Control of Parallelism in the Manchester Dataflow Machine

Carlos A. Ruggiero and John Sargeant
Department of Computer Science
University of Manchester
Manchester M13 9PL
England

Abstract

Fine-grain parallel machines, such as tagged-token dataflow machines, allow very high degrees of program parallelism to be exploited for many applications. In fact, so much parallelism can be generated that it is necessary to *control parallelism* in order to bound store usage.

This paper reviews software mechanisms for parallelism control, which rely on merely planting extra code to control execution order. Such methods are found to be inadequate, so a fundamental architectural mechanism known as a **throttle** is considered necessary. Various attempts to design a throttle for the Manchester Dataflow Machine are described. The eventual solution, a coarse-grain, process-based throttle, is explained, and simulation results are presented which demonstrate its effectiveness.

1 Background and Terminology

1.1 Introduction

In the future, computers will have to rely on a high degree of parallelism exploitation if large increases in speed are to be achieved. Numerous architectures for parallel machines have been proposed, but there are a number of unsolved problems and it is still not clear whether wide-purpose parallel computers will ever be cost-effective.

One of the principal problems in some architectures is excessive usage of resources for highly parallel programs. This paper describes the work done in the Manchester Dataflow project on this problem. We present not only our final solution, but also some of the ideas which were tried and found inadequate along the way.

1.2 Our Eventual Aim

The sort of parallel processing system which we would regard as ideal would have the following basic properties:

- **Wide-purpose hardware** - the machine should be able to execute a rich variety of application programs efficiently.
- **Wide-purpose language** - there should be a powerful, expressive language, suitable for many sorts of computation. This language must be implementable efficiently, and must be able to exploit the full potential of the hardware.
- **Simple, cost-effective hardware** - The cost per processor should not exceed the cost of a serial processor by more than a small constant factor.
- **Linear scalability** - A machine composed of N hardware units should be N/(N-1) times faster than one with only N-1 units.

It is quite clear that the above properties are difficult to achieve, and certainly no current system achieves them, but they give us some criteria for assessing any real parallel machine.

1.3 Granularity

The granularity of a parallel machine is the size of the units by which work is allocated to processors. Conventional multiprocessors are **coarse grain**; work is allocated by processes. Dataflow machines are **fine-grain**; the allocation unit is a dataflow single instruction. Architectures with intermediate granularity are now emerging, such as the Flagship rewrite-rule machine which is also being designed at Manchester [WaWo85].

Fine-grain machines tend to be easily scalable because they can exploit a very high proportion of the total parallelism available in a program, and because they can use simple hardware mechanisms to distribute work evenly between processors. (See [BaGu85] for a description of the method used in the Manchester machine). However, this scalability is achieved at the cost of extra hardware complexity.

Coarse-grain machines are very cost-effective in hardware, since long sequences of instructions can be executed serially without any overhead. On the other hand, they do not exploit all the available parallelism, and it is very difficult to design a coarse-grain machine which is both wide-purpose and scalable.

Even in a machine which is basically fine-grain, there is usually some notion of processes, and some operations may be performed in a coarser, process-based manner.

1.4 Dataflow Architecture

It is not the purpose of this paper to explain dataflow principles or the architecture of the Manchester Dataflow Machine (hereafter MDFM). These are described in numerous papers, e.g. [GKWa85,

Gurd85,GWGl78]. However, two particular points are explained here because they are necessary to understanding the rest of the paper.

The first point is the meaning of a "process" in our machine. The MDFM implements **tagged token** dataflow, in which each token (ie. each item of data which flows) is tagged with an **activation name** (AN), in order to distinguish it from other tokens flowing through the same piece of code. An activation name may correspond to an invocation of a function body, or of a loop cycle. We will refer to such a task as a process. Typically, a process contains 100 - 1000 dataflow instructions.

The mechanism for allocating activation names must ensure that no two processes executing the same code at the same time can possibly be allocated the same one. In the current hardware, this is achieved by merely incrementing a counter on each allocation. Clearly, this counter can overflow, and it is eventually necessary to recycle ANs. This implies that a mechanism for detecting the termination of a process is required.

