LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Managing Resources in a Parallel Machine

CSG Memo 257
October 11, 1985

Arvind

David E. Culler

This work was presented at the
IFIPS TC-10 Conference on Fifth Generation Computer Architecture,
UMIST, Manchester, England, July 16-19, 1985.
It is to appear in the Proceedings to be published by
North-Holland Publishing Co.

Source File = ti.mss, Last updated 15 October 1985 at 1:21pm

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Managing Resources in a Parallel Machine

Arvind
David E. Culler

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts
U.S.A.

We discuss resource management in parallel machines that support a general purpose programming model, *i.e.*, one that allows dynamic creation of tasks. Concurrent execution of tasks, in general, requires more resources than sequential execution. Thus, a policy that eagerly allocates resources to tasks can cause a deadlock not often seen in sequential implementations. We present a method of exploiting parallelism while avoiding deadlock for a large class of programs in the context of dataflow machines. The method is based on the concept of *Resource Bounded Dataflow Graphs*, and allows "parallelism parameters" of tasks to be set based on the availability of resources at time of task initiation.

1. Overview

Our experiments with dataflow programs running on a simulated parallel machine have demonstrated that even small programs coded without particular concern for parallelism exhibit substantial amounts of parallelism. While we herald these results as validation of our initial claims, we also view them as a subject of some concern. Exposing parallelism tends to increase the resource requirements of a program. Automatic program unfolding, if unconstrained, may expose far more parallelism than the underlying machine can exploit and suffer performance degradation due to inordinate resource demands. We would like to expose sufficient parallelism to fully utilize the machine, while keeping resource demands within reason.

This resource usage problem is perhaps most serious for dataflow machines, but is certainly not limited to that class of machines. We believe that any programming model offering a suitably general means of expressing parallelism will give rise to this concern regardless of the underlying parallel computation model. Thus, the problem will appear to some extent in any parallel implementation of a functional or relational language and in parallel implementations of imperative languages augmented with *tasks*, such as ADA.

This paper illustrates the potential hazards in dynamic unfolding of programs and addresses how these problems might be overcome in a dataflow context. Sections 2 and 3 outline the general problem and argue that it should appear in any general purpose parallel machine. Sections 4 through 6 describe our work to overcome this problem in a dataflow machine. The concept of a

resource bounded graph is developed as a means of exploiting program structure to alleviate the burden of resource management.

2. Context: General Purpose Parallel Machines

The goal of our work is to demonstrate the feasibility of a general purpose parallel machine in which all processors cooperate to solve a single problem. As this is the context in which the above claims are to be understood, we should make clear what we mean by a general purpose parallel machine. In our view, the following properties are essential:

1. The programming model should not reflect the number of processors nor the vagaries of the interconnection topology.
2. Higher performance should be achievable through the simple addition of processors.
3. The programming model must support dynamic allocation of storage, *i.e.*, heaps.
4. A task can create new tasks dynamically, and thus may request additional resources at run time.

In putting forward these properties we have intentionally blurred any distinction between hardware, system software, and compiler. These are perceptible properties of the system as a whole.

The first property is particularly important. Specific topologies have been shown to provide efficient solutions to particular problems [1, 14, 18, 20, 21, 22]; our comments may not pertain to such specialized machines. We also recognize that the programming task becomes far more difficult if the underlying machine structure is reflected in the programming model. Parallel machines can only exacerbate an already onerous task, unless the programming model is extremely *clean.*

The second property is often referred to as *scalability.* However, scalability can be manifested in a variety of forms. Taking advantage of additional processors may require rewriting parts of the application code; this is especially likely if the first property is not met. A more appealing approach would involve only recompilation of the application code, but note this assumes the compiler embodies rather intimate knowledge of the machine configuration. Perhaps the most general approach would involve merely initializing the resource management system running on the machine. We hope to achieve this latter form of scalability.

Dynamic storage management is essential for a variety of reasons, not the least of which is its contribution toward ease of programming. A static storage model, such as embodied in Fortran, forces the programmer to manage storage explicitly within the application code. Even for a

sequential program it can be tricky to determine when storage areas may safely be reused, and the problem is far more difficult when many parts of the program execute in parallel. Moreover, in many cases the number of distinct versions of a data structure that must exist concurrently depends on the number of tasks executing in parallel.

We see dynamic task creation as a fundamental aspect of general purpose parallel machines. Tasks may be created *implicitly*, as in functional programs or pure logic programs, or *explicitly*, as in CSP based languages (*e.g.*, Occam or ADA), Milner's CCS [16], Halstead's Multilisp [9] (via futures), or annotated functional and logic programs. The key point is that tasks can spawn additional tasks and request resources at run time. A program in execution can be thought of as a dynamic complex of tasks, each possessing certain resources.

3. Goal: Exploiting Parallelism without Deadlocking

The goal of resource management in a parallel machine is to execute a program as fast as possible within a given set of resources. Some attention has been paid in the literature to how work and data are distributed over processors. A variety of straight-forward load-leveling techniques [7, 13, 17, 25] have been implemented and a good deal of discussion has centered on the need to exploit locality to reduce communication overhead [2, 5, 12, 15, 24]. However, we feel that a more fundamental problem has been largely overlooked: In order to keep resource demands within reason it may be necessary to defer execution of tasks even if resources are available to initiate the tasks.

The resource problem can be illustrated by a simple example. Consider the functional program fragment `def F = ••• G ••• H •••`. The three tasks F, G, and H can potentially execute concurrently, as suggested by the task invocation structure shown in Figure 1. Task H may consume enough resources that G can not go to completion, and the resources held by G may prohibit H from completing. The two tasks are deadlocked. Note that this situation is difficult to predict, since tasks can create subtasks and request resources dynamically. Note also that serializing the execution of G and H may alleviate the resource problem.

