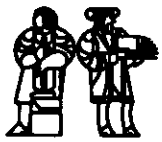
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Resource Requirements of Dataflow Programs†

Computation Structures Group Memo 280

December 15, 1987

Arvind
David E. Culler

†Paper three in a sequence of four, including CSG Memos 278, 279, 281

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

Abstract

The dataflow approach exposes ample parallelism, even in programs where little attention has been paid to special parallel algorithms, but the approach may be *too successful* in this respect. As more parallelism is exposed, more resources are required. Unless precautions are taken, programs with tremendous parallelism will saturate, and even deadlock, a machine of reasonable size. We examine the waiting-token requirements of realistic programs through *resource profiles*, which depict the number of outstanding tokens over time under an idealized execution model. The requirements are shown to be large, regardless of the number of processors executing the program. We develop a mechanism for reducing resource requirements, called *loop bounding*, which constrains the unfolding of loops to control the amount of exposed parallelism, and show its effectiveness. This mechanism is shown to be a key step towards managing other resources, such as I-structure storage and tags. Comparisons are made throughout with more conventional approaches to parallel computing, in order to show that this resource problem is not peculiar to dataflow, even though it may be most evident in that context.

Resource Requirements of Dataflow Programs

1 Introduction

The dataflow approach exposes ample parallelism, even in programs where little attention has been paid to special parallel algorithms, but the approach may be *too successful* in this respect. As more parallelism is exposed, more resources are required. Unless precautions are taken, programs with tremendous parallelism will saturate, and even deadlock, a machine of reasonable size. This claim is clearly evident in resource profiles presented below which correspond to parallelism profiles described in a companion paper[4]. We claim that this resource problem arises in any system which allows dynamic generation of concurrent tasks, although it may be most acute in dataflow models since parallelism is so aggressively exploited. Ideally, enough parallelism should be exposed to fully utilize the machine on which the program is executing, while minimizing the resource requirements of the program. We show that limiting the maximum number of simultaneously active iterations of a loop is highly effective in reducing the resource requirements of typical scientific programs without sacrificing performance. The implementation of this idea is based on compiling loop programs into dataflow graphs with a "loop-bounding" parameter which can be set at run time according to some policy.

We begin Section 2 by articulating the resource requirements of programs on conventional, sequential machines and show that certain properties of high-level languages complicate storage management. We then argue that parallel execution generally requires more storage than sequential execution. Both concerns must be addressed to support dataflow execution of programs in a language like Id. Also, since the execution of dataflow programs is usually given in terms of propagating data *tokens* through a graph, it is not obvious what resources are needed to execute a dataflow program; Section 2 relates the resources associated with dataflow programs and compares these to familiar storage concepts. Section 3 develops the concept of resource profiles, expanding on the characterization of programs offered by parallelism profiles, and presents data for various examples. Section 4 introduces a method of controlling resource requirements by controlling the amount of exposed parallelism, and demonstrates its effectiveness. Section 5 examines broader resource management concerns, such as recycling data-structures and other resources within portions of the program graph.

2 Dynamic Resource Allocation Under Parallel Execution

2.1 Stack-and-Heap Storage Model for Programming Languages

One way to characterize the expressiveness of a programming language is by its storage model. Among high-level languages Fortran has the simplest storage model. The storage requirement of a Fortran program is determined at compile-time and does not change during the course of execution. The static storage model contributes toward the efficiency of Fortran at the cost of expressiveness; traditional Fortran does not support recursion. Any language that supports recursion needs a stack allocated storage model to provide storage for activation frames.

Languages such as Algol-60 and its modern derivatives support recursive procedures and consequently, rely on a stack-allocated storage model. However, these languages do not permit data structures to be returned by a procedure, and the lifetime of an array or record may not be longer

than the lifetime of the procedure activation that creates it. The reason is that such a data structure cannot be allocated on the procedure activation stack; a heap-allocated storage model is required to support dynamic creation of data structures which can be passed around freely. A storage heap is a directed graph of objects. At the implementation level, a distinction is always made between an object (a data structure) and its descriptor (a pointer). The storage occupied by a data structure is not released until it is determined implicitly or explicitly that no part of the data structure can be accessed. Pointers, on the other hand, are manipulated more like scalar values.

Allocation and deallocation of storage is significantly harder for heaps than stacks. Consequently, all language implementations treat stack storage differently from heap storage. Stacks are used for procedure activation frames which contain local variables and heaps are used for data structures accessible from more than one frame.

All modern programming methodologies require a heap storage model, although differences in operations on data structures permitted in various languages are considerable. For example, in Lisp, CLU and Smalltalk storage is reclaimed implicitly by a process known as Garbage Collection, whereas in Pascal and C heap storage is explicitly deallocated by the programmer, perhaps in recognition of the fact that automatic garbage collection is an expensive and difficult operation. Functional and other declarative languages such as Id, Miranda, ML and Prolog invariably require a heap storage model and insist on implicit storage reclamation. However, heaps in functional languages are usually acyclic graphs which can be scanned for garbage collection more efficiently than arbitrary cyclic graphs.

There are significant differences in storage requirements of programs written according to different methodologies. Generally, a more abstract programming style implies that more storage management is done by the system and less by the programmer. For example, Functional languages do not permit updating of data structures and, hence, often require more storage or at least more storage allocation and deallocation operations than, say, Lisp. Among functional languages also there are significant differences between those that permit non-strict functions and data structures (*e.g.*, Miranda and Id) and those that do not (*e.g.*, pure Scheme and ML). Non-strictness supports a methodology of programming with infinite objects which often simplifies control structures in programs. In addition, it allows more parallelism to be exposed in programs. In particular, non-strict data structures allow concurrent production and consumption, and thus require synchronization on an element by element basis. One cost of supporting non-strict data structures is that the lifetime of data structures is often longer than in strict implementations. Estimating the "real" storage requirements of programs under automatic storage management is difficult; non-strictness makes the problem even more difficult.

2.2 Parallelism and Storage

Exploiting parallelism in a program increases it storage requirements and complicates storage management. The simplest form of parallelism is the introduction of vector operations to replace certain loops. Consider the vectorization of the following Fortran program for inner product.

```
SUBROUTINE IP (A, B, N)
DIMENSION A(N), B(N)
S = 0.0
DO 10, I= 1,N
  T = A(I) * B(I)
  S = S + T
```

```
10      CONTINUE
        END
```

Following the recipe given in the literature[14, 17] this would be vectorized by expanding T, a scalar, into a vector of size N and splitting the loop. The result is shown below.

```
        SUBROUTINE IP (A, B, N)
        DIMENSION A(N), B(N), T(N)
        DOALL  5, I= 1,N
          T(I) = A(I) * B(I)
5       CONTINUE
        S = 0.0
        DO 10, I= 1,N
          S = S + T(I)
10      CONTINUE
        END
```

Vectorization has clearly increased the storage requirement of the program. However, the subtlety here is that vector T can not be declared locally, because its dimension is determined when the subroutine is called. It may be introduced in the same manner as A if every call is modified to pass in a temporary vector, but this requires global analysis. Temporary vectors like T could be allocated dynamically when the subroutine is called, but this fundamentally changes the storage model of Fortran. Presumably the compiler sets aside a special common block and uses it for temporary vectors throughout the program, but it is still unclear how the size is determined.

With the recent availability of commercial multiprocessors, emphasis has shifted to program transformations that can spread a loop over multiple processors. One approach, developed by the NYU Ultracomputer group[12] and implemented on IBM's RP3[15], is to maintain the loop index variable in shared memory which a processor atomically increments when it attempts to execute an iteration of the loop. Still, each processor must maintain its own copy of local variables. In the parallel version of inner-product, each processor could have its own stack frame containing the local value of I, T and pointers to various arrays, as suggested by Figure 1. Each processor has its own local data area, in addition to the shared data area, which changes the storage model of the language in a subtle and significant manner. This approach only supports parallelization of one loop level. If a loop within a parallel loop were parallelized, an additional level of local data areas would be required, whereas the storage paradigm is one global address space and one local address space per processor.

