LABORATORY FOR CONCURRENT COMPUTING SYSTEMS

COMPUTER SYSTEMS ENGINEERING
School of Electrical Engineering
Swinburne Institute of Technology
John Street, Hawthorn 3122, Victoria, Australia.

A Comparison of Structure Accessing Techniques in IdA and SISAL on the CSIRAC II Dataflow Multiprocessor

Technical Report 31-024

P.G. Whiting†
G.K. Egan‡

†Division of Information Technology
Commonwealth Scientific and Industrial Research Organisation
723 Swanston Street, Carlton, 3053

‡Laboratory for Concurrent Computing Systems
Swinburne Institute of Technology
John Street, Hawthorn 3122

To be presented at the Third IEEE Symposium on Parallel and Distributed Processing, Dallas, Texas, December 1991.

Version 1.0 October 1991

Abstract:

The paper describes the implementation of the functional language IdA on the CSIRAC II dataflow multiprocessor. IdA is a derivative of MIT's ID Nouveau language and CSIRAC II is a dataflow architecture which combines the features of static queued and dynamic dataflow architectures and as such is outside the generally accepted taxonomy. The non-strict implementation of structures in IdA exploits the overlap in the production and consumption of structures. This eliminates the unnecessary serialisation of the computation evidenced in the implementation of another functional language SISAL. To illustrate this, results for the model numerical weather prediction code, Shallow, are presented for both IdA and SISAL formulations. These results show that for similar instruction counts a 39% reduction in runtime is obtained for the IdA implementation over the SISAL implementation in a representative scientific application.

1 Introduction

The Joint Parallel Systems Architecture Project officially commenced in May 1986 as a collabarative project between the Royal Melbourne Institute of Technology (RMIT) and the Commonwealth Scientific and Industrial Research Organisation (CSIRO). Given the human and financial resources, the project was studying a hybrid dataflow computer - CSIRAC II - and conducting research into programming languages for this high-speed computer. This work is now continuing in the Laboratory for Concurrent Computing at Swinburne Institute of Technology.

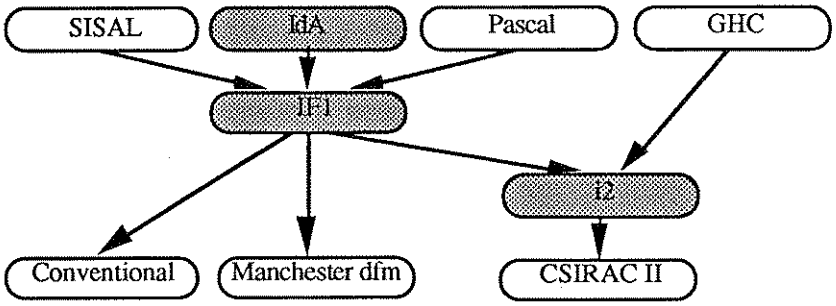


Figure 1 - experimental environment

The experimental environment into which the IdA study fits is shown in Figure 1. Compilers have been developed for IdA [23], Pascal [21] and GHC [25][17]. The IdA and Pascal compilers produce IF1 code [19] although, as will be described in this paper, IF1 has been augmented to support the I-structures required by IdA. The IF1 code is translated by the CSIRAC II IF1 backend [22] to the i2 block structured assembly language [9]. Both IdA and Pascal can take advantage of the competent IF1 optimisers now available to support SISAL [15]. GHC makes extensive use of tag manipulation primitives in the architecture and targets directly to i2.

2 IdA and CSIRAC II

2.1 IdA

IdA [23] is a derivative of the ID Nouveau language developed at MIT [16]. IdA includes features not found in ID Nouveau including explicit typing of variables and input /output. The feature of IdA, and the various ID dialects, which is of interest here, is the non-strict implementation of arrays.

SISAL is a functional language which has been targeted at a wide variety of systems including current generation multiprocessors such as the Encore Multimax and research dataflow machines [14][11]. The multi-targeting feature is accomplished by compiling SISAL to an intermediate language IF1 [19][4][22]. The IF1 representation is then compiled to the appropriate target instruction set. The optimising SISAL compiler (OSC) from Colorado State University yields performance competitive with FORTRAN [5][6]. Most of the optimisations in OSC are performed as IF1 to IF1 transformations.

2.2 CSIRAC II

The architecture of the CSIRAC II dataflow multiprocessor [8] is based on early work by one of the authors at Manchester [7] and encompasses both the static queued and the dynamic or unravelling dataflow schemes [2][14] and exhibits the advantages of both. The particular features of the architecture which are of interest to the IdA implementation are:

- generic node functions with implicit type coercion;
- sequence functions for tag and structure index generation;
- variable length strongly typed tokens including compound tokens containing heterogeneous data and vectors;
- deferred and non-deferred structures;
- random static allocation of nodes to processors at compile time qualified by colour at run time (double hashing scheme).

3 Implementation

3.1 IF1

IF1 was developed as an intermediate directed graph representation for SISAL compilers and as such it incorporates the strictness assumptions of SISAL extending most importantly to structures. Machine dependent analysis such as memory management or partitioning a graph over multiple processors is done using supersets of IF1. It is based on acyclic graphs and there are four components to a graph:

- nodes which denote operations such as add and subtract,
- edges which represent values that are passed from node to node,
- types attached to each edge or function for identification, and
- graph boundaries which surround groups of nodes and edges.

Arrays and records are the two main structures available in IF1 and there exists seventeen nodes for the manipulation of arrays and seven for the manipulation of records. We are primarily concerned with arrays in this paper. Any of the structure primitives which modifies elements of an array does so by creating a new array, placing the new values in the relevant elements and then copying the rest. The problem associated with copying structures in such a manner are well known, and the work of Cann [5][6] has virtually eliminated this from many SISAL programs.

```
define main

type OneDim = array[integer];

function init(n:integer; returns OneDim)
  for k in 1,n
  returns array of
    k*k
  end for
end function init

function main(returns OneDim)
init(10)
end function
```

Figure 2a - example SISAL program producing an array

Figure 2a shows a small SISAL program which creates an array filled with the squares of the numbers in the range one through to ten. Figure 2b represents function `init` which contains the parallel `Forall` loop responsible for the creation of the array. At the top of the graph, the argument `n` enters the Generator subgraph for the Forall loop which produces the sequence of numbers from `1` to `n`. For each number in the sequence, there exists a Body subgraph where the loop body calculation takes place. Finally, in the Returns subgraph, the

values produced by each instance of the Body subgraphs are collected together in their correct order by the `AGathers` node to produce an array of length `n`.