This resource deadlock problem represents a basic difference in sequential and parallel execution. The execution of a program can be represented by a tree of task invocations, as suggested in Figure 2. A sequential execution traverses this task invocation tree in a depth-first manner. The maximum resource demand corresponds to the heaviest single branch in the tree. Parallel execution tends toward a breadth-first traversal, and the resource demand may be the weight of the entire tree. To characterize this problem more precisely, let R_i represent the resource requirement of task i, not including the resources of its subordinate tasks. Then sequential evaluation of the program

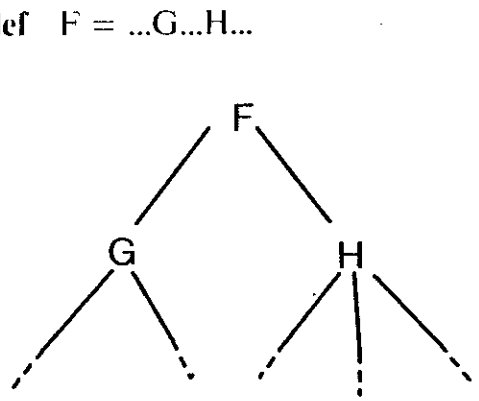

Figure 1: Task Invocation Structure for a Simple Example

described by Figure 1 has a resource requirement of MAX{ $R_F + R_G$, $R_F + R_H$}; whereas parallel evaluation has a resource requirement of $R_F + R_G + R_H$.

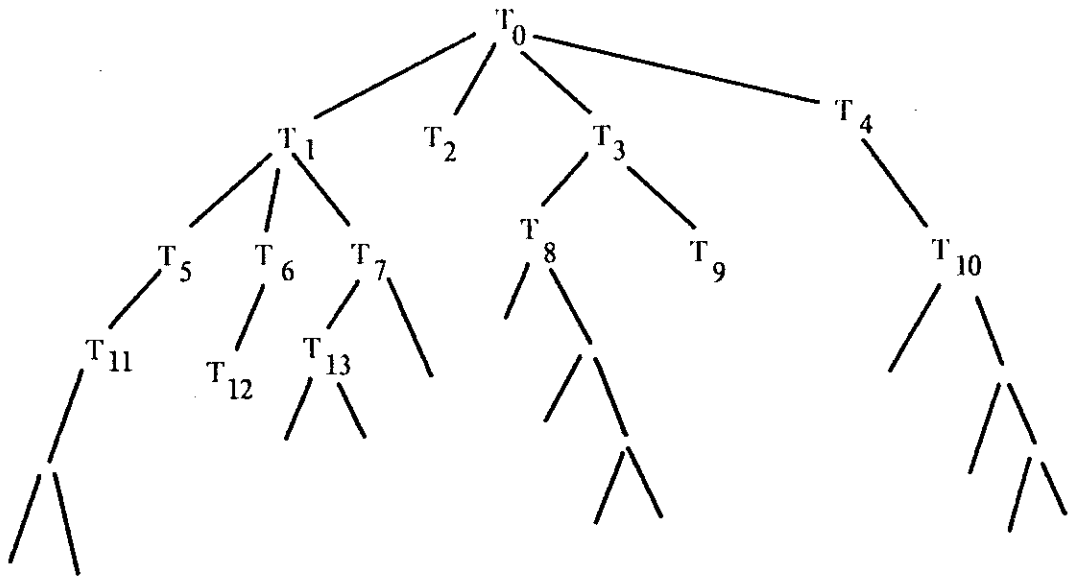

Figure 2: Task Invocation Structure

Our experiments with dataflow programs show that this problem is indeed serious. Straightforward coding of a matrix multiplication routine for N X N matrices generates N^2 concurrent dot-products, but also requires N^2 resources in the form of "constant areas" and "contexts" (these resources are explained more fully in Section 4.4). A simple loop (or equivalent tail recursive procedure) such as shown in Figure 3 is likely to cause a deadlock. If G involves

substantial computation, as many as N instances of G may be in execution concurrently, regardless of the number of processors. They all compete for a finite pool of resources.

```
def F n = initial s ← 0
          for i from 1 to n do
              new s ← s + G(i)
          return s

            or

def F' n i s = if i ≤ n then F' n (i+1) (s+G(i))
                        else s
def F n = F' n 1 0
```

Figure 3: Potentially Hazardous Computation

We want to examine strategies for controlling program execution such that enough parallelism is generated to utilize the machine fully, but resource demands are kept low enough to allow large programs to run without deadlock. One possibility is to pursue a breadth-first traversal until sufficient parallelism is generated, and to switch to a more depth-first strategy when the machine becomes busy [6, 23]. In essence the scheduling discipline for the task queue is dependent on the *status of the machine.*

We briefly note the resource management strategy in Halstead's Multilisp implementation [9] because it offers an interesting variation on the approach suggested above. There the run-time system maintains a central queue of runable tasks. Each processor can perform one task at a time. When a task creates a new subtask, the parent is placed on the task queue and the child pursued. If another processor is free, the parent will be resumed; otherwise, it will wait till the child (or some other task) completes. Thus, if the program in Figure 3 were executed on a machine configured with k processors, at most k instances of G would be active at any time.

The Multilisp approach is attractive in many respects, but, unfortunately, it is based heavily on architectural features which are not particularly well suited to a multiprocessor setting. We have argued extensively that effective parallel machines must be based on processors that are tolerant of long, unpredictable communication delays [3]. Such tolerance seems to require each processor to maintain a substantial number of independent activities and sizable process state. The Multilisp approach relies on cheap context swaps (even between machines), achieved by keeping all but a tiny amount of process state in global memory [8]. Controlling program behavior without compromising the effectiveness of a parallel architecture appears to be a subtle problem.

4. Background: Tagged Token Dataflow

In our work on managing resources in a dataflow machine we have examined how program structure can be exploited to control program unfolding and reduce the severity of the resource deadlock problem. Before describing that work some background should be provided on the computational model with which we are working, Tagged Token Dataflow [3].