More generally, if subroutines can be spawned dynamically as tasks, which in turn may spawn tasks, a tree of stack frames is required, not just a stack, and allocation and deallocation is no longer a simple, constant time operation. Management of trees is only slightly simpler than general heap management, and multiple tasks can be supported by allocating local variables on the heap as well[13]. The tricky part is, of course, reclaiming unused storage in the heap, and especially doing so in parallel. Moreover, parallel execution may require exponentially more storage than sequential execution, because where the calling depth is n the size of the tree is 2^n if each task spawns two subtasks.

Parallel execution and storage management are related in an even more subtle way involving names for synchronization points. The easiest way to explain this is by example. Take a Load/Store architecture like the Cray which is able to issue a memory read and continue to execute instructions

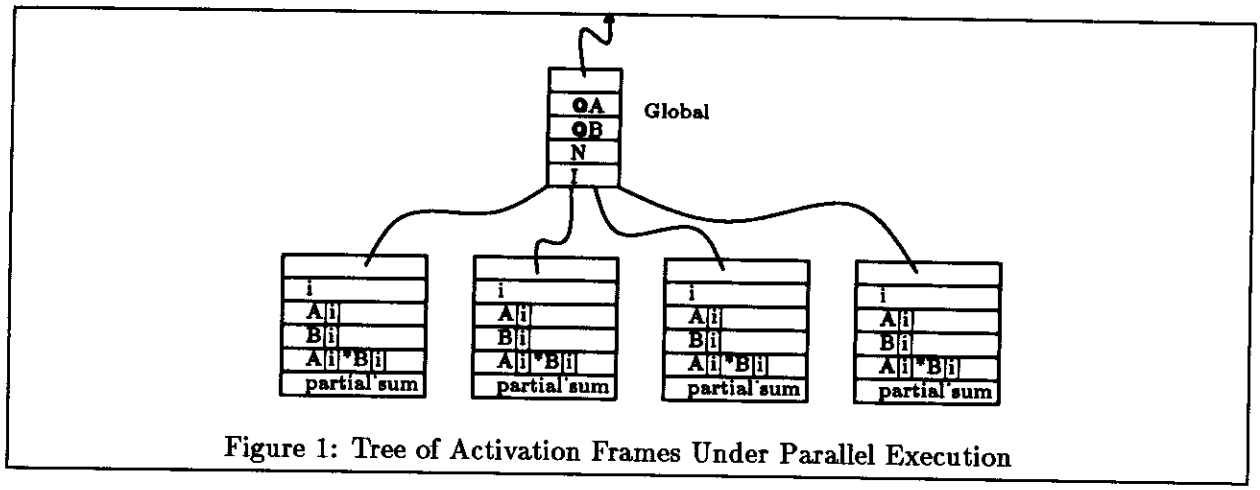


Figure 1: Tree of Activation Frames Under Parallel Execution

while the read takes place. To do this, a register is used as the target of the read, and instructions can be issued as long as they do not refer to this register. Clearly, the number of concurrent read requests is bound by the number of such registers. The register serves as a synchronization point for two asynchronous events. A similar situation arises with semaphores, locks and other high level synchronization mechanisms - as more parallelism is exploited, more concurrently active synchronization points are required.

2.3 Storage in the Dataflow Model

The resources involved in executing a dataflow program are influenced by both the factors discussed above: a powerful functional language augmented with non-strict arrays and highly parallel execution. Not surprisingly, this requires a tree of activation frames and a heap. Each user-defined function and loop in the program is represented by a dataflow graph called a *code-block*. At any point an executing dataflow program comprises a tree of concurrent code-block invocations, each identified by a *context*. Loop code-blocks may unfold allowing many iterations to be active concurrently, so the invocation tree can branch with arbitrarily large degree.

Variables in the dataflow program do not denote storage directly, but rather denote arcs in the graph. While a token is present on the corresponding arc, however, a certain amount of storage is required to represent the data value. Thus, one measure of the storage requirement of a program is the number of tokens in existence. We can consider the tokens associated with a context to be the activation frame for the code-block invocation. However, unlike stack frames which are laid out statically by the compiler, token storage is typically managed dynamically by the hardware, *c.f.*, the description of the waiting-matching store in [1]. Often we distinguish tokens which are waiting for a partner from those that are not, because the first occupy more crucial storage resources. Waiting tokens correspond to temporary variables held on the stack, whereas short lived tokens are more akin to values held in high-speed registers.

I-structures are allocated dynamically and filled by the program and are implicitly reclaimed. They can be passed into and out of functions, so their lifetime is independent of the code-block invocation that creates them. Thus, I-structure storage corresponds with the heap in many language implementations. Element by element synchronization is provided to allow concurrent production and consumption of structures.

In the remainder of the paper we focus on the token storage requirements of programs, as this is a critical resource in dataflow machines. Once the token store is exhausted, a dataflow machine will deadlock. I-structure requirements are, of course, extremely important as well, for generally large scientific problems are limited by storage for arrays, not temporaries, but as we will show in Section 5, the techniques for reducing token storage are an essential step toward solving the harder problem of effectively managing I-structures.

3 Resource Profiles of Dataflow Programs

In a companion paper[4] we developed an ideal model of dataflow program execution where in each step all enabled operations execute concurrently producing results for the next step, *i.e.*, unbounded parallelism, unit time operation, and zero communication latency. Relative to this model we defined the *Parallelism Profile* of a program to be the number of concurrent operations over time. Similarly, we can define the *Token Storage Profile* of a program to be the number of tokens in existence during each step of the ideal execution.

We again consider the simplified inner-product example. Figure 2 shows the graph (top), the parallelism profile (middle) and token storage profile (bottom). The token storage profile is annotated with arc names for each token, represented by the destination node number concatenated with l or r as needed. Initially three tokens are input, one carrying the initial value of **sum** and two carrying the initial value of i. The $\leq$ predicate is enabled and fires, producing two tokens, one for each switch node, to make a total of four. The two switches fire, the left producing a single token while the right produces three. At this point nodes 4,5, and 6 are enabled and a token is waiting at the left input to the +. The rest of the token storage profile is obtained employing the dataflow firing rule and observing that when a node fires the total number of tokens changes by the number of outputs minus the number of inputs. The maximum of the token profile is the token storage requirement of the program.[1].

Under this ideal execution model the average parallelism approaches 8/3 operations per step, while the token storage requirement is five tokens. Thus, approximately two tokens worth of storage are required to support each operation. It is not surprising that the token storage requirement is proportional to the exposed parallelism, as indicated here, since a certain amount of storage is required to hold the operands of each of the concurrent activities, although we shall see that the storage requirements can in fact be worse.

The *waiting token profile* is similar to the token profile, but more subtle. Note that the token produced by the left switch must wait until the result of the multiply is available. The data input to the right switch must wait until the predicate is evaluated. The boolean input to the left switch must wait for the data input. Stepping through the idealized execution yields the waiting token profile shown by the thick line in the bottom portion of Figure 2. As these waiting tokens are architecturally most significant, we focus on waiting token profiles.

