A Comparative Study of Data-Flow Architectures

University of Manchester
Department of Computer Science
Technical Report Number
UMCS-94-4-3

April 1994

Dr. David F. Snelling
Department of Computer Science
University of Manchester
snelling@cs.man.ac.uk

Professor Gregory K. Egan
Laboratory for Concurrent Computing Systems
Swinburne University of Technology
gke@swin.edu.au

Abstract

Partly as a result of the diversity of the architectures and system software, Data-Flow systems have not been compared directly. In this paper, three widely different Data-Flow systems are compared using a relatively uniform metric which is representative of the actual amount "work" performed by these systems to execute a small collection of common benchmarks. One feature, common to all Data-Flow systems, forms the basis for a metric for "work", the creation of tokens. By counting the creation of tokens in all parts of a Data-Flow system, an accurate measure of the work performed can be obtained. The three systems compared are the Manchester Data-Flow Machine, the Stateless Data-Flow Architecture (also from Manchester), and the CSIRAC II machine. All machines perform approximately the same amount of work when solving a collection of benchmark problems, but two systems (CSIRAC II and SDFA) exhibit successful exploitation of a memory hierarchy.

1. Introduction

Data-Flow systems represent a unique class of computer architecture which combines a heterogeneous, fine-grain model of computation with latency hiding mechanisms. In contrast to the von Neumann model of computation, the execution of an instruction in the Data-Flow model relies on the availability of its operands, rather than on a pre-defined sequence on instructions. Even in parallel versions of the von Neumann model, sequencing of instructions is controlled explicitly by the programmer or compiler. In a Data-Flow system, the selection of instructions, for execution, is performed by the hardware at execution time and is constrained only by the partial order implicit in the program's data dependency graph. The resulting computation is fine-grained and exhibits a much higher degree of parallelism than code written for parallel von Neumann machines. This fine-grained parallelism is then used for exploiting replicated hardware for increased performance, masking memory access latency, and maintaining a uniform distribution of workload.

In Data-Flow systems, data values rather than being stored at particular addresses, are *tagged*. The tag includes the address of the instruction for which the particular data value is destined, and other information defining the computational context in which that value is used. This context is called the value's *colour*. The data value, together with its tag, is called a *token*. By considering the creation of a token as a single unit of ``work'', an effective metric for ``work'' can be derived. For example, where a *proliferate* instruction counts as only one instruction it may create many tokens. To count instructions only would obscure the actual work performed by the system. The number of *Created Tokens* is, therefore, our measure of work.

In order for an instruction, requiring two operands to execute, both tokens must exist and be brought together. This synchronisation process is called *matching*. Once these input tokens are matched, the instruction is performed, and the result token(s) sent on to subsequent instructions. Note that tokens which do not require matching may go directly to the execution unit. These tokens are called *by-pass* tokens.

The management of data structures, arrays in particular, is one of the major problems in Data-Flow research. Given that the semantics of Data-Flow languages are basically functional in nature, the modification of a single element of an array necessitates the creation of another array, identical to the original, except for the altered element. Multiple references to an array require multiple copies of the array, even when only one element is needed. Solutions are varied and depend largely on the architecture, but several approaches predominate.

Data structures have two modes of reference, to the data structure as a whole and to the individual elements. The data structure is passed, as a unit, from one part of the program to another, and references to, and modifications of, data structures are performed on the elements. This dichotomy gives rise to various semantics for data structures. The following techniques, representative of Data-Flow computing in general, are drawn from the two major Data-Flow languages Id [Nikhil87] and SISAL [SISAL85].

SISAL streams - are one dimensional arrays, created element by element in order. Once elements have been written they cannot be modified; they can only be removed from the head of the stream; but they may be referenced randomly by an index. They are non-strict, in the sense that elements in a stream may be referenced before they have been created.

I-Structures and non-strict SISAL arrays - are both non-strict in the same sense as SISAL streams, however, they are of a fixed size and shape. Elements may be modified, which implies that the entire array is (logically) copied and the element modified in the process. There are compilation techniques to avoid excess (physical) copying of arrays [Cann88, Sargent86].

Strict SISAL arrays - are distinct, from the non-strict variety, only in that before the pointer to an array is released the entire array must have been created. In this case, no references can occur prior to the creation of a given element, since a reference requires a pointer. This mechanism is supported in software and requires no special hardware support, but all elements of an array must synchronise before the pointer can be released. This can prove costly [Arvind88, Egan91].

M-Structures - are part of the Id language and are the most explicit form of state in Data-Flow computation. They are initialised explicitly and then exist until destroyed. Elements of an M-Structure may be modified asynchronously in a variety of ways.

As is apparent from above, data structures in Data-Flow systems require special treatment. In most systems, there is an additional, specialised function unit (e.g. the structure store in the Manchester machine) that provides the storage and performs token colouring. The most important aspects of this mechanism are the use of pointers and reference counting.

This variety of approaches to data structure management gives rise to many of the distinctions in Data-Flow systems and the difficulty in comparing them. Most studies of Data-Flow systems in the past have relied on metrics similar to those commonly used in conventional computing, e.g. instruction count or execution time. Execution time, clearly the best metric for all such studies, has been avoided because of the experimental nature of these architectures and their diverse and relatively slow cycle times.

By taking the creation of a token as a single unit of work, an effective metric for "work" can be derived. For example, where a proliferate instruction counts as only one instruction it may create many tokens. To count instructions only would obscure the actual "work" performed by the system. The number of *Created Tokens* is, therefore, our measure of work.

In the remainder of this paper, the three Data-Flow systems are described briefly with particular reference to the techniques used to manage data structures, the source of greatest diversity and most of the Created Tokens. The next section outlines the experimental framework used to compare these systems, and the last two sections present the results of the comparative experiments and draw conclusions from them.

2. Three Data-Flow Systems

Although widely different, all three of these Data-Flow systems have their origins in early research into Data-Flow architectures at the University of Manchester. All are based on the data driven model described above, but vary significantly in the details of their implementation. For complete details of these architectures, the reader is referred to the literature [Egan91,Gurd87,Snelling93]. In the following sections, the architectural details relevant to this paper are presented, in particular the memory organisation, data structure colouring, the instruction-set architecture, and workload distribution mechanisms.

