Virtual Parallelism by Self Simulation of the Multiple Instruction Stream Associative Model

Darrell R. Ulm and Johnnie W. Baker
Department of Mathematics and Computer Science
Kent State University: Kent, OH 44242 U.S.A.
dulm@mcs.kent.edu jbaker@mcs.kent.edu

Abstract
The ASC model for parallel computation supports a generalization of an associative style of computing that has been used since the introduction of associative SIMD computers in the early 1970's. In particular, this model supports data parallelism, constant time maximum and minimum operations, one or more instruction streams (ISs) which are sent to an equal number of partition sets of processors (PEs), and assignment of tasks to the ISs using control parallelism. Since problems often need more processors than a machine has, it is useful to have virtual processors simulated by existing architectures. This paper shows how virtual parallelism is possible where more PEs and ISs are simulated than the actual hardware possesses. The extra time needed for an ASC(n,j) machine to simulate an ASC(N,J) machine where $N \geq n$ and $J \geq j$ is $O(\frac{N}{n} + \frac{J}{j} + min(\frac{N}{n}, J) \times min(n, \frac{J}{j}))$ while the extra space required is $O(\frac{N}{n} + J)$ per PE. The overall results provide an important step in defining the usability of the ASC model in terms of both existing one IS machines and future machines supporting multiple ISs.

Keywords : computational models, simulation, associative computing, ASC, data parallel, massively parallel, SIMD, MSIMD, virtual parallelism, parallel algorithms

1 Introduction

Virtual parallelism allows the number of processors in terms of the programmers view to be larger than the actual number of processors. Self simulation algorithms ensure that large problems can be solved on a virtual machine without the user having to 'hard-code' his program for the actual number of processors on the machine. Here we will show a method for self simulation on an associative architecture with the possibility of having more than one instruction stream. The simulation algorithm is general enough for the host architecture to be essentially SISD, SIMD, or MIMD, and some comments are made later in this paper concerning how well such a simulation would work in a distributed environment. In the case where the host machine is actually the same model that the simulation uses (e.g. ASC or MSIMD) the assumption is made that that the communication steps take constant time. Times are also given considering times needed for the required communication on generic networks.

2 The multiple IS associative computing model (ASC)

This section introduces the associative model of computation presented in the IEEE Computer article "ASC: An Associative Computing Paradigm," which is based on work done at Kent State University[1]. In this paper algorithms are investigated for ASC to simulate an ASC machine with more processors and instruction streams. In the most basic terms, the *associative model (ASC)* has an large number of processing elements (PEs) and one or more instruction streams (ISs) that broadcast their commands to partition sets in a dynamically reconfigurable partition of the PEs. A likely implementation of ASC is to have one electronic or optical bus joining each IS to all PEs although many networks could be used to connect these components. An ASC machine with j ISs and n PEs will be written as $ASC(n, j)$. The number of ISs is normally expected to be small in comparison to the number of PEs. The multiple ISs supported by the ASC model allows greater efficiency, flexibility, and reconfigurability than is possible with only one IS. Typically, an IS is not to be confused with a host processor which sends its instructions rather inefficiently to PEs over a loosely coupled network; instead an IS sends its instructions to the PEs over a bus, and the ISs should be considered to be local to the PEs in terms of hardware. Each PE has a local memory and ASC supports the associative searching concept, which is to locate objects in the memory by content instead of by location. This is accomplished by checking a specified field of each active PE for a given data item[2]. Each PE is capable of performing local arithmetic and logical operations and the other usual functions of a sequential processor, but each PE is assumed to be reasonably basic so that a maximum number of PEs can be integrated on a chip. It is argued that this trade off of complexity for more processors gives a better cost for performance ratio over implemented MIMD system[3]. ASC is a model for a currently buildable, massively parallel multi-purpose computer which is easy to program and can execute many types of programs efficiently. A parallel computer with these characteristics is greatly needed if parallel computing is ever to be generally accepted by industry and the general public. The ASC model is intended to standardize the meaning of associative computing and to provide a basis for complexity analysis for associative algorithms. Moreover a programming language (also called ASC) has been designed for ASC with one instruction stream and implemented on several SIMD computers including Thinking Machine's CM-2, the Wavetracer, and the Goodyear/Loral ASPRO. Plans are in the works for a distributed version of ASC that could run on any UNIX system that supports ANSI-C and the MPI communication library. In addition, an efficient simulator has been implemented on both PCs and Workstations running UNIX[4][1]. A wide range of different types of algorithms and several large programs have been implemented using the ASC language including a parallel optimizing compiler for ASC, two rule-based inference engines, and an associative PROLOG interpreter[4][1].