4.1. Dataflow graphs

Figure 4 shows the graph for the simple loop of Figure 3. A dataflow graph is a directed graph whose nodes correspond to machine instructions and whose arcs correspond to (data) dependencies between instructions. Conceptually, a dataflow instruction sends values, or *tokens*, to instructions connected to it by outgoing arcs. An instruction is *enabled* for execution when all required input values are present. Thus, instructions are sequenced in accordance with the dependence arcs. Tokens carry a *tag* which specifies the arc upon which the token (conceptually) resides and additional contextual information to distinguish between different instances of the destination instruction.

The graph shown in Figure 4 is somewhat stylized. The node marked G represents the instructions required to invoke function G, which is itself a graph. The D, D^{-1}, L, and L^{-1} act as identity operators on data values, but are essential for manipulating contextual information on the token. A wave of tokens on the input arcs provides the initial inputs to the Switches and the loop predicate. If the loop predicate evaluates to 'true', the Switches route data into the body of the loop. The results of an iteration flow back to the predicate and the Switches via the D operators. When the loop predicate turns 'false', the final data values are routed to the loop trailer via the D^{-1} operators. Note that the left-hand loop variable can initiate new iterations even if the right-hand loop variable lags behind. The L operator provides a new *context* which distinguishes different activations of the loop; the context is carried on the token. The D operator increments the *iteration number* carried on the token, thereby distinguishing between different iterations within a particular activation. The D^{-1} and L^{-1} reset the iteration number and context, respectively.

A dataflow processor provides a mechanism for propagating tokens along arcs in the graph and detecting when instructions become enabled. Upon execution of an enabled instruction, the input tokens are absorbed, and output tokens for the following instructions in the graph are produced. A program is said to terminate when no enabled instructions remain.

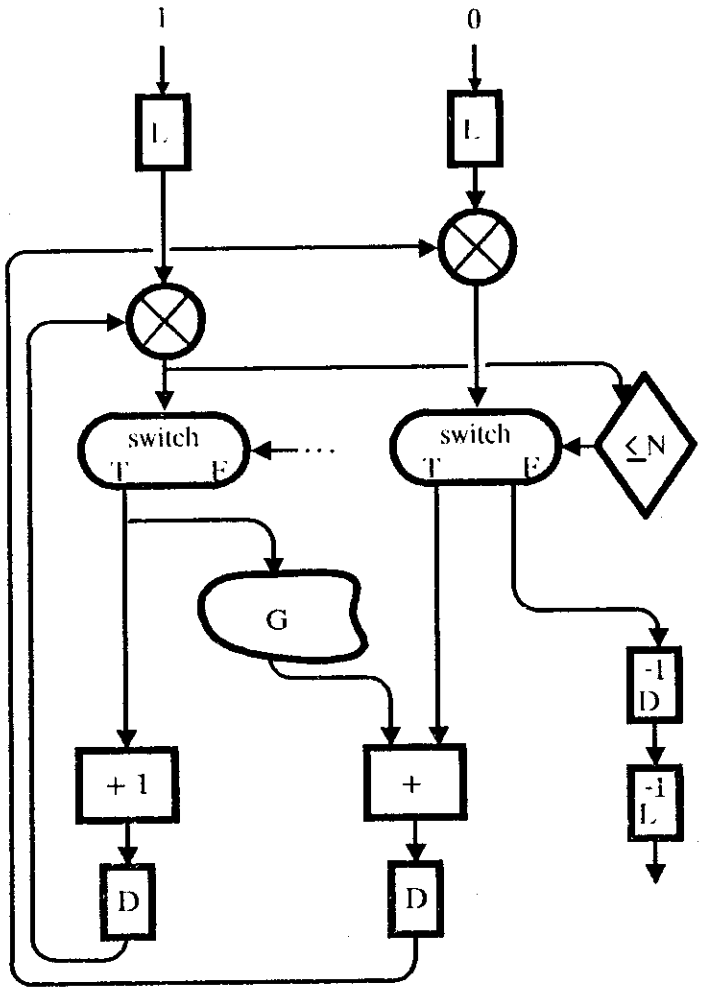

Figure 4: Dataflow graph for F in Figure 3

4.2. Instruction Processing in the TTDA

The Tagged Token Dataflow Architecture under development at the Massachusetts Institute of Technology comprises a collection of processing elements (PEs) connected via a packet switched communications network. Each PE is a complete dataflow computer. The basic PE architecture is shown in Figure 5. The *waiting-matching* store is the key component of this architecture. Tokens carry data and a tag. Two tokens are destined for the same activity, *i.e.*, a particular instance of an instruction, if and only if their tags are identical. When a token enters the waiting-matching stage its tag is compared against the tags of the tokens resident in the store. If a match is found, the matched tokens are purged from the store and forwarded to the instruction fetch stage. Otherwise, the incoming token is added to the matching store to await its partner. (Instructions are restricted to

at most two non-constant operands so a single match enables an activity.) Tokens which require no partner, *i.e.*, are destined for a monadic operation, bypass the waiting matching stage.

Once an activity is enabled it is processed in a pipelined fashion without further delay. Information in the tag allows *instruction fetch* stage to locate the instruction and any required constants. The op-code and data values are passed to the ALU for processing. In parallel with the ALU, the *compute tag* stage accesses the destination list of the instruction and prepares result tags, using information stored in *mapping registers*. Result values and tags are merged into tokens and passed to the network, whereupon they will be routed to the appropriate waiting-matching store.

4.3. Code-block activations

A dataflow program comprises a collection of *code-blocks*; these correspond essentially with individual loops or procedures in the high-level language program. A code-block activation can be viewed as a *task*; it is assigned to a collection of processors, where it will run to completion. There may be substantial parallel activity within a code-block activation, and this activity may be distributed over a number of processors. To keep the overhead of distributing work low we follow a two-level strategy; code-block activations are assigned dynamically (by the resource management system) to collections of processors called *domains*, whereas parallel activity within a code-block activation is distributed automatically over the processors in a domain without further interaction with the resource manager.