The second point is the function of the various stores in the machine. The machine consists of **processing rings** (actually only one in the prototype machine) and **structure stores**, connected by a switching network. The structure stores are fairly similar to conventional stores, and hold the ordinary stored data structures of the program [SaKi86]. A processing ring contains several stores, two of which are significant here. The **token store** is simply a queue of tokens waiting to be processed. The **matching store** collects together matching pairs of tokens (i.e. tokens with the same AN flowing to the same instruction). It operates in a pseudo-associative manner [SiWa83], and its speed largely determines the maximum performance which a processing ring can attain. The token store and matching store represent the main storage overhead in the machine, and their occupancy is a key factor in determining the cost-effectiveness of the architecture.

1.5 Control of parallelism

Many programs have (for at least some of the time) **TOO MUCH PARALLELISM**; orders of magnitude more than that available in the machine. If not controlled, excess parallelism causes excessive store usage. [1] This effect has been observed in several places, for instance at MIT [ArCu85], in the Japanese dataflow projects Sigma-1[Shim86], DFM[Amam86] and PIM-D[ItoN86], in the ALICE reduction machine [DaRe81] and in several candidate architectures which have been simulated for Flagship rewrite-rule machine [WaWo85]. However, in most cases these observations have not been published.

Consider the execution of a program as a tree. Serial machines traverse the tree in a depth-first manner. This usually requires very little store apart from that used for the global data structures of the program. The "natural" execution order for a parallel machine is breadth-first; anything which *can* be done in parallel *is* done in parallel. Unfortunately, this requires an amount of store proportional to the total area of the

[1] Readers not fully convinced that there *is* a problem should take a glance at the simulation results in section 5 at this point; they leave little room for doubt!

tree to hold intermediate results. What is needed is **limited-breadth** execution; go breadth-first until the machine is busy, and depth-first thereafter.

Ideally, we would like a mechanism to turn the level of activity in the machine up or down at will, in order to dynamically match the parallelism in the program to the resources available in the machine. We will call such a mechanism a **throttle**[2]. The rest of this paper describes the hunt for an effective, implementable throttle for the MDFM.

2 Defining the problem

For many programs, the usage of the temporary token stores, the token store and matching store, is excessive compared to the space used in the structure store to hold the actual data structures of the program.

Besides excessive parallelism, there are other effects which contribute. Dataflow works by *eager evaluation.* Everything that can be executed will be and intermediate results can be produced long before they are really needed. These results will wait a long time in the stores, increasing their occupancies.

A particularly nasty case occurs in certain loops. Consider the outermost loop of a typical numeric problem, which does sums until either some convergence test is satisfied or a limiting number of cycles is reached:

```
for initial
Cycles :=0;
Converged := false;
repeat
Cycles := old Cycles + 1;
.....%lots of complex calculations
until (cycles=N) or Converged
.....%lots of results returned
end for
```

In general, tagged-token dataflow systems attempt to **unfold** loops like this, to try to execute loop cycles in parallel whenever possible [ArGo78]. In this case, the convergence test is calculated near the end of each cycle, so the (K+1)th cycle cannot start before the Kth has ended. If the convergence test is removed, however, the (K+1) cycle can start well before the Kth has terminated. In fact, the values of the variable Cycles are calculated very rapidly and so all the cycles start at about the same time. The store usage is proportional to N, the number of cycles. Disaster!

From the above we conclude that for any throttle to work, it must :

[2] As far as we know, this use of the word "throttle" was originated by Arthur Veen at a dataflow workshop in 1982.

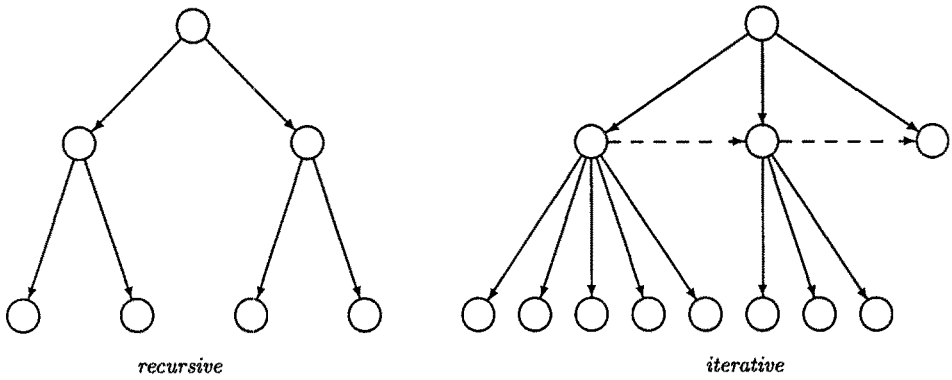

Figure 1: Recursive and iterative process trees

1. Limit the activity in the machine : excessive parallelism implies excessive usage of store.

2. Reinforce locality : related instructions or processes should be executed close together in time to limit the number of results waiting in the matching store.

