JOINT ROYAL MELBOURNE INSTITUTE OF TECHNOLOGY AND
COMMONWEALTH SCIENTIFIC AND INDUSTRIAL RESEARCH ORGANISATION
PARALLEL SYSTEMS ARCHITECTURE PROJECT

Numercial Examples on the RMIT/CSIRO Dataflow Machine

TR 118 082 R

G.K. Egan
N.J. Webb

W. Bohm †

Department of Communication and Electrical Engineering
Royal Melbourne Institute of Technology,
P.O. Box 2476V, Melbourne 3001,
Australia.

† Department of Computer Science,
Victoria University of Manchester,
Manchester, M13 9Pl,
England

Version 1.0 May 1989

Presented at the Workshop on Dataflow Computing: A Status Report, Eilat, Israel 1989

ABSTRACT:

The dataflow architecture of CSIRAC II encompasses both the static queued model and dynamic or unravelling model and exhibits the advantages of both.

The paper will present some distinguishing features of the architecture using as illustrations simple representative numerical examples.

Introduction

The Joint Parallel Systems Architecture Project at the Royal Melbourne Institute of Technology (RMIT) under funding from the Commonwealth Scientific and Industrial Research Organisation (CSIRO) is directed at the design, construction and application of parallel computing systems. The architecture of CSIRAC II [1][2][8] which is being constructed as part of the Project encompasses both the static [7] and dynamic dataflow schemes[5] and exhibits the advantages of both. The architecture is characterised by:

generic node functions with implicit type coercion;

sequence functions for tag and index generation;

variable length strongly typed tokens including vectors and compound tokens;

extended matching function set ;

support of heterogeneous nested lists or streams[21];

support for re-entrant graphs;

deferred and non-deferred structures;

integrated input-output;

random static allocation of nodes to processors at compile time qualified by colour at run time.

Completed application studies using the DL1 language [17] include object recognition [9], manipulator control [10], logic simulation [3] and resource allocation[4]. Application studies currently in progress include those in geomechanics and weather modelling.

This paper will present some distinguishing features of the architecture using as illustrations simple representative numerical examples.

1. Development Environment

The Project has under development several language implementations including SISAL [15], IDA [23] (an Id derivative [16]), Guarded Horn Clauses (GHC) [20] and Pascal. SISAL, IDA and Pascal are targetted on IF1 [18] while GHC and IF1 are targetted on i2 [11]; i2 is a graph assembler at the CSIRAC II instruction set [12] level. The availability of several back end translators for IF1 permits comparative studies of languages independant of the dataflow hardware. The i2 back end generates code for a multi-processor based instruction set emulator and the CSIRAC II hardware which is currently under construction [1] (Figure 1.1).

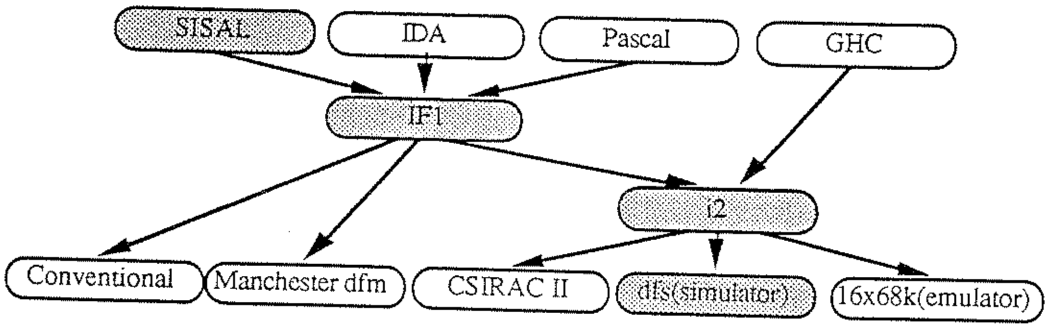


Figure 1.1 Development Environment

2. Computing Π

The computation of Π by numerical integration was used by Babb [6] to assess informally some parallel machines and their language systems. The recursive divide and conquer scheme is used to illustrate how graph re-entrancy is supported, while the loop solutions illustrate some tradeoffs between loop unravelling and streaming.

2.1 Recursive Solution

The first solution is by recursive binary subdivision of the integration interval. A SISAL program which implements this is given below:

```
% compute pi by recursive binary subdivision of integration interval
define recursive
function recursive(returns real)
function Area(D: integer; dx, L,R: real returns real)
 let
  Mid := (L + R) * 0.5;
  NewD := D / 2;
 in
   if D = 0 then
     (R - L) * 4.0/(1.0 + L * L)
   else
     Area(NewD, dx, L, Mid) + Area(NewD, dx, Mid, R)
   end if
 end let
end function
 let
  Rectangles := 1000;
  dx := 1.0 / Real(Rectangles);
 in
   Area(Rectangles,dx,A,B)
 end let
end function
```

The compiled graph region corresponding to bold text is shown in Figure 2.1.1. The node functions are **crc** (create colour), **evc** (exchange value and colour), **stc** (set colour), **srl** (set return link) and **e** (exit).

