The Price of Parallelism[†]

Computation Structures Group Memo 278

December 15, 1987

Kattamuri Ekanadham
IBM T.J.Watson Research Center
Hawthorne, New York

Arvind
David E. Culler
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

[†]Paper one in a sequence of four, including CSG Memos 279, 280, 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.

Abstract

Dataflow models of computation have been proposed as an elegant approach to parallel computation, but have been criticised as *inefficient* compared to conventional approaches. In this paper we examine the additional computing resources demanded in dataflow models of computation. In our view, a program specifies essential arithmetic and logical operations and additional operations to affect control, addressability, synchronization, etc. By isolating this latter group of overhead operations we can identify what fraction of it is inherent to parallel computing and what is an artifact of the dataflow approach. In this paper we present measurements on sample programs written in a declarative language (Id) and executed on dataflow machine (MIT Tagged-token Dataflow Architecture) and programs written in an imperative language (Fortran) and executed on a sequential machine IBM 370. This means that two factors, language and architecture, come into play and the combined effects must be examined. The metric used for comparisons is the number of instructions executed. While we recognize this is an imperfect measure, it is good to first order and can be measured precisely. Biases it introduces are discussed. To articulate the nature of the overhead we develop a structural model of program execution and verify intruction counts it predicts with those observed on benchmark programs. Finally, we present preliminary data on the overhead introduced in parallelized Fortran programs.

The Price of Parallelism

1 Introduction

Dataflow models of computation have been proposed for faster execution of programs using many processors, but they have also been viewed by critics as *inefficient* compared to conventional approaches. Although it is hard to make concrete comparisons of such diverse approaches, often the inefficiency is attributed to large quantities of memory and processing resources demanded by dataflow models. Indeed, parallel computation *requires* additional memory in order for processors to access private copies of values concurrently and execution of additional instructions for data communication and synchronization among concurrent computations. Since dataflow models attempt to execute all possible subcomputations (down to a single instruction) in parallel, the amount of synchronization, communication and memory requirements may conceivably be large. Furthermore, it is generally believed that an unrealistically large amount of parallelism is needed to keep a significant number of processors busy. However, our experiments demonstrate the contrary — the amount of parallelism found in test programs is so large that provisions must be made to constrain the parallelism to match finite resources. This subject is dealt with in [1]. A detailed study of the memory requirements is presented in [8]. In this paper we examine the additional computing resources demanded in dataflow models of computation.

In our view, a program specifies essential arithmetic and logical operations and additional operations to affect control, addressability, synchronization, *etc.* By isolating this latter group of overhead operations we can begin to understand what fraction of it is inherent to parallel computing and what is an artifact of the dataflow approach. Currently, we can perform high-quality measurements on programs written in a declarative language (Id) and executed on a dataflow machine (MIT Tagged-token Dataflow Architecture), and programs written in an imperative language (Fortran) and executed on a sequential machine (IBM 370). As performance is the issue, we insist that sophisticated optimizing be employed and non-trivial programs be considered. Unfortunately, this means that two factors, language and architecture, come into play and the combined effects must be examined. We comment on some of the effects introduced by differences in declarative and imperative languages and confine our investigation to programs expressed easily in either and with similar control structures. Preliminary measurements of "parallelized" Fortran programs on a collection of sequential machines is offered as well.

The metric used for comparisons is the number of instructions executed. While we recognize this is an imperfect measure, it is good to first order and can be measured precisely. In some cases redundant computation can be introduced to avoid synchronization and communication or to reduce the critical path in a parallel computation, blurring the distinction between essential operations and overhead. We do not consider such cases here. Certainly instruction set complexity cannot be ignored. We argue that TTDA instructions are roughly comparable to those of load/store architectures and that comparisons with IBM 370 instruction counts should be viewed in light of the ratio of instruction counts on a RISC and CISC. Although TTDA instructions may be less powerful than IBM 370 instructions, the cost in terms of hardware complexity may be greater because of dynamic instruction scheduling. On the other hand, this manner of instruction scheduling provides tolerance to memory latency by interleaving arbitrary threads of computation, and thereby may mask the execution of a portion of the overhead instructions. Further, we ignore the overheads due to dynamic resource management, which is essential in the TTDA environment; the interface to the routines that make such decisions is accounted for, but not for their algorithms. While proper treatment of the myriad of lesser factors may bias the outcome slightly, articulating the overhead in terms of the number of additional instructions executed is a valuable starting point in understanding the price of parallelism.

In order to identify the sources of the overhead operations, we develop a structural model of the cost of program execution. In both dataflow and sequential machines there is a certain cost associated with iteration, procedure calls, conditionals, structure accesses, and so on. We assign a weight to each class and measure the number of times each occurs in the execution of a program. Taking the weighted sum, together with the essential operations, we should arrive at the total number of instructions executed, if we have properly accounted for the cost of the events in the various classes. We stress that the comparison is made between similar programs in the two settings, not arbitrary programs solving a given problem.

We assume that the reader is generally familiar with dataflow model of computation and in particular with the Tagged Token dataflow Architecture (TTDA) which is described in [6] and its execution of programs written in the declarative language, Id which is described in [15, 4]. The language is functional, except for the introduction of I-structures, which is a facility for asynchronous access to structures [5]. I-structures have built-in synchronization on an element by element basis. A read from an element is delayed until a value is written into it. Multiple writes to the same element result in run time error. The language supports higher order functions and currying. The Id compiler [19] translates Id programs into machine code for TTDA. The machine code is one particular implementation of abstract dataflow graphs described in [3] and is executed by an interpreter called, GITA [15]. GITA also produces execution statistics such as the frequency of instructions executed, time profiles of instructions and various queues, *etc.* The execution of a dataflow graph in TTDA is data driven. An operator is fired when all its operands are ready. Loops and functions are instantiated as soon as they are invoked and they can start execution even before their operands are ready. Multiple executions of the same operators are distinguished by distinctly coloring each set of corresponding data values.

The rest of the paper is organized as follows: In Section 2 we present the raw data of instruction counts collected from a sample of six programs on both TTDA and IBM 370. This data is interpreted and explained further in the rest of the paper. In Section 3, we give an overview of the instruction set and instruction processing in a dataflow machine and contrast it with a conventional machine. We argue that the instruction-set of TTDA is roughly comparable to load/store architecture; hence, we can view the number of instructions executed as a reasonable measure of work. Section 4 highlights the effects due to differences in the two languages and the optimizations performed by their compilers. We argue that, with certain precautions the two are fairly even on these issues. Section 5 presents a classification of the structural features of a program execution and illustrates how instruction overheads can be related to the classes. These are illustrated through the example of matrix multiplication. Section 6 presents measurements of dynamic instruction counts on our benchmark program, called Simple, in Fortran and Id, where reasonably sophisticated compiler optimizations have been applied in both cases, to show how the approaches compare in real terms. In Section 7 we examine similar overheads incurred by a conventional machine in a parallel environment. We exemplify this using the EPEX simulation of the RP3 multi-processor system. Finally, Section 8 gives some concluding remarks.

2 Dynamic Instruction Counts: The raw data

We begin by presenting the dynamic instruction counts observed for six programs on TTDA and on IBM 370. Fortran programs are coded in Fortran-77 and are compiled using the IBM VS FORTRAN compiler version 1.4.1, with optimization level 3 (highest available) and are run on an IBM 370/3081 processor. Instructions are traced using the PER execution trace facility available on the IBM 370 machines and are classified based on opcode and instruction characteristics. Equivalent programs are coded in Id and are compiled using Id Compiler version 2.1, producing code for TTDA. The programs are run under GITA version 2.1, on the lisp machine, TI Explorer I, version 2.1. GITA execution assumes infinite resources — that is, all instructions that are enabled are executed in each step. Actually this has no implication on this study, as the number of operations is not affected by the number of processors. First we give a brief description of thr programs and then present their instruction counts.

2.1 Program descriptions

Figure 1 shows 5 of the Id programs used in this study. They are self-explanatory. Program *ip* computes the inner product of two vectors of size n. Program *vsum* returns the vector sum of two vectors of size n. The program *ip_sum* illustrates the invocations of these two functions. It doubles each given vector by invoking *vsum* and returns the inner product of the result vectors. The program *matmult* multiplies two matrices using the standard 3 nested loops. We use this program with $l = m = n$ and in all cases, n is referred to as the problem size.

