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

Parallel Manipulation of Arrays in SISAL

TR 112 078 R

D. Abramson †
J-L. Gaudiot §
W. Heath †

† Division of Information Technology
C.S.I.R.O.
c/o Department of Communication and Electronic Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001
Australia.

§ Department of Electrical Engineering-Systems,
University of Southern California,
Los Angeles, California, U.S.A

Draft 1.0 November 1988

ABSTRACT:

SISAL is an applicative, single assignment language designed for parallel computing. It allows easy side-effect free expression of parallel algorithms without the need to consider parallel programming issues such as synchronisation. A major efficiency problem with many applicative languages is that they require much more data copying than imperative languages. Much work has already been performed on reducing the amount of copying performed by the SISAL runtime system. However, in many cases these optimisations severely reduce the available concurrency. This paper examines some examples in which many arrays are constructed in parallel and must be combined into one result, which would normally involve a large amount of copying in SISAL. We propose a new reduction operator called *merge*, which both simplifies the solutions and also can substantially reduce the amount of copying. An implementation technique is described which allows *merge* to operate efficiently.

1. INTRODUCTION

SISAL is an applicative language based on VAL [1] which has been designed by a consortium of industrial and research organisations [2] for the specification and execution of parallel programs. It differs from conventional imperative languages by

- providing a safe execution model free of synchronisation constraints
- using a single assignment rule
- not allowing global variables
- providing a powerful parallel loop construct

SISAL can be compiled for a number of different machines and architectures. Code can be generated for shared memory machines like the Sequent Balance [3], Encore Multimax [3] and Cray X-MP; message passing machines like the Inmos Transputer [4], and dataflow machines like the Manchester machine [5] and the RMIT/CSIRO dataflow machine [6].

The most mature code generator is the shared memory version, developed by Colorado State University (CSU) in collaboration with Lawrence Livermore National Laboratory [3]. This runtime system includes a reference counting scheme to assist in detecting when structures are no longer being accessed, but more interestingly, to determine when it is possible to perform *update in place* rather than *data copying*.

A problem experienced with SISAL is that it is not possible to specify parallel updates on large structures. Further, the optimisations used in the CSU run time system allow efficient update-in-place for serially executed loops, but cannot assist with parallel loops. This paper addresses both of these issues. First, we suggest a new *reduction* operator for SISAL, which allows concise specification of parallel updates on large structures, without loss of functional semantics. Second, we suggest an efficient implementation for use in the run time system. The technique is demonstrated by a number of examples.

2. LOOPS, REDUCTION OPERATORS AND ARRAYS

SISAL provides two different loop constructs, one for serial *while/repeat* loops and the other for parallel *for* loops. Serial loops can refer to values produced in previous iterations using the prefix *old* before the variable. Thus, it is possible to carry information from one iteration to another in serial loops. Because the language is applicative, every expression must return a result. Loops may return a number of different types of result by using special *loop reduction* operators. Current reduction operators allow the following results of a loop:

value	return the **last** value computed by the loop expression
least	return the **minimum** value computed by the loop expression across all iterations
greatest	return the **maximum** value computed by the loop expression across all iterations
array	collect all loop results and place them in an array
catenate	collect all loop results and catenate them into an array
sum	add all the loop results into one result

The combination of loops and reduction operators provides an extremely powerful facility for building and operating on arrays. To illustrate this power, the following serial and parallel loops construct an array which contains the square of the index in each cell.

```
for index in 1,arraysize


    cell := index * index

    returns array of cell
end for
```

```
for initial
    index := 1;
while index <= arraysize repeat
    cell := index * index;
    index := old index + 1;
    returns array of cell
end for
```

Parallel Loop Serial Loop

A special *replacement* operator is provided for producing a new array which is identical to an old array except for one element which has been changed. For example, the following code would create a new array A2, which was identical to A1 except that element j had been changed to newval.

```
A2 := A1[j:newval];
```

The replace operator makes it possible to apply changes iteratively to a structure by using a serial loop.