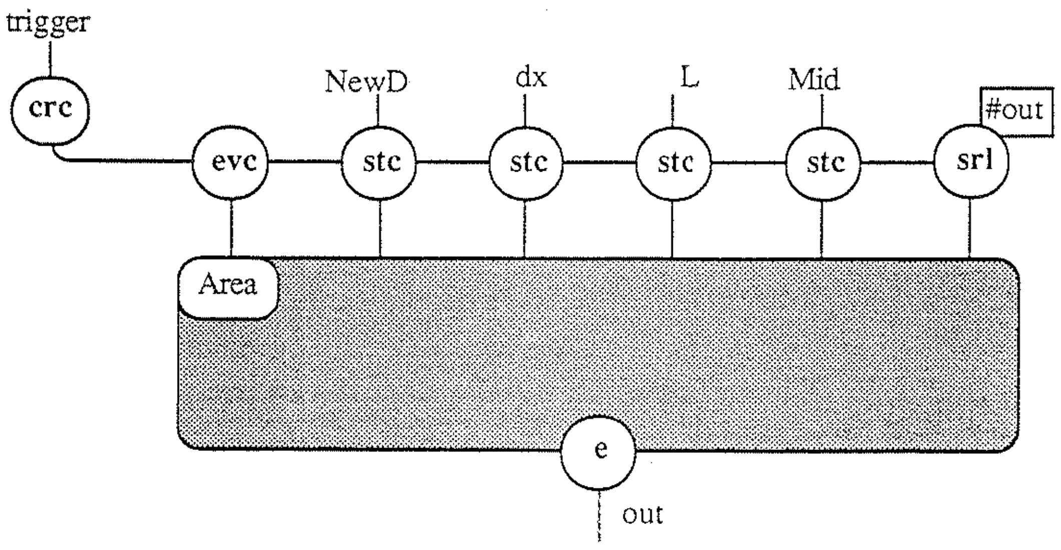


Figure 2.1.1 Recursion Fragment

Although the process is logarithmic it takes some considerable time to reach the leaves of the recursion tree where the computation of the integral takes place; this is a consequence of the complexity of the decision and new integration subrange generation combined with latency due to the number of network and processing-element stages; latency is set to 10 for all following simulations. The tail of the computation where the summation of the partial integrals occurs is comparatively fast. The upper line of the performance measures graph of Figure 2.1.2 is nominal MIPs and the lower is MFLOPS; it can be seen clearly that the peak of the floating point computation lies towards the end of the computation. The other graphs show processing- element activity and the number of tokens both in transit and unmatched over time in the system. Workload distribution is excellent as shown by the even granularity of the individual processing-element activity graph.

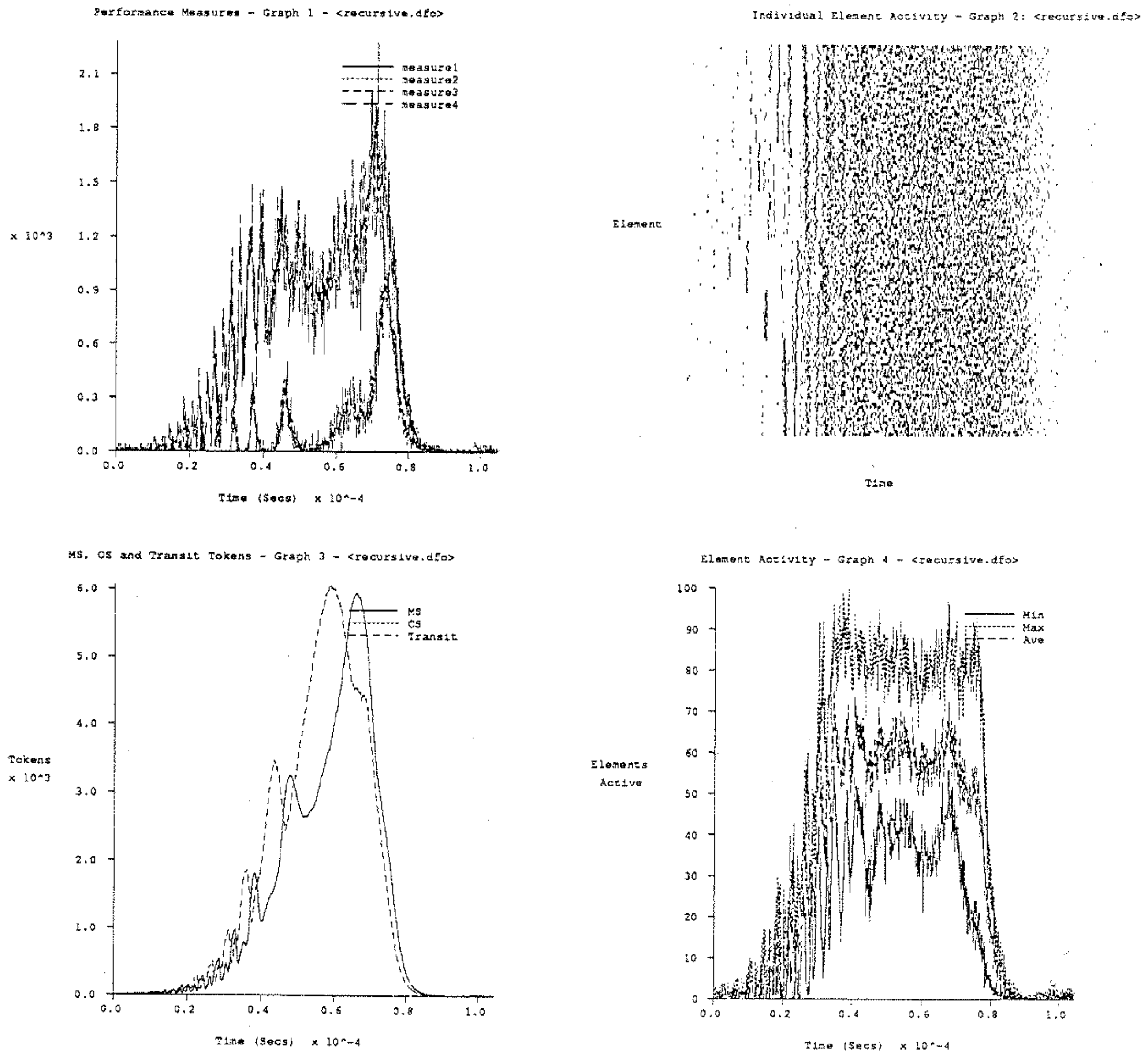


Figure 2.1.2 Recursive Solution

2.2 Loop Solutions

2.2.1 Single Forall Loop

A program for a solution using a forall loop is:

```
define loop1s
function loop1s(returns real)
 let
  Rectangles := 1000;
  Dx := 1.0 / Real(Rectangles);
 in
  for r in 1 , Rectangles returns value of left sum
    4.0*Dx / (1.0 + Dx*Real(r)*Dx*Real(r))
  end for
 end let
end function
```

The compiled graph for the forall loop is shown in Figure 2.2.1.1.