Furthermore, the associative computer is assumed to be equipped with an interconnection network between the PEs[5]. There is no restriction on the networks allowed for the ASC model and the network could potentially be the mesh, shuffle-exchange, Batcher's Flip Network, hypercube, or the De-Brujiun network[6]. However, some of the most obvious choices are a linear array or a mesh because of their ease to implement in VLSI and their expandability. It is also reasonable that there is some network connecting the ISs however it is not necessary since IS communications could be handled through the network connecting the ISs to the PEs. While all of the algorithms and programs mentioned at the end of the preceding paragraph were implemented without the use of any network, numerous associative programs using one IS have also been implemented using a network. The fact that the algorithms and programs mentioned in the preceding paragraph have been implemented without the use of a network illustrate that the simplest ASC model is quite powerful. In the remainder of this paper, we will not assume the ASC model has a network, which will determine how well the single IS and the multiple IS models without an interconnection network perform self simulation. A diagram of ASC is shown in Figure 1.

3 Simulation of a parallel model

When the *simulation* of a model is performed, there are various operations to consider. For parallel models, the operations that need to be simulated are parallel execution of processors and the data movement between processors[7][8]. These operations are defined by mapping processor resources from one model to another in terms of the operations that need to be simulated. When the time complexity of each operation performed in a cycle of the simulation is divided by the complexity to perform the same operations on the machine being simulated, the maximum resulting time gives the slowdown of the simulation and also an indication of the relative powers of the two machines [9][10][11][12][13]. The crucial operations for simulation of the associative model (ASC) are specified in the following text. Assumptions concerning control parallelism and IS synchronization times are also given in order to simplify the simulation somewhat[3]. Another form of simulation is *self-simulation* (e.g., *virtual parallelism*) which is useful when real problems exceed the size of a machine and virtual processors are

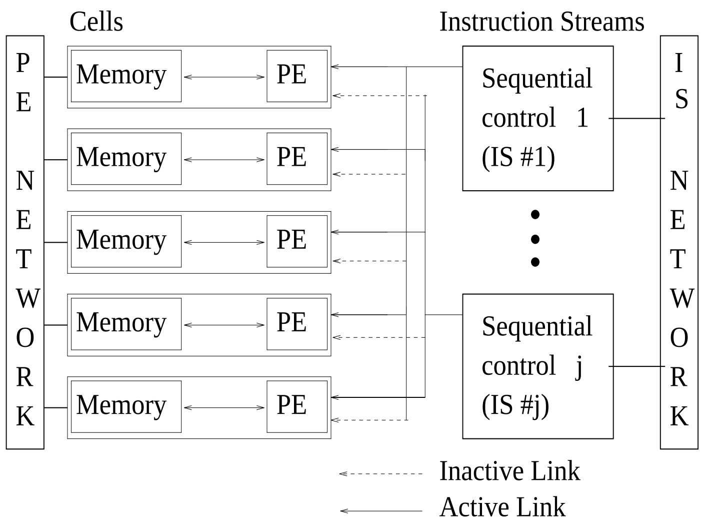

Figure 1: The ASC Model

needed by an algorithm since no machine can have an infinite number of processors. Self-simulation shows how efficiently a parallel machine can execute an algorithm for a problem requiring more processors than are available[14]. A diagram showing an example of ASC virtual parallelism is shown in Figure 2.