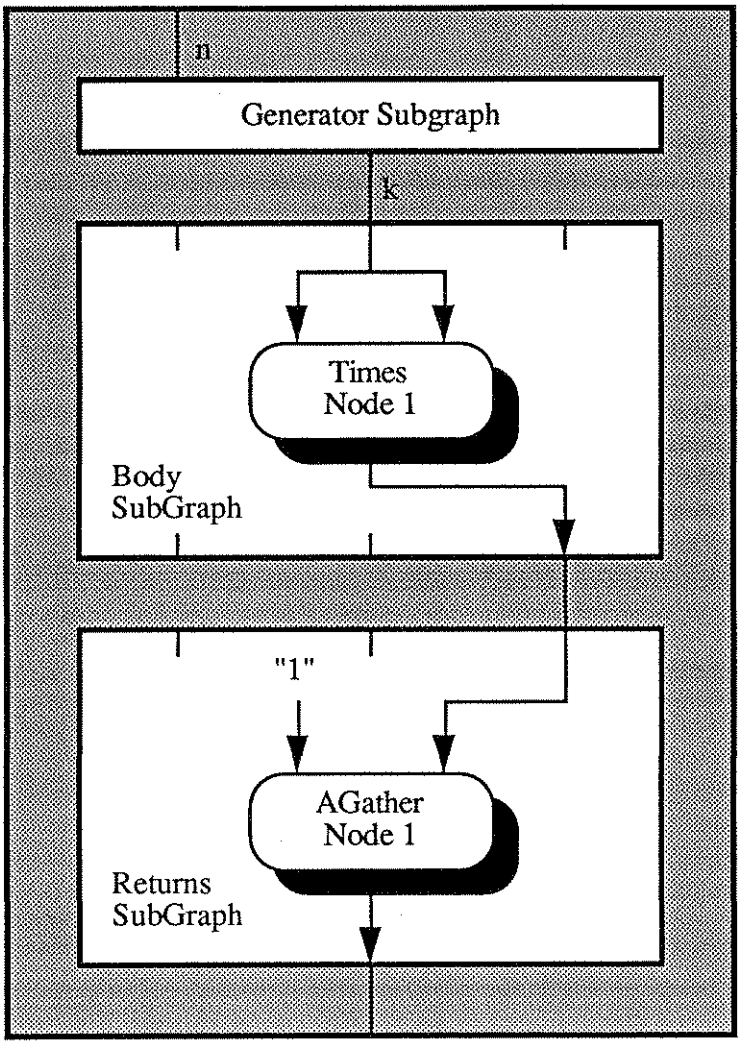


Figure 2b - simplified IF1 graph for the SISAL function `init`.

3.2 I-structures

Having just looked at a small SISAL program and its translation into IF1, let us now consider the same program written in IdA (Figure 3a). The source code is very similar except that in the IdA version there are more declarations and the value is explicitly assigned to its element in the array. MIT's Id Nouveau provides the programmer with a traditional strict array and a non-strict I-structure[3]. I-structures overcome the rigid semantics on computations responsible for filling in components of data-structures and make the language non-functional, yet they still allow Id to remain determinate. For various reasons discussed in [24] all array operations in IdA have been replaced with I-structure operations.

The semantics of I-structures made them difficult to implement in IF1. The existing IF1 nodes were not suitable as they were strict and would not allow the non-strict I-structures to be gradually defined across a number of functions. The defining of an element of an I-structure is considered to be a side-effect in Id and this was not allowed in IF1. Even when a scheme was designed to reflect this by not connecting the output of the node responsible for the definition of an element of the array, the optimisers and other tools available would assume it was dead-code and roll-up the graph eliminating the loop Body subgraph. A variety of schemes for their

implementation were considered before a suitable one was developed. The method makes use of four procedures which are intrinsics of the CSIRAC II instruction set. These procedures are treated as imported functions by the IF1 tool set whereby the IF1 translator allows the definition of the procedure and assumes the actual body will be defined somewhere else.

The four procedures used are:

- `Ialloc(arg1) -> address`
  Allocates a block of contiguous memory in the object store to contain the I-structure. The number of components in the I-structure is represented by `arg1` and the function returns the starting address of the I-structure in the object store.
- `Iread(arg1) -> value`
  Reads a value from a component of an I-structure. The address of the component is represented by `arg1` and is obtained by adding the address from `Ialloc` to the result of the index calculation.
- `Iwrite(arg1 arg2) -> signal`
  Writes a value into a component of an I-structure. The address of the component is `arg1` and is calculated in the same way as the address for `Iread`, while the value is represented by `arg2`. The `signal` is a dummy edge, carrying the address of the component and is used to satisfy the semantics of IF1.
- `Ikill(arg1 arg2) -> signal`
  Gathers the outputs of the `Iwrite` procedures. The arguments used are in general, the signals from calls to `Iwrite`, while the result is either one of the inputs.

```
const
  n = 10;

type
  OneDim = array(1..n) of integer;

def init
  y : integer
returns OneDim =
let
  var
    k : number;
    x : OneDim;
in
  for k from 1 to y do
    x[k] = k * k;
  returns x;

init(n);
```

Figure 3a - IdA version of Figure 2a

To comply with the functional semantics of IF1, *Ikill chains* are generated to handle the side-effects of I-structures. For every `Iwrite` procedure called, there's a redundant edge that must be generated. This edge has no destination and becomes the argument to an `Ikill` procedure. The shape of the resulting graph dealing with I-structure writes, resembles a chain. It depends on the source program as to what actually happens to the output of the chain.

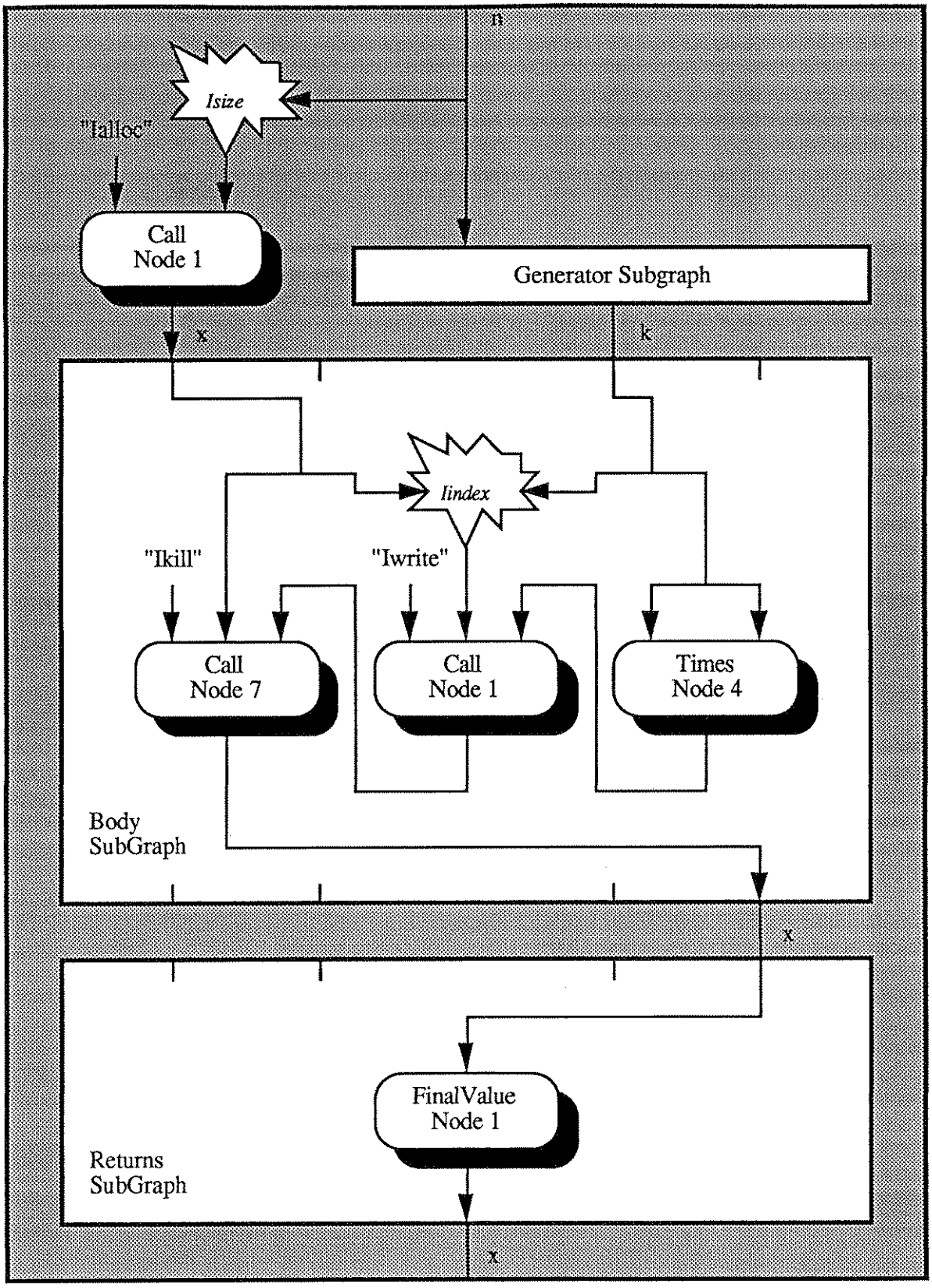


Figure 3b - simplified IF1 graph for the IdA function `init`.