3. SOME INSTANCES OF ARRAY MANIPULATION IN SISAL

This section considers a number of different classes of problems, each of which requires parallel access, and in some cases update, to some large structure. These problems are quite difficult to solve using the existing SISAL constructs because they yield unwieldy code and have inefficient implementations.

The first example illustrates problems of sparsely updating arrays in parallel. The second example considers a divide and conquer algorithm which manipulates very large structures. The third problem is one which is traditionally hard to solve in SISAL, for it involves several processes collaborating in constructing a histogram (implemented as a shared structure). In the last instance, synchronisation is implemented by sharing the access to the same global structure.

3.1 Sparse Updating of Arrays

Consider the following FORTRAN loop and its equivalent in SISAL, which sets the elements of an array to zero given index values held in a second array:

```
   do 10 i = 1,updatearraysize        for initial i := 0;
                                          while i <= updatearraysize repeat
      mainarray(updatearray(i)) = 0            mainarray := old mainarray[updatearray[i]:0];
                                               i := old i + 1;
10 continue                               returns value of mainarray
                                      end for
```

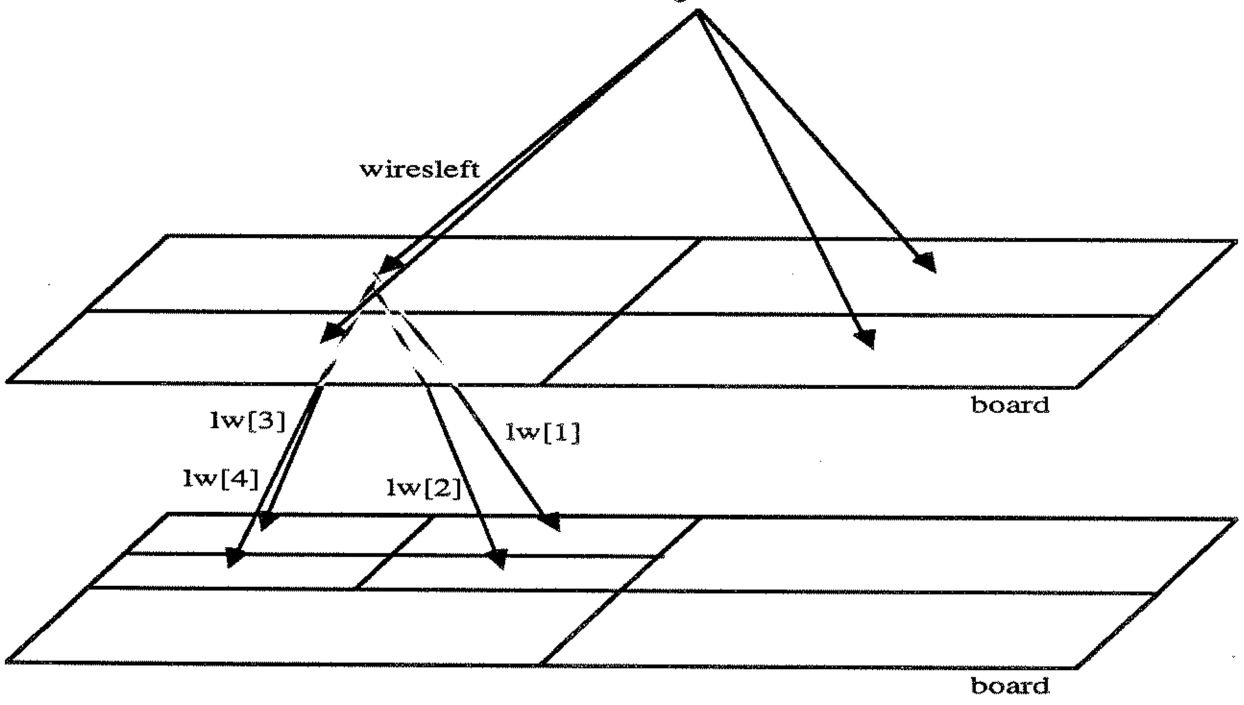
FORTRAN SISAL

It may appear that the SISAL code would involve much more work than the FORTRAN because each loop iteration requires a complete copy of the *mainarray* structure; logically there are as many *mainarrays* created as iterations of the loop. However, the SISAL compiler and runtime system plants reference counting code to determine whether the update cannot be made in place, or whether the creation of the new array simply involves a pointer reassignment. In the example, it is clear that no other code requires the old *mainarray* structure, thus the updates can be performed on the original array. This scheme is implemented in the CSU SISAL runtime system.

A problem with SISAL is that there is no way to specify that the above updates could be performed in parallel, even though each loop iteration is independent of the other. In fact, in this example, it is not important whether the *updatearray* contains any duplicates because they all assign the same value.

3.2 Divide and Conquer on Large Structures

A more complex example is that of a parallel circuit board router algorithm [7]. This algorithm places the tracks which connect the points of a printed circuit board by dividing the board into four routing areas recursively. Prior to each subdivision the wires are split into 5 lists; one list for the wires which are completely contained in each of the four quadrants, plus one list of wires which do not have both endpoints in any of the new quadrants. The routine *route* returns with either no wires left in the wire list, or a number that could not be routed in the given area. These 4 lists are then merged together again, and the remaining wires routed. Any wires that can not be routed are returned to the calling instance of *route*.



```
procedure route ( input     board, wiresleft, bounds;
                  output    board, wiresleft)
begin
    if keepsplitting(...) then
        begin