Before considering methods to implement a throttle, it is useful to show how the execution of a dataflow program can be viewed as a tree of processes. Such trees tend to take two different forms; one for iterative programs and one for recursive ones. Consider the two trees in figure 1.

The nodes represent portions of computations e.g. function activations or loop cycles. A solid arc represents the relationship between a "father" process and its "child" (ie the father creates the child). Dashed lines represent data dependencies between "brothers". Such dependencies occur most often between loop cycles, but there may also be others, as in:

Y = F(X);

Z = G(Y)

The recursive tree is typical of a divide-and-conquer algorithm. It is quite deep and very wide at the bottom. The iterative tree represents a loop with parallel **forall** executions within each cycle. It is not very deep but can be extremely wide.

The area is of the tree is proportional to the total amount of computation. In a tree with no data dependencies, the minimum execution time for the program (assuming infinitely many processors are available) is proportional to the depth of the tree, so a wide tree with few data dependencies represents a program with high parallelism.

An iterative tree should be executed left-to-right to follow the data dependency between processes. For a simple recursive tree, left-to-right or right-to-left execution are equivalent because there is no data dependency between brothers.

3 Methods considered inadequate

3.1 Software Methods

By software methods we mean methods which do not use any support from the hardware, except maybe some way of determining the level of activity in the machine. Control of parallelism is achieved by extra code planted by the compiler. There are static software methods where the degree of parallelism is completely determined at compile time, and dynamic methods where decisions are made at runtime on the basis of some activity level information.

The problem with the iteration in the preceding section could be solved by synchronising a termination signal from each cycle with the result of the test which initiates the next cycle so that the (K+1)th cycle could not start until the Kth has terminated. This creates a bottleneck at the end of each cycle, but considerably reduces the store occupancies. We could go further with this idea by allowing two or more cycles to execute at the same time, thus avoiding the bottleneck.

Another software technique is Function Input Syncronisation (FIS). FIS means that a function will be executed only when all its input parameters are available. In the example:

D,E,F := F1(A,B,C);

X,Y,Z := F2(E,F,G,H)

F2 would not be allowed to start until E and F had been produced. In that way, F2 would not produce partial results that would increase matching store usage.

The implementation of FIS involves planting extra code to synchronise all the input parameters of a function or loop. This can be a considerable overhead if done indescriminately. In our implementation, care was taken to minimise this overhead by imposing FIS only when a new activation name was allocated (ie. only when a new process was created). Simulation results showed that FIS had a nice effect on the store usage of some programs without excessive instruction overhead.

Loop serialisation and FIS are useful techniques but they alone are not enough for an effective throttle. The problems are :

1. They cannot cope with some forms of excessive parallelism, like that found in divide-and-conquer recursive programs.

2. They do not limit the amount of activity in the machine although they tend to reduce it.

3. It is very difficult to determine statically how much synchronisation is required. Overly enthusiastic use of synchronisation can kill parallelism altogether.

Some more dynamic software methods have been suggested. One of them is the **K-bounded loops** technique proposed by Arvind [ArCu85]. The compiler analyses the code and determine the maximum

store usage for a loop cycle (This is not difficult to do for non-recursive code). At run time, the hardware decides how many loop cycles are allowed to execute in parallel from the activity level of the machine (dynamic) and the static information about maximum store usage per cycle.

There are two problems with the K-bounded loop method. Firstly it is not a general solution, since it cannot handle recursion. Secondly, there is the overhead of the extra instructions needed to control the loops.

Another idea for dynamic software throttling is to plant two types of code for any parallel program : one serial and the other parallel. The machine switches from one style of code to the other at run time, according to how busy it is. Work on program transformation [BuGu85] shows that it is possible to turn serial, singly recursive, programs into parallel, double-recursive, ones (and vice-versa) in some cases. These transformations however, are not fully automatic at present and often the programer must help the transformation process. Although this method could be useful in the future, achieving a complete solution in this way is well beyond the state of the art.

The conclusion is that despite being useful, software techniques are not enough to implement a general and effective throttle. We need some help from the hardware.