This analysis assumed that arrays A and B were available. Suppose instead, that only array A is available, and the elements of B have yet to be filled in. Select operations against B are deferred until the stores are performed, but the index variable i continues to circulate causing elements of A to be selected, resulting in as many as n tokens waiting at the left input to the * node. Tags associated with the tokens ensure that correct pairing is achieved with elements of B, and when

[1]This rule can be formalized to give a system of integer linear constraints whose solution is the worst-case token storage requirement of the graph.[10]

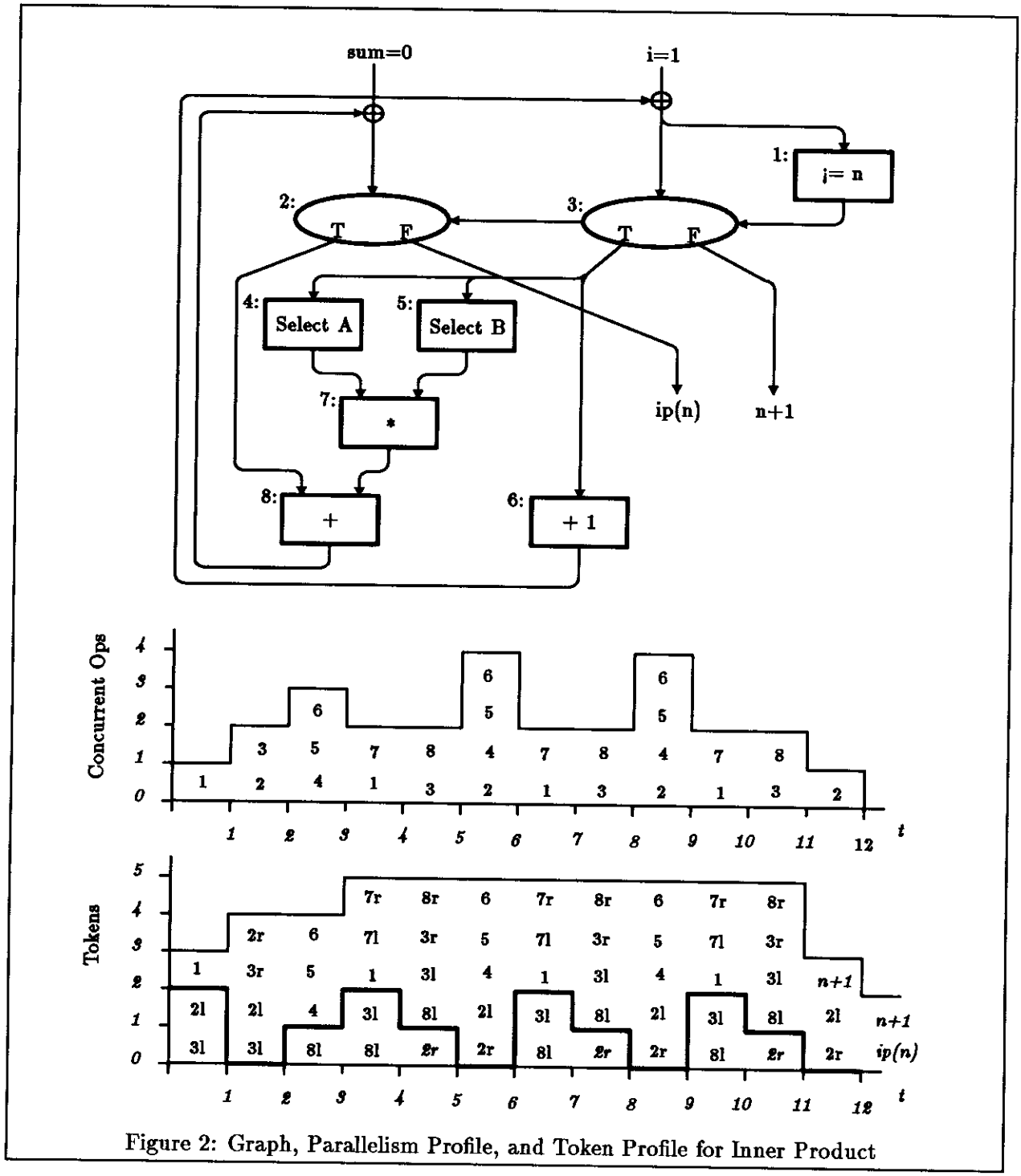


Figure 2: Graph, Parallelism Profile, and Token Profile for Inner Product

the elements of B are available all the multiplies may potentially execute in parallel. This scenario is quite similar to the vectorization example discussed above, and pays a similar storage penalty. Since the summation is still serial, the storage requirement is $\mathcal{O}$(n), while the average parallelism is constant.

We now present three examples which will be used in the remainder of the paper. First, matrix multiply using the straight-forward algorithm generates the parallelism and waiting-token profiles shown in Figure 3 on 16x16 matrices. The Id code appears in a companion paper.[11] The outer two loops unfold, allowing n^2 inner products to proceed in parallel for matrices of dimension n, and the waiting-token requirement is proportional to the amount of parallelism. The parallelism profile shows a bell shape because each loop spawns instances of inner loops in a staggered fashion. Second, a hydrodynamics and heat conduction application (Simple) run for eight timesteps on a 16x16 mesh (see [11, 5] for details) generates the nearly periodic parallelism profile in Figure 4, with a stair-step waiting-token profile. In this case, the storage requirements are excessive compared to the exposed parallelism. This example shows eight iterations on a 16x16 mesh; real problems involve 100,000 iterations on a 100x100 mesh. Finally, Figure 5 shows the parallelism and waiting token profiles of a 2-dimensional Gaussian relaxation for four iterations on a 16x16 grid; the parallelism and waiting token profiles are both periodic. Again, the waiting-token requirement is proportional to the amount of exposed parallelism.

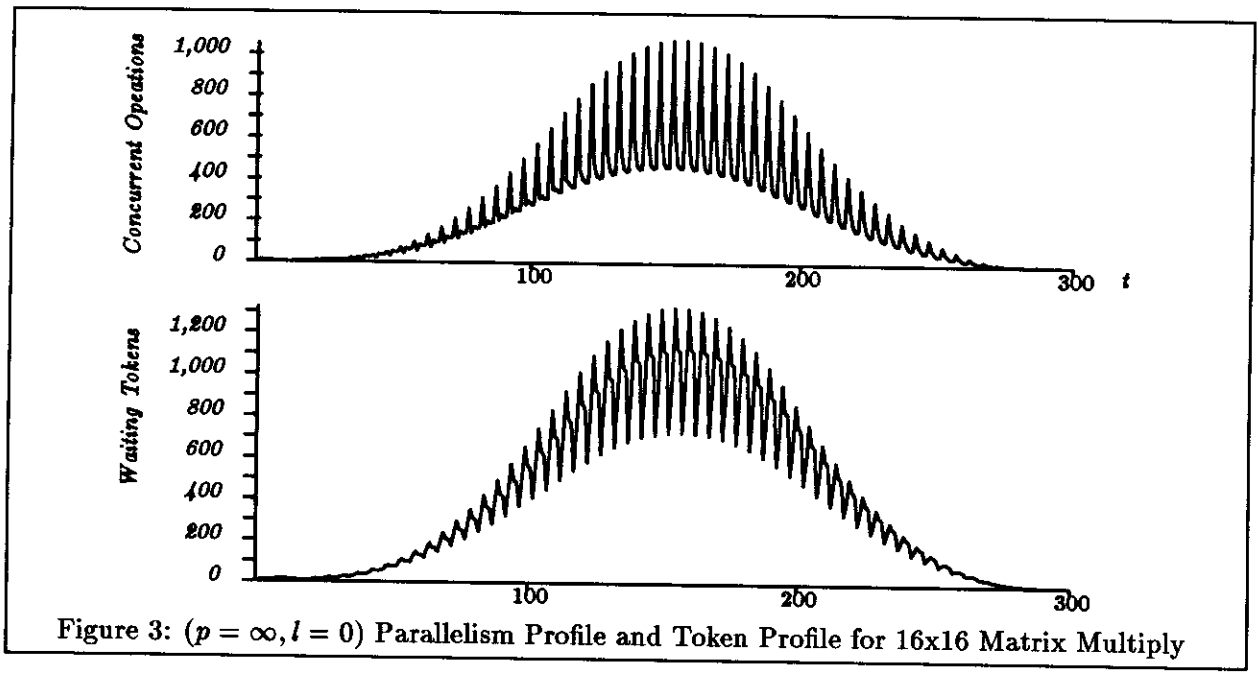


Figure 3: ($p = \infty, l = 0$) Parallelism Profile and Token Profile for 16x16 Matrix Multiply