2.1. Manchester Data-Flow System

The Manchester Data-Flow Machine (MDFM) represents one extreme in this collection of systems. It has a relatively complex instruction-set which includes vector style operations. The MDFM relies exclusively on latency hiding to manage non-local activity, as opposed to attempting to exploit locality directly.

2.1.1. Memory Organisation

The system is composed of four types of elements: one or more processing elements, one or more structure stores, one global allocator, and one throttle. The processing element contains the matching store where tokens are stored until they are matched by tokens destined for the same instruction. Tokens issued from the function units are placed onto the switch network and routed to one of the processing elements or structure stores, the global allocator, the throttle, or the host.

The structure stores are managed dynamically as a single logical memory which contains all data structures. Scalars are implemented as single tokens and are stored in the matching stores.

There are several approaches to managing this kind of dynamic memory [Kawakami86]. The strategy adopted in the MDFM involves dividing the virtual address space into two parts, one used for smaller structures (typically less than 10 locations) and one for larger structures. This dual allocation strategy involves both local and global memory management. The global allocation of "large" structures is designed to take advantage of interleaving, whereas, the distributed allocation of small structures avoids contention at the global allocator. Memory management is based on a reference counting scheme performed by software, but the hardware performs the garbage collection function.

2.1.2. Data Structure Colouring

Data structures are stored colourlessly in the structure stores. They are, therefore, accessible from all contexts via a pointer. As the pointer to a data structure is passed from one context to another, it is re-coloured. When a data structure reference occurs, the pointer (along with information describing what part of the data structure is required and where it is to be sent) is routed to the appropriate structure store. The structure store uses the context described by the pointer token to colour the data structure tokens.

2.1.3. Instruction-set Architecture

The instruction-set of the MDFM is relatively complex. Although instructions may have only one or two inputs, each instruction may produce several results. In particular, the proliferate instruction produces a stream of values with different index fields in their tags.[1] Even two result instructions may produce tokens with differing tags and values. There is also a tuplicate instruction which sends copies of the incoming token to a number of successive instructions. These complex instructions obviate the need for many intermediate tokens in a data flow graph, e.g. the tuplicate instruction eliminates the need for the intermediate arcs of a duplicate tree.

---

[1] Related instructions can also modify the value of these tokens, such as incrementing the value for each successive token.

2.1.4. Workload Distribution

Workload distribution in the MDFM is based on a pseudo-random hashing function applied to the tag field of each token. In particular, this function uses not only are colour and index fields, but the destination address as well. The result is that a token created by one instruction and destined for another planted "near by" will invariably be routed to a different processor. This prevents the MDFM from exploiting temporal locality within code blocks.

2.2. Stateless Data-Flow Architecture

This system was designed specifically to redress the "latency-hiding-only" approach of the MDFM. A SDFA system is a homogenous collection of processors connected by a network. Each processor includes a matching store, a token queue and a processing element.

2.2.1. Memory Organisation

As its name implies, the SDFA system has no explicit notion of state. There are no structure stores, and only *extract-wait* functionality is provided in the matching stores. However, the SDFA architecture has a hierarchical memory organisation. Within each processing element there is small matching store (Short Stay Matching Store) where tokens wait a short time in the expectation that their matches will arrive soon. If no match occurs in the SSMS after a prescribed number of cycles, the token is forwarded to the main matching store for that processor. The main matching store, in each processor, is composed of multiple banks (8 in the current design), each with two levels. The SDFA machine, therefore, has a three level memory hierarchy.

2.2.2. Data Structure Colouring

Data structures, arrays in particular, are collections of data values distinguished by the index fields of their tags. There are four index fields within the tag: X, Y, Z, and S. The S field is a scratch field for use in reductions and is not used as an array dimension. Arrays are not stored at a particular address, and pointers do not exist. The matching stores provide the storage space for all array elements, which are stored and matched using the same associative mechanism as scalar values.

2.2.3. Instruction-set Architecture

The instruction-set for SDFA is based on a classical two-input, two-output Data-Flow model. The basic philosophy of the instruction-set is that it should be simple and RISC-like. The goal of simplicity (motivated by the desire for speed of computation, ease of compilation, and cost of hardware) is addressed by adhering to a short list of requirements:

Each instruction:

a) has 1 or 2 inputs;
b) must have at least one input which is a token;
c) may have one input which is a literal;
d) produces at most one result value;
e) produces results with exactly one tag value;
f) has 0, 1, or 2 defined destinations.

These instructions may perform up to three operations as a result of a single match, representing a kind of super-scalar approach. The restrictions are that 1) the second two instructions must be simple integer operations (on either the data or tag fields of the token), 2) the second two instructions must each have only one token as input, and 3) no more than two tokens can result from the collection of instructions. Therefore, the tokens passing between these instructions are not included in the total token count.

2.2.4. Workload Distribution

Spatial locality is exploited in SDFA through the hashing function used to route tokens to processors. One of the aspects of this function is a *locality factor*. It represents how many of the low order bits in the index fields are not used in the hashing function. This insures that tokens with neighbouring indices are, for the most part, processed on the same processor. This simple mechanism is exploited naturally in applications, either as a result of template structures, as in finite difference schemes, or algorithmic techniques, such as unit stride vector processing. In SDFA, these structures and techniques can be exploited in several dimensions, rather than just the dimension corresponding to the linear organisation of conventional memories. The part of the tag which keeps track of the recursive structure of the program is also used in this hash function. This provides distribution of unstructured applications. The instruction address is not used.

2.3. CSIRAC II

The CSIRAC II architecture is directly descended from one of the author's early work at Manchester [Egan79]. Its roots lie in the study of Data-Flow machines as applied to image processing and robot control [Egan81,Egan85]. The architecture is unusual in that the temporal order of tokens with the same colour on the same graph arc is maintained.

2.3.1. Memory Organisation

Data structure storage, provided by the Object Store of the CSIRAC II, is similar to the MDFM and MIT machines. The major difference being that large complex data-structures may be associated with a single Object Store Cell. Read-before-write as well as standard read and write functions are supported. All Object Store cells carry state information and reference counts. The state of cells may be interrogated to determine memory state e.g. full or empty. Data structures are stored colourlessly as in the MDFM. The Object Store itself is partitioned and the partitions are associated directly with processing elements. The particular partition is determined by the least significant bits of the descriptor.