The program *decompose* is used in solving simultaneous linear equations. It decomposes a given matrix A by Gaussian elimination with partial pivoting, so that $A = LU$, where L is a lower triangular matrix and U is an upper triangular matrix. It uses other functions *find_pivot, make_matrix, fill_matrix* which are not shown in Figure 1. They do the obvious things suggested by their names and further details on this may be found in [2]. The parallelization of a Fortran version of this program is discussed in [12]. This program is chosen to illustrate the problem of structure copying. The Fortran version of the program obtains the L and U matrices by updating A in place. In the Id version shown in Figure 1, the final matrix D is allocated at the beginning and it is filled in n iterations. The i-th iteration fills portions of the i-th column and i-th row. Each time a new right sub-matrix is created to remember the new values. Thus, the solution avoids structure copying. We use this Id program, after substituting for the functions it invokes. We use an $n \times n$ matrix in this problem and n is the size of the problem.

Finally we present instruction counts on a large benchmark program called Simple. This program computes numerical approximations to the physical properties of a fluid under compression [7]. The program simulates a cross section of the fluid, which is treated as a rectangular grid, so that the four corners of each fluid parcel are four neighboring grid points. A program parameter, n, can be set to choose the size $n \times n$ of the grid and hence the total number of fluid parcels being simulated. The program consists of an initialization phase followed by a loop. Each iteration of the loop advances the time by some computed value and computes the properties for all the fluid parcels in the grid for that time step. The solution adopted a Lagrangian Formulation and the motion of the fluid parcels in the grid is tracked by computing the coordinates of each parcel as time progressed. In each iteration, new values of position, volume, velocity, pressure, energy, artificial viscosity and temperature of each fluid parcel are computed. To enforce proper boundary conditions, the computations for the $n-1 \times n-1$ interior fluid parcels is done differently from those

```
Def ip A B n =
   { s = 0;
   In {For i From 1 To n Do
         Next s = s + A[i] * B[i];
       Finally s }};
```

```
Def vsum A B n =
   { C = array (1,n);
     {For i From 1 To n Do
         C[i] = A[i] + B[i] };
     In C};
```

```
Def ip_vsum A B n =
  ip (vsum A A n) (vsum B B n)
```

```
Def decompose A n =
   { D = matrix ((1,n),(1,n));
     B = {For k From 1 To n-1 Do
              r = find_pivot A;
              Call fill_matrix D ((k,k),(k,n)) row_fill;
              row_fill (i,j) = A[r,j];
              Call fill_matrix D ((k+1,n),(k,k)) col_fill;
              col_fill (i,j) = -A[i,k] / A[r,k];
              Next A = make_matrix
                     ((k+1,n),(k+1,n)) new_elem;
              new_elem (i,j) = D[i,k] * D[k,j] +
                     A[ (If i==r Then k Else i), j];
           Finally A};
     D[n,n] = B[n,n];
   In D};
```

```
Def matmult A B l m n =
   { C = matrix ((1,l),(1,n));
     {For i From 1 To l Do
         {For j From 1 To n Do
             s = 0;
             C[i,j] = {For k From 1 To m Do
                           Next s = s + A[i,k] * B[k,j];
                        Finally s}}};
   In C};
```

Figure 1: Programs in Id

around the boundary. Computations of successive iterations have very little overlap, as the value of the new time step computed at the end of an iteration must be used throughout the next iteration. The counts reported in Table 1 are for an $n \times n$ grid and for one iteration. In this experiment we used a version of the Simple program obtained from Los Alamos National Laboratory[1]. Using this program and the documentation [7], an equivalent program is written in Id. The latter program is encoded using higher level abstractions as explained in [13]. The Fortran version has roughly 1200 lines and the Id version has 680 lines.

2.2 Data Collected

Table 1 gives the instruction counts for the programs described above. Instruction counts shown do not include initialization, input-output etc. For each program and size, the Instruction counts for Fortran program are given with the corresponding Id program counts shown in italics on the side. Instruction counts are grouped into 6 major categories. *Float* and *fixed* are the arithmetic and logical operations performed in the floating and fixed point units respectively. The *Load* and *Store* categories for IBM 370 report all move operations involving memory, but do not perform any arithmetic or logic. Thus, an integer-add operation with a memory operand is counted under the *Fixed* category. (An equivalent operation in a load/store architecture would take two operations - a

[1]There are many versions of the Simple program and they vary substantially in their computational properties. The exact version used in this experiment may be obtained from the authors.

load and an add.) For TTDA, the *Load* and *Store* are respectively the I-fetch and I-store operations used to access the I-structure memory. The *Switch* category gives the number of *branch* instructions for the IBM 370 and the number of *switch* instructions in TTDA. The *Other* category gives the count of all remaining operations. For IBM 370, this category consists of only operations that move data between registers. For TTDA, this includes operations for the manipulation of tags, contexts and resources. The *Total* category gives the sum of all the above mentioned categories. Finally, *Ratio* shows the ratio of the total operations for Id to that of the corresponding Fortran program. We recognize that the problem sizes depicted in Table 1 are not representative of realistic programs. Large sizes may affect the relative importance of the overheads compared to the work being done, but they do not affect the characterization that we will present in Section 5.

2.3 Observations

In Table 1, the floating point operations for Fortran and Id versions are about the same, allowing for minor variations in the Simple program, which are explained in detail later in Section 6. Since most of the primary calculations involve floating point numbers, this confirms that the programs are doing roughly the same work. In the remaining categories, the differences are significant. Notably the total number of operations for the Id version is 1.4 to 3 times that of the corresponding Fortran version. There could be many reasons for this: differences in program characteristics due to different coding styles in the two languages, differences in the compiled code due to different levels of optimizations of the two compilers, differences due to architectural considerations - one is a von Neumann machine with complex instruction set and the other is a dataflow machine with a much simpler instruction set, and finally differences due to the fact that one is executed sequentially and the other is executed on a parallel machine. In the sequel we analyze these differences.

3 Architectural Issues

Comparison of instruction counts is meaningful when we understand the relative complexity of the instruction sets used, in relation to each other. Dataflow instructions, in general, have the complexity of the instructions for load/store architectures like CRAY-1 (also see [11]). In this section, we briefly describe the nature of the instruction set of TTDA and comment on some of the issues of complexity.

3.1 Instruction processing

The principal components of a TTDA processor are: wait-match unit, program/constant memory, I-structure unit and an ALU consisting of a tag-manipulation unit, a fixed point unit and a floating point unit. These are organized as a circular pipeline around which data tokens flow. The processing of an instruction roughly corresponds to the flow of a token through this pipe once. Tokens may leave the pipe to go to other processors and tokens from other processors may enter this pipe. Instructions and constants are stored in the program/constant memory. An instruction can have at most two operands (besides a constant in the constant memory) and it specifies an opcode and destination addresses. An instruction execution is triggered when all its operands are available.

The instruction processing cycle of a dataflow machine is different than a conventional machine as follows. A data item presented to the wait-match stage is accompanied by a tag that identifies the instruction for which it is an operand. The wait-match unit uses this tag to locate the other operand

	inner product				vector sum				sum of products			
	size=10		size=64		size=10		size=64		size=10		size=64	
	Fort	*Id*	Fort	*Id*	Fort	*Id*	Fort	*Id*	Fort	*Id*	Fort	*Id*
float	20	*20*	128	*128*	10	*10*	64	*64*	40	*40*	256	*256*
fixed	7	*21*	7	*129*	8	*21*	8	*129*	30	*63*	30	*387*
load	32	*20*	86	*128*	35	*20*	89	*128*	124	*60*	286	*384*
store	28	*0*	82	*0*	30	*10*	84	*64*	111	*20*	273	*128*
switch	16	*130*	70	*130*	16	*23*	70	*130*	56	*66*	218	*390*
other	1	*340*	1	*340*	1	*74*	1	*345*	4	*261*	4	*1071*
total	104	*153*	374	*855*	100	*158*	316	*860*	365	*510*	1067	*2614*
ratio		*1.47*		*2.29*		*1.58*		*2.72*		*1.40*		*2.45*

	matrix multiply				LU decomposition			
	size=10		size=16		size=10		size=16	
	Fort	*Id*	Fort	*Id*	Fort	*Id*	Fort	*Id*
float	2000	*2000*	8192	*8192*	669	*669*	2735	*2735*
fixed	1708	*2352*	5794	*9042*	901	*1094*	2618	*3802*
load	1813	*3027*	6055	*12327*	1273	*1149*	3506	*3983*
store	474	*116*	1122	*278*	827	*445*	2445	*1592*
switch	226	*2464*	4646	*9316*	615	*1374*	2056	*4493*
other	171	*10215*	363	*33189*	230	*7089*	516	*18947*
total	7392	*20174*	26172	*72344*	4515	*11820*	13876	*35552*
ratio		*2.73*		*2.76*		*2.62*		*2.56*

	simple			
	size=10		size=16	
	Fort	*Id*	Fort	*Id*
float	28650	*30297*	82848	*87261*
fixed	14787	*10972*	41803	*29560*
load	31523	*20054*	84860	*55226*
store	18795	*3692*	49443	*9056*
switch	11687	*16992*	32801	*47244*
other	2675	*114841*	6747	*297253*
total	108117	*196848*	298502	*525600*
ratio		*1.82*		*1.76*