3.2 Fine Grain Hardware Methods

It was initially thought that for fine grain machines like the MDFM, a fine grain throttle would be ideal. The reason was that there would be no need for any software modification; a simple hardware mechanism would do it all for us. The Manchester Dataflow Group has spent considerable effort in that direction. There have been several suggestions to modify the architecture to implement an effective fine grain throttle. They all try to give priority to some tokens to the detriment of others. The tokens with highest priority should be those ones most likely to produce urgently needed results. In the MDFM, this could be achieved by replacing the token store with a special hardware unit which would order tokens in accordance with some simple, pre-defined rules.

A first attempt[3], was that we should turn the token store into a stack. The usual order of token processing traverses the execution tree of fig. 1 breadth-first and left-to-right. With a stack, depth-first and right-to-left execution of the tree is favoured.

The token stack worked quite well for divide-and-conquer recursive programs (see recursive tree in fig. 1). Simulation results showed considerable improvement in the store requirements of this type of program without any apparent increase at execution time. It does not work, however, for iterative programs where it even makes things worse. The reason for this is that, as we have seen, iterative programs need left-to-right execution. To favour the last cycles of a loop, as the token stack does, is exactly the wrong thing, and will *increase* the store usage. In this case, a token queue performs better.

[3]suggested and implemented by Ian Watson

Queue-like behaviour is fair and it is better for iterative programs Stack behavior is better for recursive ones. After the token stack, an intelligent token queue (ITQ) was proposed. Recall that a process corresponds to an activation name. In the ITQ, there would be a "current process" (or a set of current processes) and all tokens which belonged to that process would be queued for immediate execution. All the other tokens would be stored. When a current process runs out of tokens it is replaced by another process and all tokens of these new process are sent out of the store to execute.

It is easy to see that the critical part of the method was to select a new process to replace the one which had finished. One possibility is to always take the latest created process (this was called highest activation name rule - HAN). This produced stack behaviour, favouring right-to-left execution. Alternatively, we could always take the oldest process in the system (lowest activation name rule - LAN), favouring left-to-right execution. As might be expected, the HAN rule worked well for purely recursive program while the LAN rule was good for purely iterative ones. The first problem with the ITQ is that the rules are opposite and it is not clear how to combine them sensibly. Secondly, the hardware to implement the idea would have to be fairly complicated and fast, and therefore expensive.

There were various other failed attempts, which are described in [Ruggie]. We conclude that the problem of fine-grain throttling in a tagged-token dataflow machine is far from solved. We could not find a method which worked, but on the other hand we did not prove that it is impossible to find an effective method to control parallelism at the instruction level.

4 The eventual solution

4.1 The basic idea

The ITQ idea described in the previous section can be regarded as a first step towards a more coarse grain approach to the throttling problem. Nodes in the program execution tree can be regarded as processes and a process corresponds to a generation of an activation name. In the MDFM, an activation name is created by an instruction called GAN (generate activation name). Note that a process can contain quite a lot of computation, for instance all cycles of one loop may be one process.

If we could control the number of active processes according to availability of resources in the system we would be able to solve the throttling problem. A first suggestion for a throttle is as follows. On receiving a request for starting a new process (every execution of a GAN node), the machine would decide, based on some information of availability of resources, whether to grant a new activation name or not. If it decides not, the process would be **suspended** and would be reactivated some time later. When an active process finishes, it releases its activation name, which can be used for another process. When sufficient resources become available, suspended processes can be unsuspended. Note that we use the term *suspension* to refer to delay in initially granting an AN to a process. There is no notion of suspending a process once it is running.

4.2 Scheduling issues

We have to decide *when* processes should be suspended and unsuspended, and *which* processes should be unsuspended.

Of course, we should not suspend a process if the machine is not busy, so we need some **measure of level of activity** in the machine. Initially, we used a fixed number of leaf processes in the tree. This works quite well, but it is rather sensitive to variations in the sizes of processes. There is also possibility of deadlock in certain cases. The **length of the token queues** provides a more accurate measure, and this is what we actually use. When the token queue length reaches some specified limit, the throttle starts suspending processes. When it drops below the limit again, it starts unsuspending them. Since processes take a while to start up, it would be dangerous to unsuspend too many processes at once, since the store usage could shoot up once they "got going". We therefore impose a small delay between unsuspensions. Deadlock is avoided (apart, of course, from the ultimate deadlock of running out of store!), since it would show up as empty token queues, and this would trigger unsuspensions.