Figure 3b shows the simplified IF1 graph for function `init` in Figure 3a. Due to the declarations in IdA, the size of the I-structure is known at compile-time and this is represented by the explosion-like subgraph labelled *Isize*. Memory in the object store of the CSIRAC II is

allocated for I-structure `x` by the call to `Ialloc`, which occurs inside the body of function `init` but outside the `Forall` loop. The function of the Generator subgraph has been discussed earlier. In the Body subgraph there are considerably more nodes, than in the SISAL implementation. Node 1 is the call to `Iwrite`, nodes 2 through to 6 with the exception of node 4 (calculation of `k*k`) is involved with the indexing calculation for I-structure `x` and node 7 is the call to `Ikill` to deal with the signal generated by node 1. In the Returns subgraph, the `FinalValue` node is used in the implementation of IdA to signify when all the different loop bodies have terminated.

It is important to note, that while the method used does provide a working implementation of I-structures in IF1, it does not effectively reflect the side-effects occurring or how deferred reads are handled. Deferred reads, and for that matter all I-structure operations, are handled by special hardware on CSIRAC II called the Object Store. Banks of the Object Store are associated with each processing element (PE) of the dataflow machine with the appropriate bank being selected conventionally using the least significant bits of the I-structure elements address. I-structures are accordingly spread over several or all of the PEs. Any attempt to access an as yet uninstantiated I-structure element or object results in the request being queued in a linked list associated with the object. When the object is instantiated all outstanding requests are honoured. All objects in the object store have a tag field giving the current state of the object. The tag field permits efficient determination and modification of the objects state however a penalty of a full read modify-write is incurred on every object stoer access involving I-structure write operations. The strict structure implementation of SISAL does not require this as the state of the object is known by construction. The four I-structure procedures used at the IF1 level are converted by the IF1 translator into a combination of six nodes at the hardware level [8][22].

4 A Structure Accessing Example

Many numerical computations contain iterative computations involving array structures. Within each iteration arrays are transformed into or used as arguments for the computation of new arrays. In many cases the accessing pattern is similar with each transformation or use. Numerical weather prediction codes exhibit this type of computation and are used here to suggest the gains which may be made with a non-strict implementation of structures through overlapping the production and use of structures.

The results were obtained using the CSIRAC II simulator. The simulator is event based and models latency (10 network and processor pipeline stages) [12]. The simulator produces various measures of system performance in six graphs:

- *Unmatched and Transit Tokens* shows the number of data tokens waiting for their partners and the number of tokens in the communication network;
- *Element Activity* shows the number of processing-elements active;
- *Performance Measures* show miscellaneous measures (not important here);
- *Individual Element Activity* shows the workload distribution with a dark area indicating a processing-element is active;
- *Object Store Disposition* shows the number of instantiated structure elements and the number of outstanding deferred accesses to I-structures;
- *Object Store Access Pattern* shows the accessing pattern for structure cells with a dark area indicating an access to a cell in that processing-element.

4.1 A Model Numerical Weather Prediction Code

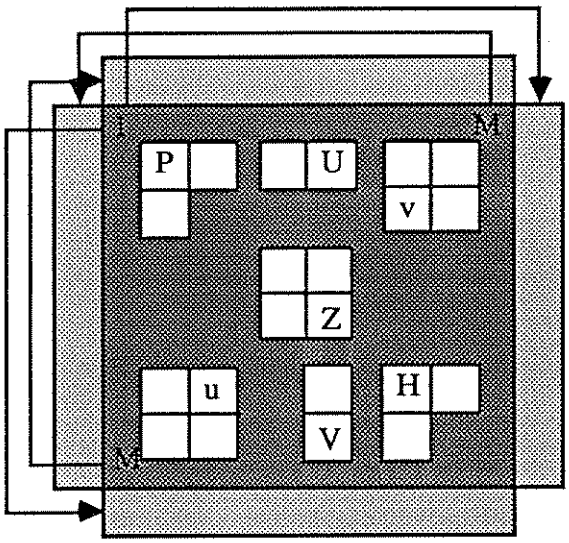
The model numerical weather prediction code Shallow is described by Sadourny in [18]. The original FORTRAN implementation of the Shallow code is due to Swartztrauber [13] and is believed to be representative, albeit greatly simplified, of full shallow water NWP codes and NWP codes in general [1].

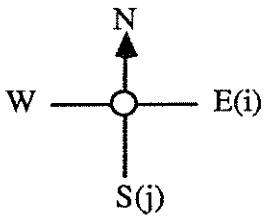
In a description from Snelling, Shallow models a square single layer of fluid (Figure 4). The model variables are the:

- cartesian fluid velocities u and v,
- pressure P,
- field height H,
- cartesian mass fluxes U and V, and
- potential velocity Z.

The variables are related by the following equations:

- $\delta u/\delta t - ZV + \delta H/\delta x = 0$
- $\delta v/\delta t + ZU + \delta H/\delta y = 0$
- $\delta P/\delta t + \delta U/\delta x + \delta V/\delta y = 0$





The "adjacency" figures indicate which cells are involved in the computation of P,u,v,U,V,Z and H. The boundaries are mapped as shown to make the mesh periodic.

Figure 4 - the Shallow Domain

Figure 4 shows how the boundaries 'wrap around' south to north and east to west providing a periodic continuation of the mesh. Shallow has two input parameters, the size of the mesh and the number of time steps.

4.2 Results for SISAL

The SISAL version of Shallow (Appendix A) described in [10] was used as the basis for comparison with IdA's non-strict structure implementation. In the case of SISAL programs, the IF1 intermediate form produced by OSC R1.8/V3.0 (*sisal -old_loops -no_opt -no_offset; if1opt -di -x -l -e*) is translated to the CSIRAC II instruction set [8] by the IF1 translator [22].

Figure 5 shows the workload distribution and processor activity for the SISAL version of Shallow with a mesh size of 32 for one time step. The processor activity clearly shows the sequential sections between the various functions where the number of active processors falls dramatically. These boundaries are due largely to inter function synchronisation. Although the IF1 structure accessing primitives used by SISAL are strict, the underlying implementation of these primitives on CSIRAC II is not. This permits SISAL to take some advantage of the deferred structure access mechanisms of CSIRAC II; the deferred reads can be seen in the Object Store Disposition graph of Figure 5. If the implementation of these primitives enforced strictness, then the sequential sections would be more evident than they are with consequent substantial increase in runtime. The work on tradeoffs between strict and non-strict SISAL implementations is continuing and will be reported elsewhere [22].