            newbounds := computemidpoints(bounds)
            splitwires(newwires,wiresleft);
            for corner in 1,4 do
                    fork(route(board, newwires [corner], newbounds[corner]));
            mergewires(wiresleft,lw);
        end;
    process(board,wiresleft, bounds);
end;
```

It is possible to spawn the four calls to route as parallel procedure calls because each operates in a different area of the board and with different wire lists. The structure *Board* holds the state of each grid position on the board. Even for modest sized boards it is quite large. It is possible to simply pass a pointer to the main board structure to the four calls, and no data is copied between calls.

However, it is difficult to specify an equally efficient program in SISAL. The algorithm remains basically the same. However, *route* must return the appropriate section of the board as a new sub array, which can then be recombined later to form a composite board.

```
function route (    board: Grid;
                    wiresleft :Wirelist;
                    bounds :corners;
                    returns Grid, Wirelist)

  if keepsplitting(...) then
    let
    newbounds := computecorners(bounds);
    newwires:= splitwires(wiresleft);
    newboard, wiresnotdone :=
    for corner in 1,4
      newboard,wires:= route(board,newwires[ corner ],newbounds[ corner ]);

      returns array of newboard, value of catenate wires
    end for

    currentwiresleft := wiresleft || wiresnotdone;
    currentboard := mergeboard(newboard);
    in
    processboard(currentboard,currentwiresleft, bounds)
    end let
  else
    processboard(board,wiresleft, bounds)
  end if

end function;
```

It is clear from these two code fragments, that the SISAL version must involve much more computation, and even worse, use much more space. It is not possible specify that the board structure can be safely shared between calls to the function route because the algorithm keeps each instance separate.

3.3 The Histogram Problem

Often one wishes perform some operations in parallel, each of which yields a frequency, and then to add frequencies into a histogram in parallel. Below is a serial FORTRAN program which shows the type of operation. The function, *compute*, forms a histogram address in the array histogram. The cell is then incremented. The equivalent serial SISAL code is shown on the right.

```
   do 10 i = 1,samplesize             for initial i := 0;
      compute(histaddr)                  while i <= samplesize repeat
                                            histaddr := compute(...);
      histogram(histaddr) =                 newval := old histogram [histaddr] + 1];
                histogram (histaddr)+1   histogram := old histogram [histaddr:newval];
                                            i := old i + 1;
10 continue                              returns value of histogram
                                      end for

                FORTRAN                         SISAL
```

A parallel solution is shown below. It involves splitting the computation across the processors, each producing a histogram corresponding to its portion of the input space. The partial histograms are them summed in parallel to form the global histogram. A drawback of the solution is that many partial histograms must be formed before they are added together, rather than one structure being updated as the computations are performed.

