to build a circular list of frames and matrices for the k loop and to cycle through them. The language semantics, which guarantees that data-structure components are defined uniquely, allows the compiler to exploit dataflow I-structure hardware fully, so that it can initiate many the iterations concurrently, leaving it to dynamic scheduling to achieve the wavefront parallelism.

12 Current status

Two dynamic dataflow machines have been built to date. The first was the Manchester Dataflow Machine at the University of Manchester, England, and was operational in 1982. It is only a single-processor prototype (with multiple function units) without I-structure storage, but it has confirmed the basic viability of fine-grained dataflow parallelism. The second machine was the Sigma 1, built at the Electro-Technical Laboratory in Japan. It has just become operational and consists of 128 processors and 128 I-structure units.

Both these machines use an earlier design in which the wait-match operation is done using an associative memory. The ETS model described in this article, because of directly addressed wait-match memory, is much easier to implement and provides much more flexibility in controlling the use of storage. At MIT, we are constructing a machine based on this model called "Monsoon". A single-processor wire-wrap prototype has been operational since October 1988, and an 8 processor, 8 I-structure machine is expected to be complete by the fall of 1990.

Concurrently, our software effort concentrates on two related areas. One is on compiler optimizations, building on the excellent Id compiler implementation by Ken Traub. The second is on resource management— the efficient dynamic management of frames and data structures. Of course, we are also continually upgrading "Id World", our incremental programming environment for Id and Monsoon.

The ETS model has also given us deeper insight into the essence of dataflow and von Neumann models, which, in turn, has led to many ideas on integrating the best of both. Research projects in this area include the Hybrid von Neumann/Dataflow machine of R. A. Iannucci of IBM Research, the EM-4 at Electro-Technical Laboratory, Japan, the Eps-88 at Sandia National Laboratory, New Mexico, and the P-RISC processor here at MIT.

Dataflow research is at an extremely exciting stage. We are confident that these projects will take us closer towards the "right" building-blocks for scalable, programmable, general purpose parallel machines. Perhaps by 1995 we will have commercially viable parallel machines which are actually a pleasure to program.

Further reading:

All the technical reports cited below are available from:

Computation Structures Group,
MIT Laboratory for Computer Science
545 Technology Square, Cambridge, MA 02139, USA.

The original "classic" by Jack Dennis:

Dennis, J. B. First Version of a Data Flow Procedure Language. In *Proceedings of Programming Symposium, Paris 1974*, Lecture Notes in Computer Science 19, Springer Verlag, Berlin, 1974.
(Revised: technical report MAC TM61, May 1975)

This paper is an overview of MIT Tagged-token dataflow approach, including example Id programs, their translation to dataflow graphs, and an overview of the Tagged-token Dataflow Architecture (TTDA):

Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. To appear in *IEEE Transactions on Computers*. An earlier version appeared in *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. Springer-Verlag LNCS Volume 259, June 1987*. Also: CSG Memo 271.

This memo is the current Id language reference manual:

R. S. Nikhil. Id (Version 88.1) Reference Manual. CSG Memo 284. August 29, 1988. 26 pages.

This paper develops excerpts of the SIMPLE hydro-dynamics code written in Id, demonstrating the level of programming and comparing it with the FORTRAN version:

Arvind and K. Ekanadham. Future Scientific Programming on Parallel Machines. In *Proceedings of the International Conference on Supercomputing (ICS), Athens, Greece*, June 8-12 1987.

This paper develops an Id solution to David Turner's "paraffins problem", demonstrating the level of programming and the inherent parallelism obtained:

Arvind, S. K. Heller and R. S. Nikhil. Programming Generality and Parallel Computers. In *Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy, September 18-22, 1988.* Also CSG Memo 287.

This thesis is a detailed description of the Id compiler and basic optimizations.

K. R. Traub. A Compiler for the MIT Tagged-Token Dataflow Architecture. MIT/LCS/TR-370, August 1986.

This paper is a survey of data-driven architectures, including static dataflow machines, the Denelcor HEP, the MIT TTDA, the Japanese ETL Sigma-1, *etc.* :

Arvind and D. E. Culler. Dataflow Architectures. Annual Reviews in Computer Science, Volume 1, Annual Reviews Inc., Palo Alto, CA, 1986, pages 225–253. Also: MIT/LCS/TM-294.

This thesis develops the Explicit Token Store (ETS) dataflow model and Monsoon, a specific implementation:

> G. M. Papadopoulos. The Monsoon Dataflow Architecture. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, August, 1988.

This paper discusses the role of latency and synchronization in parallel machines, and an analysis of how von Neumann and dataflow architectures address these issues:

> Arvind and R. A. Iannucci. Two Fundamental Issues in Multiprocessing. In *Proceedings of DFVLR - Conference 1987 on Parallel Processing in Science and Engineering, Bonn-Bad Godesberg, W. Germany*, June 25-29 1987. Also: CSG Memo 226-6.

This paper describes P-RISC, a processor element whose instruction set and architecture properly extends a conventional RISC element, giving it a fine-grained dataflow capbility:

> R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th Annual Symposium on Computer Architecture*, Jerusalem, Israel, May 29-31, 1989, pages 262–272. Also: CSG Memo 292.