In order to enforce depth-first execution, the first child of a process is never suspended. Subsequent children may be suspended it the limit on activity level has been reached. The ideal process to unsuspend is the one in "bottom left-hand corner" of the tree, in order to enforce left-to-right execution. This seems to imply that the throttle needs an explicit representation of the whole process tree, and that a search of this tree is necessary on each unsuspension. In fact, a simpler data structure will suffice. For instance, each level of the tree can be represented as a queue of suspended processes. The next process to unsuspend is then chosen by taking the queue for the deepest existing level (depth-first) and taking the first process in the queue (left-to-right). In fact, it is not necessary to globally choose the "bottom left" process, provided that the general left-to-right, depth-first trend is maintained. A more fragmented data structure, which can be distributed among multiple throttle units, will therefore suffice. Clearly there is a tradeoff between the complexity of this data structure and the quality of throttling obtained.

Although using dynamic information, such as the token queue length, avoids the deadlock problem, we should still try to avoid activating processes which do not have all their input parameters, since that could make the throttle less effective. Suppose we have a process A which depends on B which depends on C and so on. There is little point in trying to activate C before B before A! Imposing FIS solves that problem but it is maybe too strong a condition (it introduces some overhead). Just guaranteeing an order of execution of GAN nodes consistent with the data dependencies is simple to implement at compile time and costs very little.

4.3 Variable size of processes

It is desirable to have processes of fairly uniform size. If they are too small, the overheads of the operation of the throttle will be high, and it may become a bottleneck. If they are too large, and have a high degree

of internal parallelism, the throttle will be less effective in controlling store usage.

Small processes can be avoided at compile time, by using techniques (e.g. function inlining, index-insensitive loops; details have been published previously [BoSa85]) which are already done for efficiency reasons. A minumum size of process can therefore be guaranteed most of the time. To deal with processes which are too large is rather more difficult. In principle the software can guarantee a maximum process size, although non-trivial implementation effort is required.

4.4 Termination Detection for Processes

As mentioned above (section 1.4), recycling of ANs is by itself a problem that we have to solve. Since the throttle hands out activation names, it can recycle them provided that it is provided with messages to signal the termination of processes.

Termination detection is a quite a tricky problem. Before a termination signal can be generated for a process, we must make sure that every instruction in that process has been executed; no weaker condition works. The full details of this are beyond the scope of this paper, but will appear in [Ruggie]. There are basically three possible strategies to detect termination:

1. By software: it is possible to plant code to synchronise all results produced by a process. The problems with this method are the excessive use of synchronisation (in the matching store) and the fact that the work has be repeated for every new language implemented. Nevertheless, it is the most obvious solution and most dataflow groups have adopted it.

2. Token reference counting : It is possible to simply reference count the activation names[4]. Suppose a reference count for each AN is held in a hardware table. Instructions can send messages to modify the counts. For instance, if an instruction takes two input tokens and produces one token with the same AN, it will reduce the count for that AN by 1. Of course this would be far too inefficient to implement in this form in the hardware, but it is easy to implement in the simulator, and was used to get some initial results.

3. Instruction counting : We eventually adopted a scheme which counts not tokens but instructions. Essentially, the idea is that it is only necessary to count **terminal** instructions, ie. ones which do not produce further tokens belonging to the same process. The number of terminal instructions can be determined statically, thereby determining initial values for counters which are decremented as terminal instructions are actually executed. The message overhead is much smaller than for simple token counting.

[4] this was first suggested by Rob Jarratt

4.5 Efficient hardware implementation for a scalable multiprocessor

The throttle is a special hardware unit which processes messages such as requests for ANs, activity level reports, and termination signals. It allocates ANs, and suspends and unsuspends processes; in short, it is a hardware resource manager. The throttle hardware for the prototype machine is currently being commissioned. Since it is just a message processor with store, the hardware is identical to that used for the structure store [KaGu86], with different microcode.

The throttle's data structure can be partitioned among multiple throttle units, in order to maintain the scalability of the architecture. One throttle unit should be able to service several processing rings. This, along with other load balancing issues, is currently being investigated by detailed multiprocessor simulation.

5 Simulation results

The results presented here are concerned purely with parallelism control and store usage. Results concerning other issues such as speedup and code efficiency have been published previously, in the papers referred to above.