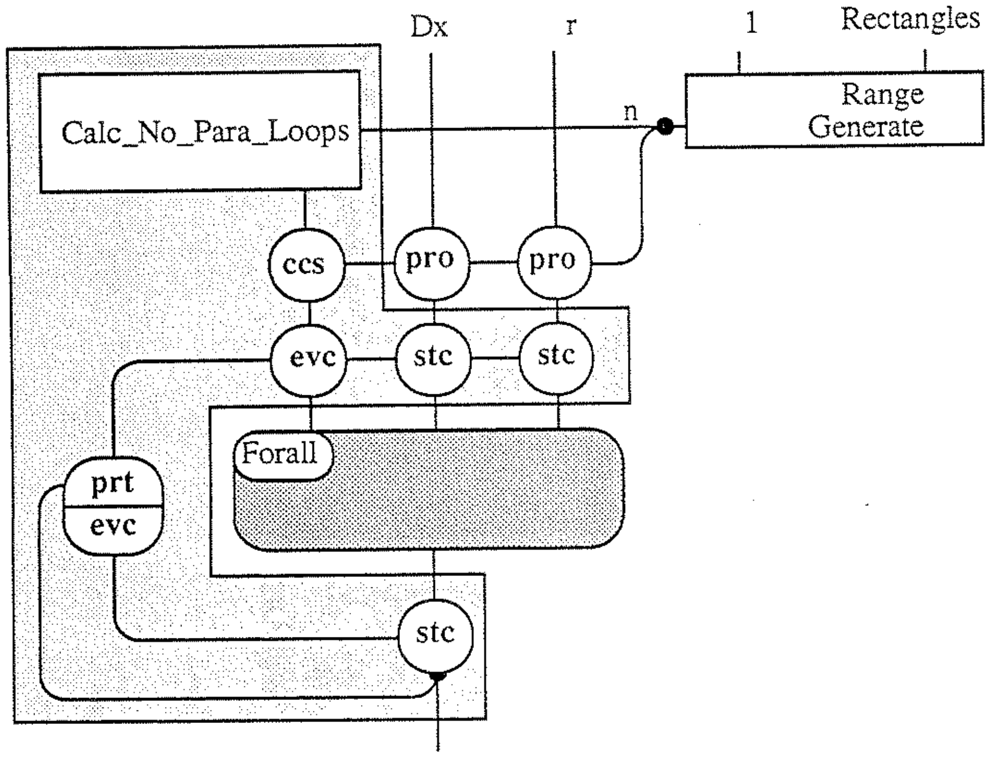


Figure 2.2.1.1 Forall Loop Fragment

The highlighted graph construction is the loop call and throttling mechanism used on the architecture. The calc_no_para_loops function determines to what degree the loop body should be unravelled or conversely to what degree data should be streamed through the loop body; this is done by examining the length of the processing-element input queue and varying the arguments of the **ccs** (create colour sequence) node to generate sequences of colours varying from all unique colours (fully unravelled) through cycles of colours (partially throttled) to one colour (fully streamed) [22]. The additional node and match functions used over and above that for the fragment of Figure 2.1.1 are **pro** (proliferate), and **prt** (protect).

It is clear that this solution will not yield good results as the "length" of the loop body is such that very few loop bodies will be active at any instant even if initiated by sequence generating functions (Section 5.). This is confirmed by the simulation results shown in Figure 2.2.1.2.

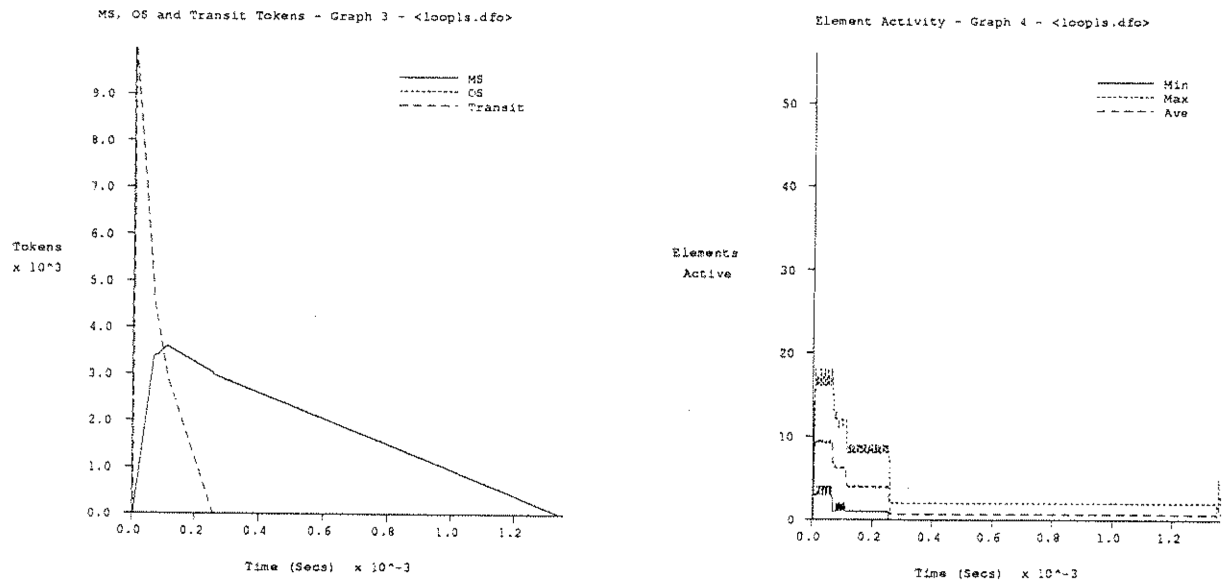


Figure 2.2.1.2 Single Forall Loop

2.2.2 Techniques for Lengthening Loop Bodies

Where the bounds of the forall loop are known at compile time, two possible strategies to increase the "length" of loop bodies are:

i) take factors of the loop range and generate a nested forall loop using these factors as bounds;

ii) statically elaborate the loop body some number of times (usually small) within a single loop of appropriately reduced range.