Table 1: Summary counts for various examples

for the specified instruction. If present, the instruction can be processed, otherwise the operand is placed in the wait-match store to await its partner. The tag provides the address for instruction fetch. Then, just as in a conventional pipelined machine, the operands and the operation code can be delivered to the ALU pipeline for processing. The result is combined with successor instruction addresses specified in the instruction to yield <tag, data> pairs for the wait-match stage. Thus, if the wait-match store is viewed as a large register set, the instruction processing cycle is similar to current pipelined machines except the operands specify the address of the instruction, rather than the reverse. The wait-match stage of TTDA performs roughly the role of result store, instruction counter update, and operand load in a conventional machine. It is clear that this stage of instruction processing is fairly complex[2], but also that a complete dataflow instruction accomplishes about the same amount as an instruction on a load/store architecture. The distinction gets blurred when we look at the complex pipelines in some of the very high performance machines.

Data structures are stored in the I-structure store. Data is moved between the I-structures and the processor pipeline by explicit I-fetch and I-store instructions, similar to the load and store instructions in a load/store architecture. Thus, the register set in a conventional machine corresponds to the set of tokens that circulate around the processor pipeline including the tokens stored in the wait-match unit and the set of constant values stored in the constant area set up for each context. All ALU operations must have their operands ready in the processor (tokens in TTDA and registers in load/store architectures) and no operands can be in memory.

3.2 Instruction set

The following classes of instructions are included in TTDA.

1.Arithmetic and Logical instructions: TTDA has the usual repertoire of arithmetic and logical instructions such as `+,-,*,/,^,<,<=,>,>=,=,<>,and, or, not`. Each of these instructions is as complex as a corresponding register-to-register operation in a conventional machine. The two operands are available in the input tokens and the result goes directly into the output token. The data tokens carry types with them and hence there is some logic to examine the type of operands and direct the data to the appropriate functional unit (eg. fixed or floating point). At the present time, type conversion is done on the fly. However, in future, we expect the compiler to deduce the types and insert explicit code for type conversions.

2.Structure-handling instructions: An array is contiguous storage in the I-structure store and tokens carry a descriptor for it, specifying its beginning address and size. Address computation for elements is done by a separate instruction, *I-fetch*, which takes a descriptor and an index, computes the address of an element by adding the offset and sends the result to the I-structure store. The token sent to the I-structure also contains the return address. Actual reading of the element takes place asynchronously in the I-structure memory, and the result is sent to the return address. The ALU proceeds unblocked, while the read is in progress. For writing into an element, its address must be computed first using a *form-address* instruction. An *I-store* instruction takes a computed address and data and sends it to the I-structure memory. At the present time, multiple dimensioned structures are implemented as arrays of arrays. Thus, for example a matrix is an array of rows. To read the matrix element $[i, j]$, the machine executes two *I-fetch* operations as in *I-fetch(I-fetch(A,i), j)* .

[2]It is complex on a conventional pipelined machine as well when register reservations and pipeline scheduling are accounted for.

3.Resource-related instructions: Instructions *get-context, release-context, make-I-structure, etc.,* take a data value and a manager address and simply send the data to the specified manager. The structure manager allocates I-structure memory and returns the address. The context manager allocates a context on a processor, initializes it and returns the new context address.

4.Tag manipulation instructions: A tag-manipulation instruction takes a token and produces a token with an altered tag. The complexity of any of these instructions is comparable to a register-to-register arithmetic/logical operation. Instructions D-N, D-1, D-N+-1 and D-N+-2 are used to manipulate the tag so that the data values can be passed between iterations of a loop. Instructions *form-tag, change-tag* and *adjust-offset* are used to pass arguments and return results across function boundaries.

5.Context-related instructions: These are instructions that perform miscellaneous operations related to the context of a function. Instruction *load-constant* is used for setting up constants as explained later in Section 3.3. Instructions *closure-ready, closure-arity, closure-cbname, closure-chain* and *closure-ncdr* are used for implementing Currying as illustrated later in Section 4.1. Instructions *loop-req* and *loop-rel* are some artifacts instituted for bounding the number of iterations simultaneously active in a loop. Details of these instructions are not important for the purposes of this paper. Each of them is no more complex than a register-to-register add operation.

6.Switch instruction: *Switch* is used for conditional expressions. It takes two inputs, a data item and a boolean value, has two destination lists, and sends the data item to one or the other of the destination lists, based upon the boolean value. If the chosen destination list is empty, the data item is simply discarded and a signal is produced for termination detection as explained later in 5.5.

7.The identity instruction: This distinguished No-op instruction is like a gate that takes two inputs, one of which is a trigger to let the other input proceed. All synchronization operations are encoded explicitly using this instruction.

3.3 Instruction complexity

In the repertoire of instructions discussed above, almost all of them fit the category of a one cycle register-to-register style instruction of a conventional load/store architecture. There are three exceptions to this, which we list below.

1.Multiple destinations: When an instruction has more than two destination nodes, we consider that instruction as complex because many more cycles are needed to implement it. Strictly speaking for each pair of additional destinations an extra *identity* instruction ought to be used to perform the forking. However, in this study we ignore this and permit an instruction to have an arbitrary number of destinations. It should however be noted that the number of destinations is a compile time constant and hence we do not permit complex instructions, such as the *Proliferate* instruction in the Machester machine [10], that receives a value n at run time and produces that many copies of the token.

2.Bounds checking: In TTDA both address addition and bounds checking are performed in one instruction (see instructions *Form-address* and *I-fetch* in Section 3.2). We consider this complex, and feel that there must be explicit instructions for bounds checking, just as load/store architectures will have. We ignore it here. The Fortran implementation on IBM 370 we are studying does not implement bounds checking and hence this is inconsequential as far as this study is concerned.

3.Loading Constants: As part of the instantiation of a function or a loop, constant values for that program must be stored in the program/constant memory. This is done using the *load-*

constant instruction which takes the data value and an offset into the constant store. Often an instantiation may span many processors. For instance, when iterations of a loop is distributed over many processors, loop constants must be stored in all the processors. At the present time, this is treated as one instruction. In reality, we must treat loading of a constant in each processor as a separate instruction.

3.4 Termination Detection

One of the quintessential characteristics of parallel execution is the need to determine when a computation or subcomputation has completed its execution. The importance of this issue may not come across naturally when one reads about a parallel architecture and we would like to emphasize this by explaining a little bit of detail about it here.

Termination is *not* the same as result being produced. For example, consider the function in Id: `f A x = {A[x] = sqrt x; In x+1}`. When we invoke f, the result value `x+1` may be returned very quickly, whereas the square root computation and storing into the array may take some time. Thus, the resources used for the instantiation of `f`, such as the program and constant memory, the unique tags used for the tokens in its computation *etc.*, cannot be reused until `f` terminates. The need for detection of termination separately arises when we have all the following three features: (1)programs have *side-effects*, (2)programs are executed in *parallel* and (3)resources are finite and must be reused. Pure functional languages do not have this problem as they do not have the first feature. Sequential computations do not have this problem as they do not have the second feature. Id as well as other conventional parallel processing systems possess all three features and must worry about termination detection. The effort involved for this is proportional to the size of the granularity for parallelism. Since TTDA makes use of parallelism even at single instruction level, termination detection is a concern down to an instruction level.

A computation terminates when it returns all the results and when all its side-effects take place. In TTDA a loop terminates when all its iterations terminate; an iteration terminates when all its instructions terminate. Since the body of an iteration or a function is an acyclic graph, and the termination of a node implies the termination of all its predecessor nodes, it is sufficient to detect the termination of *sink* nodes in the graph. That is why each *sink* node, such as *I-store* is required to send a signal back after its completion. All these signals are combined using an inverted tree of *identity* instructions. It turns out that the number of *identity* instructions is significant in TTDA execution. Although *identity* instructions are used for other purposes as well (such as distributing arguments to a function), the synchronization trees described above dominate their use. For example, in Table 1, the Simple program for 16×16 executes a total of 133,619 *identity* operations which is about 25% of the total number of operations. Therefore, throughout the rest of the paper we account for these *identity* instructions separately, wherever the need for termination detection arises.