The hardware provides significant flexibility in distributing the work involved in a task over a domain. For acyclic code-blocks each instruction executes at most once, so the work is distributed simply by partitioning the graph and placing a part on each processor in the domain. Each processor performs the operations for the instructions it contains. A copy of the graph is literally spread over the domain. Loops offer another degree of flexibility, since it is possible to distribute iterations as well as partitioning the graph. When a loop code-block is loaded into a domain, a complete copy of the graph is placed in each subdomain. Distinct iterations are dynamically assigned to subdomains by hashing the iteration identifier in the tag.

4.4. Machine Resources

The resources in the machine are structured around code-block activations. In order for a code-block to execute in a domain, there must be a copy of the code there; this requires allocation and loading of program memory. For loops, a copy of the code must be placed on each subdomain. In addition, each code-block activation must be assigned a unique identifier (a *context*) to distinguish it from concurrent activations within the domain. Each code-block activation is assigned a code-block register (CBR) in the domain on which it is to execute. The CBR serves a number of

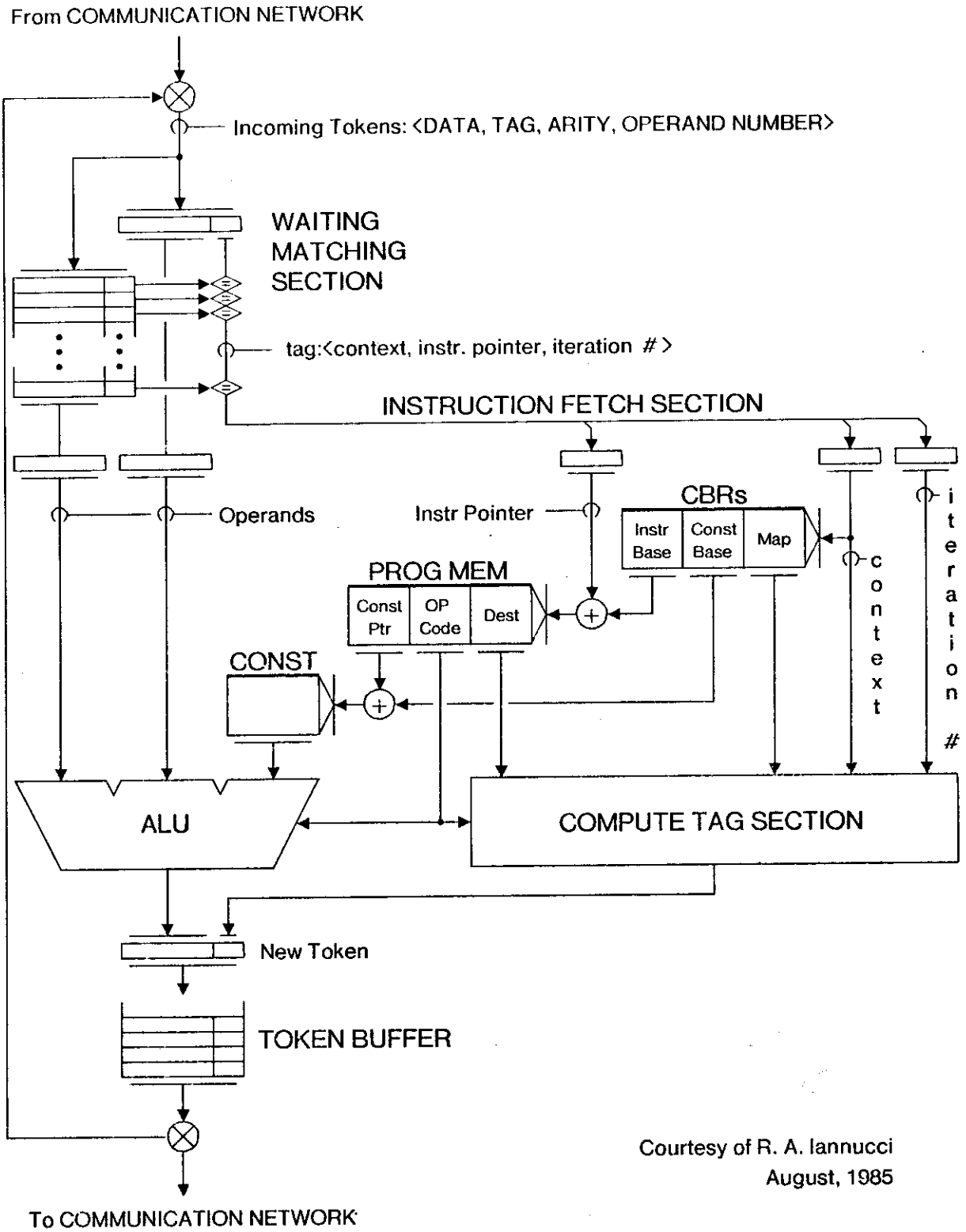

Figure 5: Processing Element of the MIT Tagged Token Dataflow Machine

functions; it provides the unique identifier and a handle on the machine resources associated with the activation. All tokens for an activation carry the CBR number in their tag. The CBR contains the base address of the code in program memory, a description of how the code is partitioned over the PEs in the domain, and a pointer to the state information, *e.g.*, constants, associated with the activation.

In total, four kinds of resources are associated with a code-block activation:

1. Token Storage, *i.e.*, waiting-matching store and token buffers;
2. Program Memory (for code and constants);
3. Code Base Registers, *i.e.*, distinct context names; and
4. Structure Storage.