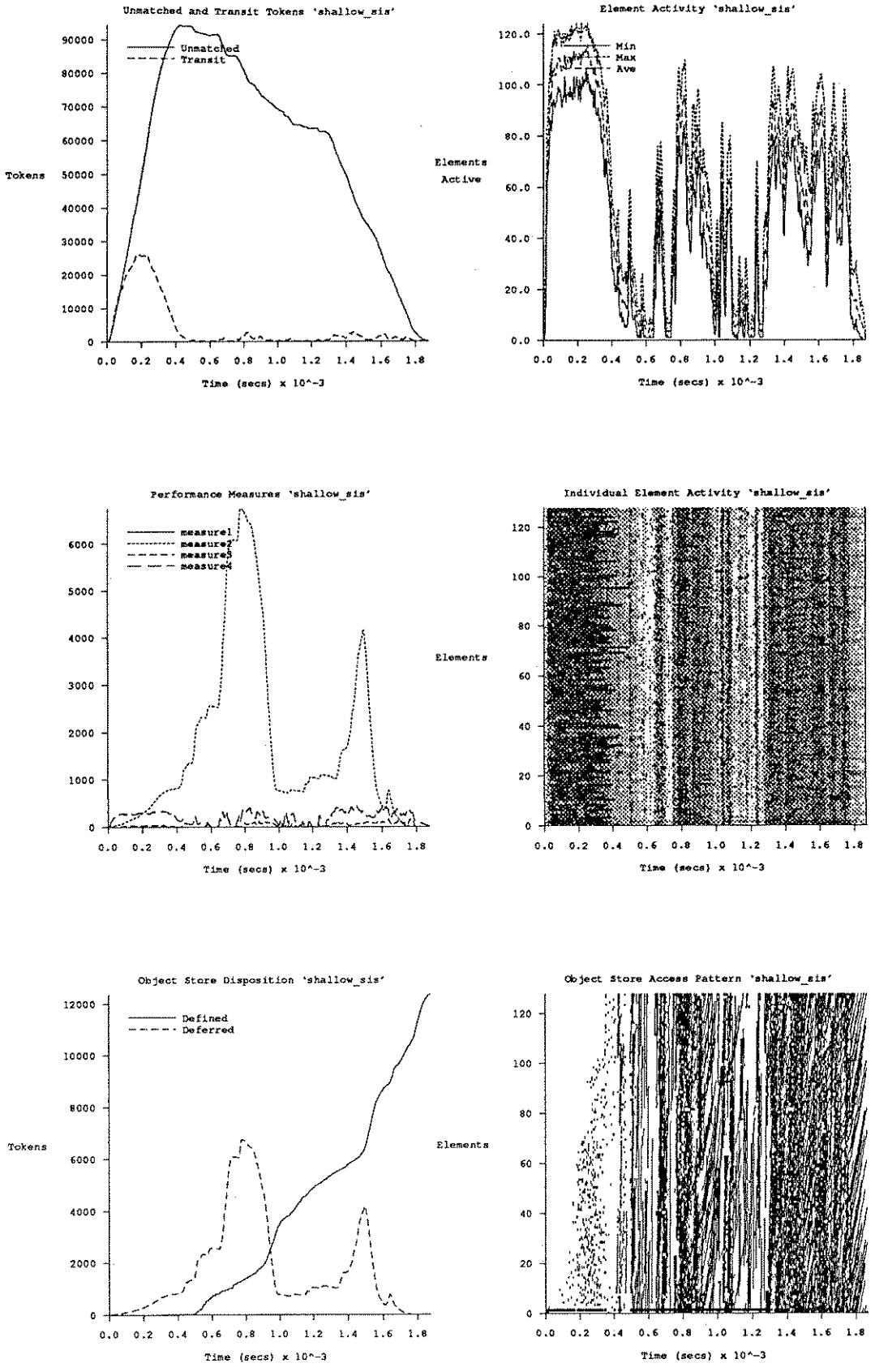


Figure 5 - simulation results for SISAL

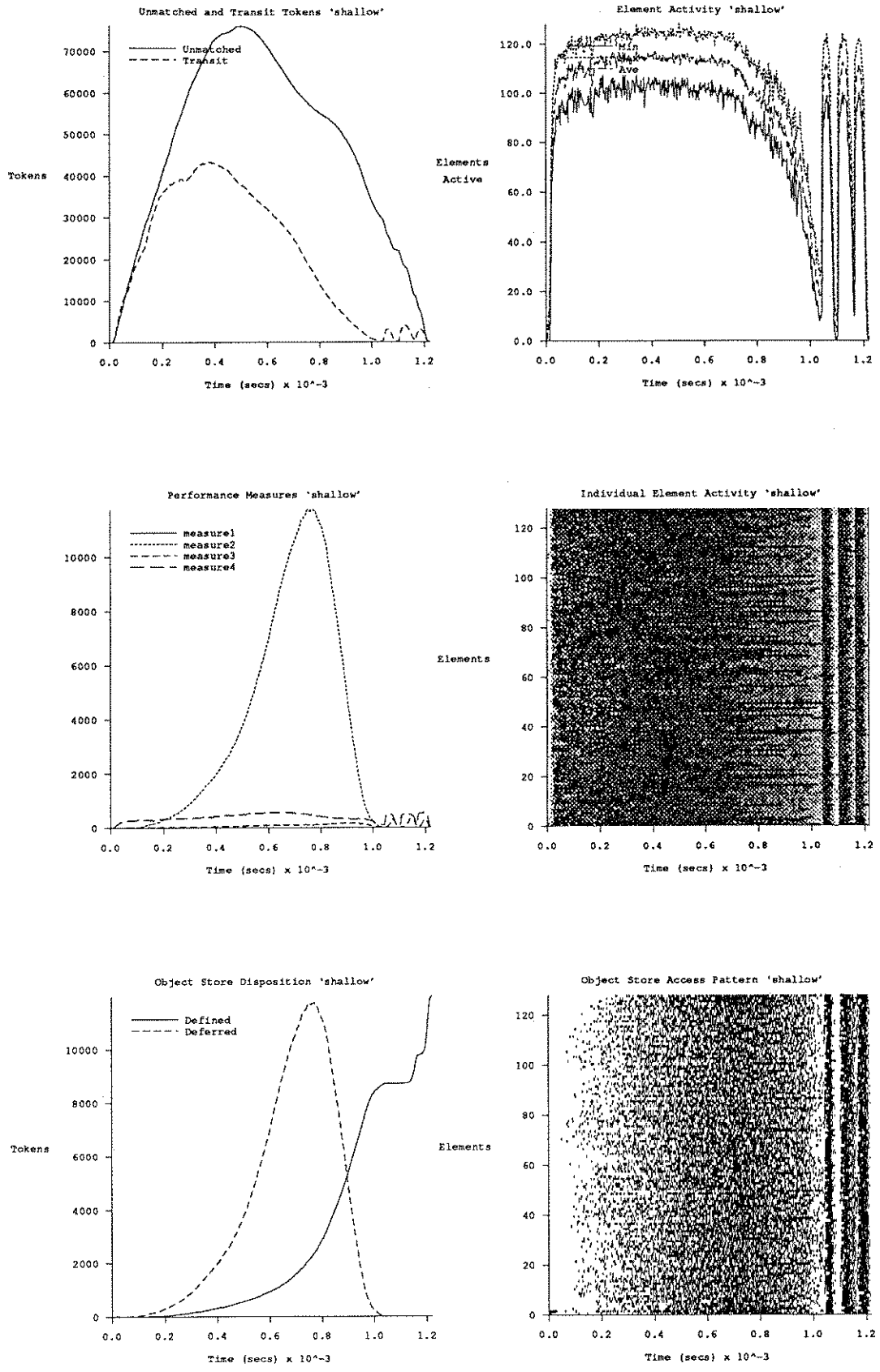


Figure 6 - simulation results for IdA

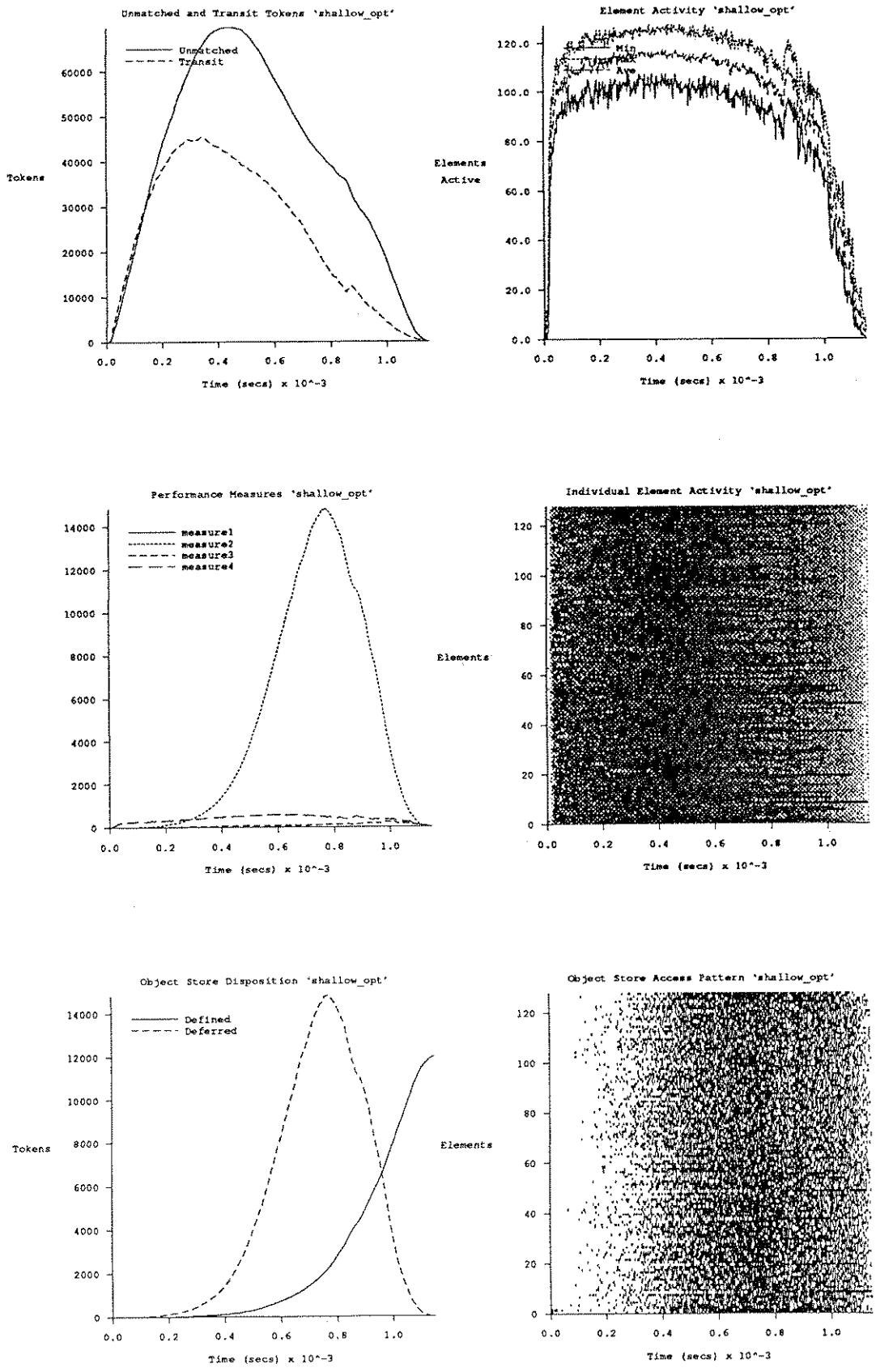


Figure 7 - simulation results for optimised IdA

4.3 Results for IdA

The IdA version of Shallow (Appendix B) was produced by direct transliteration of the SISAL version with no changes to the computational algorithm.

IdA is compiled to IF1 and then translated to the CSIRAC II instruction set. Figure 6 shows the simulation results for the IdA version at the same model resolution as was used for SISAL.

The processor activity profile is markedly different from that for SISAL (Figure 5). This is due almost entirely to the non-strict arrays referred to as I-structures [3] in IdA. Array elements for which values have been computed may be used immediately by functions or other computations for which they are arguments. As expected this permits substantial computational overlap of functions and the almost total elimination of sequential segments.

It is also of interest to compare the Object Store access patterns of SISAL and IdA. SISAL exhibits systematic diagonal striping while there is no pattern evidenced by IdA. SISAL uses indirection vectors of pointers to access multi-dimensional arrays; this is required to facilitate the support variable sized structures at runtime. In IdA array element addresses are computed directly from known array bounds.

In general the workload distribution for IdA is much more uniform than that for SISAL which contributes to the improved runtime.

4.4 Removal of Strict Loop Synchronisation