3.5 Architecture Summary

In summary, the instructions for arithmetic and logic operations specified by a program will be comparable for executions of equivalent programs on TTDA and on a load/store architecture. A load/store architecture will have additional arithmetic/logical operations for computation of indices of multi-dimensional structures and for local memory address computations. These will appear as additional memory accesses in TTDA execution. Movement instructions for finite register management in a conventional machine have no counterpart in TTDA, as the equivalent of register set (*i.e.*, wait-match unit and buffers in the pipeline) is virtually infinite. Operations corresponding to structure accesses specified in the program (subject to similar optimizations) should correspond to each other in TTDA and load/store architectures. The picture will of course be distorted for IBM 370, as many accesses may be coupled with arithmetic using memory operands. Operations to load and save registers for setting and restoring contexts on procedure boundaries bears partial resemblance to setting up constants in TTDA before a function is invoked and detecting the termination of a function. However, the correspondence is fuzzy in this regard. Finally in the category of branches, subroutine jumps/returns correspond to change-tag instructions in TTDA and conditional jumps correspond to switch instructions. However, as explained later in Section 5, there is a change-tag instruction for every argument/result value in TTDA as opposed to a single subroutine jump in von Neumann machine. Similarly, there is a switch for each value passed to/from a conditional as opposed to a single branch per condition in a von Neumann machine. Thus, for comparable executions, TTDA is likely to have more of these instructions.

4 Language and Compiler Issues

In this section we examine the implications of using the declarative style of programming. We elaborate certain effects of this style and argue that we have taken great care to minimize the effect of language differences in our study. Declarative languages, in general, are known to face three principal problems: (1)inefficient code because of higher levels of programming abstractions are advocated, (2)excessive copying of structures in certain situations, because of their single assignment restriction (which is vital for the resulting syntactic as well as semantic cleanliness of the language) and (3)the need for special optimizations in view of the above two. Below, we address each of these issues separately.

4.1 Coding style

A commonly held view is that programming in a functional language with parallel semantics complicates the programming task, because a programmer must think of a parallel model in mind. In addition, the single assignment restriction constrains the view taken by a programmer familiar with imperative style. However, we believe that for the class of problems we are studying the coding style is natural. Use of function definitions that reflect higher levels of abstractions that are closer to the problem domain greatly enhances programmer productivity. Further details on the abstractions used and their advantage over Fortran style of coding may be found in [2, 13]. There are some problems for which it is inherently difficult to encode them in a declarative language. But we confine our discussion to only the class of problems for which it is natural to express them in Id.

While this style of coding is elegant and provides great ease for a programmer, it introduces a large number of function calls and consequently precludes certain optimizations because of function boundaries. However, our compiler optimizations, elaborated later in Section 4.3, eliminate most of these inefficiencies. Here we illustrate one crucial aspect that manifests extensively in our programs. The instruction counts for Simple shown in Table 1 are obtained from a program that uses a fair amount of higher order functions and Currying [13]. Implementation of higher order functions and currying is quite tricky and involves considerable overheads. For instance, currying is the operation of composing new functions by supplying only a subset of arguments required by another function. Suppose we have the following bindings in Id:

```
multiply x y = x*y; double = multiply 2; w = double 7;
```

Multiply is a function of arity 2. Instantiation of the function takes place when the arity is satisfied. When we compute *multiply 2*, the body of the *multiply* function is not activated. Instead, a structure is formed to remember the name *multiply* and the argument value 2. The value of *double* is this structure, which actually represents a function of arity 1 and can be invoked with different arguments many times. Evaluation of *double 7* will result in the instantiation of the function *multiply* with the two arguments 2 and 7. Thus, function invocation requires checking the arities and forming these intermediate structures. In our implementation, all this overhead is accounted by counting the closure instructions listed in Section 3.2. Compile time substitution for the functions eliminates these overhead operations. In the above example, the current Id compiler can substitute `w => double 7 => multiply 2 7` and eliminate all the intermediate structures and closure operations. In all of the Simple code, we ensured that no closure operations are executed despite the heavy use of currying in the program. Hence, this does not effect the results presented here.

4.2 Structure handling

It is often viewed that dynamic creation of structures is very expensive and that functional languages with parallel semantics necessarily require this. We believe that for any model of parallel computation, dynamic allocation of structures is an inherent requirement; otherwise parallelism may have to be compromised. In TTDA new structures are created by invoking the memory manager. Our instruction counts reflect the additional instructions executed for dynamically allocating the memory, but not for any policy determination.

Another concern related to functional languages is the requirement for constructing structures in a strict manner - that is, the computation for all the elements of a structure must be specified before the structure can be used. This diminishes programming convenience significantly in certain computations. Introduction of I-structures is an important step in overcoming this problem [5]. Use of I-structures removes this constraint and permits non-strict construction of structures, at the same time preserving the determinacy of computation.

A well-known problem with functional languages is structure copying. When a program has to modify only a few elements of a structure, the declarative style requires the creation of a new structure with some new elements and copying all the unmodified elements into it from the old structure. Sometimes it is possible to overcome this problem, if the modified elements form a well-defined structure. An example of this is the LU decomposition algorithm discussed in Section 2.1. Similarly if more information is known about the algorithm, one can be more clever and reduce the number of new matrices created. There is still a class of problems for which the structure copying problem cannot be solved elegantly. Examples of this are transitive closure using Warshall's algorithm [17] and collection of a histogram of frequencies of certain events [5]. Unless some language extensions are made, it is extremely inefficient to do them in Id. We exclude them from our study at the present time.

4.3 Code optimizations

In general, compiler optimizations are more crucial for functional languages than for imperative programs. The essential optimizations used here are not complex. Most of them are present in many compilers for imperative languages. We feel that performing these optimizations for functional languages is much easier compared to imperative languages, as the former provides a cleaner structure for representing programs and therefore requires much less global analysis than the latter. To reinforce these views, below we comment on the optimizations used in the Id compiler.

1.Inline substitutions: Substituting for functions at call site is an important optimization, specially when higher level abstractions are used by defining functions that correspond to them. The absence of the concept of global variables in Id can make function calls expensive, because all values used in a function must be explicitly passed as arguments. Besides the saving of call overhead, substitution opens up opportunities for other optimizations in the inline code. For example, the LU decomposition example in Section 2.1 repeatedly invokes the function *fill_matrix*, which is defined as a separate abstraction (for programming convenience) to fill in portions of rows and columns. However if these are not substituted inline, the LU decomposition of a 16×16 matrix executes 109,106 operations as compared to the 35,552 operations reported in Table 1 which used inline substitutions. Thus, inline substitutions have a dramatic effect on the performance. It should be noted that function call substitution in Id is fairly complex as the compiler has to perform arity analysis (as illustrated in Section 4.1. Fortran programs generally avoid subroutine calls as they tend to be very expensive because of context switching. However, inline substitution is difficult in Fortran because of side effects involving common statements. Great care is needed to map the common areas to preserve the semantics.

2.Fetch elimination: This is to eliminate unnecessary fetches from memory, when the value is available in the processor (as a token in TTDA and as a register content in conventional machines). In Fortran programs this is fairly common as great care is taken to keep values in registers and eliminate memory accesses. In fact Fortran compilers do sophisticated index analysis to take maximum advantage of this optimization. In Id, besides the saving of one fetch, this opens up further optimizations. For example, when the element to be fetched is a function name, the compiler can detect this and perform the arity analysis for possible inline substitution. In Id, a special case of this occurs very frequently, which is discussed next.

3.Tuple Ellision: Tuples are implemented using I-structures. Often values are passed and returned as tuples and unfolded by the recipient. A tuple is created and elements are stored into it by the sender and read by the recipient. (In Id multiple-valued functions are required to return a single tuple of the values for providing consistent meaning for multiple bindings. For details see [14].) Inline substitution can get rid off the tuple entirely. For example, the simple program, whose high level encoding is discussed in [13], extensively uses set abstractions represented by a tuple of a generating function and a range. When this program was run for a 10×10 grid, without any substitutions, tuple ellision could not be done because of function boundaries. As a result it took 1.9 millions operations. When inline substitutions are performed, the same program took only 196,648 instructions as shown in Table 1.

4.Dead code elimination: Both Fortran and Id compilers easily eliminate code that is unreachable. However, a more substantial benefit of this optimization is achieved when dead code can be detected from data dependence analysis. For example, index analysis and constant propagation can lead to the detection of conditional values at compile time and eliminate the condition testing all together. The Id compiler does not do sophisticated analysis at the present time.