It should be noted that there has been an important departure from "accepted" functional programming language philosophy: while the programmer is usually to be concerned with the structure of the problem rather than the structure of the machine, it has been recognized here that spawning of processes would only occur efficiently if the number of processors available for the computation was explicitly specified.

```
function hist (localsamplesize : integer ; s: OneDim ; returns OneDim )

  for initial
    slot := s ;
    i := 0;

    while i < localsamplesize
    repeat
    i := old i + 1  ;
    histaddr := compute(...);
    slot := old slot [ histaddr : old slot[histaddr] + 1]
    returns value of slot
  end for
end function % hist

function mergehist ( samplesize,numprocessors : integer ; slot : OneDim ;
          returns OneDim)

 let
    newslot := for j in 1,numprocessors
          returns array of hist ( samplesize / numprocessors, slot)
        end for;
 in
    for i in 0,array_size(slots)-1
      sumofelements :=
      for j in 1,numprocessors
        returns value of sum newslot[j,i]
      end for
     returns array of sumofelements
    end for
 end let
end function % mergehist
```

3.4 The Arbitration Problem

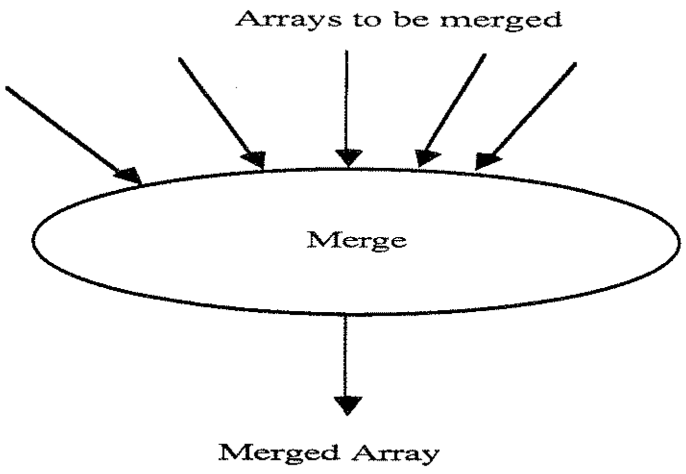
The arbitration problem occurs in computations which are spread across several processors, but require mutual exclusion and synchronisation when accessing some shared resources. This problem can be encountered in parallel Monte-Carlo algorithms such as simulated annealing problems [8]. Each of the processors chooses a number of shared resources from a pool, and then must prevent any other processor from gaining access to the same resources. Traditionally, this problem can be solved with semaphores or indivisible locks. These constructs are not functional and are not available in SISAL.

This problem has many similarities with the histogramming problem described in the last section. A solution can be found by allowing each processor to choose in parallel the resources it requires, and then forcing them to *vie* for the resources before proceeding. Arbitration can then be implemented by representing the requests of each processor in a structure, and then applying an arbitration process to these structures. The results of the arbitration process can then be read by each processor to determine which requests were successful. This scheme is shown below. A list of required resources is constructed by each worker in parallel. A conflict vector is created with one entry per resource, and is initialised with large values. The conflict vector is then processed for each worker, and the worker number is stored in the appropriate entry if it is less than the current value. At the end of the arbitration process, each worker can examine the conflict vector to determine whether it has been successful. A major problem with this approach is that the arbitration process must be sequential in order to be efficient because, typically, the number of resources is much larger than the number of workers. The effect of a slow serial arbitration is that the performance is lowered in accordance with Amdahl's law.

```
% conflict_vector is initialised to a large value

let

 resource_list :=
     for w in 1,numworkers
        resource := choose(w);
     returns array of resource

 end for;   %build list of chosen resources, one for each worker

  in

 for initial
     w := 0;
 while w < numworkers repeat

 %  process resources, giving priority to low numbered workers

     w := old w + 1;
     conflict_vector := if old conflict_vector[resource_list[w]] > w then
        old conflict_vector[resource_list[w]:w] else
        old conflict_vector
     end if
     returns value of conflict_vector
 end for
 end let

% Now proceed to check whether successful
```