We present results for two programs. The first is the standard, recursive, divide-and-conquer N-queens (all solutions) problem. The second is Simple, an iterative 2-dimensional hydrodynamics problem. The latter is a semi-realistic program, comprising about 1500 lines of SISAL. In each case, we use a variety of problem sizes, and show behaviour unthrottled, and with throttling using an optimum token queue length of 8. Using higher values for this parameter gives intermediate results. A more complete set of simulation results can be found in [Ruggie].

Most of the results given below were obtained using a detailed timing simulator for a single-ring MDFM. The quantities measured are as follows:

S1 The total number of dataflow instructions executed.

S∞ The length of the critical path.

π The average parallelism, $S1/S\infty$.

TSO The maximum token store occupancy.

MSO The maximum matching store occupancy.

TimSt The total number of timesteps required.

ANs The number of different activation names allocated.

Processes The total number of processes created.

unthrottled version

N	S1	S∞	π	TSO	MSO	TimSt	ANs	Processes
3	5050	543	9.3	79	246	26446	35	70
4	19005	993	19.1	223	683	88404	101	254
5	76372	1306	58.5	644	2114	344096	303	1006
6	303075	1646	184.1	2281	7865	1369041	1179	4262
7	N/A							

throttled version

N	S1	S∞	π	TSO	MSO	TimSt	ANs	Processes
3	5050	584	8.6	32	187	26356	30	70
4	19005	1329	14.3	46	366	88282	48	254
5	76372	4598	16.6	61	610	346852	64	1006
6	303075	16799	18.0	72	741	1359343	76	4262
7	1301346	70613	18.4	87	925	5816388	92	18120

Table 1: Simulation Results for the N-queens program

The results for N-queens are shown in table 1. The amount of token storage required for the unthrottled program increases exponentially with N, and indeed it was not possible to run an unthrottled simulation of 7 queens because too much store was needed. The throttle controls store usage beautifully, but does so at the cost of reducing parallelism dramatically. The total time taken to run the program changes little, and in fact tends to decrease. This effect is quite common, and is due to the *smoothing effect*; the usage of resources over time is more even in throttled execution than in breadth-first execution. The number of activation names required is also greatly reduced by throttling.

Tables 2 and 3 give results for Simple. Due to the limitations of the timed simulator, only very small data sizes could be used. To get some results for slightly larger sizes, a more idealised simulator was also used. In fact, the results from the two simulators are generally very similar with respect to throttling, so we believe the results in table 3 to be quite realistic. In table 3, TSOt, MSOt etc. denote figures for the throttled version of the program.

The results for Simple are rather different from those for N-queens [5]. Again, the store usage of the unthrottled version grows rapidly with the program size, and larger data sizes could not be simulated unthrottled. In fact, the version of the program used here was compiled with FIS. Without this synchronisation, the unthrottled store usage is even higher. The throttle does indeed control parallelism and reduce store usage considerably. However, it it not as effective as for N-queens. Store usage grows quadratically with N in the unthrottled version, and linearly in the throttled one. The problem is that there are a few very large processes in the code, due to our compiler being over-enthusiastic about optimisation. Tuning the compiler would therefore fix this.

The extra instructions planted to detect process termination add about 14% to the total for N-queens, and about 10% for Simple. However, almost all these extra instructions are single-input instructions which do not use the matching store; this reduces their real impact considerably. The analyser which plants these instructions can also be optimised to reduce this overhead.

6 Conclusion

A process-based throttle can successfully control parallelism in the MDFM, and has been integrated with a mechanism for colour recycling. The solution of the throttling problem brings the overall store usage of dataflow machines into line with that expected of conventional architectures, and so removes one of the major deficiencies of such machines.

[5] This illustrates the danger in the recent trend of baseing assessments of parallel machines on results from N-queens alone!

unthrottled version

N	S1	S∞	π	TSO	MSO	TimSt	ANs	Processes
6	113716	1731	65.7	1880	2800	551177	101	612
7	165074	1762	93.7	3286	3969	797741	156	920
8	226588	1794	126.3	4395	5660	1095401	223	1292
9	298328	1830	163.0	6114	7431	1441187	302	1728
10	380152	1862	204.2	8246	9456	1836806	393	2228
11	472441	1893	249.6	10011	11879	2283997	496	2792

throttled version

N	S1	S∞	π	TSO	MSO	TimSt	ANs	Processes
6	113716	4698	24.2	628	1494	550597	34	612
7	165074	6342	26.0	826	1903	794607	35	920
8	226588	8593	26.4	945	2403	1090701	33	1292
9	298328	11166	26.7	1064	2862	1432852	36	1728
10	380152	14002	27.1	1380	3544	1823772	35	2228
11	472441	17128	27.6	1152	4363	2263461	38	2792