5.Common subexpression elimination and code hoisting: Both these are fairly common in Fortran compilers and the Id compiler also implements them. They move invariant expressions outside loops and eliminate multiple evaluations of the same expression in a context. Again, the Id compiler does not use any index analysis for this purpose; but performs these optimizations in all other situations.

6.Other optimizations: Id compiler performs limited constant propagation. It does not attempt any loop jamming or interchange etc. Loop jamming significantly effects the parallel costs, as can be seen later.

5 Overheads of parallel execution

In this section we classify the instructions executed by a program and identify the instructions executed as part of the overhead for parallel execution. First we separate the instructions that correspond to the basic work to be done - this is the set of all fixed and floating point operations that are specified in the program, after all the optimizations are done. We are tacitly assuming that these instructions can be identified in an execution trace. In practice, this is not so easy and we will take approximations as we go along. The rest of the instructions executed are for a variety of purposes including loop control, function calls, movement of data, termination detection etc. We lump the latter as control overhead. This dichotomy gives us

$$\begin{aligned} \text{total count in von neumann execution} &= \text{basic work} + \text{sequential control overhead} \\ \text{total count in dataflow execution} &= \text{basic work} + \text{parallel control overhead} \end{aligned}$$

If we are considering equivalent programs in Fortran and Id and if they do not have any redundant computations, then the basic work must be the same for both. If the control structures of the programs are same, that is, they both have same number of loops, iterations, function calls *etc.*, then the difference between the two overheads must be contributed by the declarative and dataflow characteristics of the programming and execution models we adopt. In the sequel we will further subdivide the control overheads and arrive at a cost structure.

We use the 10×10 matrix multiply example of Section 2.1 throughout to illustrate the analysis. Table 2 lists the parameters, which we define as we go along. The values shown against each parameter in Table 2 are the actual counts of events observed for that parameter in the executions of the 10×10 matrix multiply example in Fortran and Id respectively. Table 3 lists categories of overheads and it gives an expression for each in terms of the parameter names listed in Table 2. Expressions given for both Fortran and Id are derived as we go along. Table 3 also shows the values of these expressions when the values of the corresponding parameters are substituted. The sum of all these costs should be the total number of instructions executed. This estimate is compared with the actual number of instructions executed at the end.

5.1 Basic work

The parameters *float* and *fixed* in Table 2 are respectively the total number of floating point and fixed point operations executed by the matrix multiply program. The data structure references specified in the program are also basic, but we treat them separately for many reasons explained in Section 5.2. In any machine, floating point operations are rarely used for purposes other than what is directly specified by the program. That is why we see 2000 floating point operations in both Fortran and Id versions. However, we see 1708 and 2352 fixed point operations for Fortran and Id. There are no problem-related fixed point operations in the matrix multiply program. In the Id version incrementing and comparison of the loop indices for the three loops take 2331 operations. The Fortran version eliminates 2000 of these by using the BXCE instruction in the inner-most

Parameter name	Fortran	Id
float	2000	2000
fixed	1708	2352
allocations	0	100
loads	1000	3027
stores	100	116
loops	111	111
constants	562	456
iterations	1110	1110
feedbackvalues		222
feedbackfrequency	2220	120
calls	0	0
arguments	0	0
conditionals	111	0

Table 2: Parameter Counts for 10 X 10 matrix multiply

Category name	Expression for Fortran	cost	Expression for Id	cost
Float	*float*	2000	*float*	2000
Fixed	*fixed*	1708	*fixed*	2352
Dynamic alloc	0	0	$2.7 * allocations$	270
Str reads	*loads*	1000	*loads*	3027
Str writes	*stores*	100	$2 * stores$	232
Loop iteration cost	$iterations +$ $2 * feedbackfrequency$	1350	$3 * iterations +$ $3 * feedbackfrequency$	9990
Loop setup cost	*constants*	562	$5 * loops +$ $2 * constants +$ $3 * feedbackvalues$	1911
Fn call setup	$23 * calls$	0	$2 * calls$	0
Argument Passing	$2 * arguments$	0	$2 * arguments$	0
Condition Cost	*conditionals*	111	*conditionals*	0
Total estimated		6831		19782
Total measured		7392		20174
Identities measured				2358

Table 3: Expressions and Overhead costs for 10 X 10 matrix multiply

loop. However, it executes roughly 1360 instructions for computing array indices. All the fixed point operations are for loop index and array index calculations and should really be considered as overhead. Unfortunately, using the IBM 370 execution trace for an arbitrary program, it is hard to separate these overhead fixed point operations from those required in solving the problem. Here we take an approximation and assume all the counted fixed point operations are essential in both Fortran and Id executions.

5.2 Structure accesses

Here we are interested in isolating the overhead in implementing the data structures used by the program, including any work done for creating structures or selecting components, *etc.* To read an element of a structure, TTDA executes one *I-fetch* operation. Multi-dimensioned structure access is also accounted here, as a separate *I-fetch* is executed for each dimension. For writing a value into the store, TTDA executes 2 instructions on average - an *I-store* and an *identity.* Recall that the latter is for detecting the termination. Each *I-store* generates a signal after its completion and all these signals are collected using an inverted binary tree of *identity* operations. Although some of the *identity* operations can be optimized away, in general we estimate one *identity* for each *I-store* operation. Structure allocation in TTDA is done by two library routines *matrix* and *array.* The number of instructions executed in them depends upon the size of the structure being created. In our measurements the actual cost of structure creation is measured accurately for the whole program and added as a lumpsum overhead to the total. On average about 2.7 instructions are executed for each element allocated. Thus, we collect 3 parameter counts: number of structure elements allocated, read and written. Then the structure overhead is given by the expression

$$loads + 2 * stores + 2.7 * allocations$$

5.3 Loops

In IBM 370, the most efficient form of a loop is implemented using the BXLE instruction. Initially 3 registers are set up with the starting value, limit value and the step size for the loop index. At the end of each iteration, the single instruction BXLE performs the increment, test and conditional branch. Since we classify BXLE as a branch instruction, we do not count the two fixed point operations of incrementing and comparing. At other times, the loop index is kept either in a register or memory and explicit instructions are used to increment and test it. In our estimate we will treat all loops as if they were implemented most efficiently using the BXLE instruction and specify an overhead of one branch instruction per iteration. However, whenever incrementing and counting are explicitly done, we account for them as essential fixed point operations.

In TTDA, all loops are implemented using the while schema. For example, consider the innerloop of the matrix multiply example shown in Section 2.1. It is implemented as the following equivalent while loop:

```
{While k<=m Do Next k = k+1; Next s = s + A[i,k] * B[k,j] Finally s}
```

Here A,B,i,j,m are loop constants and k,s are feedback values that circulate from one iteration to the next. The following sequence of events take place in executing a loop in TTDA.

A context manager is invoked using the *get-context* instruction. The manager assigns a processor to the program body and initializes its state. Loop constants, such as A, B,i,j,m in the above example are stored in the constant memory using *load-constant* instructions. Execution of the loop is not allowed to proceed until all the loop constants are in place. This is accomplished by building an inverted tree of *identity* instructions to detect the termination of all *load-constant* instructions used. This signal is used to gate all the inputs to the loop body.

There is a *switch* instruction for each circulating value, such as **k,s** in the above example. The switches pass the circulating values either to the loop body or to the loop result expression, based on the value of the loop predicate. Each time a circulating value is passed to the next iteration of the loop body, it is associated with a new tag, using D-N instruction, so that successive iterations execute concurrently without interference. For example, the successive values of **k,s** trigger computations in successive iterations, even while previous iterations may still be computing. Unfolding of iterations is automatic as subcomputations that depend upon previous iterations are delayed until the corresponding values are ready, and other subcomputations that have their inputs can proceed.

At the end of the loop, the circulating values from the last iteration are diverted to the loop result expression. Their tag is reset to that used at loop initiation time using D-1 instruction, so that the loop body with individual tags for iterations remains oblivious to the global flow of values through the loop.

Loop termination is determined by circulating a signal indicating the termination of each iteration. The termination of each iteration is detected by an inverted tree formed from not only the *sink* nodes in the loop body, but also the D-N instructions that feed values to the next iteration. Finally, in order to limit the unfolding of iterations, (to some value that can be set by a manager), each iteration executes a *loop-req*, a *loop-rel* and a D-N+-2 instruction.