4 Controlling Resource Requirements by Controlling Program Unfolding

4.1 Excessive Parallelism

While it is to be expected that the storage required by a program would be at least proportional to the amount of parallelism, since each operation requires operand values, we should consider a more

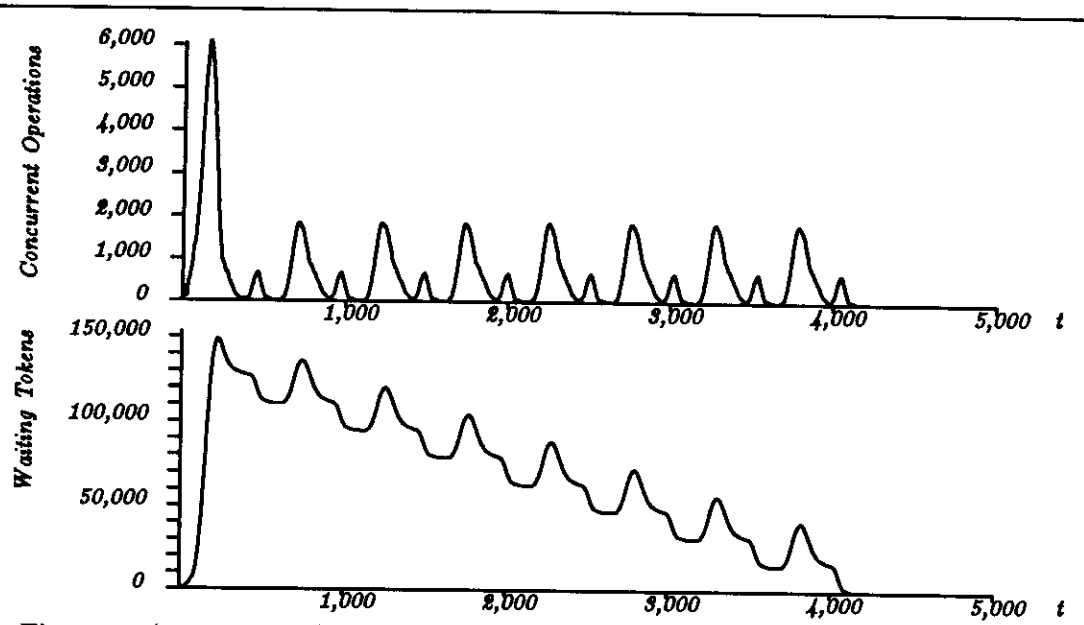


Figure 4: $(p = \infty, l = 0)$ Parallelism Profile for Simple, 8 iterations on 16x16 mesh

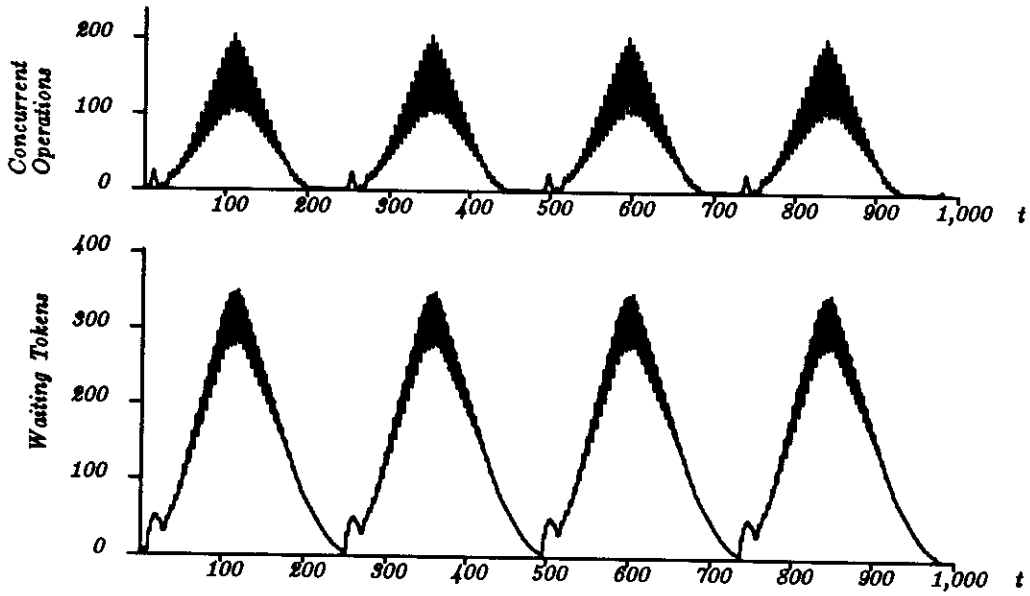


Figure 5: $(p = \infty, l = 0)$ Parallelism Profile and Token Profile for Gaussian Relaxation, 4 iterations on a 16x16 grid

realistic scenario where the amount of parallelism that can be exploited is fixed by characteristics of the machine, *i.e.*, number of processors, latency, etc. It is a serious problem if the potential parallelism in the program is large compared to that which the machine can exploit and resource requirements track the potential parallelism. In a companion paper[4] we introduced a finite processor execution model as a refinement of the ideal execution model. In each step, a fixed number of operations may be performed. Operations are selected from the set of enabled operations on a FIFO basis.

Figure 6 shows the parallelism and token profiles for 16x16 matrix multiply under this finite processor model with at most 50 operations per step. The number of steps required has increased to 1,511, and the waiting token storage requirement has not diminished. In fact, it has increased substantially. *We observe that limiting the amount of parallelism that can be exploited does not reduce the resource requirement under fair scheduling. To reduce the resource requirement, we must reduce the* exposed *parallelism.*

The source of parallelism here is the unfolding of loops; the outer loop initiates n instances of the middle loop, each of which initiate n inner products, for nxn matrices. By limiting the number of concurrent iterations of various loops we can limit the exposed parallelism. If iterations of the outer loop were executed serially, the $\theta(n^2)$ parallelism should be reduced to θ(n) parallelism, with the critical path extended from θ(n) to $\theta(n^2)$. Serializing the middle loop should have a similar effect. The inner loop is nearly serial because of how the inner product is coded.

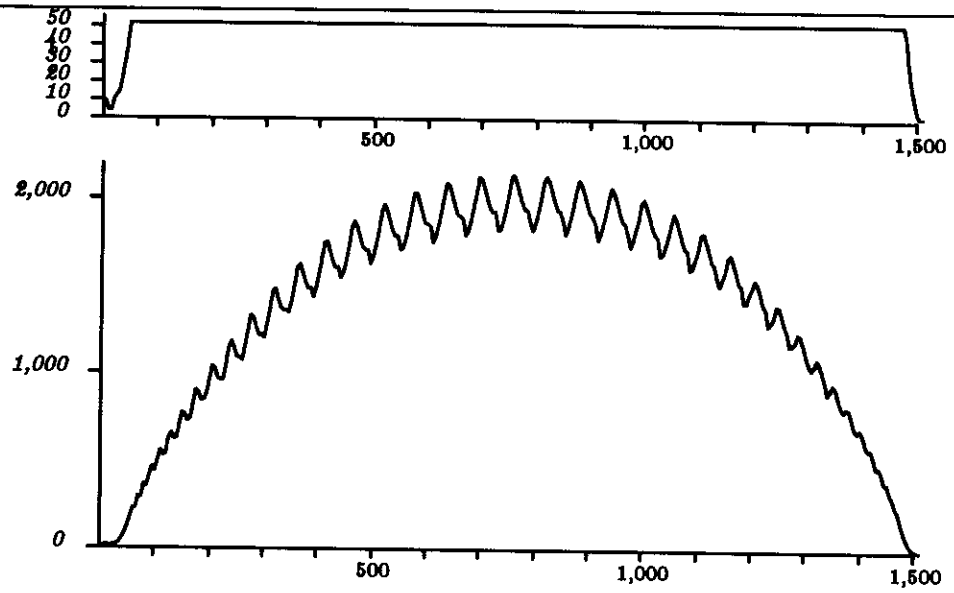


Figure 6: ($p = 50, l = 0$) Finite Processor Parallelism and Token Profiles for 16x16 Matrix Multiply