Interestingly the CSIRAC II architecture still retains a comprehensive set of in-graph storage nodes [Egan87], with the Object Store being a comparatively recent addition. The default on CSIRAC II is to transmit structures as a sequence of datum in separate tokens (or as several datum in a single token) and not use the Object Store at all. In this sense it is almost identical to the 'stateless' operation of the SDFA.

The matching store has a two-way set associative cache to capitalise on temporal locality. The arriving token's node number and colour is hashed to a store address. In pipe-lined graphs, where no colours are used at all and temporal ordering of tokens is assured by the hardware, this corresponds to a direct indexing operation with no address collisions. The address is used in a conventional manner to access the cache. The cache and associated large backing memory is managed by the matching store's controller. The matching store queues arriving tokens (with the same colour and input arc) thus preserving temporal ordering.

2.3.2. Data Structure Colouring

CSIRAC II tokens carry a single undifferentiated colour tag; it does not separate index and colour fields. This single tag is used to distinguish between different function or loop body instantiations. The descriptor, used to reference a data structure, in the Object Store contains an index and a vector of zero or more destinations to which the accessed data structure should be returned; the descriptor is contained in a single token. The colour of the returned object is the same as the descriptor. Unlike the MDFM, the CSIRAC II Object Store does not produce sequences of tokens with different index field values, relying instead on the temporal ordering on tokens. Data structure elements, where necessary, are coloured explicitly by the execution graph. In the case of pipe-lined execution the hardware maintains tokens in producer consumer, and possibly index, order [Egan91].

2.3.3. Instruction-set Architecture

The instruction set of CSIRAC II, like the MDFM is also relatively complex, although in practice only a small subset is used. Instructions may have one or two inputs and many outputs usually with a single value. Instructions may thus emit many copies of a single value to different destinations without recourse to duplicate trees and their associated latencies. Exceptions to this include instructions, responsible for loop throttling, the generation of loop indices, and proliferate nodes. CSIRAC II's proliferate nodes, unlike those of the MDFM, generate multiple copies of the input value and pass them to the output arc with the same colour as the original input token. In the context of this study the proliferate token is used to generate multiple copies of loop arguments for pipe-lined loops [Egan87].

2.3.4. Workload Distribution

Workload distribution is performed by hashing the token colour with the compile time processor 'allocation' of the destination instruction. It needs to be said that a copy of the entire graph exists on each processor and this scheme may be viewed as a double hashing. The compiler may set the processor field of all instructions to zero. The effect of this is to have all instructions involved with a particular colour, usually a loop or function body, execute on a single processor. The allocation scheme under these conditions is similar to that used SDFA. This compile time option is used in this study to give high locality, which can cause load imbalance under some situations.

3. Experimental Framework

The difficulty in comparing systems, as distinct as the MDFM, SDFA and CSIRAC II machines, arises from the need to measure something in common to all systems that represents the amount of work done by each systems. Instruction count is inappropriate since the complexity of the MDFM's and the CSIRAC II's instruction-sets far exceeds that of the SDFA system. Operation count is ruled out as defining what constitutes an operation can be rather difficult, and the use of cycle count assumes that the architecture is independent of technology, clearly a falsehood. Likewise, using execution time is out of the question.

3.1. Token Count as a Metric

Since all data values, in these systems, are carried on tokens, the number of tokens generated by all components of the system is taken as a measure of the total work. The creation of a token, as an abstraction of work done by a Data-Flow system, is quite natural and is a common statistic gathered by the simulators. Notice that this count includes tokens generated by the MDFM and the CSIRAC II's structure stores as well as by their processing elements.

A token in most Data-Flow machines, is a well defined entity.[2] Although the size of the data value and the tag vary, the token is relatively standard. It is straight forward to identify where a token is created or copied within a Data-Flow system. These token creations serve as a measure of the amount of work performed by the system. Each time a token is created in a function unit or copied from a structure store, the token count is incremented. This measure is independent of the complexity of the instruction, e.g. proliferate type instructions contribute counts depending on the number of tokens created.

Token count includes the ability of the software to reduce the number of tokens created, but is independent of the cycle time, a necessity when comparing such diverse systems.

3.2. Approximations

The following assumptions underlie the experiments discussed below:

- CSIRAC II has the ability to transmit multiple values, such as vectors, as a single "token" in the form of a multiple word network packet. For comparison purposes, each such CSIRAC II network word (containing two values) is counted as containing two tokens. It should be noted that, in this respect, CSIRAC II is 50% more efficient at transmitting structured data than the other two systems.

- The structure store garbage collection in the CSIRAC II and MDFM is not included in the token count metric. There are no such overheads for the SDFA or CSIRAC II when using transmitted data structures.

3.3. Benchmarks

The following are summaries of the benchmark programs used in this study. The codes are all available in SISAL from the authors by e-mail and listings can be found in [Snelling93].

3.3.1. Adaptive Quadrature

Adaptive quadrature is a multiply-recursive program that computes the area underneath a function by refining a trapezoid integration method until an error tolerance is reached. Its interesting properties are that it is both numerical and requires multiple recursion. It does not use arrays. The amount of work done by Aq depends on a parameter, called the tolerance, which determines the accuracy of the area computation. The reciprocal of the tolerance is used as the x-axis in the plots presented later.

3.3.2. Gaussian Elimination

Gaussian elimination is a computation with cubic complexity, which uses both array constructs and recursion. It has bad load balancing characteristics and is, therefore, useful in workload distribution studies.

---

[2] The compound tokens in the CSIRAC II are counted based on the number of elements in the multiple token, see approximations below.

3.3.3. *Matrix Multiply*

Matrix multiply is the most common of all computational benchmarks. It is both trivial and complex. Although load balancing well, it produces high degrees of parallelism and can create "explosions" in resource needs.

The MDFM and CSIRAC II matrix multiply programs are based on that used regularly in MDFM studies [Sargent86, Teo91]. It is written under the assumption that the second matrix is already transposed. The SDFA version performs the transpose as part of the computation.