Both these techniques serve to increase the length of the loop body such that an appropriate level of concurrency is maintained. They can also increase code size and organisational overhead. The simulation results for both strategies are presented in Figures 2.2.2.1 and 2.2.2.2.

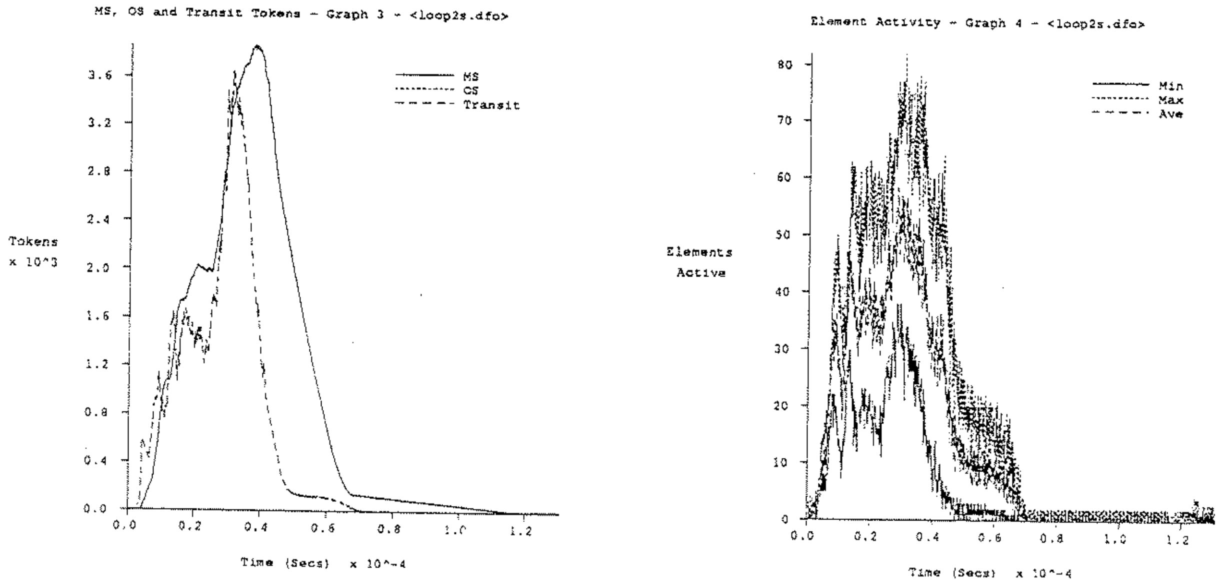
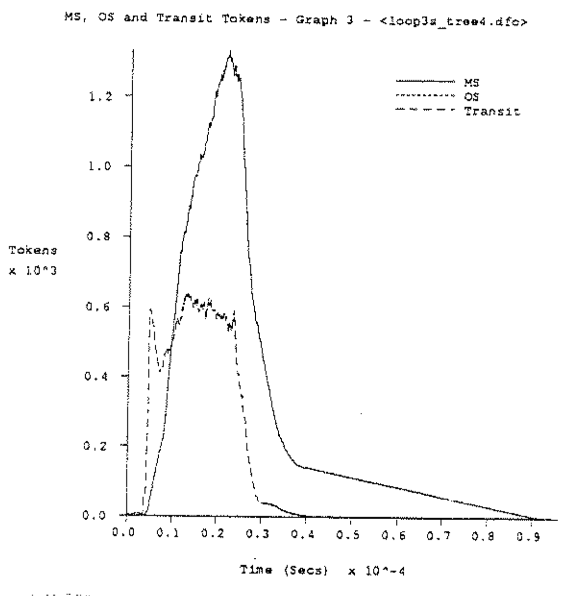


Figure 2.2.2.1 Nested Forall Loops (outer loop 50, inner 20)



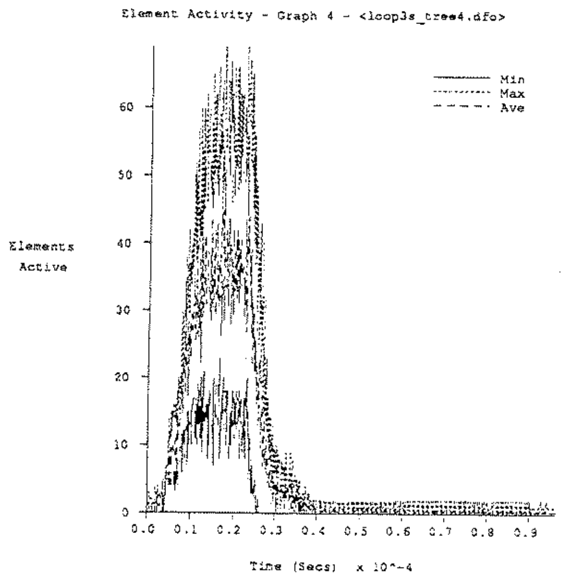
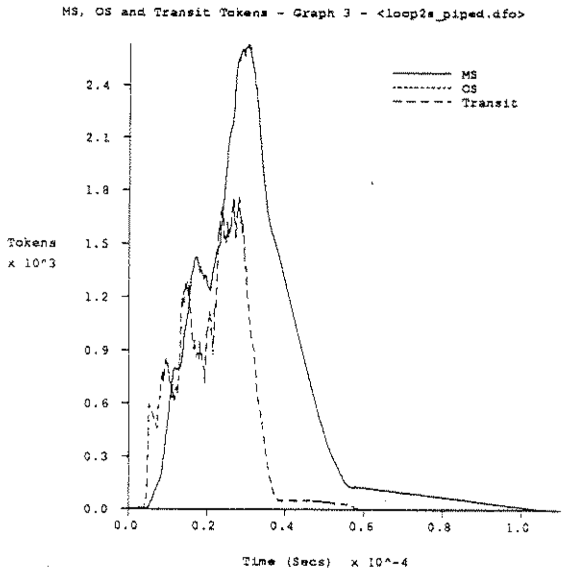