In order to exploit the full parallelism in matrix multiply using Fortran annotated for multiple processors as outlined in Section 2.2, an inner product would have to be initiated on each of n^2 processors, requiring $o(n^2)$ storage in stack frames, which is comparable to what we see in the waiting token profile. In general, it is difficult to exploit parallelism in nested loops with these approaches, although in this case it is possible with a certain amount of additional control overhead. If only a small number of processors were available, this would be extremely wasteful, so we might elect to parallelize only one loop, forcing the other to execute sequentially. Unfortunately, having

done so the program is no longer well-suited to a large number of processors. In addition, the loop we have chosen to parallelize may have few iterations, depending on the shape of matrices passed to the matrix multiply routine, resulting in little parallelism. Vectorization is difficult to apply in this example, because it must work from inner loop outward, but we observe that in order to reduce the amount of temporary storage introduced it is possible to "strip-mine" the loops, *i.e.,* break a loop into a loop of smaller vectorized loops.

4.2 Loop Bounding in Dataflow Programs

Under a dataflow approach parallelism can be constrained by bounding the unfolding of loops in a flexible manner, which does not "compile in" a limit on the parallelism that can be exploited. We expect the runtime resource management system to handle the differing requirements of large and small configurations, thus loops are compiled to have bounded unfolding, where loop bound can be set at the time the loop is invoked. In a dataflow model, the only way to ensure that one event follows another is to introduce an (artificial) data dependence between the two events. It has been shown that loops have bounded unfolding under any execution order if and only if all loop variables are mutually dependent[10]. We can use this result to develop a loop schema with parameterized unfolding, *i.e.,* no more than some number k iterations, determined dynamically, can be active concurrently. One such bounded loop schema is shown in Figure 7. A gate is placed on the output of the predicate, which inhibits new iterations from starting unless a trigger token is present at the control input to the gate. Initially, k such trigger tokens are provided, and one is consumed each time an iteration starts. A termination tree detects when an iteration has terminated and supplies the trigger token which enables another iteration to start. In effect, initiation of a new iteration is made dependent on the termination of the one k earlier. Exact details of how tags for these tokens are manipulated and how the graphs are made self-cleaning are beyond the scope of this paper[9, 7], but we may consider the effect of this mechanism.[2]

Figure 8 shows the parallelism profiles and Figure 9 the waiting token profiles for matrix multiply example under the ideal execution model, with only the outer loop constrained (top), with only the middle loop constrained (middle), and finally with both loops constrained (bottom). All of these show greatly reduced waiting token requirements and moderate potential parallelism, making them well-suited for a machine of say 16 or 32 processors. Which setting of loop parameters will be best for a given invocation of matrix multiply depends on the shape of the matrices and the availability of resources.

To see the importance of this technique, we must consider its effect under the finite processor model. Figure 10 shows parallelism and waiting token profiles for matrix multiply with instances of the middle loop bound to allow at most two concurrent iterations and the execution model allowing at most 50 operations per step. In comparison with Figure 6 the resource requirement has been reduced to less than 20% while the total number of steps has increased less than 1%. By bounding all the loops, the behavior of matrix multiply becomes independent of the problem size.

Under a model where iterations are distributed over processors and each processor executes one iteration at a time, the number of concurrent iterations is effectively bound by the number of processors. It might be possible to affect a similar mechanism in a dataflow machine, but here we have articulated control of parallelism without stepping outside the formal model. Any execution

[2]Bounding loops introduces a degree of strictness in the computational model, and as such changes the semantics slightly. It is possible to write programs which will terminate only if arbitrarily many iterations can be active concurrently, but these programs are rather peculiar can could not be written in any conventional language.

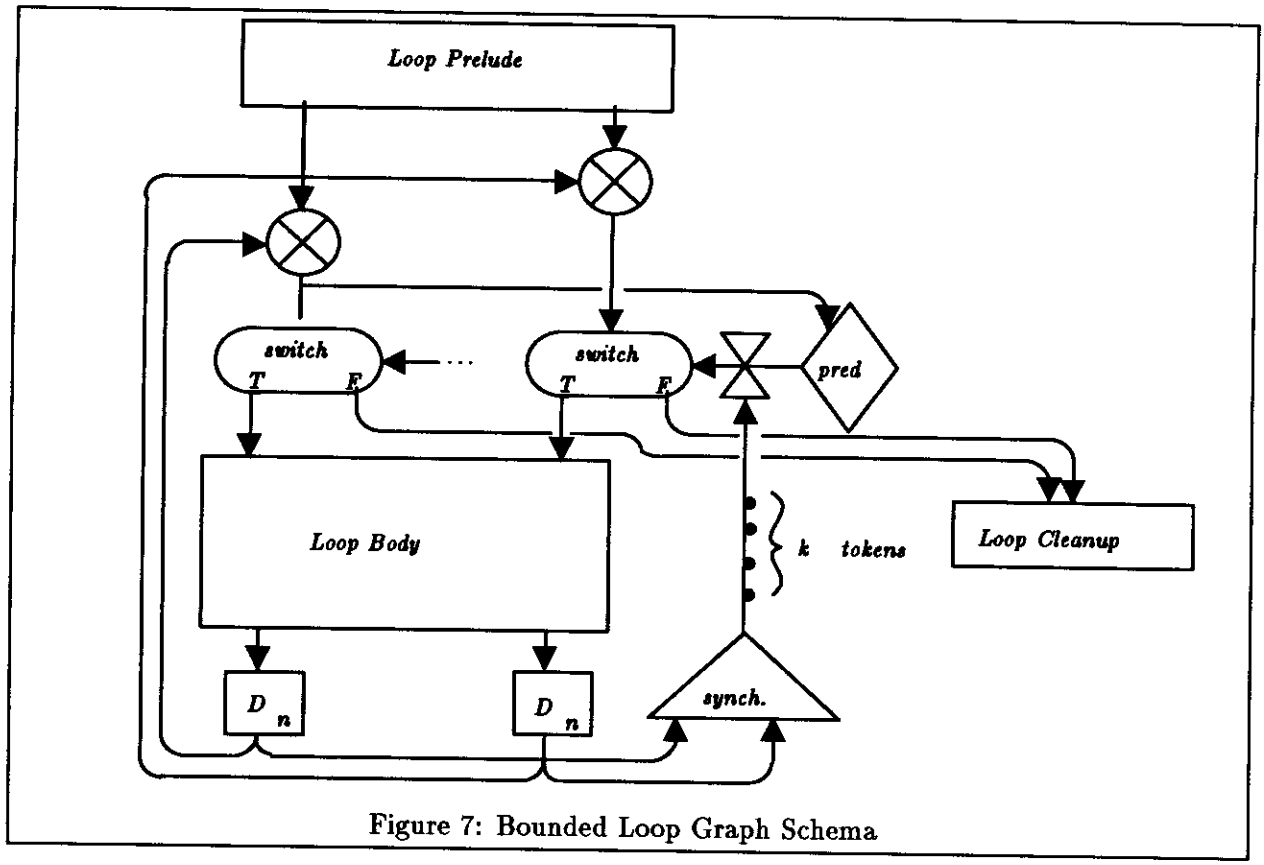


Figure 7: Bounded Loop Graph Schema

model which treated each inner product as a task and scheduled them fairly would experience similar resource requirements.

The current version of the Id compiler generates parameterized bounded-loops for all loop code blocks. If the bound is large enough, no restriction is placed on program unfolding. This is how all the unbounded profiles in this paper were generated. Profiles with bounded loops are generated by setting loop bounds for certain code blocks prior to execution.