3.3.4. *Queens*

Queens is a program which computes the number of arrangements of N queens on an NxN chess board, such that no queen is threatened by another. It is intensively multiply-recursive and uses many small arrays. The load balancing characteristics of Queens are unpredictable.

3.3.5. *Shallow*

Shallow is the closest to an application program used in this study. It requires several array variables, including temporaries, and uses both array constructs and recursion. It is the largest and most complex code in the suite. A discussion of this bench-mark can be found in [Hoffmann88].

3.4. Compilation Method

In this experiment, the above codes are executed on the Manchester Data-Flow Machine Simulator (NMR) [Teo91, Gurd92], the SDFA simulator and the CSIRAC II simulator/emulator. The following sections describe the implementation of the benchmark suite on the three systems.

3.4.1. *MDFM*

All the following options are applied at the IF1 to Data-Flow code translation phase of the compilation process [Sargent85]:

-v	Vectorize the code in loops, eliminating redundant addressing.
-clb 0	Generate code assuming all arrays have zero lower bound.
-inl N	Inline functions until their size exceeds N IF1 nodes.
-blo N	Inline recursive functions until their size exceeds N IF1 nodes.

Their use in the various programs was as follows:

Aq[3]	-blo 250
Gauss[4]	-v -clb 0

---

[3] The recursive inlining was tried for increasing values until the performance improvement became constant. In this case, as with Queens below, this was a substantial increase in the number of nodes, namely, an increase of 1180%.

[4] This implementation of Gaussian elimination is performed "upside-down" so that the constant lower bound optimisation could be applied.

Mm	-v -clb 0
Queens[5]	-clb 0 -blo 500
Shallow	-v -clb 0 -inl 1000

3.4.2. SDFA

The SDFA codes are hand written in assembler, since no compiler exists. Recall that the SDFA instruction-set is simple and therefore, frequently, several instructions are required where one suffices on the MDFM or CSIRAC II. These codes were written before the super-scalar facility was added to the simulator, but performance improved nonetheless, see [Snelling93].

Two factors limit the degree of concern here. For the numeric codes, the SISAL compilation system for MDFM is known to be good. This is highlighted by the instructions per floating point operation ratios for the various codes (MDFM: Aq - 2.6; Mm - 2.0; Shallow 4.6; and Gauss - 3.0).[6] The CSIRAC II with no low level optimisation achieves (CSIRAC II: Aq - 2.03; Mm - 0.37; Shallow - 4.4; and Gauss - 6.35).

The SDAL code is written for a macro assembler; therefore, a small library of frequently used SDAL code fragments is employed to simplify code generation. Many redundant operations could be removed by techniques such as loop unrolling and loop invariant removal, if the macros were not used.

3.4.3. CSIRAC II

The benchmark programs were compiled using the Optimising SISAL Compiler (osc) [Cann89] with full IF1 optimisation. Optimisation includes common sub-expression elimination, loop fusion and unrolling but does not include IF1/2 structure optimisations.

The resulting IF1 intermediate code was compiled using the CSIRAC II IF1 translator [Webb93] generating pipe-lined loops for all benchmarks excluding Aq which has no loops. By pipe-lined code we mean that no colour tags are used and execution relies solely on the ability of the hardware to maintain the temporal ordering of tokens. It may be observed that for many scientific applications there is no need for hardware support of colours. The Aq, Gauss, and Queens benchmarks involve recursion and require colours to support this.

Code generated for Aq, Queens and Mm uses transmitted structures and is, in SDFA terms, 'stateless'. Gauss and Shallow are the only benchmarks using the Object Store.

The only optimisation performed after translation from the IF1 is dead code elimination. Low level optimisers, equivalent to those for the MDFM, have not been implemented for the CSIRAC II, resulting in significant penalties for Gauss and Shallow.

The IF1 to CSIRAC II code translation flags used were:

-a use structure stores (default - transmitted stateless structures)

---

5 In this case, the code size increased by 1560% due to recursive inlining.

6 This ratio has its failings as a measurement, but it has nonetheless been used in the past [Böhm90a].

-p use pipe-lined loops (default - dynamically unravel loops)

-e1 all nodes comprising a loop or function body for a given colour execute in the same processor

The translation flags used for each benchmark were:

Gauss, Shallow	-a -p
Aq, Mm	-p
Queens	-p -e1

The following additional optimisations were performed manually for the Mm benchmark in an attempt to explore the CSIRAC II performance on vector operations:

replaced sdot function with the inner product intrinsic function;

performed constant lower bound optimisation;

removed code associated with SISAL Fibre compatible output formatting.

4. Analysis of Results

Two comparative experiments were conducted on the three systems. The first compares the total work performed (tokens created), and the second examines the locality behaviour of the three systems.

4.1. Total Work

It is clear that the 'stateless' computational model (as in SDFA and transmitted structures versions of CSIRAC II codes) carries with it some specialised costs. These costs manifest themselves in two ways. First, there is the potential that additional copies of data structures might be required that are not required in state based solutions. Second, since 'stateless' data structures are passed in their entirety through function boundaries, rather than just the pointers, the re-colouring data structure elements could prove costly.

There is reason to believe that the impact of these factors is not excessive. In most Data-Flow machines, the bulk of the extra copying and colouring must be performed anyway. However, the part of the machine performing the colouring may vary depending on the computational model. For example, in the MDFM colouring of data structures is done in the structure stores rather than in the processing element. In CSIRAC II colouring, where necessary, is performed by the processing-elements. Also, these extra costs could be partially compensated for by inlining techniques, and balanced against the costs of garbage collecting in the structure memories. A comparison between the SDFA, the MDFM and CSIRAC II is, nonetheless, warranted.