The allocation and deallocation of token storage is handled directly by the hardware; storage is allocated when a token arrives and finds no partner, and is released when a match is detected. However, the waiting-matching unit cannot refuse an incoming token when the store becomes full, as this would lead to deadlock. Token storage can only be released by detecting matches. Thus, in activating a code-block in a domain a certain amount of token storage is implicitly put into use, and to avoid over committing this resource the resource manager must account for the potential load placed on the token store. Accounting for token storage load is somewhat tricky, since it depends on the order in which instructions happen to execute. CBRs and program memory are managed explicitly by the resource manager; they are allocated when a code-block is activated and released when it terminates. Many CBRs can reference the same code in program memory. Structure storage presents a serious management problem. Storage is allocated whenever requested by the program. It is reclaimed automatically when a structure is no longer accessible. In general, the termination of a code-block which caused the allocation of a structure does not result in the deallocation of that structure; structures may be passed to other activations or stored in structures. Performing efficient storage reclamation in a parallel machine is currently an active research topic, and is beyond the scope of this paper. Depending on the amount of various resources provided by the machine and on the particular program, any one of these resources may become critical.

5. Resource Bounded Graphs

Dataflow programs offer a great deal of structure which can be exploited to alleviate, if not overcome, the resource deadlock problem. Sizable chunks of a program can be shown to have bounded resource requirements which can be determined in advance. In many cases the resource requirement will be a simple function of the amount of parallelism permitted. Our approach is (1)

to develop extremely well-behaved graph structures whose dynamic unfolding can be easily controlled, and (2) to develop sophisticated analysis techniques for predicting the resource requirements of graphs. In this discussion we focus entirely on token storage requirements; CBRs and program memory can be treated similarly; management of structure storage, as suggested earlier, is more complex.

We say a *resource bounded graph* (RBG) is a program fragment for which a bound on the resource requirement of the fragment can be derived at compile time as a simple function of a collection of *parallelism parameters.* The role of the parallelism parameters will become clear in the course of the discussion. A program is viewed as a collection of RBGs. In general, recursion will delimit the fragments which can be grouped into RBGs, but special cases such as loops and tail recursion may be handled within an RBG.

Consider the acyclic graph shown in Figure 6. An easy upper bound on the token storage requirement of F is the number of arcs plus the token storage requirement of G. This bound is somewhat loose, since not all arcs can be occupied simultaneously. We have $R_F = 4 + R_G \leq 7 + R_G$. Culler has shown [6] that tight bounds can be derived efficiently using a linear programming technique. Where conditionals appear, slightly loose bounds must suffice. The resource requirements of entire invocation trees can be solved by working up from the leaves.

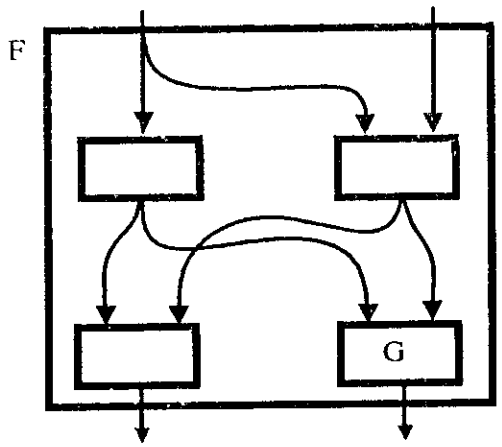

Figure 6: An Acyclic RBG

Loop code blocks may or may not have bounded resources, depending on the structure of the graph. Figure 7 shows two loops. The first has bounded resources; at most one instance of G may be in execution at any time, since the result of a given iteration is needed before the next can start. Thus, $R_{F''} = R_G + c''$, where c" is a constant bound on the number of tokens in F" itself. The second loop does not have bounded resources, according to our definition. The index variable i

does not require the output of G in order to initiate the next iteration. Thus, as many as n instances of G may be in execution concurrently, and $R_F = nR_G + c$. A loop has bounded resources if the number of potential concurrent iterations is bounded. This occurs when all loop variables are mutually dependent (or equivalently when the graph for the loop forms a single strongly connected component [6]).

```
def F'' a = (initial x ← a
             while p(x) do
                 new x ← G(x)
             return x)

def F n = (initial s ← 0        ! Same as F n in Fig. 3-3 !
               i ← 1
           while i ≤ n do
               new s ← s + G(i);
               new i ← i + 1
           return s)
```

Figure 7: Example of Bounded and Unbounded Loops

By introducing auxiliary control arcs it is possible to control the maximum unfolding of a loop. Given some maximum number of concurrent iterations *k*, iteration i must be delayed until iteration i-*k* has completed. This is accomplished basically as shown in Figure 8. A *gate* operator is placed on the output of the predicate; the gated must be *triggered* in order for an iteration to proceed. Initially *k* trigger tokens are available, allowing the first *k* iterations to begin without delay. When these are used up the loop will pause. Trigger tokens must be regenerated as iterations finish. To do this, we add a *synchronization tree* (simply a tree of identity operators) on the outputs of the D operators. When a token emerges from the synchronization tree with iteration number i+1, we are assured that iteration i has completed and iteration i+*k* may be triggered. The last node of the synchronization tree increments the iteration number by *k*-1. Once loops are controlled in this way, it is possible to operate on iteration numbers modulo *k*+1. (The extra iteration number is necessary because in aberrant cases it is possible for one of the D operators to fire for the k^{th} time before one of the switches fires once.) The D_n operators in Figure 8 increment the iteration number modulo n, where n = k+1. The synchronization tree decrements the iteration number by 2 modulo n. A variety of hardware tricks can be employed to obviate the clean-up of trigger tokens on completion of the code-block.

It is instructive to work through an example with $k = 1$. The initial wave of tokens coming from the caller pass into the predicate and queue at the switches. The gate has a trigger, so the first iteration can commence immediately. Suppose the left-most loop variable is produced very quickly; it will activate the predicate, but a new iteration can not begin since the gate deprives the switches from firing again. The second iteration can not begin until all the D operators have executed for the first iteration and a token has emerged from the synchronization tree.