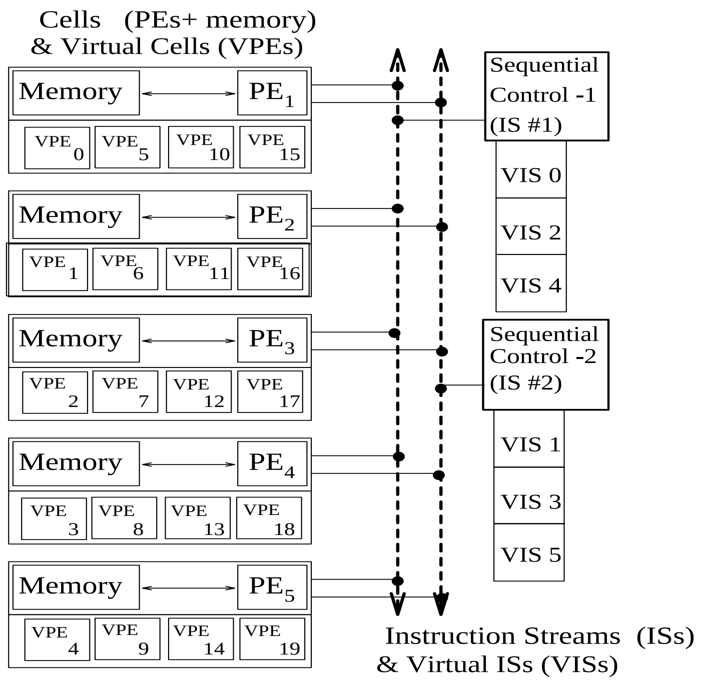

Figure 2: Example of Self Simulation on the ASC Model

4 Properties of ASC

The properties of the ASC model are in this section with the rules that govern each property of ASC, and the rules are in list form for ease of reading.

1. THE PROPERTIES OF THE CELLS:

- Each cell contains a processing element (PE) and local memory.
- The memory of the associative processor is an array of cells.

- Each PE may only access the memory contained in its own cell.
- All the PEs are connected by a network (e.g. a linear array or a grid).
- Related data items are grouped into records. In addition, more cells are assumed to exist than records, and at most one record is stored in each cell.

2. THE INSTRUCTION STREAM (IS) PROPERTIES:

- Each IS is a processor with a bus (actual or logical) to all cells. The IS processors themselves are connected in some way (e.g. by shared memory, bus, or generic network), and can communicate control information with each other in constant time. Each IS has a copy of the program being executed and may broadcast an instruction or a word sized data item to all cells in unit time. The PEs execute the IS instructions synchronously.
- Each cell listens to only one IS. Some or all of the cells can switch to a different IS in response to instructions from the current IS and local data in a cell.
- An active cell executes the instruction sent from its IS, while an inactive cell listens to its IS but does not execute any of its commands. An IS may unconditionally activate all cells listening to it.
- The number of cells is much greater than the number of IS's.

3. ASSOCIATIVE PROPERTIES:

- An IS can make all its active cells execute an associative search in constant time. Cells which test positive on the search are called responders while the unsuccessful active cells are called non-responders. The IS can make either the set of responders or non-responders active. The previous set of active cells can also be restored. Any of these actions are performed in constant time.
- Each IS has a dedicated responder bus.
- An IS can select a cell from the list of active cells in constant time. The IS cannot specify which cell is selected.
- An IS can command the selected cell to place data on the responder bus. The other cells listening to this bus receive this data value in constant time.

4. CONSTANT TIME GLOBAL OPERATIONS:

- An IS can compute the AND or OR of a boolean value in all active cells in unit time.
- An IS can find the cells containing the maximum or minimum of a value stored in each active cell of its PEs in constant time.

5. CONTROL PARALLELISM:

- Cells with nothing to do are called idle cells and are assigned to a specified IS which manages idle cells. A subset of cells can be deallocated and reassigned to the group of idle cells in unit time. Any idle cell can be allocated to an IS in unit time.
- When an IS is running a task that needs the results from two or more subtasks involving data in strictly disjoint subsets of the active cells, control parallelism can be invoked by assigning subtasks to idle ISs. The cells are returned to the originating IS when all of its subtasks are completed.