4. A NEW REDUCTION OPERATOR

All of the problems raised in the previous section can be overcome by using a new reduction operator which allows the programmer to specify that a number of structures are to be combined to form one new structure. At the graph level, merge can be viewed as a node which accepts a number of arrays and produces one new array.



By adding a number of modes to the operator, it is possible to provide a wide range of functions.

4.1 Unique Merge

The basic operator, called *merge*, accepts a number of arrays, and merges them into a new array. The cells of the new array can take on values in three cases.

1) If all of the corresponding elements of the arrays to be merged have the same value, then the element of the reduced array is the same as the corresponding elements in the original arrays.

2) If one element differs from all other corresponding elements in the arrays to be merged then the differing value is placed in the reduced array

3) If more than one element differs from corresponding elements in the merged arrays, then the corresponding cell in the reduced array is set to *error value.*

The following examples illustrate the above definitions.

Elements Merged	Result
7 7 7 7 7 7 7 7 7	7
10 10 8 8 7 7 7 7 7	Error
10 7 7 7 7 7 7 7 7	10
10 10 7 7 7 7 7 7 7	Error
10 0	0

The unique merge can be used in the sparse updating of arrays problem, and also in the circuit router. The resulting code is much simpler, and it is possible for the runtime system to make some optimisations. The following code can replace that shown in 3.1 and 3.2.

```
for i  in updatearraysize

   reducedarray := mainarray[updatearray[i]:0];
   returns value of merge reducedarray

end for
```

```
function route (    board: Grid;
                    wiresleft :Wirelist;
                    bounds :corners;
                    returns Grid, Wirelist)

   if (keepsplitting) then
      let
      newbounds := computecorners(bounds);
      newwires:= splitwires(wiresleft);
      currentboard, wiresnotdone :=
      for corner in 1,4
         newboard,wires:= route(board,newwires[ corner ],newbounds[ corner ]);

         returns value of merge newboard, value of catenate wires
      end for

      currentwiresleft := wiresleft || wiresnotdone;
      in
      processboard(currentboard,currentwiresleft, bounds)
      end let
   else
      processboard(board,wiresleft, bounds)
   end if

end function;
```

4.2 Sum Merge

The unique merge operator can be usefully extended by the introduction of the sum merge. This variant operates similarly to the unique merge, but corresponding elements in the input arrays are added together to form a merged array. This is in contrast to the original merge, in which only one element of the input arrays contributes to the final result. The sum merge provides an extremely elegant solution to the histogram problem, as shown below. The programmer no longer needs to form and sum many separate histograms, because the addition is performed by the reduction operator.

```
   for i  in samplesize

      histaddr := compute(...);
      newhistogram :=  histogram [histaddr:1];

      returns value of sum merge newhistogram

   end for
```

4.3 Least and Greatest Merge

One further extension to the basic merge is the least/greatest merge. It differs from the conventional merge in that the contributing element is the minimum/maximum of the corresponding elements of the input arrays, rather than being the differing element. This form of merge provides a solution to the arbitration problem, in which the entire arbitration process can be performed in parallel. The new code is shown below. It replaces the code in italics in 3.4.