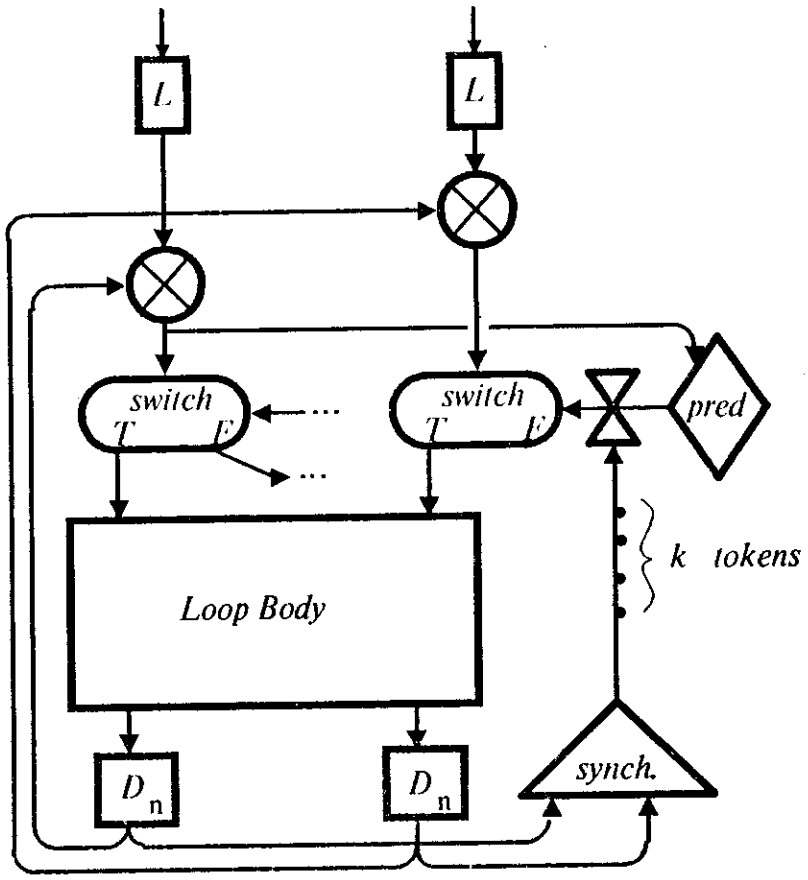

Figure 8: Resource Bounded Loop Schema

The value of the parallelism parameter does not affect the total computation performed by a loop, but it does affect the amount of parallelism the loop exhibits. One caveat is in order: According to the definition of Id, the language in which we program our dataflow machine, it is possible to write pathological loop programs which will not terminate if any bound is placed on the number of concurrent iterations. In our current redesign of the language such constructs will be excluded. A terminating loop will terminate even if iterations are executed one at a time, that is, with k set to 1. The loops encountered in practice have this property in any case. Thus, for the new language definition our compiler can legitimately generate Resource Bounded Loop Schema for *all* loop programs.

Transformed in this way, the resource requirements of the second loop in Figure 7 are given by $R_F = kR_G + c$, for parallelism parameter k. The auxiliary dependencies act like a governor on the loop; they constrain the loop when a certain amount of parallelism is realized. Certainly, if a loop is

inherently bounded or if the maximum number of iterations is known when a loop is invoked, there is no point in setting k greater than this number. The parallelism parameter allows the resource management system to control the amount of parallelism, and thereby the resource requirement, of an executing program. When a loop code-block is activated, the parallelism parameter for that particular activation can be set in accordance with the availability of machine resources. To avoid deadlock we ensure that at the time a code-block is activated there will at least be resources available to support execution with k equal to 1. It is important to note that the value of k does not affect the program structure; it is a part of the state associated with the task invocation.

6. A Resource Management Policy

Given loops with parameterized unfolding, the primary question is how to set the various parameters. Our efforts at this point are aimed at developing a simple, effective policy, rather than striving for optimality. One policy that appears attractive is a direct analog of the breadth-first/depth-first approach presented in Section 3. The idea is illustrated by an example.

The (somewhat contrived) program in Figure 9 initializes a matrix with zeros, except on the the left and upper boundaries, and then repeatedly relaxes the matrix by averaging points with their left and upper neighbors.

First, we would like to derive resource expressions for the various code-blocks in the program. In general a loop will have a resource requirement of the form $A + kB$, where k is the parallelism parameter for the loop. A simple way to compute these coefficients is to let A be the requirement of the initial portion and B be the requirement of the body. This approach is conservative for two reasons. First, the loop body generally requires the result of the initial portion, so it will lag behind the initial portion. However, this effect is often quite small because some parts of the loop begin almost immediately, *e.g.*, generation of index values. Moreover, since structures in Id (*i.e.*, I-structures [4, 10]) are *non-strict*, significant overlap is possible; the initial portion may produce a structure while the loop consumes it. Nonetheless, we expect the initial portion to overlap with only the first few iterations of the body. Secondly, some part of the computation within the body portion must be performed before new iterations can be initiated, *e.g.*, the generation of index values, and thus contribute to the additive coefficient A, rather than the multiplicative coefficient B. If loops contain nested loops, the resource expression are composed in the obvious way.

Below we give the resource expressions for the various code-blocks in Figure 9. These numbers were derived from the graphs produced by the Id compiler [11] using the approach developed in [6]. They give a fairly tight bound on the maximum number of tokens that can exist in the graph at any time. The actual numbers are not critical to this discussion, but they give a feel for the scale of numbers involved.

```
def relax n maxsteps =
  (initial A ←
        (initial
            rows    ← allocate(0..n);
            rows[0] ←
                (initial row0 ← allocate(0..n)
                 for j from 0 to n do
                      row0[j] ← top(j)
                 return row0)                    ! loop init-row0 !
         for i from 1 to n do
              rows[i] ←
                (initial rowi    ← allocate(0..n);
                         rowi[0] ← left(i)
                 for j from 1 to n do
                      rowi[j] ← 0.0
                 return rowi)                    ! loop init-rowi !
         return rows)                            ! loop init      !

   for steps from 1 to maxsteps do
        new A ←
         (initial
              rows    ← allocate(0..n);
              rows[0] ←
                 (initial row0 ← allocate(0..n)
                  for j from 0 to n do
                       row0[j] ← A[0,j]
                  return row0)                   ! loop row0      !
          for i from 1 to n do
               rows[i] ←
                 (initial rowi    ← allocate(0..n) ;
                          rowi[0] ← A[i,0]
                  for j from 1 to n do
                       rowi[j] ← (A[i,j] + A[i-1,j] + A[i,j-1]) / 3
                  return rowi)                   ! loop rowi      !
          return rows)                           ! loop step      !
   return A)                                     ! loop Relax     !
```