Table 2: Simulation Results for Simple

N	π	TSO	MSO	πt	TSOt	MSOt
14	389.6	28621	21467	27.5	2318	6580
16	506.3	38935	28975	27.4	2994	7470
18	634.6	50833	37619	27.7	3292	9748
20	773.2	64315	47399	27.5	3909	9494

Table 3: Results for Simple using the idealised simulator

7 References

Amam86 Amamiya et al, **Implementation and Evaluation of a List-Processing-Oriented Data Flow Machine**, 13th Annual Symposium on Computer Architecture, June 1986

ArCu85 Arvind and Culler D.E., **Managing Resources in a Parallel Machine**, in Fifth Generation Computer Architectures, ed. J.V. Woods, North Holland, pp. 103-121, April 1986

ArGo78 Arvind and Gostelow K.P., **Some Relationships between Asynchronous Interpreters of a Dataflow Language**, Formal Description of Programming Concepts, ed. Neuhold E.J., North Holland, pp. 95-119, 1978

BaGu85 Barahona P.M.C.C. and Gurd J.R., **Processor Allocation in a Multi-Ring Dataflow Machine**, Technical Report UMCS-85-10-3, University of Manchester, October 1985.

BoSa85 Bohm A.P.W. and Sargeant J., **Efficient Dataflow Code Generation for SISAL**, Proceedings International Conference on Parallel Computing, September 1985

BuGu85 Bush V.J. and Gurd J.R., **Transforming Recursive Programs for Execution on Parallel Machines**, Lecture Notes in Computer Science, Vol. 201, pp. 350-367, September 1985

DaRe81 Darlington, J. and Reeve, M, **ALICE - A Multiprocessor Reduction Machine for the Evaluation of Applicative Languages**, Proc. conf. on functional programming languages and computer architecture, 1981.

GKWa85 Gurd J.R., Kirkham C.C. and Watson I., **The Manchester Prototype Dataflow Computer**, Communications of the ACM, Vol. 28, no. 1, pp. 34-52, January 1985

Gurd85 Gurd J.R., **The Manchester Dataflow Machine**, Computer Physics Communications, Vol. 37, no. 1, pp. 49-62, July 1985

GWGl78 Gurd J.R., Watson I. and Glauert J.R.W., **A Multilayered Dataflow Computer Architecture**, Internal Report, Department of Computer Science, University of Manchester, January 1978 (1st edition), March 1980 (3rd edition)

ItoN86 Ito N. et al, **The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D**, 13th Annual Symposium on Computer Architecture, June 1986.

KaGu86 Kawakami K, Gurd J.R., **A Scalable Dataflow Structure Store**, Proc. of the 13th Annual Int. Symposium on Computer Architecture, June 1986

MSAg83 McGraw J.R., Skedzielewski S.K., Allan S., Grit D., Oldehoeft R., Glauert J.R.W., Dobes I. and Hohensee P., **SISAL - Streams and Iteration in a Single-Assignment Language**, Language Reference Manual, Version 1.0, Lawrence Livermore National Laboratory, July 1983

Ruggie Ruggiero C., **Throttle Mechanisms for the Manchester Dataflow Computer**, Computer Science Dept., University of Manchester, Ph.D. thesis, in preparation

SaKi86 Sargeant J. and Kirkham C.C., **Stored Data Structures on the Manchester Dataflow Machine**, Proc. of the 13th Annual Int. Symposium on Computer Architecture, Vol. 14, no. 2, pp. 235-242, June 1986

SiWa83 da Silva J.G.D. and Watson I., **A Pseudo-Associative Matching Store with Hardware Hashing**, Proc of the IEE, Vol. 130E, no. 1, pp. 19-24, January 1983

Shim86 Shimada T. et al, **Evaluation of a Prototype Data Flow Processor of the Sigma-1 for Scientific Computations**, 13th Annual Symposium on Computer Architecture, June 1986

WaWo86 Watson, I, Watson, P. and Woods, J.V., **Parallel Data-Driven Graph Reduction**, in Fifth Generation Computer Architectures, ed. J.V. Woods, North Holland, April 1986