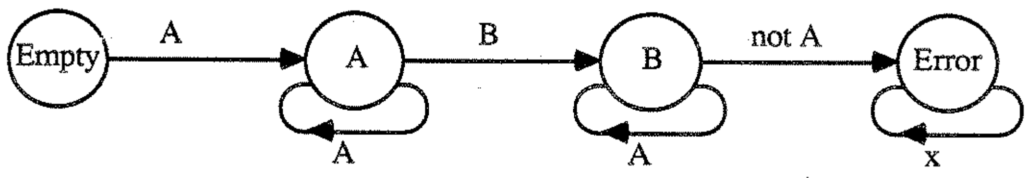
```
% process resources, giving priority to low numbered workers

      for w in 1,numworkers

      newconflict_vector := conflict_vector[resource_list[w]:w]
         returns value of least merge newconflict_vector
  end for

% Now proceed to check whether successful
```

5. AN IMPLEMENTATION TECHNIQUE

There are a number of different techniques for implementing the merge operators. This section describes a simple method, and then proposes some optimisations which allow the update to occur in place.

5.1 Simple Implementation - All arrays present

In the simplest implementation of merge, all of the arrays are produced in parallel, and are then merged element by element to produce the new array. Since all of the input arrays are available, the merge can simply sweep across the input elements. In the case of unique merge, only one differering value is allowed. In sum merge, the corresponding elements are added together. In the least/greatest merge the minimum/maximum value is chosen in each sweep. The unique merge can be controlled by a finite state machine, as shown below.



The name of each state indicates the output value of the merge for any particular element of the output array. The values *A* and *B* indicate different input values. The value *x* indicates any value.

5.2 Simple Implementation - one array at a time

In many cases not all of the arrays will be available at the same time, and therefore cannot be reduced simultaneously. It is still possible to implement an incremental merge, however, extra data must be stored along with the reduced array. It is necessary to store the the values of *A* and*B* for all elements or the output array, as well as the state of each finite state machine.

In the case of sum, least and greatest merge, it is not necessary to retain any state information because the incoming value can simply be added, or compared, to the current output value.

5.3 Update in Place

One of the most attractive features of the merge operator is that it is possible to recognise that a number of structures are being merged into one new structure. Under certain circumstances it is possible to perform the merge *in place*, rather than by constructing a number of temporary arrays and then merging them. Update in place is clearly more efficient because it removes the need to construct the temporary arrays, and also removes the merge as a separate operation. An update in place can occur when

1) No consumers of the array are active at the time of the merge
2) The resultant array is produced by a number of independent iterations
3) Each of the contributing iterations is using the *array-replace* operator in forming their sub-array

Under these conditions each loop iteration is producing new arrays which differ from the same input array. If no other computation requires the input array, then this input array can be directly modified by the merge, without the need to construct new structures. The reference counting scheme used in the CSU runtime system can be used to test condition 1. Conditions 2 and 3 can be detected by some relatively simple graph analysis.

The semantics of the merge operator arbitrate *producer-producer* conflicts and clearly define the outputs to be obtained in those conditions. Thus is it possible to have many independent loop iterations contribute to one merged array without the need for separate temporary arrays. In this section we describe two different methods for implementing the producer-producer conflict resolution when the update is performed in place. Consumer-producer conflicts, which occur when a loop instance both consumes a structure and produces a new structure, will be dealt with in section 5.4. The sum, greatest and least merge operators do not require any producer-producer conflict resolution.

5.3.1 State Vectors

The simplest technique for detecting multiple producers is to attach a state bit to each element of the array. Initially all of the state bits are cleared, and as each element of the array is filled, the corresponding state bit is set. If any producer attempts to fill a cell which is already full, then an error value may be placed in the cell.

5.3.2 One Time Identifiers

A problem with using state vectors on very large arrays is that the state vector must be cleared before each parallel loop which merges a result into the array. This clear operation may take a long time. The solution consists of expanding the single bit state vector values to many bits per element. For example, each cell may be associated with an 8 bit state value. Each array has an additional state value called a *one-time-identifier* [9], which is initially set to zero. Similarly, the state values associated with the elements are also set to zero. Each time a parallel loop which merges into the array is started, the *one-time-identifier* is incremented. When a data item is stored in an element the state value is examined. If the state value is equal to the *one-time-identifier* then the error value is substituted for the original data. If they differ then the data is stored in the element, and the corresponding state value is set to the *one-time-identifier* value.