5 Self simulation of ASC

The self simulation algorithm performs two basic operations. The first is to simulate more PEs and ISs than actually exist so that extra processor resources are actually obtainable. Virtual PEs (VPEs) are simulated with existing PEs where each PE contains roughly an equal amount of virtual PEs. Thus each VPE must have space allocated for registers, ports, and memory as a real PE would possess. Similarly virtual ISs (VISs) are simulated by the ISs and need storage for program counters, registers, and local memory.

The second part of the simulation is to write ASC code for the real machine that will simulate a single cycle of the virtual ASC machine. This code must be able to execute local VPE operations, update VIS information as dictated by program flow and handle the transfer of data between VPEs and VISs. An ASC machine can only execute a finite number of operations, some of which transfer data between PEs and ISs. The data movement operations include broadcasting a value from an IS to a set of PEs, reading a value from a single PE to an IS, and performing data reduction from a set of PE data to a single value in an IS. The size of the virtual machine is dependent on the amount of memory in each processor, but it can be assumed that enough memory is present to simulate the amount of virtual processors needed.

The algorithm to simulate an ASC(N,J) machine with an ASC(n,j) machine is shown in Figure 4 and Figure 5 where $N \geq n$ and $J \geq j$. Many addressing methods can be used to map VPEs to PEs and VISs to ISs, and the following method is a suggested mapping. The virtual processor VIS_x will be simulated by IS_k where $k = (x \text{ MOD } j)$ and $0 \leq x \leq J-1$. Moreover, VIS_x is addressed by the IS that emulates it by the index $(x \text{ DIV } j)$ where $0 \leq x \leq J-1$. Similarly, VPE_y is emulated by PE_l where $l = (y \text{ MOD } n)$ and $0 \leq y \leq N-1$. VPE_y is addressed inside the cell it is stored with index $(y \text{ DIV } n)$ where $0 \leq y \leq N-1$.

The simulation algorithm has two phases that are executed each machine cycle. First, all instructions are sent to VPEs from the appropriate VISs, and secondly once the virtual instructions needed in a PE are present, each PE carries out the operations of the $\lceil N/n \rceil$ VPEs it simulates. This second phase also involves any operations that transfer data between PEs and ISs. To make the algorithm more efficient, the fact that each PE may not need instructions for all J VISs is employed. Communication operations for one VIS that involve several VPEs in a PE do not need more than a single communication between the PE and IS due to the nature of the ASC operations. Therefore the identifiers of the unique VISs being listened to by the $\lceil N/n \rceil$ VPEs in each PE are saved in a list. This list is of size $O(min(N/n, J))$. Only the opcodes and data of the VISs that appear in a PE's list are ever sent to that PE.

Figure 3 shows a diagram of how the data structures could be stored. The figure shows that each VPE is listening to some VIS, and a $LIST_i$ is formed by only including the unique VIS numbers required by the VPEs. One simple way to determine if a VIS has already been added to $LIST_i$ is including a boolean array of size J where the entry at index m is 1 if the VIS has been added or 0 if it has not yet been added where $0 \leq m \leq J$. Figure 3 shows this boolean array called "$IN\text{-}LIST_i$" for the VISs in the example $LIST_i$. There is also a word array or J entries used as a temporary data buffer for operations in phase 2 of the algorithm that move data either to or from the VPEs in the PE. This array is required so that any data movement operations that involve several VPEs in a single PE only need to rendezvous with an IS once. For instance, a datum broadcast from VIS number 5 is placed at location 5 of the data array. Later, when each VPE executes a local operation, the VPEs expecting a broadcast from IS 5 copy the PE local value from the $5th$ entry of the temporary data buffer. Other data movement operations are carried out similarly.