5.3.1 Per iteration costs

From the above description of the loop operation in TTDA, we can see that there are 3 overhead instructions (D-N+-2, *loop-req, loop-rel*) for each iteration and 3 overhead instructions (D-N, *switch, identity*) each time a value is circulated. Let *feedbackfrequency* denote the total number of times values are circulated. For instance, in the 10×10 matrix multiply program the value k is circulated 1,000 times and j is circulated 100 times, *etc.*For the whole program we observed $iterations = 1,110, feedbackfrequency = 2,220$. Thus we get the expression for loop iteration overhead as

$$\textit{dataflow loop iteration costs} = 3 * (iterations + feedbackfrequency).$$

In a conventional machine, circulating values are usually kept in registers and the arithmetic operations produce the values directly into the registers. Their circulation is achieved by a transferring control back to the beginning of the loop body. Thus ideally, the cost per iteration is just one (branch) instruction. However, for lack of sufficient number of registers, some circulating values may have to be maintained in memory. Then each iteration executes a store and a load for each such value. If *feedbackfrequency* denotes the counts for only those circulating values that are kept in memory, we get the following cost expression for loops:

$$\textit{von neumann loop iteration costs} = iterations + 2 * feedbackfrequency$$

In IBM 370 it is tricky to isolate the circulating values kept in memory, from an execution trace. In our measurements we used a heuristic based on the registers used, to determine the feedbacks. For the matrix multiply program our measurements showed $feedbackfrequency = 120, iterations = 1,110$.

5.3.2 Loop setup costs

We now give an estimate for the overhead instructions executed once for each loop (independent of the number of iterations) for loop setup and cleanup at the end.

In TTDA as mentioned earlier, two instructions (*get-context, rel-context*) are executed to set up and release a new context for the loop instance. For each loop constant TTDA executes two instructions (*store-constant, identity*). It should be noted that constants could also be implemented as circulating values. But the, the overhead would be much more, as can be seen from the expressions for loop iteration costs.

When the loop terminates, 3 instructions (D-N+-1, *loop-req, loop-rel*) are executed in connection with the bounded loop schema. Also, for each circulating value 3 instructions (D-1, *switch, identity*) are executed for final diversion and tagging of the result values. This count for the number of circulating values, which we denote by *feedbackvalues* differs from *feedbackfrequency* defined earlier in the following sense. Consider the circulating value s in the inner loop of the matrix multiply example. The inner loop is invoked 100 times and each time it performs 10 iterations. There will be 100 loop terminations, for each of which the final diversion costs are incurred for s, hence the circulating value s has $feedbackvalues = 100$ and $feedbackfrequency = 1000$. For the whole program we observed $feedbackvalues = 222$. Thus, we get the cost expression

$$dataflow\ loop\ setup\ costs = 5 * loops + 2 * constants. + 3 * feedbackvalues$$

A conventional machine has no additional overhead to instantiate a loop. In a conventional machine, a loop constant is usually loaded into a register at the beginning of the loop. We consider this as an overhead of one load instruction. However, in IBM 370 sometimes a loop constant may be kept in memory and accessed as a memory operand. We do not count this as overhead in our results reported. If *constants* denotes only the constants stored in registers,

$$von\ neumann\ loop\ setup\ costs = constants$$

In our measurements for 10×10 matrix multiply example, we found $constants = 562$.

5.4 Functions

In sequential machines, this involves saving and restoring the contents of processor registers and address space parameters and the overhead for this varies substantially depending upon calling conventions and the type of a call. So we use an average estimate. In TTDA, a function is invoked by executing a *get-context* instruction, which sends a request to a manager. The manager in response, allocates processor and memory to the function and returns the new context address. The caller executes a *change-context* instruction for each argument. This instruction takes a context and a value and sends a token with that value to the specified context. One of the arguments is the address of an instruction in the calling program, that must receive the result. The called function is a graph with one root node for argument and the subcomputations for each argument can proceed asynchronously as and when the corresponding value is received. A function returns its result also using the *change-context* instruction.

5.4.1 Call setup costs

For Fortran programs on IBM 370, we observed that on average, a caller executes 3 instructions to load the address of a subroutine into appropriate registers and jump to it. A callee executes about

20 instructions to set up a new context and restore it at the end. This varies from subroutine to subroutine; for instance, the calls to library routines are very fast. So we charge an overhead of 23 instructions for each call to a user-defined function or subroutine. In TTDA, two instructions *get-context,rel-context* are executed for setting and releasing the context. If *calls* the total number of function calls made, we have

$$\begin{aligned} \textit{von neumann function call setup cost} &= 23 * \textit{calls} \\ \textit{dataflow function call setup cost} &= 2 * \textit{calls} \end{aligned}$$

5.4.2 Argument costs

In conventional machines, explicit passing of arguments is often avoided by making use of the shared storage, *e.g.,common* in Fortran. Even when arguments are explicitly passed, there are many conventions and the cost varies substantially. We estimate that, on average, for each value exchanged requires the execution of two instructions – a store by the sender and a load by the receiver. In TTDA, each argument/result is passed asynchronously using a *change-tag* instruction and most often the receiving instruction is an *identity*. Hence we estimate an overhead of 2 instructions per exchanged value. If *arguments* denotes the total number of arguments and results passed, we have

$$\begin{aligned} \textit{von neumann argument cost} &= 2 * \textit{arguments} \\ \textit{dataflow argument cost} &= 2 * \textit{arguments} \end{aligned}$$

5.4.3 Function call summary

To illustrate the overheads in function calls, suppose that the innermost loop in the 10×10 matrix multiply program is made a separate function which performs the inner product of the row and column of a matrix.

The function is called 100 times, with altogether seven argument/result values. Hence the total function call overhead is $2 * 100 + 2 * 700 = 1600$. An equivalent Fortran program would incur $23*100+2*700 = 3700$. Some of the optimizations that were done earlier can no longer be done and hence the total number of operations goes up even higher. Actual runs of the above program took a total of 14,880 operations on IBM 370 and 21,600 operations on TTDA. Hence inline substitutions of functions are of tremendous importance for functional programs; they are even more important if Fortran programs are written in the same style.

5.5 Conditionals

Each condition in a Fortran program corresponds to a conditional branch instruction. In Id, evaluation of either arm of a conditional expression must be triggered only when the predicate evaluation completes. This is accomplished by passing each argument to an arm of a conditional through a switch operator. The switch operator is triggered by the result of the predicate. Thus there will be a switch for each value in the set of all input values to both arms of the conditional. Furthermore, the switch that discards a value is like a node with no outcome. A signal from it must feed into the termination tree. Hence conditional evaluations are in general more expensive in TTDA. In our measurements we counted the conditional overhead as the number of branch instructions in IBM 370, other than the branches that are used for controlling the loop iterations. In the 10×10 matrix multiply examples, the zero-trip tests made for each of the 111 loops show up as condition tests. In TTDA we counted the number of all switch instructions, other than those used for feedback values in loops. We ignored the corresponding *identity* instructions, as this entails the knowledge of the proportion of the above switches for the true and false arms of the conditionals.

5.6 Cost summary

In summary, we have defined the parameters shown in Table 2. The corresponding counts can be obtained from the execution traces. Table 3 gives the expressions for the overheads in terms of the above parameters.I In the case of Fortran programs, the disparity between the estimate and actual instruction counts should correspond to the movement of local data between registers and memory. In Id programs, the disparity is due to the approximations made in counting the identity operations.

6 Analysis of Simple

We have done detailed analysis on all the subcomputations in the Simple program and tabulated comparisons similar to the matrix multiply example illustrated in Section 5. The comparisons show nearly identical parameter values for Fortran an Id routines which are very similar in structure and show diversity wherever differences in the routines are significant. For lack of space we show only the summary comparison in Tables 4 and 5, for a 16 × 16 Simple executing one iteration. Below we comment on these tables and explain some of the differences.

6.1 Detailed comments

Basic work: The difference of about 5000 floating point operations is due to, what we perceive as, algorithmic differences, which we did not attempt to correct as it would distort the structure of the Id program. In the Fortran version of the velocity computation, the programmer performed grouping of operations using the distributive law. In the Id version, an abstraction for line integral is used. Thus, $\int(p+q)$ is computed as $\int p + \int q$. This happens about 8 times for each node in the grid. Similarly there are other situations where minor optimizations are done by the programmer. The subdivision of fixed point operations as done in Section 5.1, is very tedious to do for this big program. We simply lumped all the fixed point operations into one category and these include the fixed point operations specified by the program as well as those used for loop control *etc.*Hence the two counts differ substantially.

Structures: The overhead to allocate the structures dynamically reflects only the calls made to the resource manager and not for executing any algorithms. In TTDA only the program specified stores are done and hence the number of stores will be much less than for Fortran, as the latter includes all other book keeping operations on memory. Nevertheless, the total structure related costs balance out, 76116 in Fortran compared to 88993 in Id; the only cost that stands out is 18112 *identity* operations associated with the stores.