This technique has the advantage that if only a small section of the result array is modified by any loop, then the entire state vector need not be cleared. Only those cells which are active have their state values manipulated.

5.4 Producer-consumer conflict

In section 5.3 the conditions which allow update-in-place to occur are outlined. The first condition is that no other consumers of the array are active at the time of the merge. In almost all of the examples of the use of merge given in section 4 the merging loops did not yield any producer-consumer conflicts. Conflicts were not present because each loop only used an *array-replace* operation, and did not examine the contents of the array. However, the code for the circuit router presented in section 4.1 did contain a producer-consumer conflict. Thus, the *board* structure could not be updated unless the code which examined it was separated from the code which produced a new board.

The solution to the problem involves separating the code which examines the structure, and have it return a list of changes to the *board* structure. A separate for-all loop could then take the changes and apply them.

The producer-consumer problem described is not peculiar to the merge operation, but occurs in any array manipulation which both examines and produces a structure concurrently.

6. CONCLUSION

In this paper, we have described a new reduction operator for SISAL, which will enable the programmer to specify complex array interaction operations. In addition, we have demonstrated implementation schemes for both the simple semantics of the operator as well as more advanced semantics. We have shown how update-in-place can be used for efficient implementation of some array merge operations. Our reduction operator is not specific to any class of multiprocessor but can instead be used on shared memory machines as well as message passing systems. A prototype version of the merge operator has been included in our local version of the LLNL SISAL compiler and of the CSU runtime system.

Further research issues will include the implementation by parallel merge reduction trees. Alternatively, unconventional communication networks such as the NYU Ultracomputer "Fetch-And-Add" network could be used efficiently by merge add, for example.

Acknowledgements

The authors wish to acknowledge the support of the project team, in particular Dr G. Egan, P. Chang, M. Rawling, S. Wail, N Webb, P. Whiting and A. Young. The Parallel Systems Architecture Project at RMIT is being jointly supported by RMIT and the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) under an Information Technology joint research project. This material is also based upon work supported in part by the US Department of Energy, Department of Energy Research, under Grant No DE-FG03-87-ER25043.

References

[1] J.R. McGraw. "Data-flow computing: the val language". ACM Transactions on Programming Languages and systems, 4(1), Jan. 1982.

[2] J.R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J.R.W Glauert, I. Dobes, and P. Hohensee. "SISAL-Streams and Iterations in a Single Assignement Language", Language Reference Manual, Version 1.2. Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.

[3] R. R. Oldehoef and D. C. Cann. "Applicative parallelism on a shared-memory multiprocessor". IEEE Software, January 1988.

[4] J.-L. Gaudiot and L.T. Lee. "Occamflow: a methodology for programming multiprocessor systems". Journal of Parallel and Distributed Computing, In Press 1989.

[5] J.R. Gurd, C.C. Kirkham, and I. Watson. "The Manchester data-flow prototype". Communications of the ACM, 28(1), January 1985.

[6] D. Abramson and G.K. Egan, "An Overview of the CSIRO/RMIT Parallel Systems Architecture Project", Australian Computer Journal, Vol 20, No 3, August 1988.

[7] D. Abramson and J. Freidin, "A Parallel Router for Printed Circuit Boards", Department of Communication and Electrical Engineering RMIT Technical Report, in preparation.

[8] D. Abramson. "Constructing School Timetables using Simulated Annealing: Sequential and Parallel Algorithms", Department of Communication and Electrical Engineering RMIT Technical Report TR112 069R.

[9] A.J. Smith, "Cache Consistency Support using One-time Identifiers", Proc. Pacific Comp. Comm. Symp, Oct 1985.