5.1 Instruction send phase

For any given machine cycle the instruction broadcasting phase in Figure 4 is executed first and then the data movement stage is executed second, as shown in Figure 5, all as part of the same algorithm. The first stage forms lists in each PE of the VIS numbers used. A VIS number is only added to a PE list if it hasn't already been added (see Figure 3). In this way, multiple requests for the same VIS are handled at the same time. As mentioned above, one possible way to check whether a VIS has already been added to the list is to use a boolean

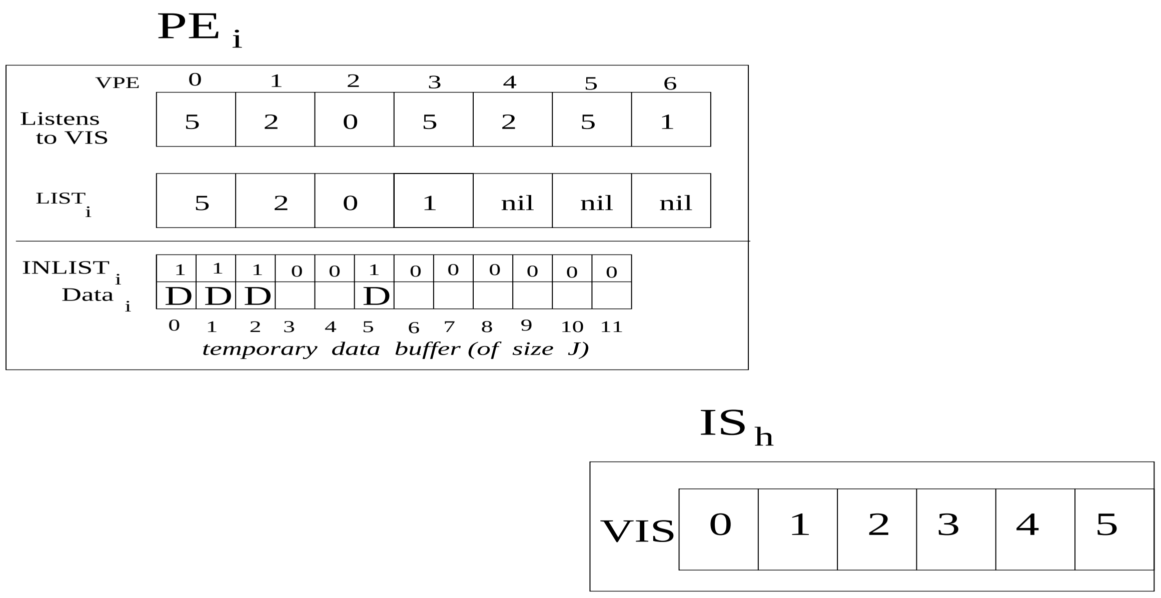

Figure 3: Data Structures used for Simulation inside each PE and IS

array of size J indicating whether a VIS has already been added.

As shown in Figure 4, line 1 handles the simulation of the J VISs with j ISs. There are $\lceil J/j \rceil$ VISs stored in each IS and, this step takes $O(J/j)$ time to simulate the actions of an IS. Such operations include updating program counters, fetching the next instruction for each VIS, performing localized operations, and any other operations an IS can perform. It is assumed that IS synchronization operations execute independently of the simulation as functions executed on top of the real ASC architecture. Step 2 starts the data parallel execution. Line 3 makes all PEs active and listening to IS_0 which is considered the master or initial IS, and line 4 clears VIS request lists. Step 5 loops through $\lceil N/n \rceil$ VPEs in parallel, and step 6 adds a VPE's VIS number to $LIST_i$ if not yet added. Figure 3 shows an example of the data structures used to form $LIST_i$.