Figure 9: A Non-trivial Dataflow Loop Program

The resource requirements of the entire code-block relax is given by a function of the form $R_{relax}(k,...)$ where k is the parallelism parameter of the main loop. The expression for R_{relax} is equal to the resources for the initial part plus *k* times the resources for the body, assuming the loop is allowed to unfold *k* steps. Working inward, we get the following:

Resource requirement for the initial part of relax:

$$R_{init}(k,k1,k2) = 12 + R_{init\text{-}row0}(k1) + k \times R_{init\text{-}rowi}(k2),$$
$$R_{init\text{-}row0}(k) = 7 + 2 \times k,$$
$$R_{init\text{-}rowi}(k) = 8 + 2 \times k.$$

Resource requirement for the body part of relax:

$$R_{step}(k,k1,k2) = 14 + R_{row0}(k1) + k \times R_{rowi}(k2),$$
$$R_{row0}(k) = 8 + 3 \times k,$$
$$R_{rowi}(k) = 9 + 10 \times k.$$

Resource requirement for the whole of relax:

$$R_{relax}(k,k1,k2,k3,k4,k5,k6) = R_{init}(k1,k2,k3) + k \times R_{step}(k4,k5,k6).$$

The parallelism parameters could be set statically, but there are advantages to deferring the decision until run time. At the time relax is activated, only k need be chosen; it determines how much loop relax is allowed to unfold. Once loop init is activated, $k1$ will be chosen. Loop init will activate loop init-row0, causing $k2$ to be chosen. Each iteration of loop init causes an instance of loop init-rowi to be activated, so $k3$ will be chosen n times, once for each activation of loop init-rowi. Similar comments hold for the other parameters. The inner-most parameter, $k6$, will be determined essentially n×maxsteps times. At the time a loop is activated, its parallelism parameter can be set in accordance with the resources available. If ample resources are available, it is advantageous to allow loops to unfold substantially, but if resources are scarce, loops must be constrained.

In Section 3, we suggested that a program should be allowed to unfold until sufficient parallelism was exposed, *i.e.*, breadth-first, and then to constrain the unfolding, *i.e.*, depth-first, to keep resource requirements low. Breadth-first unfolding for loops corresponds to a large parallelism parameter; depth-first unfolding corresponds to a parallelism parameter equal to one. This suggests that the outer block should be allowed to unfold substantially, and the inner blocks only if sufficient resources remain.

To this end, we define MIN-$R_L(k)$ to be the resource expression for L with 1 substituted for each subordinate parallelism parameter. Thus, MIN-$R_{init}(k) = 21 + 10 \times k$. When a loop L is activated, k is chosen and MIN-$R_L(k)$ resources are reserved. This ensures that all subordinate code-blocks will have enough resources allocated to them *a priori* to support fully constrained execution. Thus, for pure loop programs, *i.e.*, no recursion, resource deadlock can be avoided.

To capture the breadth-first/depth-first strategy, when outer block L is invoked we choose k such that MIN-$R_L(k)$ is large, perhaps as large as the total available resources at the time L is invoked. As the inner blocks are invoked, they are assured of enough resources to allow for parallelism parameters of one, and if additional resources are available, the parallelism parameter may be set higher. Additional resources become available whenever the resources held by a terminating task are released.

It is important to note that resource allocations can be very dynamic. In the example above, loop init and various instances of loop step execute concurrently. Suppose k is set such that MIN-$R_{relax}(k)$ accounts for half machine resources. Then the subordinate loops will be given parallelism parameters greater than 1. Initially, subordinates of loop init may be given large parallelism parameters. This means loop init will occupy a large part of the machine. Suppose the first instance of loop init-rowi is assigned a parallelism parameter of 10. MIN-$R_{init\text{-}rowi}(1)$ resources were reserved for it when loop init was invoked, thus an additional MIN-$R_{loop\text{-}rowi}(10)$ — MIN-$R_{loop\text{-}rowi}(1)$ resource must be allocated. These will be released to the global pool when the instance of loop init-rowi terminates. The original MIN-$R_{init\text{-}rowi}(1)$ resources cannot be released, as they must be reserved for later instances of init-rowi. Instances of loop step, loop row0, and loop rowi are being activated concurrently with the instances of loop init-rowi. Thus, resources released by init-rowi may be claimed by instances of these other loops. The various concurrent loops compete for a finite pool of resources and they activate subordinate code-blocks.

7. Conclusions

The work presented here has already proved its effectiveness in running large programs on the MIT Tagged Token Dataflow machine, but it is just a beginning. Techniques for deriving resource expressions should be developed further to give *practical bounds*. Resource allocation policies must be tested by executing large programs and observing their dynamic behavior. The policy outlined above is perhaps a good start, but it has some serious drawbacks. For example, unfolding loop relax a large amount is not necessarily wise. It causes many versions of the matrix to be in use simultaneously. If a single step offered enough parallelism to keep the machine busy, it would be best to constrain the outer loop and allow its inner loops to unfold.

The approach developed here applies to sizable fragments of programs, but not to programs in their entirety. When programs have an arbitrary recursive structure, we cannot predict the resource requirements of the subordinates of an activation, regardless of how the execution is constrained. Thus, at some level we must resort to fully general techniques, such as the task queue approach mentioned in Section 3. However, the fully general technique can be applied at the level of Resource Bounded Graphs, rather than individual code-block activations, *i.e.*, tasks.