Figure 2.2.2.2 Static Elaboration

2.2.3 Streamed Iterative Solution

Removing the call and throttling code (Figure 2.2.1.1) from the inner forall loop results in data being streamed through the loop body. This constrains the maximum concurrency of the inner loop to the static node count of the loop body; many data sets may however be in transit down the arcs connecting these nodes. Throttling is now controlled by the outer loop(s). The simulation results for this case which is presented in Figure 2.2.3.1 may be compared with those in previous sections.



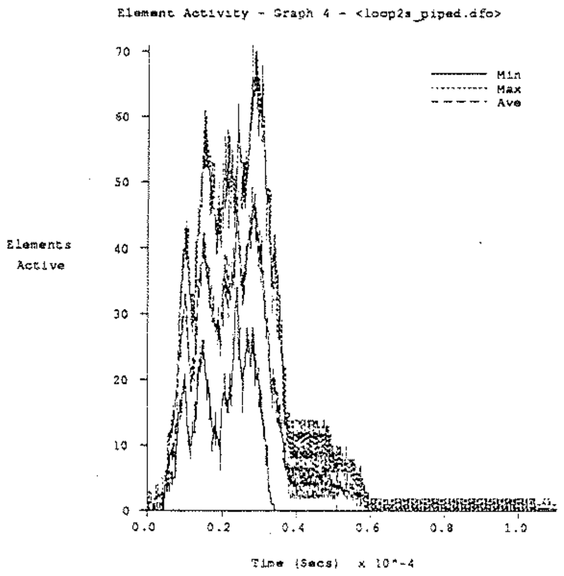


Figure 2.2.3.1 Streamed Inner Forall Loop

Left reduction was specified in the forall loop examples of Section 2.2. This results in a distinct tail to the computations which is exacerbated by the assumed latency of the system (10). If the reduction form is not specified, or if tree reduction is specified, it is possible to reduce the length of the tail significantly.

3. Lists and Vector Reductions

Figure 3.1 illustrates the combined behaviour of matching functions, object store evaluation functions and lists of tokens. The store_add (**sad**) function in combination with the normal match function (**nrm**) matches the scalar object store index with each element of the list in turn. The **sad** function forms a compound token comprised of the value, index and destination name of the result and sends it to the the object store where the values are accumulated. Arrival of the **end_of_list** token causes the **sad** to emit the sum and reset its state. No counter is necessary to keep track of how many values have been accumulated.

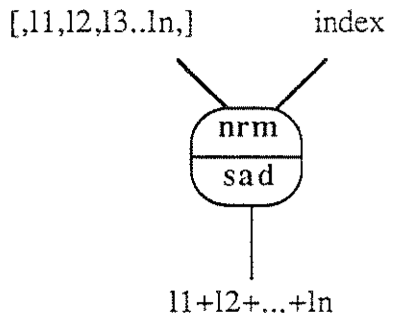


Figure 3.1 List Reduction

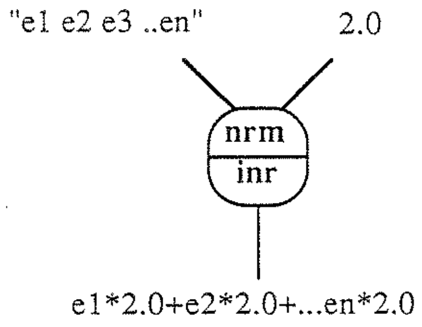


Figure 3.2 Scaled Inner Product

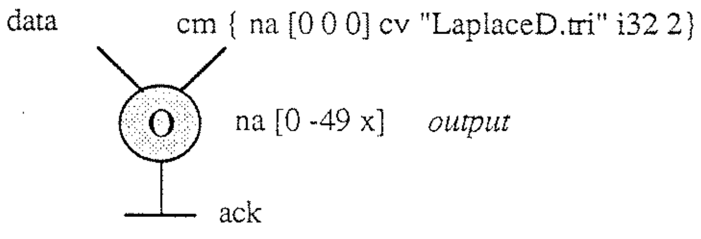
In some applications vectors may be short e.g. spatial transformations in manipulator and graphics systems. In these cases we may choose to use transmitted vectors rather than stored or list representations. The example of Figure 3.2 forms the self inner product of a scaled vector; for the more normal case both operands would be vectors.

4. Monitors and Shared Resources

The ability of the architecture to pass records or compound tokens greatly simplifies atomic transactions on shared resources. Two examples of the use of compound tokens in this context are given below. The examples are expressed in .i2.