4.1.1. Results

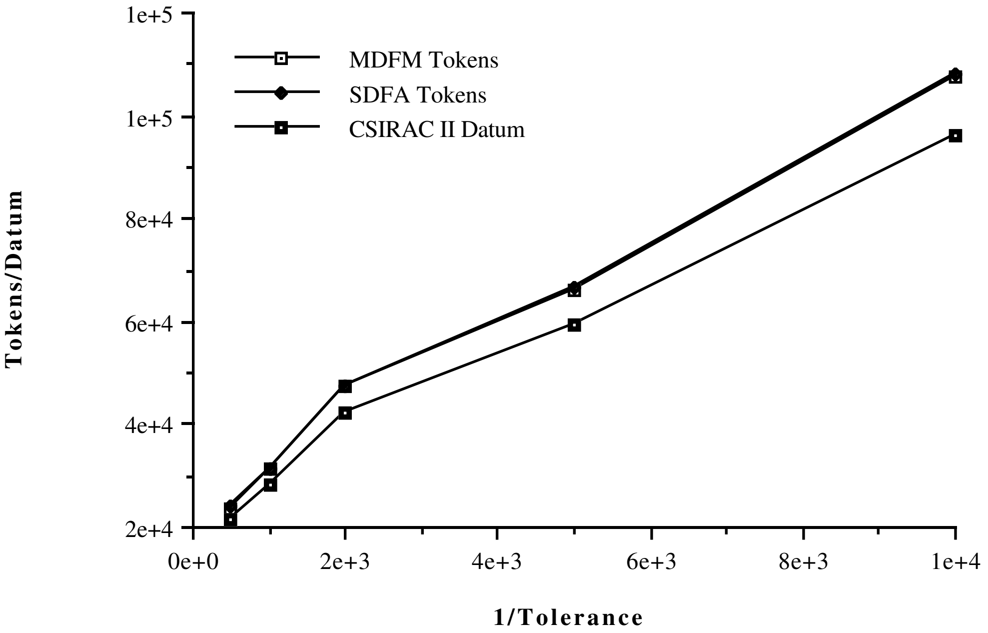

Figure 1: Token Count in Adaptive Quadrature (Aq).

The CSIRAC II performs slightly better than the MDFM or SDFA on Aq. This is due in part to the local throttle mechanism used by the CSIRAC II.

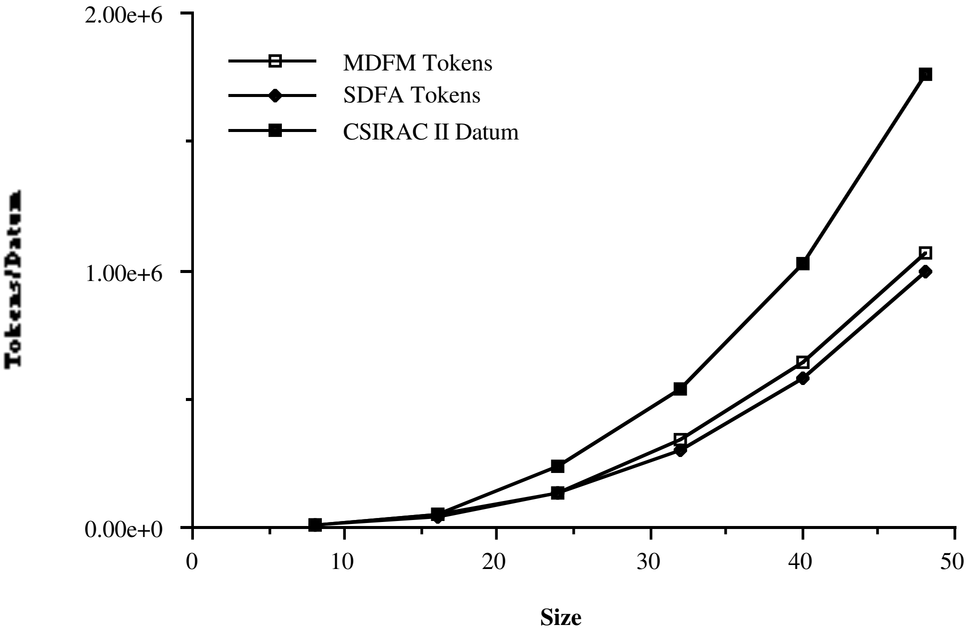

Figure 2: Token Count in Gaussian Elimination (Gauss).

The CSIRAC II performs significantly worse on Gauss due largely to excessive copying used to perform the array concatenate operation. Lack of a low level optimiser contributes to the relatively poor performance.

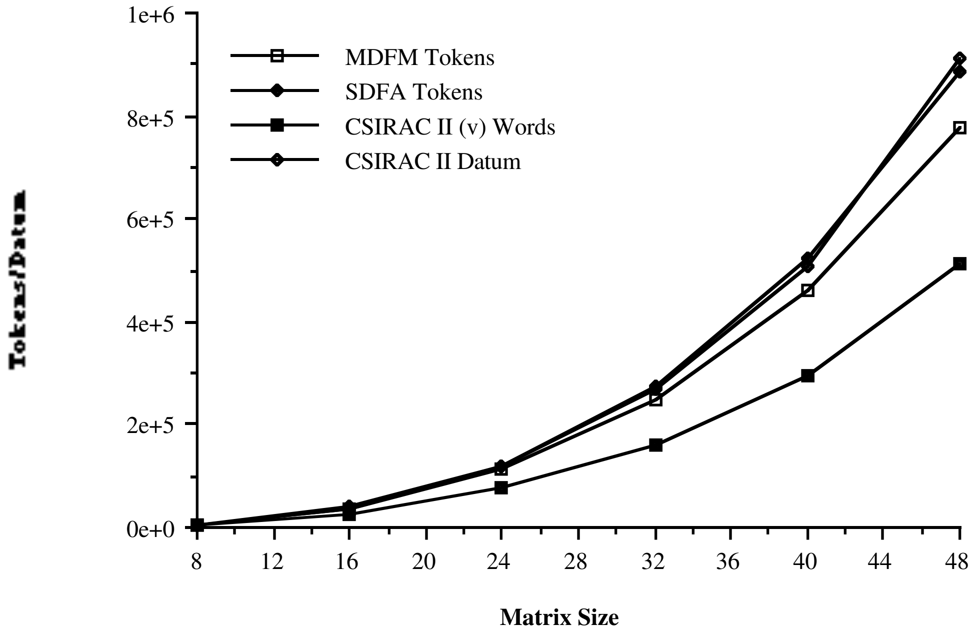

Figure 3: Token Count in Matrix Multiply (Mm).

The supplementary curve plot for the CSIRAC II is for where the comparison metric would be network words rather than datum. As can be seen aggregation of datum into a token with a single tag can be advantageous.