4.3 Useless parallelism

The profiles for Simple in Figure 4 above raise a more serious concern than the excess parallelism in the previous example - useless parallelism. The program unfolds, but major portions of it allocate resources and then wait until data becomes available. The eight iterations of the outer loop are clearly visible as there is a strong constriction point, where the new time-step is determined, between iterations. Each iteration has two distinct phases, the first is the hydrodynamics calculation, the second heat conduction work. Each phase reduces the values in the mesh to a single value used in determining the time step for the next phase. Thus, each point of the mesh in one iteration depends on the entire mesh produced in the previous iteration - there is almost no useful overlap between iterations. The outer loop simply performs a number of time-steps. All eight iterations unfold, each allocates instances of the mesh, unfolds over the dimensions of the mesh and issue reads against the mesh of the preceding iteration. As the data is filled in, the deferred reads are satisfied, and the computation sweeps across the successive iterations. In many applications this

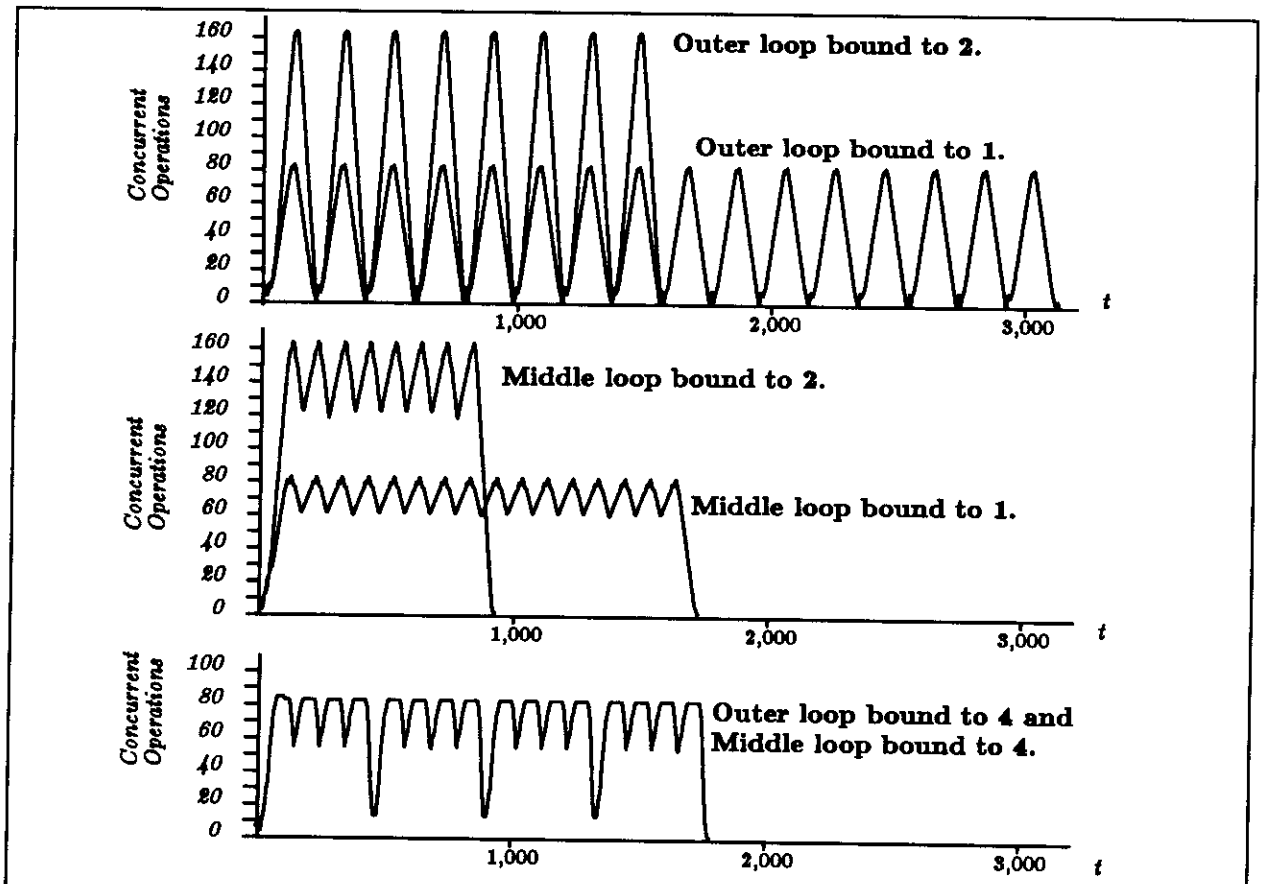


Figure 8: ($p = \infty, l = 0$) Parallelism Profiles for 16x16 Matrix Multiply Under Various Loop Bounds

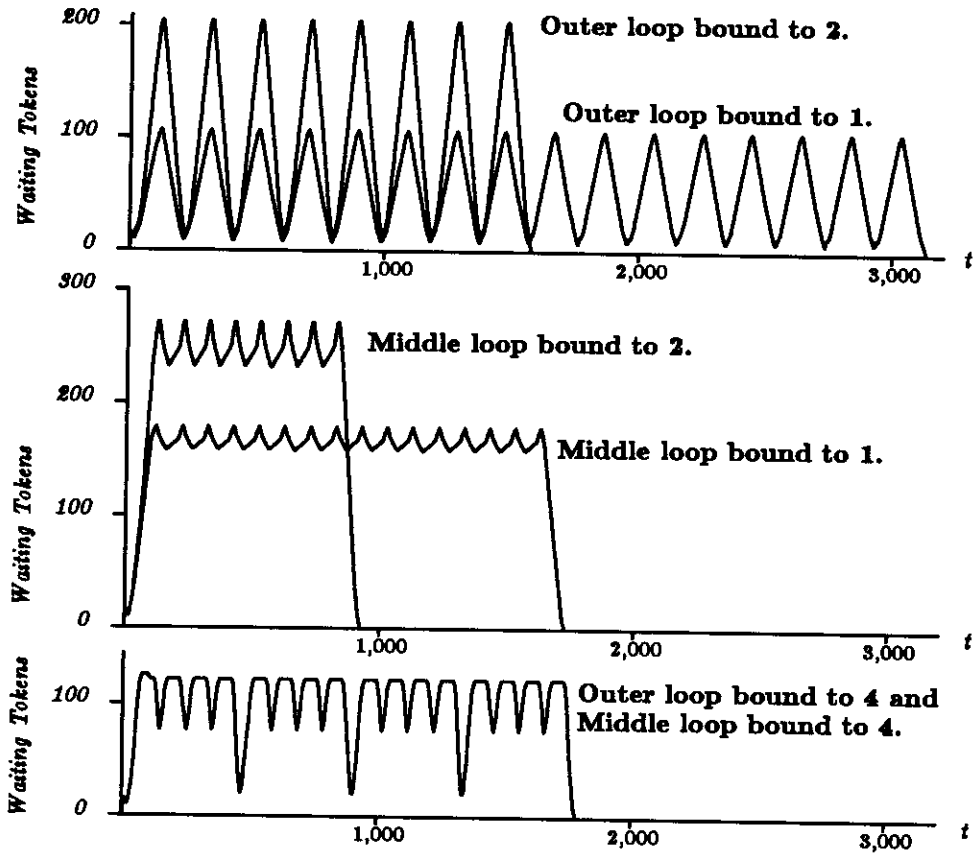


Figure 9: ($p = \infty, l = 0$) Waiting Token Profiles for 16x16 Matrix Multiply Under Various Loop Bounds