The first example is that of 'linking' an output node with a graph arc (na [0 0 0]) and associating the node with an external disk file (LaplaceD.tri) in raw integer mode (i32 2). The code fragment is taken from a laplacian heatflow problem [13].

```
define Laplace();
const
  TriFile = 'na [0 -49 0]';              \well known name of output node
  TriFileLink = 'na [0 -49 1]';
-
-
begin
 'cm { na [0 0 0] cv "LaplaceD.tri" i32 2 }'->TriFileLink;
 'n'->Start;
 InitCAndA(Start)->Initialised;
 Laplace(Initialised);
end.
```



The second is a simple object store allocator.

```
/** Object Store Allocate definitions for node functions to be found in [12]
const
 first_free = 'i8 1';
 next_free = 'i8 0';
define *_alloc(size link);
begin
 fmc(size link) -> request;                        |make request atomic
 rcl(request) -> new_request req_colour;           |set colour to "0" but retain as req_colour
 cgt(new_request) -> req_size req_link;
 pip(next_free req_size) -> non_lit_next_free;
 prt(non_lit_next_free free_updated)->read_free;
 srd(read_free)->base_address;                     | next available free memory position
                                                   |using deferred read
 add(base_address req_size)->new_free;
 ssw(new_free next_free)->free_updated;            |update next free using direct write
 stc(base_address req_colour)->base;
 stc(req_link req_colour)->requester;              |restore colour
 stn(base requester);                              |set name to requestor
end;
define _ialloc(trigger length) -> base_address;
begin
 pip(#base_address trigger) -> link;
 _alloc(length link);
end; | _ialloc
```

The **fmc** (form compound) function concatenates the requested block size and the link, or name, to which the base address of the assigned block should be returned. The **rcl** (remove colour) function translates the atomic request into the colour space of the allocation monitor. There may be many requests non-deterministically merging at the input of the **rcl** function as depicted in Figure 4.1.

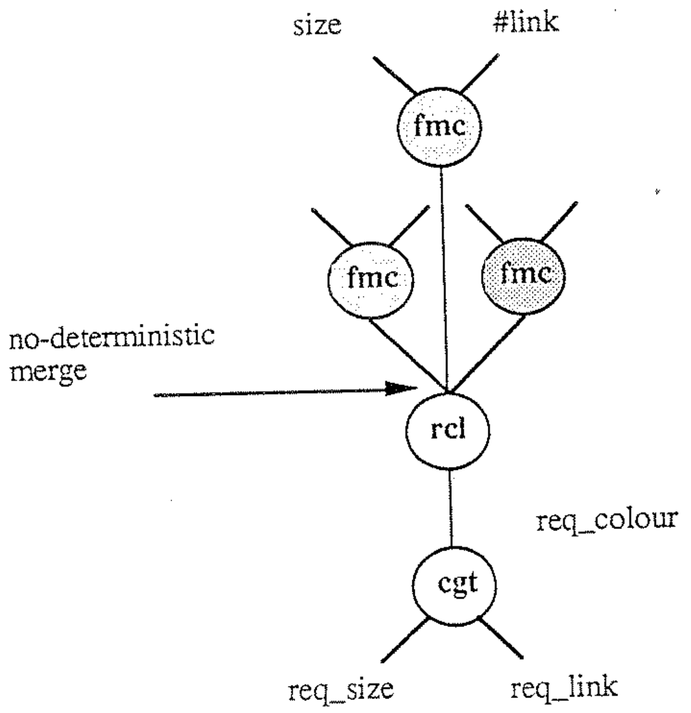


Figure 4.1 Non-deterministic Merge

5. Sequence Generators

There are three sequence generating node functions in the instruction set as shown in Figure 5.1.

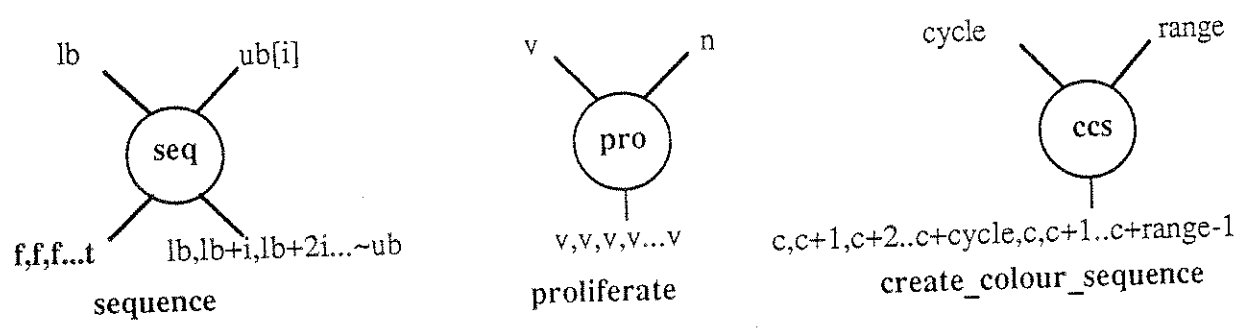


Figure 5.1 Sequence Generators

The **seq** (sequence) function takes as arguments an upper bound, lower bound and optionally step size producing two output sequences. One sequence is truth values which may be used to control gating functions and the othe is the a sequence of integers which may be used as indices or colours.

The **pro** (proliferate) function takes as arguments a value and the number of times the value should be reproduced.

The **ccs** (create colour sequence) function takes as arguments a cycle length and the number of colours to be produced.

Sequence functions may be used to quickly initiate computations (Section 6.). Some caution however is required in setting upper bounds to sequence lengths. Long sequences may produce allocation hot spots. The grains due to sequence generators may be seen in Figure 5.2.

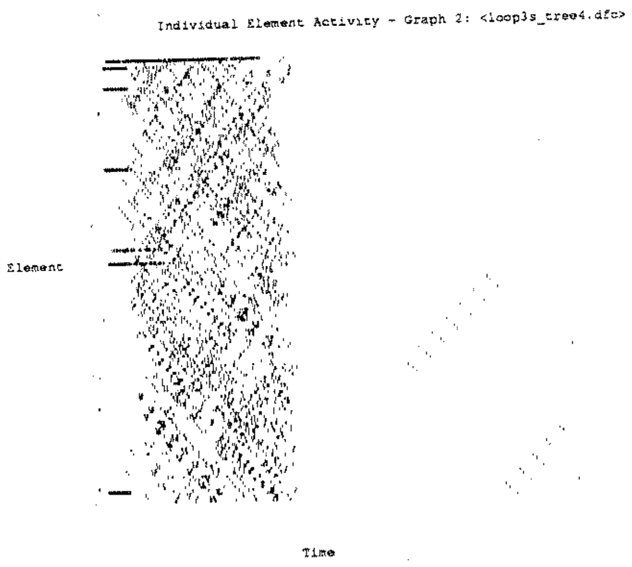


Figure 5.1 Sequence Generator Effects

6. Some Observations on Latency

It is arguably true that dataflow architectures, by virtue of overlapped communication and computation, can tolerate significant communication latency. This is not however a licence for neglecting the need to minimise critical paths in the code or the hardware. Some, possibly self evident, observations are made in the following discussion.