Two dimensional arrays are not first class constructs in SISAL, rather they are represented as an array of pointers to one dimensional arrays.[7] This has resulted in a penalty for the MDFM and would have for the CSIRAC II if it had used its Object Store. In this case, the CSIRAC II has, with modest hand optimisation capitalised, on its vector tokens and associated vector instructions while using entirely transmitted data structures.

---

7 This is partially an artefact of the base language, SISAL, which supports this model for arrays. However, if the execution model supported multidimensional arrays, a compiler could convert this to a multidimensional array representation.

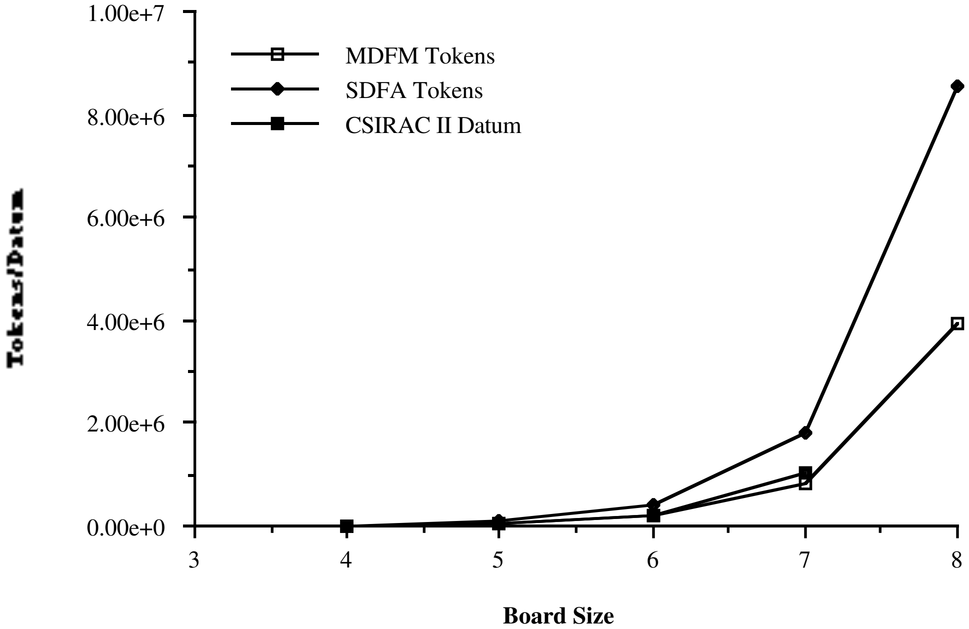

Figure 4: Token Count in the N Queens Problem (Queens).

The final point in the CSIRAC II curve is missing due to simulator limitations.

In all cases, the rapid growth in computational load with board size results in a sudden increase in the number of tokens created. Because of this exponential growth, a small difference in the number of tokens created in the multiply recursive function causes a substantial difference in the final token count. This accounts for the failure of the SDFA system to track the other two as closely as it does in the other cases.

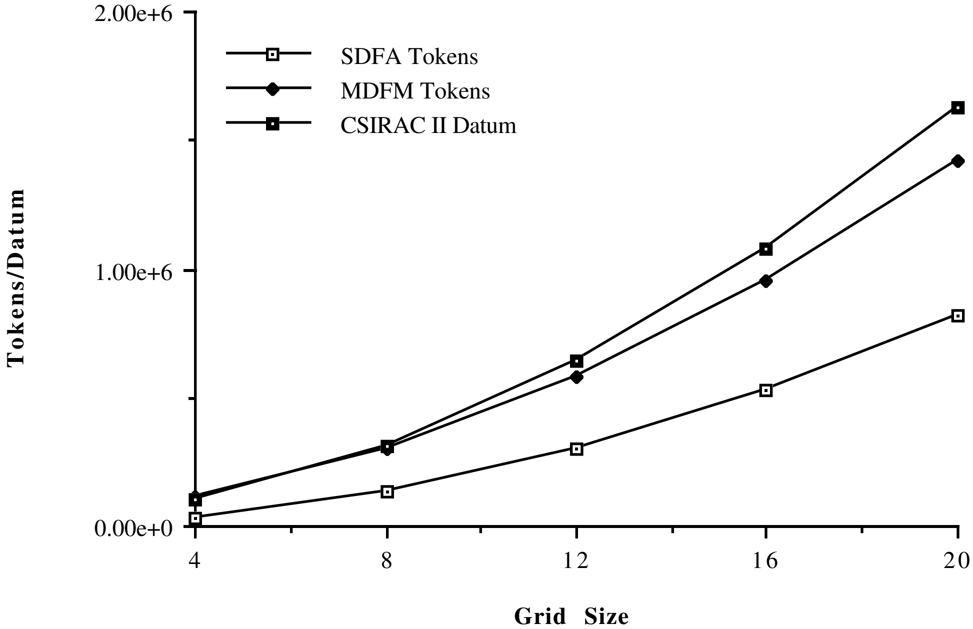

Figure 5: Token Count versus Grid Size in the Shallow Water Equations (Shallow).

Again the MDFM and the CSIRAC II have incurred a significant cost through the use of SISAL array representations. The lack of a low level optimiser has further penalised the CSIRAC II. For Shallow the number of time steps was fixed at 8.

4.2. Locality

One of the fundamental motivations behind the SDFA and CSIRAC II architectures is to benefit from the distributed memory model. This experiment compares the amount of work performed locally for each of the three machines.

4.2.1. Method

In most cases, the largest versions of the programs from above are run on a 16 processor MDFM, CSIRAC II, and SDFA systems. The exception is Queens which case a board size of 7 is used. As above, highly optimised versions of the MDFM and CSIRAC II codes are used. Only the original versions of the SDFA codes are used.

For the purposes of this study, the important part of the MDFM traffic is that which travels to and from the structure stores, because it is this that will always be global,[8] regardless of what locality facilities are incorporated in the MDFM. The same applies to the CSIRAC II when using stored structures.

For the SDFA system two measurements are taken, the total number of tokens created and the number of these which travel between processors. This is used to compute the percentage global traffic. For the MDFM and CSIRAC II three measurements are taken, the total number of tokens created, the number of these which travel over the switch network, and the number travelling to or from the structure stores. These values are used to compute the percentage global traffic and the percentage structure store traffic.