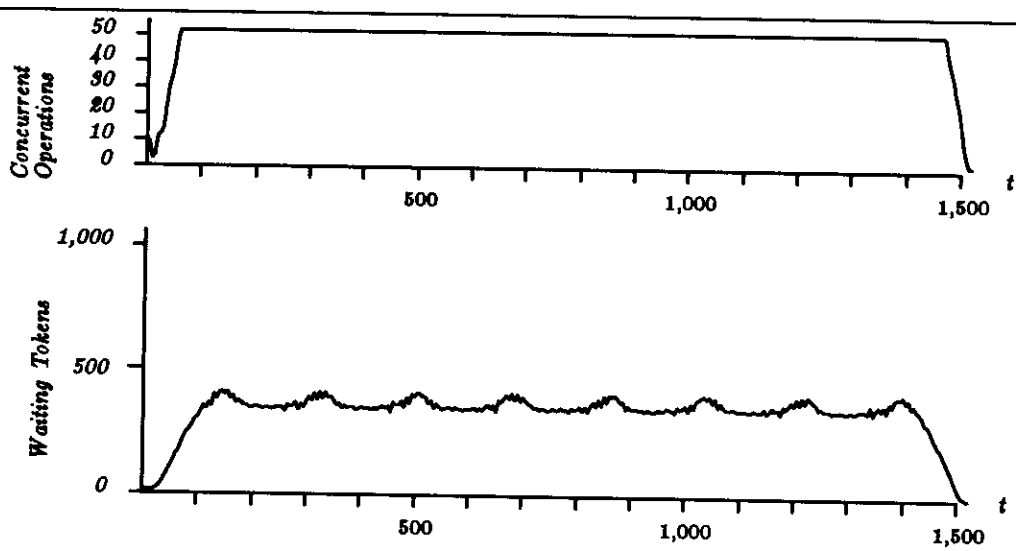


Figure 10: ($p = 50, l = 0$) Profiles for 16x16 Matrix Multiply with Middle Loop Bound to two

kind of unfolding would expose essential parallelism, but here it serves only to increase the resource requirements.

Bounding the outer loop of this application eliminates this problem, giving the ideal parallelism and resource profiles shown in Figure 11. The waiting token requirement is now independent of the total number of iterations and proportional to the amount of exposed parallelism. The critical path is only increased by 10% even under the ideal model. For finite processors the difference is even less, and loops can be further constrained to reduce excess parallelism. Figure 12 shows the waiting token profiles with the outer loop unbound and bound to 1 under finite processor execution with 32 operations per step and communication latency of 10 units.

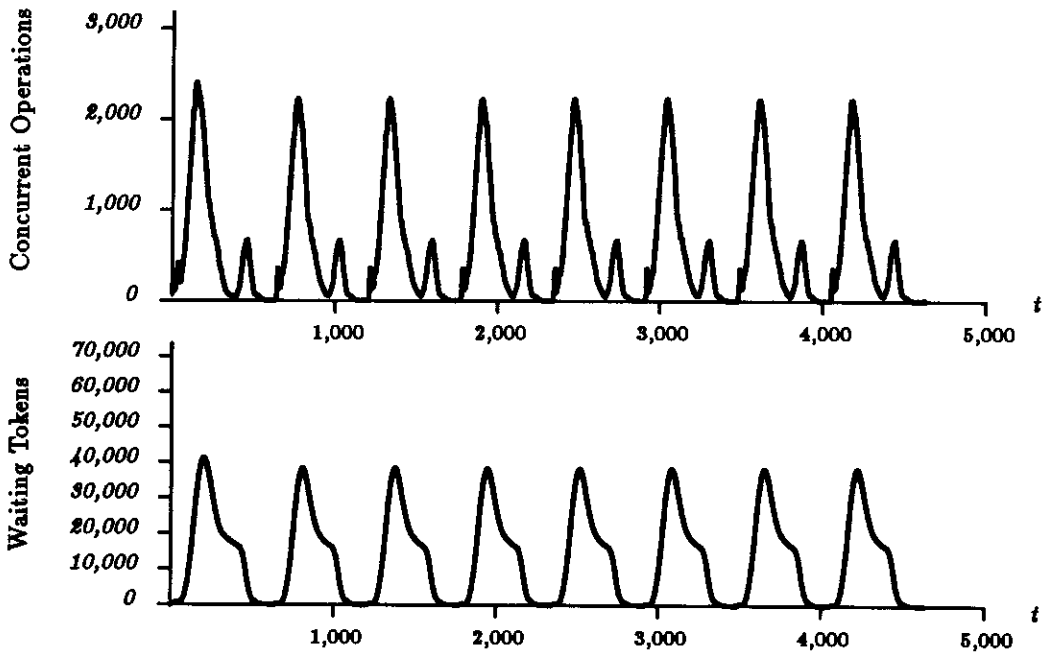


Figure 11: ($p = \infty, l = 0$) Profiles of Simple, 8 iterations on 16x16 mesh, with the Outer Loop bound to one

4.4 Inherently bounded loops

In many cases the dependencies within the application create inherently bounded loops. The Gaussian Relaxation example appearing below illustrates this situation as indicated by the profiles in Figure 5. The relaxation is performed until a given convergence criterion is met. We have used the maximum change of any point in the grid relative to the previous iteration; thus, the predicate of the loops is dependent on every point in the grid. There may be a slight overlap between iterations because of termination detection, but the useful work of two iterations does not overlap. It is not surprising that the waiting token profile is periodic and the storage requirement is independent of the number of iterations. Detecting this situation is important, because it ensures that a small unfolding parameter will not compromise the performance of such a program, thereby eliminating one variable in the resource management policy.

```
def 2D_relax A steps epsilon =
```

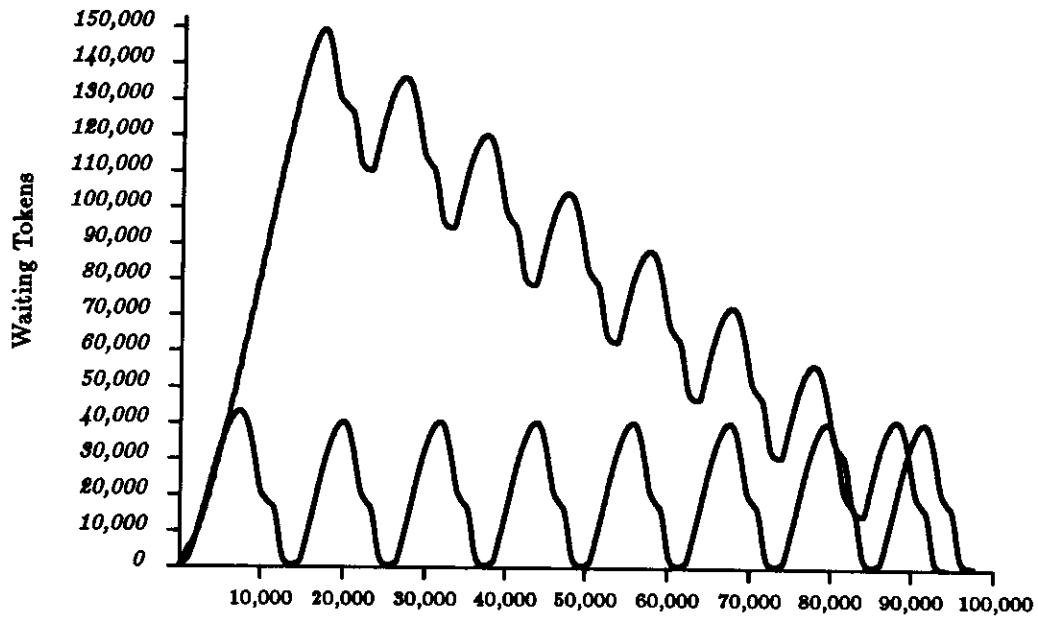


Figure 12: ($p = 32, l = 10$) Token Profiles of Simple, 8 iterations on 16x16 mesh, with the Outer Loop unbound and bound to one

```
{delta = 2*epsilon
 in {while delta > epsilon do
       next A = make_matrix (bounds A) (nine_point_stencil A);
       next delta = max_pointwise_difference A (next A);
     finally A}};
```

This example also shows how subtle differences in program structure may have dramatic effects on program behavior. If the result of the maximization were used in computing the new grid but did not appear in the predicate, the program would be ill-behaved in the same way as Simple. If this result were not used in computing the new grid, the behavior would be more like matrix multiply; all three loops would unfold, generating a wave-front of parallel activity. However, if we wished to extract more parallelism than indicated by Figure 5, while still testing for convergence, it would be necessary change the program, allowing a number of iterations between convergence tests.