Graph cycles which can occur in graph fragments need particular attention; the maximum rate at which the fragment can produce tokens may be constrained to O(1/nL) where L is the combined processing-element and network delay and n is the number of nodes in the longest cyclic path. For example the allocator of Section 4.0 exhibits a loop cycle of 4L (Figure 6.1).

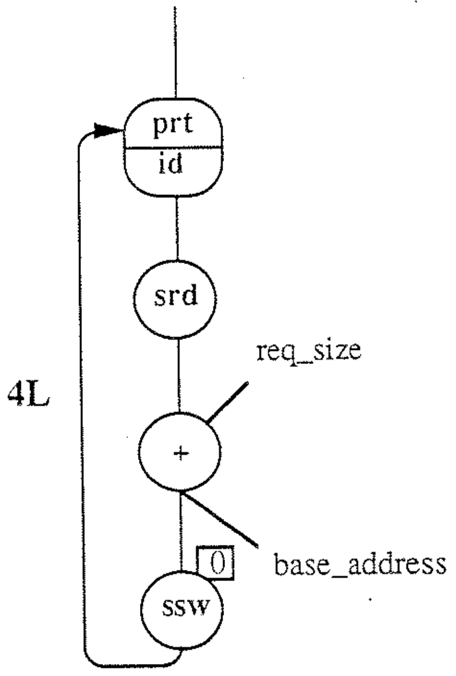


Figure 6.1 Graph Cycle from Object Store Allocator

It is of course important that simulators, such as the one used to produce the results presented here, model latency with reasonable fidelity. Figure 6.2 shows the examples of the nested forall loops (Section 2.2.2) with latencies of 1 and 10 respectively.

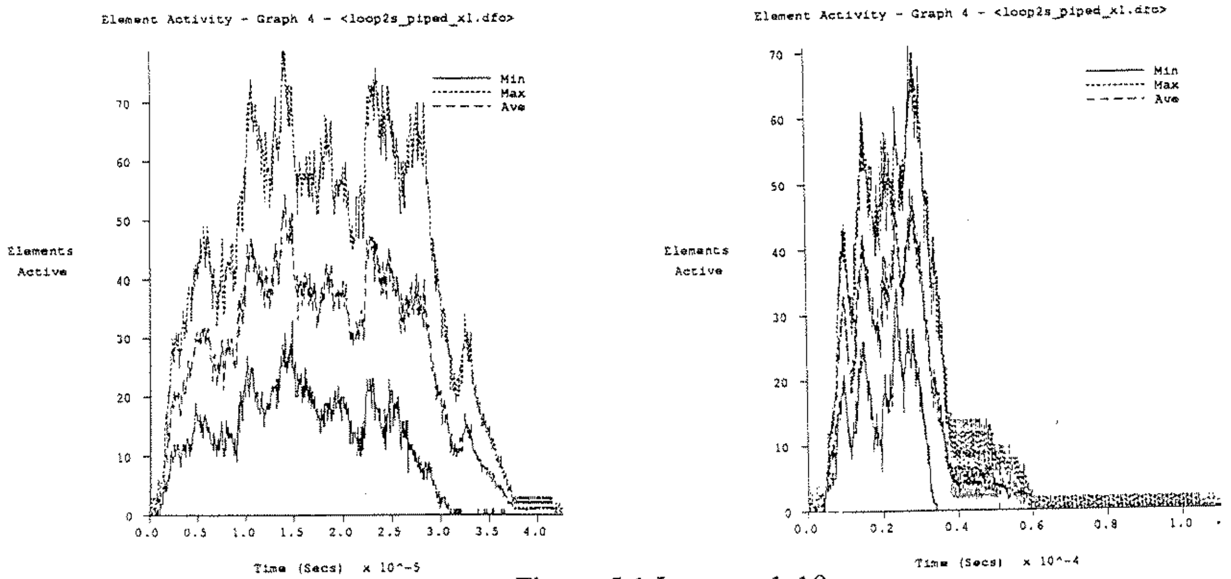


Figure 5.1 Latency 1-10

Considerations of latency of course extend to hardware design where there is usually a tradeoff between the number of processor pipeline and network stages and maximum instruction rates. Latency due to these stages is an important factor in reducing the recovery time from sequential code segments; others have demonstrated an awareness of these tradeoffs [19][14]. At the code level recursive tree based tag-generators, or generators which have cycles, exhibit slow startup. For example each level in a tree based generator may have a critical path of of 3-4 nodes. Assuming a latency of 10 there will be a delay of 30-40 before the next level of the tree is reached. In this time a sequence generator will have generated 40 tags or indices. Figure 5.2 illustrates the relationship between activity generation, thread length and expected levels of concurrency.