Comparison of the instruction counts for the SISAL and the initial IdA implementation showed that significantly more instructions were being executed for IdA (Table 1).

	Instructions	Run Time (mS.)
IdA SISAL	466268 386168	1.21 1.88
IdA (opt.)	*444385*	*1.15*

Table 1 - instruction counts for IdA, SISAL and *optimised IdA*
(one iteration on a 32x32 problem)

As previously discussed SISAL uses an array gather (`AGather`) primitive to assemble a sequence of elements emitted from the loop body into an array. In the case of IdA, arrays are assembled within the loop body but it is still necessary to anchor the loop with a loop termination primitive. The only primitive available in the current IF1 translator implementation and supported by the IF1 optimisers was `FinalValue`. The graph required by IF1 translators and optimisers is shown in Figure 3b.

`FinalValue` consumes all elements of a structure or stream of values except the last which is propagated to the successor function. This is an active process requiring the execution of a significant number of instructions. For Shallow the values returned from the computation are the base addresses of the structures constructed within the loop body. One of these addresses is selected as input to the `FinalValue` node to provide the required synchronisation; the other addresses are passed directly to the successor function permitting some overlap between production and use of these structures. It is possible to eliminate these redundant computations by passing the base addresses of all structures to be constructed by the

loop body, directly to the successor function as shown in Figure 8 and grounding, or terminating, the loop Body subgraph outputs.