Loops: The Fortran version shows fewer loops, but more iterations than the Id version. When we examined these tables for individual routines, we could spot which routines caused this difference. There is a difference in the representation of polynomial coefficients in the two versions. To evaluate a polynomial, two table lookups are performed to determine the ranges of given values of temperature and density. Each pair of ranges is associated with a matrix of polynomial coefficients. In the Fortran program, each time the ranges are determined and the corresponding coefficients

Parameter name	Fortran	Id
float	82848	87985
fixed	41331	31744
allocations	0	5393
loads	43465	56318
stores	32651	9056
loops	2020	4130
constants	11754	16452
iterations	21544	12207
feedbackvalues		11845
feedbackfrequency	3998	29489
calls	247	0
arguments	1433	45402
conditionals	8446	10642

Table 4: Parameter Counts for Simple 16 x 16 one iteration

Category name	Expression for Fortran	cost	Expression for Id	cost
Float	$float$	82848	$float$	87985
Fixed	$fixed$	41331	$fixed$	31744
Dynamic alloc	0	0	$2.7 * allocations$	14563
Str reads	$loads$	43465	$loads$	56318
Str writes	$stores$	32651	$2 * stores$	18112
Loop iteration cost	$iterations+$ $2 * feedbackfrequency$	29540	$3 * iterations+$ $3 * feedbackfrequency$	125088
Loop setup cost	$constants$	11754	$5 * loops+$ $2 * constants+$ $3 * feedbackvalues$	89089
Fn call setup	$23 * calls$	5681	$2 * calls$	0
Argument Passing	$2 * arguments$	2866	$2 * arguments$	90804
Condition Cost	$conditionals$	8446	$conditionals$	10642
Total estimated		258582		524345
Total measured		298502		554254
Identities measured				133619

Table 5: Overheads for Simple 16 x 16 one iteration

are copied into a temporary array using an extra loop, whereas in the Id program, the coefficient matrix is simply selected from a matrix of matrices. Since this happens in the innermost loop, the Fortran version shows up many more iterations all of which involve copying elements into temporary matrices. The higher number of loop initiations in the Id versions are for a different reason altogether. In the test data that was used for the program, the search for the temperature and density ranges seldom takes more than one iteration. Thus most of the time, in the Fortran version we do not even see a loop back, as the range is found in the first pass itself. Hence no loop initiations corresponding to them are recorded. Whereas in the Id version, loop initiation takes place explicitly, whether any iterations are executed or not. Thus, every initiation is counted and hence the disparity. There is also a minor difference in computing the inverse of a polynomial. The Id version sets up a loop for Newton's approximation method, whereas the Fortran version uses a different approximation avoiding a loop.

For loop control, the Fortran version incurs a total of 41294 operations, whereas the Id version incurs 214177. This 5-fold increase is the dominating factor for the increased overhead in Id programs. About two-thirds of this is due to the need (required by the language) to pass every argument explicitly since there is no notion of environment. The remaining one-third is due to the *identity* operations required for termination detection and regulation of iterations.

Functions: In the Id version, virtually all functions are substituted inline. The function calls reported in Fortran are for the subroutines that are not substitutable easily. Manual substitution for them can be done, but is tedious. These include the PROJCT function and the polynomial evaluation functions, which constitute the bulk of the calls. There are about 900 calls to library functions such sin, cos and sqrt made in both versions. Since Fortran uses a fast call mechanism for them, we ignored the overheads for their calls.

Conditionals: Simple is not a good example for the study of conditionals as there are very few conditions in it. Finding maximum, minimum, absolute value account for most of the conditionals reported.

In summary, differences in optimizations, algorithmic and representational variations do cause some differences in the instruction counts; but these are negligible. Some of them can be avoided by restructuring the code to mimic the control and data structuring used by the Fortran version. This will of course destroy the abstract nature of the code and will tremendously increase the difficulty in debugging, in general. We have tested this approach by encoding the Simple program from scratch, without paying attention to high level abstractions and adopting most of the control structuring and optimization strategies of the equivalent Fortran program. This program for 16X16 grid and one iteration took 311,117 instructions, reducing the instruction count for the more abstract program we reported here, by more than a third. One should weigh the resulting improvement from a less abstract program against the ease of coding and debugging the higher level program, as it is closer to the mathematical description of the problem.

If we ignore the above mentioned minor differences, the major contribution to the higher counts observed for the Id version stems from the overheads we identified for each class. A bulk of this excess is due to the fact that loops are explicitly initiated like functions and all arguments and results are not only passed back and forth from them, but also are circulated within the loops. Thus, the cost is proportional to the number of quantities moved in this fashion. The substantial saving for a von Neumann style machine is due to the fact that most of these values are kept in memory and instructions often combine their access with arithmetic. For load/store architectures, since the latter facility is not present, they still end up moving these values back and forth from memory and hence incur overheads to the same tune as a dataflow machine. To the extent parallelism is given up by not running all loop iterations in parallel, the above mentioned overheads can be eliminated by less expensive loop controls.

It should be noted that *identity* operations constitute a fourth of the total number of operations. This is a rough measure of the cost of synchronization for parallel execution of this program. If we go by the rule of thumb that the ratio of instructions executed for equivalent programs on RISC machines to that on CISC machines is about 1.5, we can see that parallel execution is contributing another 40% more in this program, bringing the ratio of dataflow instructions to CISC machines to 1.9.

7 Conventional multi-processors

A major portion of the costs enumerated in Section 5 is because of the parallelization of loops and and function calls in TTDA. Any other system that executes the program in parallel must incur similar costs. The cost structure may be different based on the nature of parallelization and the architecture. To illustrate this, we consider the parallel execution of some of the examples on a system that resembles the IBM RP3 multi-processor system [16].

7.1 EPEX - a parallel processing facility

EPEX is a facility on the IBM 370 machines for parallel execution of a Fortran program on multiple virtual machines [18]. Each virtual machine can be thought of as an IBM 370 machine. The shared segment feature of the system provides a means for all the virtual machines to communicate, as if they were independent processors connected by a shared memory. Most of the synchronization features are implemented by software routines. It is envisioned that this system provides a testbed for studying some of the behavior of the RP3 multi-processor system under construction.

In EPEX, a Fortran program is parallelized by user defined annotations. The annotated program is translated into a legal Fortran program with appropriate calls to some library subroutines, which perform the synchronization functions. The user specifies the number of processors on which the parallel execution of the program is envisioned and the system creates that many virtual machines with a shared segment, which behave as independent virtual processors. All the virtual processors execute the same program concurrently. The annotations designate portions of the program into one of the following 3 possible categories:

Serial sections, such as initialization of shared memory, are designated for serial execution by enclosing them within a pair of special annotations. The annotations are translated into calls to synchronization routines that ensure that only the first virtual processor that attempts to execute it will do so; all others will be forced to skip over that portion. All virtual processors will normally synchronize at the end of a serial section and proceed further when the serial execution is completed.

A parallel section is unannotated and all the virtual processors execute the same code. Typically these are portions of code that performs some initialization of the state of each virtual processor.

Finally a loop may be designated for parallel execution. In a parallel loop, the virtual processors compete and select distinct iteration numbers and execute the corresponding iteration. When an iteration is completed, the virtual processor selects another iteration number in the same manner until all iteration numbers for the loop are exhausted. All the virtual processors normally synchronize at the end of a loop and proceed further only after all the iterations are completed. When loops are nested, at most one loop within each nest is permitted to be parallelized (in the current implementation).

Fixed Cost	42 * *processors*
Loop setup Cost	130 * *parallel-loops* + 80 * *processors*
Loop iteration Cost	87 * (*parallel-iterations* + *processors* * *parallel-loops*)
Serialization Cost	156 * serial-sections

Table 6: Parallelization Costs in EPEX system

The system also provides special primitives for parallel manipulation of locations in shared segment. These are associative operations such as add, multiply, logical conjunctions *etc.*The virtual processors can perform these operations on a location concurrently without losing determinacy of the computation.

7.2 Costs in EPEX

Since only some loops are parallelized, we count the number of loops parallelized and the total number of iterations in the parallel loops. Thus, we use the following 4 parameter counts: *processors, parallel-loops, parallel-iterations* and *serial-sections.* We can then express the parallel costs as shown in Table 6.

The constants coefficients in the expressions on the right hand column of Table 6 are obtained by counting the average number of instructions executed eachtime the corresponding synchronization routine is executed in EPEX. The fixed costs correspond to getting a unique number for each virtual processor. For setting up a loop, one processor executes a routine to initialize the state representing the loop and all the other processors must skip over this (like a serial section). For each loop iteration, a processor must check and obtain an iteration number. For each loop, each processor will perform this test once more than the number of iterations, to get out of the loop. The cost expressions do not include the counts of instructions executed in waiting for other processors, as this is tricky to account for in EPEX implementation, but we recognize this could amount to significant overhead with a large number of processors.