We have not addressed the issue of structure storage at all. Structure storage reclamation in parallel machines is a very active research area, and all of the basic approaches (reference counts, copying garbage collection, etc.) raise serious implementation problems. Controlled loop unfolding offers interesting possibilities in this area too, since intermediate structures can be recycled automatically. For example, if the maximum unfolding of loop relax equals 2, three versions of the matrix should be sufficient to sustain the entire computation.

Suspending the parent task when a child is initiated is potentially attractive, but is difficult in the Tagged Token Dataflow approach. The architectural ramifications of controlling program execution must be explored more deeply.

We are encouraged to learn that other researchers are examining the issues discussed here. Experiments performed by the Manchester Dataflow team have demonstrated similar problems with unconstrained unfolding of loops. They have developed a technique for encouraging depth-first traversal of the invocation tree when resources become scarce [19]. One can reach the erroneous conclusion based on the Manchester and MIT experience that the resource management problems discussed in this paper are peculiar to dataflow; if we have encountered these problems before others it is because dataflow systems coupled with functional languages expose more parallelism than any other system.

Acknowledgments

This work was performed at the M.I.T. Laboratory for Computer Science under the Tagged-Token Dataflow project. Funding is provided in part by the Advanced Research Projects Agency of the U.S. Department of Defense, contract N00014-75-C-0661, and in part by International Business Machines Corporation, T. J. Watson Research Center. We wish to thank the IFIPs TC-10 for inviting us to present these ideas.

References

1. Amano, H.,T. Toshida, and H. Aiso. (SM^2): Sparse Matrix Solving Machine. Proceedings of the 11th International Symposium on Computer Architecture, Ann Arbor, Mich., 1984.

2. Arvind. Decomposing a Program for Multiple Processor System. Proceedings of the 1980 International Conference on Parallel Processing, August, 1980, pp. 7-14.

3. Arvind and R. Iannucci. "Two Fundamental Issues in Multiprocessing: the Dataflow Solution". Proceedings of the 10th International Symposium on Computer Architecture, Stockholm, Sweden, June, 1984. Revised version appears as MIT Laboratory for Computer Science CSG Memo 226-3, August 1985

4. Arvind, and R. E. Thomas. I-Structures: An Efficient Data Type for Functional Languages. Tech. Rep. TM-178, Laboratory for Computer Science, MIT, Cambridge, Mass., September, 1980.

5. Campbel, M. Static Allocation for a Dataflow Multiprocessor. Proceedings of the International Conference on Parallel Processing, 1985, pp. 511-517.

6. Culler, D. E. Resource Management for the Tagged-Token Dataflow Architecture. Master Th., Massachusetts Institute of Technology, Cambridge, Mass., January, 1985. Laboratory for Computer Science Tech. Rep. TR-332

7. Gao, C., J. Liu, and M. Railey. Load Balancing in Homogeneous Distributed Systems. Proceedings of the International Conference on Parallel Processing, 1984, pp. 302-306.

8. Halstead, R. The Architecture of a Myriaprocessor. Proceedings of COMPCON 81, September, 1981, pp. 299-302.

9. Halstead, R. Multilisp: A language for Concurrent Symbolic Computation. MIT Laboratory for Computer Science, August, 1984. Submitted to ACM Transactions on Programming Languages and Systems

10. Heller, S. and Arvind. Design of a Memory Controller for the MIT Tagged Token Dataflow Machine. Proceedings of IEEE/ICCD '83, October, 1983. Also appears as MIT Laboratory for Computer Science CSG Memo 230

11. Heller, S., and Traub, K. The ID Compiler User's Manual. Computation Structures Group Memo 248, Laboratory for Computer Science, Massachusetts Institute of Technology, May, 1985.

12. Ho, L. and K. Irani. An Algorithm for Processor Allocation in a Dataflow Multiprocessing Environment. Proceedings of the International Conference on Parallel Processing, 1983, pp. 338-340.

13. Keller, R. Rediflow Multiprocessing. Proceedings of COMPCON '84, 1984.

14. Leiserson, C. *Area-Efficient VLSI Computation*. MIT Press, Cambridge, 1983.

15. Mehrota, R. and S. Talukaar. Scheduling of Tasks for Distributed Systems. Proceedings of the 11th International Symposium on Computer Architecture, Ann Arbor, Mich., 1984.

16. Milner R. A Calculus of Communicating Systems. Tech. Rep. LNCS, Edinburgh University, 1980.

17. Ni, L. and C. Wu. Design Tradeoffs for Process Scheduling in Tightly Coupled Multiprocessor Systems. Proceedings of the International Conference on Parallel Processing, 1985, pp. 63-70.

18. Peterson, J. and J. Tuazon. The Mark III Hypercube-Ensemble Concurrent Computer. Proceedings of the International Conference on Parallel Processing, 1985, pp. 71-73.

19. Ruggiero, J. and Sargeant. Hardware and Software Mechanisms for Control of Parallelism. University of Manchester, Computer Science Dept.

20. Schwartz, J.T. Ultracomputers. *ACM TOPLAS 2* (October 1980), 484-521.

21. Stone, H. Parrallel Processing with the Perfect Shuffle. *IEEE Transactions on Computers c20*, 2 (February 1971), 153-161.

22. Stout, Q. Sorting, Merging, Selecting, and Filtering on Tree and Pyramid Machines. Proceedings of the International Conference on Parallel Processing, 1983, pp. 214-221.

23. Vafa, B. A Resource Management Strategy for the Tagged-Token Dataflow Architecture. Master Th., Massachusetts Institute of Technology, Cambridge, Mass., May, 1985.

24. Vrslovic, D., E. Gehringer, Z. Segall, and D. Siewiorek. The Influence of Parallel Decomposition Strategies on the Performance of Multiprocessor Systems. Proceedings of the 12th International Symposium on Computer Architecture, Boston, Mass., 1985.

25. Williams, E. Assigning Processes to Processors in a Parallel System. Proceedings of the International Conference on Parallel Processing, 1983, pp. 404-406.