4.2.2. Results

The results obtained from the percentages study are presented in figures 6 and 7. The percentages, in figure 6, represent the fraction of all created tokens which were transmitted across the network of each machine, i.e. the percentage of non-local token traffic. The percentages, in figure 7, represent the fraction of all created tokens which were transmitted to or from the structure memories. These are the tokens that must be transmitted across the network, due to the implied shared memory model of stored data structures.

---

8 In this 16 processor experiment, an average 1/16-th of this traffic could be counted as local if the structure stores were integrated into the processing elements. It would be difficult to arrange for more than 1/16-th of the structure store traffic to be local, since data structures are colourless, see Chapter 2. Therefore, all structure store traffic is considered global.

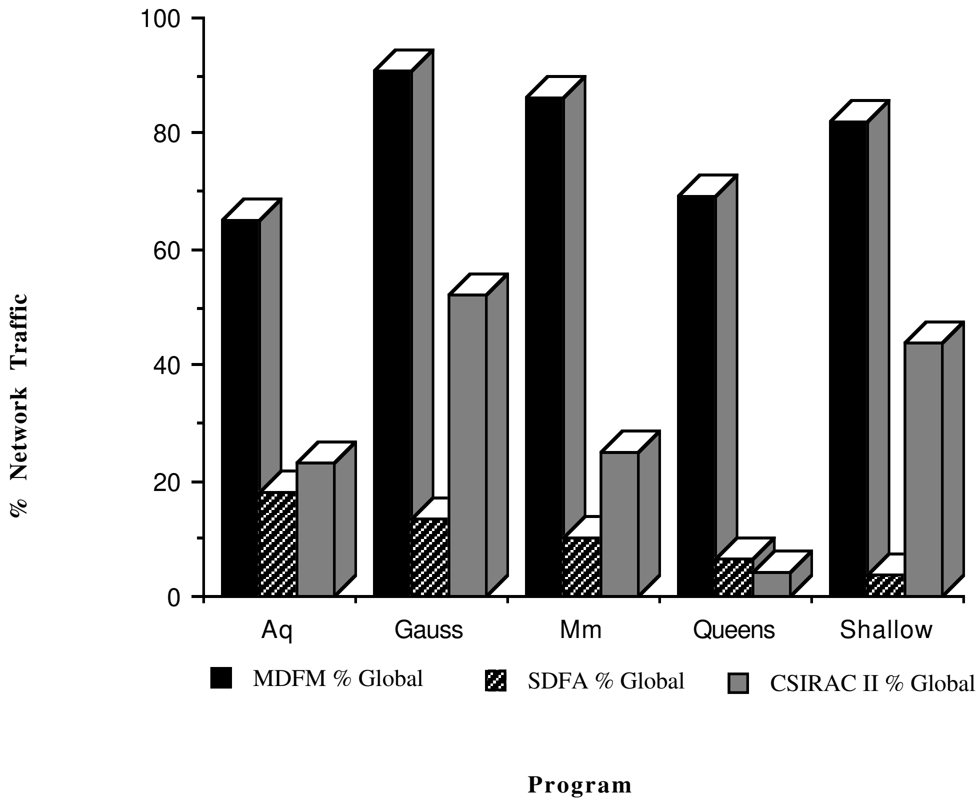

Figure 6: Network Token Traffic.

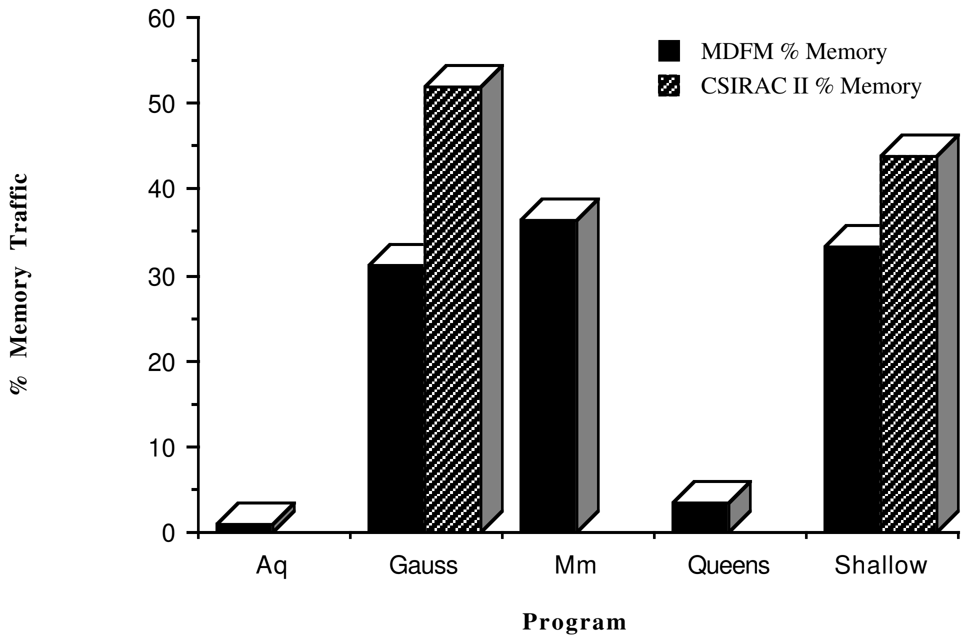

Figure 7: Network Structure Store Token Traffic.

The low figures for the MDFM and CSIRAC II memory traffic on Aq and Queens, are due to the nature of the programs. For Aq MDFM uses the structure stores only for activation name recycling counters, and Queens has one array for each board, a maximum of 7 elements long. The CSIRAC II does not use the structure store for Aq and Queens.

It is acknowledged that the global traffic in the MDFM is a worst case, due to the distribution function and basic architecture.[9] The structure store traffic, on the other hand, represents a best case since it assumes that <u>all</u> non structure store traffic is local.

5. Conclusions

First, it is possible to compare experimental systems with diverse architectures using a meaningful, simple metric which is not rendered useless by the experimental nature of the systems.

Second, although mainstream Data-Flow computing assumes the existence of a specialized structure memory, the provision of such is not a requirement. The stateless operation of the systems above did not cause substantial reductions in performance.