As an example consider the matrix multiply example of Section 2.1. The following program annotates the outer loop for parallelization. The pair of annotations @do and @endo denote a parallel loop. The annotation @shared indicates that the corresponding data structures must be placed in shared memory.

```
     subroutine matmult(A,B,C,n)
     @shared/public/ A(n,n),B(n,n),C(n,n)
     @do 50 i=1,n
     do 10 j=1,n
     C(i,j)=0.0
     do 10 k=1,n
10   C(i,j) = C(i,j) + A(i,k) * B(k,j)
     @endo 50
     return
     end
```

For $n = 10$, the program executes the 10 outer loops in parallel. Thus we have *parallel-loops = 1, parallel-iterations = 10, serial-sections = 0.* This is a parallel cost of 1167 when run on one processor and a 3090 when run on 10 processors. The cost with 100 processors (assuming we can parallelize all loops) is prohibitively high for this example. This overhead marks a sizable increase to the instruction counts in Table 1.

7.3 Comments

In a real multi-processor system, the coefficients of expressions in Table 6 are likely to be smaller, as the hardware features might implement some of the bookkeeping. But if a load/store style of processor architecture is used, it is unclear how much reduction can be obtained. In any case it is clear that for small number of processors, the overhead is smaller; and the parallelism exploited is also smaller. As number of processors increases, the cost is prohibitive. These trade-offs need further investigation, especially taking into consideration the timing characteristics of a real multi-processor system.

8 Concluding remarks

It is very clear that parallel execution of a computation calls for expending additional resources. The ultimate question of interest is how these additional resources can be traded in return for the benefits offered by parallel execution. The scene is fogged by the multitude of factors both on the cost side as well as on the side of benefits. Furthermore the factors are often ill-defined and have a very high degree of variance. The benefits of parallel execution include the reduction in total time of execution compared to a sequential machine, the ease with which additional resources can be employed to execute a program faster and the ease with which a program can be encoded and debugged on a parallel processing system. Similarly on the cost side one may examine the additional memory used, the additional processor time used in terms of instructions and the dynamic distribution of these memory and processing resources during execution, which influences the peak memory requirement and the critical path length for execution. Examining any one of these factors, disregarding the others has the danger of grossly distorting the picture. Nevertheless, one must have a coherent means of examining each of them in detail and finally assess the overall trade-off combining them.

In this paper we examined the additional processing in terms of the total number of instructions executed by a parallel processor over and above that of a single sequential processor. Since the instruction counts are affected by a variety of factors including the language, compiler, optimizations and instruction set architectures, one has to be careful in isolating these effects. In this paper we have closely examined the differences between the executions of a reasonably large program on a dataflow machine as well as on IBM 370. We have taken care to see that the differences introduced by differences in algorithms, compilers and optimizations are minimized so that the effects of architecture and parallel execution dominate. We have argued that architectural differences diminish if the von Neumann architecture is a load/store type. Although we have not presented the comparison with counts on a real load/store architecture, we have presented sufficient analysis of instruction set to convince that this is indeed the case. Isolating this, finally leaves the effects of parallel execution. We have identified the principal features of a program execution that are significantly effected by parallel execution, namely loops, values passed and circulated. By examining the patterns in which the two machines execute these control structures, we have quantified the overhead instruction counts for these features. On the large example we presented, we showed that the estimates of additional instructions due to parallel execution come very close to the actual execution counts.

We believe that these overheads must be incurred by any parallel machine. The extent of overhead might vary with the degree of parallelism that is being exploited. Dataflow machine is perhaps an extreme case in this arena, as it tries to exploit all the parallelism that can ever be extracted from a given specification. One might argue, then, that trying to utilize only a limited amount of parallelism in a program reduces the overheads substantially, thereby giving a better payoff. Our experience suggests the contrary - that the overheads due to parallel execution in a dataflow machine are not orders of magnitude higher than in a conventional parallel processor exploiting limited parallelism. Examples presented in this paper show increases of up to 3 times the number of instructions taken by IBM 370 on corresponding programs. When compared to execution on a load/store architecture, this ratio will come down to 2 or less. In contrast, we took the Simple program that was parallelized for EPEX execution by the authors of [9]. The program parallelized only the outer loops in each subcomputation. For a grid size of 16X16 and for one iteration, the program took approximately 40% more instructions than a uni-processor version, whereas a dataflow machine took roughly 100% more. If we apply the 50% rule for the load/store architectures, one can see that the excess number of instructions executed by a dataflow machine is not far from that of a conventional multi-processor using processors with load/store architecture. In addition, the dataflow machine exploits a lot more parallelism than just the outer loops.

We would like to comment on some subtle details that are often ignored in this comparison. The dataflow machine has the ability to interleave arbitrary set of ready instructions. As a result, it is possible many of these additional instructions can be hidden when one examines the critical path of the problem. Whereas, this is not the case for a conventional parallel processor. Often the synchronization involve participation by many processors and the processors usually execute busy waits during this time. In the above counts on the EPEX simulation, we have completely ignored the instructions executed by a processor during busy waits. Conventional architectures have been studied for a long time and evolved with novel instructions that efficiently implement more frequently occurring operations. In the dataflow machine, there is plenty of scope for such fine tuning of instruction set. For instance, it is not difficult to introduce new instructions that combine switch and tag manipulation instructions so that the overhead instructions for loop feedbacks is reduced to half. In the analysis we presented here, we used a crude method for termination detection by using inverted trees of identity instructions which constitute about 40-50% of the overhead instructions. Again there can be many strategies for implementing the termination detection with lesser number of additional instructions. Above all, the dataflow machine exploits a lot more parallelism than a typical conventional multi-processor system and programs do not need special strategies or recompilations for running on different number of processors.

Examination of effects of finite resources and strategies for their management are subjects of further study. It is possible to conceive of changes to the dataflow architectures, so that when automatic parallelization of all loops and function calls is undesirable, the parallel costs are not incurred. This is also a subject for future study.

References

[1] 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.

[2] 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.

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

[4] Arvind, R. S. Nikhil, and K. K. Pingali. *Id Nouveau Reference Manual, Part II: Semantics.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. *I-Structures: Data Structures for Parallel Computing.* Technical Report Computation Structures Group Memo 269, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, February 1987. (Also to appear in *Proceedings of the Graph Reduction Workshop*, Santa Fe, NM. October 1986.).

[6] 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.

[7] W. P. Crowley, C. P. Hendrickson, and T. E. Rudy. *The SIMPLE Code.* Technical Report UCID 17715, Lawrence Livermore Laboratory, February 1978.

[8] David E. Culler and Arvind. *Resource Requirements of Dataflow Programs.* 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.

[9] F. Darema-Rogers, Karp A., and Teller P. *Applications Survey Reports - I.* Technical Report RC 12743, IBM T.J.Watson Research Center, Hawthorne, New York, May 1987.

[10] John R. Gurd, C.C. Kirkham, and I. Watson. The Manchester Prototype Dataflow Computer. *Communications of the Association for Computing Machinery*, 28(1):34–52, January 1985.

[11] K. Hiraki. *Instruction set reference manual for Sigma-1.* Technical Report Tech report, Electro Technical Laboratory, Japan, 1983.

[12] Alan H. Karp. Programming for Parallelism. *IEEE Computer*, 43–57, May 1987.

[13] Ekanadham Kattamuri and Arvind. *Simple: Part I - An exercise in future scientific programming.* Technical Report RC 12686, IBM T.J.Watson Research Center, Hawthorne, New York, June 1987.

[14] Rishiyur Sivaswami Nikhil. *Id Nouveau Reference Manual, Part I: Syntax.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[15] Rishiyur Sivaswami Nikhil. *Id World Reference Manual.* Technical Report, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

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

[17] Sedjwick Robert. *Warshall's all-pairs shortest path algorithm.* Addison-Wesley, 1983.

[18] J.M. Stone, F. Darema-Rogers, V.A. Norton, and G.F. Pfister. *The VM/EPEX FORTRAN Preprocessor Reference.* Technical Report RC 11408, IBM T.J.Watson Research Center, Hawthorne, New York, September 1985.

[19] Kenneth R. Traub. *A Compiler for the MIT Tagged-Token Dataflow Architecture.* Technical Report LCS TR-370, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, August 1986. (Master's Thesis, Dept. of Electrical Engineering and Computer Science, MIT).