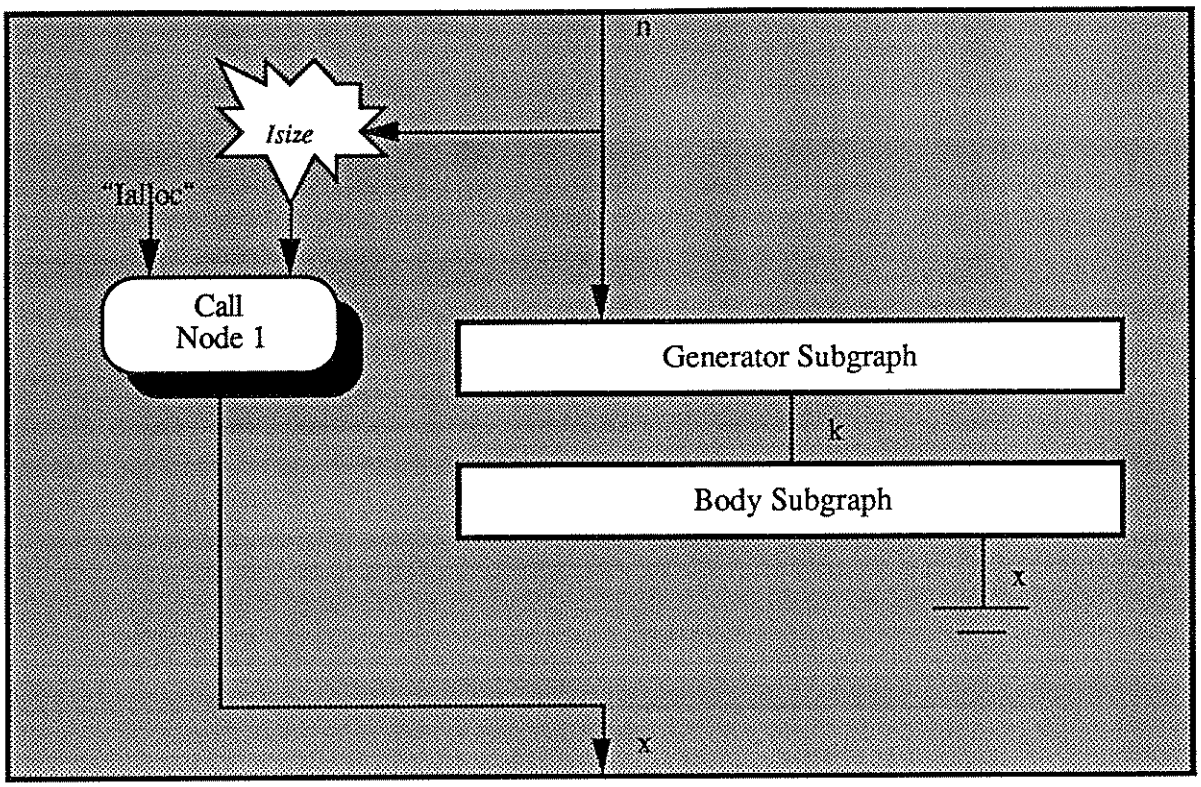


Figure 8 - optimised graph for IdA function `init`

Elimination of this code resulted in the processor activity profile in Figure 7. It can be seen that the remaining sequential sections have been eliminated. These sections were due to `FinalValue` in the closing stages of the computation. The runtime for this optimisation is shown in italics in Table 1 and is 39% less than that for SISAL.

5 Conclusions

An environment has been established which permits comparison between a number of languages targeted on the standard intermediate form (IF1). We have successfully integrated the non-strict I-structure mechanism into the strict framework of IF1 permitting the use of the sophisticated optimisation techniques developed by SISAL researchers.

In this paper, we have presented some initial comparisons between IdA and SISAL focusing in particular on strict and non-strict structure access mechanisms. The results show that for similar instruction counts, the IdA implementation provides a dramatic improvement in runtime compared with that for SISAL. This performance improvement is due to greater overlap in the production and consumption of structures permitted by the I-structure mechanism. These gains may be obtained on dataflow systems, or for that matter any other parallel system having the required hardware to support non-strict structures.

Work is continuing on the development and optimisation of both the IdA and SISAL programming environments for the CSIRAC II dataflow multiprocessor.

Acknowledgements

The authors thank Dr. David Cann of Lawrence Livermore National Laboratory, and Neil Webb for their contributions to this research. We would also like to thank the members of the Parallel Systems Architecture Project and the Laboratory for Concurrent Computing Systems.

The Parallel Systems Architecture Project was jointly funded by the Royal Melbourne Institute of Technology and the Commonwealth Scientific and Industrial Research Organisation.

The Laboratory for Concurrent Computing Systems at the Swinburne Institute of Technology is funded by the Australian Commonwealth Government under a special research infra-structure grant for parallel processing research.

References

[1] G.S. Almasi and A. Gottlieb, 'Highly Parallel Computing', Benjamin Cummings Publishing Company Limited, 1989.

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

[3] Arvind, Nikhil R.S. and Pingali, K.K., 'I-Structures: Data Structures for Parallel Computing', ACM Transactions on Programming Languages and Systems, Vol 11, #4, Oct. 1989.

[4] A.P.W. (Wim) Bohm and J. Sargeant, 'Efficient Dataflow Code Generation fo SISAL', Technical Report UMCS-85-10-2, Department of Computer Science, University of Manchester, 1985.

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

[6] D.C. Cann, 'High Performance Parallel Applicative Computation', Technical Report CS-89-104, Colorado State University, Feb.1989.

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

[8] G.K. Egan, 'The CSIRAC II Data Flow Computer - Token and Node Definitions', Technical Report 31-001, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[9] G.K. Egan, Rawling M. and Webb N.J., 'i2: An Intermediate Language for the RMIT Dataflow Computer, Technical Report 31-002, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[10] G.K. Egan, 'Some Shallow Experiences', Technical Report 31-004, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[11] G.K. Egan, N.J. Webb and A.P.W. (Wim) Bohm 'Some Architectural Features of the CSIRAC II Dataflow Computer', Technical Report 31-007, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, 1990.

[12] G.K. Egan, 'The CSIRAC II Simulation Suite', Technical Report 31-010, Laboratory for Concurrent Computing Systems, School of Electrical Engineering, Swinburne Institute of Technology, *in preparation*, 1990.

[13] Geerd-R. Hoffman et al. eds. 'Aspects of Using Multiprocessors for Meteorological Modelling', in Multiprocessing in Meteorological Models, Hoffman and Snelling eds., Springer-Verlag, Berlin,1988.

[14] J. Gurd and I. Watson, 'Data Driven Systems for High Speed Parallel Computing - part 1: Hardware Design', Computer Design, June 1980, pp 91-100.

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

[16] R.S. Nikhil, K. Pingali and Arvind, 'Id Nouveau', MIT-LCS Computation Structures Group Memo 265, 1986.

[17] M. Rawling, 'GHC on the CSIRAC II Dataflow Computer', TR118090R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, Oct. 1989.

[18] R. Sadourny, 'The Dynamics of Finite-Difference Models of the Shallow Water Equations', Journal of Atomospheric Sciences, Volume 32, Number 4, April 1975.

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

[20] D.F. Snelling, 'Experimental Guidelines for a Shallow Mapping Study', Technical Report Number 21, by David F. Snelling, Department of Computing Studies, University of Leicester, 1989.

[21] S. Wail, 'Implementing an Imperative Language on a Dataflow Computer Architecture', Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, *PhD thesis in preparation, 1990.*

[22] N.J. Webb, 'Implementing an Applicative Language for the CSIRAC II Dataflow Computer', School of Electrical Engineering, Swinburne Institute of Technology, *Ph.D. Thesis in preparation*, 1990.

[23] P.G. Whiting, 'IdA: A Dataflow Programming Language', TR118075R, Department of Communication and Electrical Engineering, Royal Melbourne Institute of Technology, Oct. 1988.

[24] P.G. Whiting, 'Compilation of a Functional Programming Language for the CSIRAC II Dataflow Computer', M.App.Sci. Thesis, Department of Computer Science, Royal Melbourne Institute of Technology, Feb. 1990.

[25] K. Ueda, 'Guarded Horn Clauses', Doctor of Engineering Thesis, University of Tokyo, Graduate School, 1986.