5 Broader Resource Management Concerns

5.1 Recycling resources

The loop bounding technique has been introduced as a means of controlling program unfolding to reduce token storage requirements, but properly generalized it may serve a broader role. In embellishing the graph to control unfolding, we begin to build resource management into the graph itself. It can be extremely effective, because it is integrated with the very structure of the program.

Each iteration of a loop requires certain resources; the trigger token fed into the gate is effectively a ticket for those resources. A block of resources is granted to the entire loop invocation and managed locally by simply recycling them in the graph.

Token storage is one such resource, but another is tag space. The tag carried on a token includes an iteration identifier so that tokens belonging to different iterations do not interact. In abstract formulations of dataflow such as the U-interpreter[6], management of tags is completely localized, but the tag space is arbitrarily large and very sparsely used. Any realistic implementation of this model must restrict that tag to some fixed size. With bounded loops, iteration identifiers are represented modulo the bounding parameter, so by picking a maximum loop unfolding tags can be of fixed size. By building resource management into the graph, localized management is regained, while using the tag space densely.

Loop bounding could play an important role in existing dataflow machine such as the NEC IPP. That machine supports a limited FIFO dataflow model[2], where at most 16 tokens can queue on an arc. Graphs are carefully hand tuned so that this limit is not exceeded. Bounded loops provide a basis for systematic code generation, making implementations of high level languages possible.

The notion that each iteration has a certain amount of resources in which to work suggests a novel approach to implementing token storage. Rather than perform an associative match or hashing to determine if a partner token is present, assign a block of token storage to each iteration. Tokens can be assigned to slots in the block at compile-time by coloring the graph so that no two tokens which could potentially coexist fall in the same slot. A lower bound on the number of slots can be found via linear programming, and actually performing the assignment is similar to sophisticated register allocation schemes.[8] Setting up a loop requires allocating as many such blocks as the loop bound permits. Determining if a partner is present only requires checking a presence bit.

We have focused primarily on token storage resources, but loop bounding plays an essential role in managing I-structure storage as well. Suppose that implicit storage reclamation were implemented by whatever means: mark-and-sweep, reference-counting, volatility regions, or any other. An I-structure can be reclaimed only when no references to the structure exist. Consider the Simple program discussed above. In each iteration, thirteen meshes are allocated and filled using the set of meshes of the previous iteration. Without loop bounding, the iterations unfold so that all the versions of the mesh are allocated, and references exist for each. Only as data moves through the sequence of meshes will any structures be reclaimed. Thus, the I-structure storage profile will have the same shape as the waiting-token profile in Figure 4. No reasonable machine could support 100,000 versions of the mesh at one time. With loop bounding, old versions of the mesh are reclaimed as new one are allocated.

This idea can be carried further to recycle I-structures within the program graph itself. Many scientific applications successively transform a large, regular data structure to model behavior over time or to reach some termination criteria. Traditional methods update the structure in place, but this can be tricky, especially under parallel execution. The Id style is to allocate a new structure during each iteration and to fill it in. All of the examples presented here demonstrate this strategy to some degree. With bounded loops the lifetime of such structures is clearly defined, so they could be reset and reused. In effect, this provides multiple buffering within parts of the program, without major programmer effort.

5.2 Controlling unfolding in general

This loop bounding approach to controlling parallelism is applicable to a restricted class of programs. For general recursive programs there is need of similar techniques. The general approach of introducing auxiliary arcs to enforce artificial dependencies still applies, but it is less efficient to implement and less clear how to apply. Researchers at Manchester University have suggested that rather than introduce artificial dependencies in the graph, the effect can be achieved by deferring context allocations when the machine appears busy.[16] Unfortunately, deferring a context allocation does not inhibit the requester from making additional requests. Simple provides a good example. Each iteration invokes numerous code-blocks to perform various parts of the two phases of the computation. Assuming all of these were deferred, the outer loop would completely unfold, making a huge number of requests. Eventually, some deferred request will be serviced and a subordinate code-block invoked, but unfortunately almost all the deferred requests would be useless to service. Only those from the first iteration, and only the hydrodynamics portion of the first iteration will result in tangible progress. Deferring requests can actually shift the resource requirements from that suggested by fair scheduling to worst-case. Certainly if code-blocks and data-structures are strict this kind of approach would be much more effective, but much of the potential parallelism in programs would be lost. Combining such heuristics with loop bounding may bring us a long way toward automatic control of parallelism of dataflow programs in general.

6 Conclusion

We have shown that parallel execution generally requires more resources than sequential execution, and complicates resource management. Dynamic generation of parallel tasks can easily "run away", generating inordinate resource demands. Some mechanism for controlling parallelism is necessary, so that enough is exposed to saturate the machine, while minimizing resource demands. We have presented one mechanism in the context of dataflow models, which allows resources to be recycled within the program graph. Given such a mechanism, the open question is what policy would be effective over a broad class of programs. Compiler analysis is part of the answer, as strictness analysis is essential to detecting useless parallelism in cases like Simple, and run-time heuristics based on availability of resources hold potential. A planning strategy based on building *resource expressions* for portions of programs has been proposed[3]. All three approaches are still subjects of study.[9]

References

[1] Arvind, Stephen A. Brobst, and Gino K. Maa. *Evaluation of the MIT Tagged-Token Datflow Architecture.* Technical Report Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[2] Arvind and David E. Culler. *Dataflow Architectures,* pages 225–253. Volume 1, Annual Reviews Inc., Palo Alto, CA, 1986. Reprinted in Dataflow and Reduction Architectures, S. S. Thakkar, IEEE Computer Society Press, 1987.

[3] Arvind and David E. Culler. Managing Resources in a Parallel Machine. In *Proceedings of IFIP TC-10 Working Conference on Fifth Generation Computer Architecture, Manchester, England,* North-Holland Publishing Company, July 15-18 1985.

[4] Arvind, David E. Culler, and Gino K. Maa. *Parallelism in Dataflow Programs.* Technical Report Computation Structures Group Memo 279, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

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

[6] Arvind and K. P. Gostelow. The U-Interpreter. *COMPUTER,* 15(2), Feburary 1982.

[7] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259),* Springer-Verlag, June 15-19 1987.

[8] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. *Register Allocation via Coloring.* Technical Report RC 8395, IBM T.J. Watson Research Center, August 1980.

[9] David E. Culler. *Effective Dataflow Execution of Scientific Programs.* PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, June 1988 (expected).

[10] David E. Culler. *Resource Management for the Tagged-Token Dataflow Architecture.* Technical Report TR-332, Massachusetts Institute for Technology, Laboratory for Computer Science, January 1985.

[11] Kattamuri Ekanadham, Arvind, and David E. Culler. *The Price of Parallelism.* Technical Report Computation Structures Group Memo 278, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, December 1987. (Submitted to the Fifteenth Annual International Symposium on Computer Architecture, Honolulu, Hawaii, May 1988.

[12] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Coordinating Large Numbers of Processors. In *ACM Transactions on Programming Languages and Systems,* January 1982.

[13] Robert H. Jr. Halstead. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the ACM Symposium on Lisp and Functional Languages*, August 1984.

[14] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and Wolfe M. Dependence Graphs and Compiler Optimizations. In *Proceeding of ACM Symposium on Principles of Programming Languages*, January 1981.

[15] G. F. Pfister and et al. The IBM Research Parallel Processor Prototype (RP3). In *Proceedings of the 1985 ICPP*, August 1985.

[16] C. A. Ruggiero. *Throttle Mechanisms for the Manchester Dataflow Machine.* PhD thesis, University of Manchester, Manchester M13 9PL, England, July 1987.

[17] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982.