Third, it has been argued recently, see [Culler92], that the increased processor state required to support latency hiding in the traditional Data-Flow systems will eventually force either a limit on the scalability of the system (due to increasing latency) or a reduction in performance (due to the cost context switching). A multiple level memory hierarchy, which can be exploited through locality in computations, reduces the impact of this argument. In two of the systems presented here (SDFA and CSIRAC II), a memory hierarchy exists and the systems employ a variety of techniques to extract locality from the computation. Although there is a barrier inhibiting the scaling of traditional Data-Flow systems, there is no fundamental limit.

Lastly, in the context of evolving work on multi-threaded architectures, the tokens created locally to the processor may be regarded as being equivalent to register transfers, thus providing us with the multi-threaded equivalent of *Tokens Created*. Aggregation of low level instructions into threads is still not well understood, but what seems clear is that in practical codes the larger the multi-processor configuration the smaller the aggregations must be to permit effective load balancing and work distribution. Tokens passing across the network are equivalent in both cases, and as hardware designers, we know that this traffic is an essential parameter in sizing the communication network. Unlike *Tokens Created* or its multi-threaded equivalent, the more conventional measure of instruction count is of almost no use in this critical element of multiprocessor design.

Bibliography

[Abramson91]
D.A. Abramson and G.K. Egan, 'Design of a High Performance Data Flow Multiprocessor', in Advanced Topics in Data-Flow Computing, Prentice-Hall, 1991, pp121-141.

[Arvind88]
Arvind, D.E. Culler, and G.K. Maa. Assessing the Benefits of Fine-grained Parallelism in Data-Flow Programs. Tech. Rept. 279, Computational Structures Group Memo, Massachusetts Institute of Technology, Cambridge, Mass. (March 1988).

---

[9] To improve locality in the MDFM, the destination node address could be removed from the distribution function. It should be noted that the latter could have an impact on the machine's load balancing ability.

[Cann88]
D.C. Cann and R.R. Oldehoeft. Reference Count and Copy Elimination for Parallel Applicative Computing. Tech. Rept. CS-88-129 Colorado State University, Fort Collins Colorado (November 1988).

[Cann89]
D.C. Cann and R.R. Oldehoeft, 'Compilation Techniques for High Performance Applicative Computation', Technical Report CS-89-108, Colorado State University, May 1989.

[Culler92]
D.E. Culler, K.E. Schauser, and T. von Eicken. Two Fundamental Limits in Data-Flow Multiprocessing. Tech. Rept. UBC/CSD 92/716 Computer Science Division, University of California, Berkeley, Elsevier (1992).

[Egan79]
G.K. Egan, 'Data-flow: Its Application to Decentralised Control', Ph.D. Thesis, Department of Computer Science, University of Manchester, 1979.

[Egan81]
G.K. Egan and Richardson C.P., 'Object Recognition Using a Data-flow Computing System', EuroMicro Microprocessing and Microprogramming 7, North-Holland, 1981.

[Egan85]
G.K. Egan and Richardson C.P., 'Manipulator Control using a Data-driven Multi-processor Computer System', invited paper, Mechanical Engineering Transactions of the Institution of Engineers, Australia, Vol. ME10,No. 3, Sept. 1985.

[Egan87]
Egan, G.K., 'The RMIT Data Flow Computer: Token and Node Definitions', TR 112 060 R, Department of Communication and Electronic Engineering', Royal Melbourne Institute of Technology, Jun. 1987.

[Egan91]
Egan, G.K., N.J. Webb and A.P.W. Bohm. 'Some Features of the CSIRAC II Dataflow Machine Architecture', in Advanced Topics in Data-Flow Computing, Prentice-Hall 1991, pp143-173.

[Gurd87]
J.R. Gurd, C. Kirkham, and W. Bohm. The Manchester Dataflow Computing System. In *Experimental Parallel Computing Architectures*. North-Holland, J.J. Dongarra, Ch. 6, Amsterdahm (1987) pp. 177-220.

[Gurd92]
J.R. Gurd and D.F. Snelling. Manchester Data-Flow: A progress Report. In *1992 International Conference on Supercomputing,* ACM, ACM Press (1992) pp. 216-225.

[Hoffmann88]
G.R. Hoffmann, P.N. Swarztrauber, and R.A. Sweet. Aspects of Using Multiprocessors for Meteorological Modelling. In *Multiprocessing in Meteorological Models,* G.R. Hoffmann and D.F. Snelling eds., ECMWF, Springer-Verlag, Berlin (1988) pp.125-196.

[Kawakami86]
Kawakami, K. and Gurd, J.R., "A Scalable Dataflow Structure Store", ACM Comp. Arch. News 14/2, 1986, pp. 243-250.

[Nikhil87]
R.S. Nikhil. *Id Nouveau: Reference Manual Parts I and II*, MIT Laboratory for Computer Science, Cambridge, Mass. (April 1987).

[Sargent85]
J. Sargent and W. Böhm. *IF1 Compilation System User's Guide*, Department of Computer Science, University of Manchester, 2nd (October 1985).

[Sargent86]
J. Sargent and C.C. Kirkham. Stored Data Structures in the Manchester Dataflow Machine. Tech. Rept. UMCS-86-4-3 University of Manchester, Dept. of Computer Science (1986).

[SISAL85]
SISAL: Streams and Iteration in a Single Assignment Language, University of Manchester, LLNL, DEC, and Colorado State University, Manchester, Version 1.2 (January 1985).

[Snelling93]
Snelling, D.F., The Stateless Data-Flow Architecture, Ph.D. Thesis, Dept. Comp. Sci., Univ. Manchester, Technical Report Number UMCS-93-7-2, (July 1993).

[Teo91]
Y.M. Teo and W. Böhm. Resource Management and Iterative Instructions. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, J.L. Gaudiot and L. Bic eds., Ch. 18, Englewood Cliffs, N.J. (1991) pp. 481-499.

[Webb93]
N. Webb, 'Implementing an Applicative Language for the RMIT/CSIRO Dataflow Machine', Laboratory for Concurrent Computing Systems, Swinburne University of Technology, *PhD Thesis in preparation*, 1993.