Appendix A- SISAL Version of Shallow

```
% Shallow - G.K. Egan 1989-91
% Laboratory for Concurrent Computing Systems
% Swinburne Institute of Technology, Australia

define
  Main,
  SisalMain,          % Main entry point to program
  Initialises,        % Define initial condition
  Fluxes,             % Compute the mass fluxes
  Height,             % Compute the field height
  Potential,          % Compute the total potential velocity
  TimeStep            % Compute next time step

% convention used for cell neighbour indices is:
%
%                     j-1 is jn (north)
%  i-1 is iw (west)         i,j          i+1 is ie (east)
%                     j+1 is js (south)

% --- type specifications ---
type  GridVariables = array[array[real]];

% --- imported functions ---
global sin (x: real returns real)
global cos (x: real returns real)

% --- initialize ---
function Initialize(M:integer; delta:real
  returns GridVariable, GridVariables, GridVariables)

let
  P:GridVariables;
  Psi:GridVariables;
  U:GridVariables;
  V:GridVariables;
  Amp:real := 1.0E6;        % Amplitude of waves in initial condition.
  pie:real := 3.1459265359;
  el:real := real(M)*delta;
  pcf:real := pie * pie * Amp * Amp(el*el);
  delta_i : real := 2.0 * pie/real(M);
  delta_j : real := 2.0 * pie/real(M);

  Psi := for j in 0,M cross i in 0,M
         returns array of
         Amp*sin(delta_i*(real(i+1)-0.5))*sin(delta_j*(real(j+1)-0.5)
         end for
in
  for j in 0,M cross i in 0,M
  returns array of
  pcf*(cos(2.0*delta_i*(real(i))+cos(2.0*delta_j*real(j)))+50000.0
  end for,

  for j in 0,M
  returns array of
    let
      js := if j=M then 0 else j+1 end if
    in
      for i in 0,M
      returns array of
      -(Psi[js,i]-psi[j,i])/delta
```

```
      end for
    end let
  end for,

  for j in 0,M
  returns array of
    for i in 0,M
    returns array of
      let ie := if i=M then 0 else i+1 end if
      in
        (Psi[j,ie]-Psi[j,i])/delta
      end let
    end for
  end for
end let
end function

% --- Fluxes ---
function Fluxes(M:integer; P,U,V:GridVariables
  returns gridVariables, GridVariables)
for j in 0,M
returns array of
  for i in 0,M
  returns array of
    let
      iw := if i=0 then M else i-1 end if
    in
      0.5*(P[j,i]+P[j,iw])*U[j,i]
    end let
  end for
end for,
for j in 0,M
returns array of
  for i in 0,M
  returns array of
    let
      jn := if i=0 then M else i-1 end if
    in
      0.5*(P[j,i]+P[jn,i])*V[j,i]
    end let
  end for
end for
end function

% --- Height ---
function Height(M: integer; P,U,V: GridVariables returns GridVariables)
for j in 0,M
returns array of
  let
    js := if j=M then 0 else j+1 end if
  in
    for i in 0,M
    returns array of
      let
        ie := if i=M then 0 else i+1 end if
      in
        P[j,i]+0.25*(U[j,ie]*U[j,ie]+U[j,i]*U[j,i]+
        V[js,i]*V[js,i]+V[j,i]*V[j,i]
      end let
    end for
  end let
end for
end function
```

```
% --- Potential ---
function Potential(M: integer; P,U,V: GridVariables; delta:real
  returns GridVariables)
let
  fsdx: real := 4.0/delta;
  fsdy: real := 4.0/delta;
in
  for j in 0,M
  returns array of
    let
      jn := if j=0 then M else j-1 end if
    in
      for i in 0,M
      returns array of
        let
          iw := if i=0 then M else i-1 end if
        in
          (fsdx*(V[j,i]-V[j,iw])-fsdy*(U[j,i]-U[jn,i]))/
          (P[jn,iw]+P[jn,i]+P[j,iw]+P[j,i])
        end let
      end for
    end let
  end for
end let
end function

% --- Time Step ---
function TimeStep(M: integer; deltat, delta: real;
  Psmooth, Umsooth, Vsmooth: GridVariables;
  Uflux, Vflux, H, PotVel: GridVariables
  returns GridVariables, GridVariables, GridVariables)
let
  deltat_8 :real := deltat/8.0;
  deltat_d :real := deltat/delta;
in
  for j in 0,M
  returns array of
    let
      js := if j=M then 0 else i+1 end if
    in
      for i in 0,M
      returns array of
        let
          ie := if i=M then 0 else i+1 end if
        in
          Psmooth[j,i]-deltat_d*(Uflux[j,ie]-Uflux[j,i])-
          deltat_d*(Vflux[js,i]-Vflux[j,i])
        end let
      end for
    end let
  end for,

  for j in 0,M
  returns array of
    let
      js := if j=0 then M else j-1 end if;
      jn := if j=M then 0 else j+1 end if
    in
      for i in 0,M
      returns array of
        let
          iw := if i=0 then M else i-1 end if
          ie := if i=M then 0 else i+1 end if
```

```
        in
          Usmooth[j,i]+deltat_8*(PotVel[j,ie]+PotVel[j,i])*
          (Vflux[js,i]+Vflux[j,i])+(Vflux[j,iw]+Vflux[js,iw])-
          deltat_d*(H[j,i]-H[j,iw])
        end let
      end for
    end let
  end for,

  for j in 0,M
  returns array of
    let
      jn := if j=0 then M else j-1 end if;
      js := if j=M then 0 else j+1 end if
    in
      for i in 0,M
      returns array of
        let
          iw := if i=0 then M else i-1 end if;
          ie := if i=M then 0 else i+1 end if
        in
          Vsmooth[j,i]-deltat_8*(PotVel[j,ie]+PotVel[j,i])*
          (Uflux[j,ie]+Uflux[j,i]+Uflux[jn,i]+Uflux[jn,ie])-
          deltat_d*(H[j,i]-H[jn,i])
        end let
      end for
    end let
  end for
end let
end function

% --- Smooth ---
function Smooth(M: integer; X,Xsmooth, Xnext: GridVariables
  returns GridVariables)
let
  Alpha : real := 0.001;    % Time filtering parameter.
in
  for j in 0,M cross i in 0,M
  returns array of
    X[j,i]+Alpha*(Xnext[j,i]-2.0*X[j,i]+Xsmooth[j,i])
  end for
end let
end function

% --- Main Program ---
function SisalMain(Minput: integer; lastiter: integer
  returns GridVariables, GridVariables, GridVariables)

for initial

  P: GridVariables;          % Pressure
  U: GridVariables;          % Velocity in east/west direction
  V: GridVariables;          % Velocity in north/south direction
  Psmooth: GridVariables;    % Time smoothed P
  Usmooth: GridVariables;    % Time smoothed U
  Vsmooth: GridVariables;    % Time smoothed V
  M: integer;                % Deminsionality of system
  iter: integer;             % Iteration count
  deltat: real;              % Time step size in seconds
  delta: real;               % Grid spacing

  M := Minput;
  iter := 0;
```

```
    deltat := 90.0;
    delta := 1.0E5;

    P,U,V := Initialize(M, delta);
    Psmooth := P;
    Usmooth := U;
    Vsmooth := V
  while (iter < lastiter) repeat
    P, Psmooth, U, Usmooth, V, Vsmooth :=
      let
        Uflux: GridVariables;       % Mass flux in east/west direction
        Vflux: GridVariables;       % Mass flux in north/south direction
        H: GridVariables;           % A value related to height of fluid
        PotVel: GridVariables;      % Potential velocity
        Pnext: GridVariables;       % Intermediate results
        Unext: GridVariables;       % Intermediate results
        Vnext: GridVariables;       % Intermediate results

        Uflux, Vflux := Fluxes(M, old P, old U, old V);
        H := Height(M, old P, old U, old V);
        PotVel := Potential(M, old P, old U, old V, delta);
        Pnext, Unext, Vnext := TimeStep(M, old deltat, delta,
          old Psmooth, old Usmooth, old Vsmooth,
          Uflux, Vflux, H, PotVel)
      in
        if (old iter = 0) then
          Pnext, old P, Unext, old U, Vnext, old V
        else
          Pnext, Smooth(M, old P, old Psmooth, Pnext),
          Unext, Smooth(M, old U, old Usmooth, Unext),
          Vnext, Smooth(M, old V, old Vsmooth, Vnext)
        end if
      end let;
      deltat := if (old iter = 0) then
                  old deltat*2.0
                else
                  old deltat
                end if;
      iter := old iter + 1;
  returns
    value of P
    valie of U
    value of V
  end for
  end function

  function main(returns GridVariables, GridVariables, GridVariables)
    SisalMain(32,1)
  end function
```