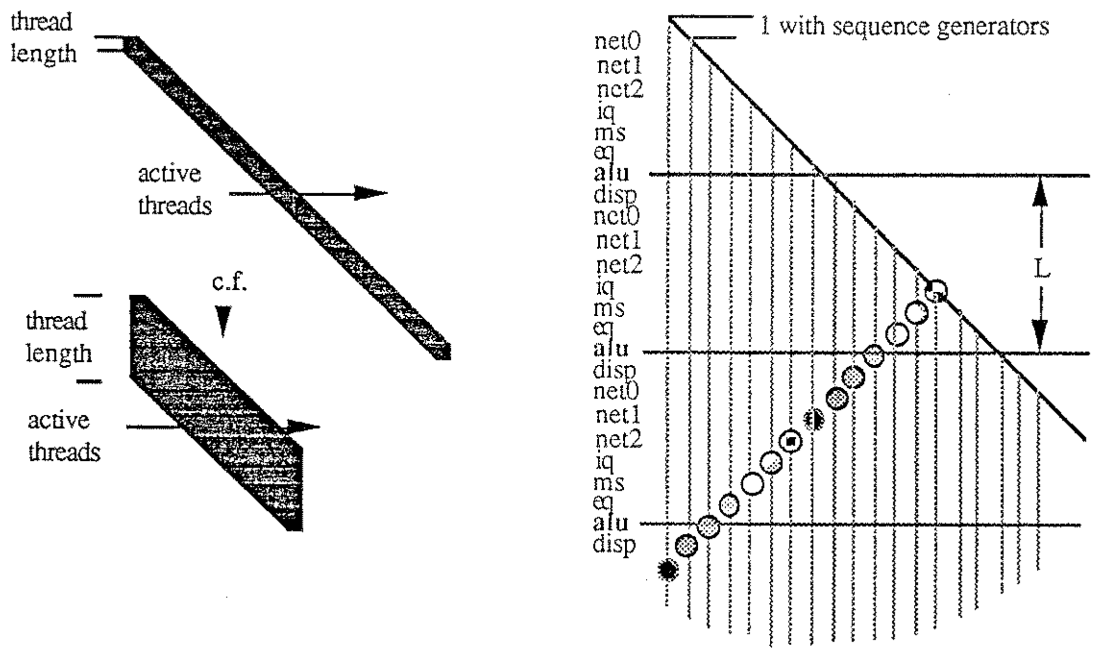


Figure 5.2 Threads

6. Conclusions

The architecture of CSIRAC II permits the exploration of tradeoffs between unravelled and streamed computations and has distinct advantages in environments where preservation of temporal ordering is important.

The examples presented here are based on initial code generated by our SISAL implementation and although we are at the beginning of a number of optimisation cycles, early results are promising. We have commenced evaluating our IDA implementation and expect some advantages over SISAL in the overlap of computational phases through the non-strict implementation of arrays.

Acknowledgements

The authors wish to acknowledge the contributions of the Project members, and in particular Dr. David Abramson and Mr. Mark Rawling, to the preliminary work presented here. The Parallel Systems Architecture Project at RMIT is being supported by the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) under an Information Technology joint research project.

References

[1] D. Abramson and Egan G.K, "Design Considerations for a High Performance Dataflow Multiprocessor", TR 122073R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1988. *Presented at the Workshop on Dataflow Computing: A Status Report, Eilat, Israel 1989*

[2] Abramson D. and Egan G.K. 'The RMIT Data Flow Computer: A Hybrid Architecture', The Computer Journal (British Computer Society), *accepted for publication,1988.*

[3] D. Abramson, "Using A Dataflow Multiprocessor for Functional Logic Simulation", Proceedings of ICS 88,pp 288-297.

[4] D. Abramson, "Constructing School Timetables Using Simulated Annealing: Sequential and Parallel Algorithms", TR112069R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1987.

[5] Arvind and Gostelow K.P., "The U-Interpreter", Computer, Vol. 15, No. 2, Feb 1982.

[6] R.G. Babb, "Programming Parallel Processors", Addision-Wesley, 1988.

[7] J.B. Dennis and Misunas D.P., "A Preliminary Architecture for a Basic Data-Flow Processor", Proc 2nd Ann. Symp. Computer Architecture, New York, May 1975.

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

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

[10] 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.

[11] G.K. Egan and Rawling M., "i2: An Intermediate Language for the RMIT Dataflow Computer, TR112068R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1987.

[12] G.K. Egan, "The RMIT Data Flow Computer - Token and Node Definitions", TR11260R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1989, *Internal Report..*

[13] G.K. Egan and J-L Gaudiot, "Data Driven Nearest Neighbour Iterative Numerical Computations", TR118083R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1989. *To be presented at the 9th International Conference on Computational Techniques and Applications, Brisbane, July, 1989.*

[14] J. Gurd and Watson I., "Data Driven Systems for High Speed Parallel Computing - part 2: Hardware Design", Computer Design, July 1980, pp 97-106.

[15] McGraw et al, "SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual", Lawrence Livermore National Laboratories, M146.

[16] R. Nikhil, K. Pringali and Arvind, "Id Nouveau", Computation Structures Group Memo 265, Massachusetts Institute of Technology, Laboratory for Computer Science.

[17] C.P. Richardson, "Manipulator Control Using a Dataflow Machine", Department of Computer Science, University of Manchester, Ph.D. Thesis, 1981.

[18] S. Skedzielewski and Glauert J., "IF1 An Intermediate Form for Applicative Languages", Lawrence Livermore National Laboratories, 1985.

[19] T. Shimada, K. Hiraki, K. Nishida and S. Sekigucki, "Evaluation of a Prototype Data-flow Processor of the SIGMA-1 for Scientific Computations", Proceedings of 13th Annual International Synposium on Computer Architecture, pp 226 - 234.

[20] Kazunori Ueda, "Guarded Horn Clauses", Doctor of Engineering Thesis, University of Tokyo, Graduate School, 1986.

[21] K.S. Weng, "Stream Oriented Computation in Recursive Data-Flow Schemas", Technical Memo 68, Laboratory for Computer Science, Massachusetts Institute of Technology, Oct 1975.

[22] N. Webb, "Implementing an Applicative Language for the RMIT/CSIRO Dataflow Machine", Department of Computer Science, Royal Melbourne Institute of Technology, *M.App.Sci. Thesis in preparation*, 1989.

[23] P. Whiting, "IDA: A Dataflow Programming Language", TR112075R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, 1988.