Appendix B- IdA Version of Shallow

```
% Shallow - G.K. Egan 1989-91
% Laboratory for Concurrent Computing Systems
% Swinburne Institute of Technology, Australia
const
  M = 32;
  LastIter = 1;

type
  mesh = array (1..M, 1..M) of number;
  PUV_tuple = tuple(mesh; mesh; mesh);
  UV_tuple = tuple(mesh; mesh);

def  Initialize
  M:number
  delta: number
returns
  PV_tuple =

let
  const
    Amp = 1.0E6;
    pie = 3.14159265359;
  var
    Psi, P, U, V: mesh;
    el, pcf, delta_i, delta_j, i, j, ie, iw, jn, js: number;
  assign
    el = M*delta;
    pcf = sqr(pie) * sqr(Amp) / sqr(el);
    delta_i = 2.0 * pie / M;
    delta_j = delta_i;
in
  for j from 0 to M do
    = for i from 0 to M do
      js = if j = M then 0 else j+1;
      ie = if i = M then 0 else i+1;
      Psi[j, i] = Amp * sin (delta_i*((i+1)-0.5)) *
        sin (delta_j*((j+1)-0.5));
      P[j, i] = pcf * (cos (2.0*delta_i*i) +  cos (2.0*delta_j*j)) +
        50000.0;
      U[j, i] = -(Psi[js, i]-Psi[j, i]) / delta;
      V[j, i] = (Psi[j, ie]-Psi[j, i]) / delta
    returns()
  returns (P,U,V);

def Fluxes
  M:number
  P:mesh
  U:mesh
  V: mesh
returns UV_tuple =
let
  var
    NewU, NewV: mesh;
    i, j, iw, jn: number;
in
  for j from 0 to M do
    = for i from 0 to M do
      iw = if I = 0 then M else i-1;
      jn = if j = 0 then M else j-1;
      NewU[j, i] = 0.5 * (P[j, i] + P[j, iw]) * U[j, i];
```

```
        NewV[j, i] = 0.5 * (P[j, i] + P[jn, i]) * V[j, i]
      returns()
  returns(NewU, NewV);

def Height
  M:number
  P:mesh
  U:mesh
  V:mesh
returns mesh =
let
  var
    H: mesh;
    i, j, ie, js: number;
in
  for j from 0 to M do
    = for i from 0 to M do
      js = if j = M then 0 else j+1;
      ie = if i = M then 0 else i+1;
      H[j, i] = P[j, i] + 0.25 * (sqr(U[j, ie]) + sqr(U[j, i])+
        sqr(V[js, i]) + sqr(V[j, i]))
    returns()
  returns(H);

def Potential
  M:number
  P:mesh
  U:mesh
  V:mesh
  delta: number
returns mesh =
let
  var
    PotVel: mesh;
    fsdx, fsdy, i, j, iw, jn: number;
  assign
    fsdx = 4.0 / delta;
    fsdy = 4.0 / delta;
in
  for j from 0 to M do
    = for i from 0 to M do
      jn = if j = 0 then M else j-1;
      iw = if i = 0 then M else i-1;
      PotVel[j, i] =
        (fsdx * (V[j,i] - V[j,iw]) -
         fsdy * (U[j,i] - U[jn,i])) /
         (P[jn,iw] + P[jn,i] + P[j,iw] + P[j,i])
    returns()
 returns(PotVel);

def  TimeStep
  M:number
  deltat:number
  delta:number
  Psmooth:mesh
  Usmooth:mesh
  Vsmooth:mesh
  Uflux:mesh
  Vflux:mesh
  H:mesh
  PotVel: mesh
returns PUV_tuple =
```

```
let
  var
    P, U, V: mesh;
    i, j, ie, iw, jn, js, deltat_8, deltat_d: number;
  assign
    deltat_8 = deltat / 8.0;
    deltat_d = deltat / delta;
in
  for j from 0 to M do
    = for i from 0 to M do
      jn = if j = 0 then M else j-1;
      js = if j = M then 0 else j+1;
      ie = if i = M then 0 else i+1;
      iw = if i = 0 then M else i-1;
      P[j, i] =
        Psmooth[j, i] - deltat_d * (Uflux[j, ie] - Uflux[j, i]) -
          deltat_d * (Vflux[js, i] - Vflux[j, i]);
      U[j, i] =
        Usmooth[j, i] + deltat_8 * (PotVel[js, i] + PotVel[j, i]) *
          (Vflux[js, i] + Vflux[js, iw] + Vflux[j, iw] + Vflux[j, i]) -
           deltat_d * (H[j, i] - H[j, iw]);
      V[j, i] =
        Vsmooth[j, i] - deltat_8 * (PotVel[j, ie] + PotVel[j, i]) *
          (Uflux[j, ae] + Uflux[j, i] +
           Uflux[jn, i]+Uflux[jn, ie]) -
           deltat_d * (H[j, i] - H[jn, i])
    returns()
  returns(P,U,V);

def Smooth
  M: number
  X:mesh
  Xsmooth:mesh
  Xnext: mesh
returns mesh =
let
  const
    Alpha  = 0.001;
  var
    SmoothX: mesh;
    i, j: number;
in
  for j from 0 to M do
    = for i from 0 to M do
      SmoothX[j,i] = X[j, i] + Alpha
                     * (Xnext[j, i] - 2.0 * X[j, i]+Xsmooth[j, i])
    returns()
  returns(SmoothX);

def Shallow
  M:number
  LastIter: number
returns number =
let
  const
    delta = 1.0E5;
  var
    P, U, V, Psmooth, Usmooth, Vsmooth: mesh;
    Uflux, Vflux, H, PotVel, Pnext, Unext, Vnext: mesh;
    iter, deltat: number;
  assign
    iter = 0;
    deltat = 90.0;
```

```
      P, U, V = Initialize M delta;
      Psmooth = P;
      Usmooth = U;
      Vsmooth = V;
  in
    while (iter < LastIter) do
      Uflux, Vflux = Fluxes M P U V;
      H = Height M P U V;
      PotVel = Potential M P U V delta;
      Pnext, Unext, Vnext = TimeStep M deltat delta
                            Psmooth Usmooth Vsmooth
                            Uflux Vflux H PotVel;
      new P, new Psmooth,
      new U, new Usmooth,
      new V, new Vsmooth = if (iter = 0) then
                             Pnext, P,
                             Unext, U,
                             Vnext, V
                           else
                             Pnext, Smooth M P Psmooth Pnext,
                             Unext, Smooth M U Usmooth Unext,
                             Vnext, Smooth M V Vsmooth Vnext;
      new deltat = if iter = 0 then
                     deltat * 2.0
                   else
                     deltat;
      new iter = iter + 1;
    returns(P[0,0]);

  Shallow M LastIter;
```