The lists of VIS requests are now formed in each PE, and the algorithm is now ready to send the VIS instructions to the appropriate PEs based on these lists. Line 7 proceeds to loop through the lists, which must be of size $\min(\lceil N/n \rceil, J)$ since at worst $\lceil N/n \rceil$ unique VIS requests can exist, and there are only J different VISs to choose from. At line 8 each PE_i retrieves the next unprocessed entry in the VIS $LIST_i$, and line 9 instructs each PE_i to listen to the IS that contains the requested VIS. Step 10 loops while any PEs are still active, or in other words, have not had their VIS request met by an IS. At step 11 all ISs begin acting in parallel, and step 12 commands each IS to arbitrarily select a PE from all active PEs listening to an IS. At line 13 all PEs listening to an IS requesting the same VIS as the arbitrarily chosen PE are also selected while other PEs are temporarily unselected. In line 14 the selected PEs receive the VIS operation, and then these satisfied cells are temporarily made inactive at line 15. The inner loop at line 10 iterates until all PEs receive a VIS operation from an IS which afterwards means that the work done for one entry in the parallel $LIST_i$ is completed. The outer loop at line 7 simply proceeds to the next entry in $LIST_i$. At the end of an outer loop iteration line 16 resets all PEs to and active status listening to the initial IS_0. This step is necessary because when the loop at line 10 is finished, all PEs are inactive but need to be made active for the next iteration. Also, all PEs need to be listening to the same IS at step 7 so the while loop test can be checked by a single IS. The inner while loop at line 10 iterates no more than $O(min(n, J/j))$ times since in the worst case all of the n VISs needed are inside the same IS, however there are at most J/j VISs at each IS. The outer while loop at line 7 iterates $O(min(N/n, J))$ times since this is the maximum length of $LIST_i$. Note that these VIS request lists can be of different sizes. Thus after the outer loop completes at line 7, all PEs have only the VIS operations used by the VPEs they are simulating.

```
PHASE 1: SEND ALL VIRTUAL INSTRUCTIONS PHASE

1 In  $O(J/j)$ time SIMULATE $J$ VISs with $j$ ISs
      (e.g. update program counters, fetch next instruction, etc.)

*** In each cell, form a list of requested VISs for VPEs ***
2 FOR ALL $PE_i$ do in parallel
3     ALL $PE_i$ are now ACTIVE and LISTENING to $IS_0$
4     CLEAR list ($LIST_i$) in $PE_i$
5     FOR $k = 1$ to $\lceil N/n \rceil$ do
6         ADD $VIS_i[k]$ number to $LIST_i$ if not yet a member of $LIST_i$

*** Process parallel lists of VIS requests by looping through responders ***
7     WHILE (any VIS requests in $LIST_i$ are unprocessed)
8         SELECT NEXT $VIS_i$ request in $LIST_i$
9         $PE_i$ LISTENS to the IS that simulates $VIS_i$
10        WHILE (any $PE_i$ has unsatisfied VIS requests)
11            FOR ALL $IS_h$ do in parallel (ISs act in parallel on responding PEs)
12                SELECT an arbitrary $PE_i$ from active cells of $IS_h$
13                SELECT all cells requesting same the VIS as the arbitrary cell
14                ALL ISs BROADCAST VIS information to their selected cells
15                MAKE the selected (satisfied) PEs temporarily INACTIVE
16        ALL $PE_i$ are now ACTIVE and LISTENING to $IS_0$
```

Figure 4: First phase of ASC(n,j) Algorithm which simulates sending J virtual ASC instructions to N VPEs where $N \geq n$ and $J \geq j$

5.2 Perform all instructions phase

The algorithm in Figure 5 performs instructions handling data movement between VISs and VPEs. Lines 17-23 handle local VPE functions and operations that need to move data from the VPEs to the VISs. These operations include a VIS reading a value from a single VPE and a VIS executing a reduction operation on VPE data. Line 17 indicates that data parallel execution is beginning. Step 18 forces all PEs to be active and to receive instructions from the master IS_0. At line 19, there are $\lceil N/n \rceil$ iterations for the VPEs stored in each PE. Line 20 uses a case statement to check if the opcode for each VPE is data movement operation or a local operation. At line 21, when a VIS is reading from a single VPE, the data is put into PE_is temporary buffer at the index of the VIS if the address of the VPE matches the read address in the opcode of the VIS (see "temporary data buffer" in Figure 3). For line 22, data is moved to PE_is temporary buffer after a reduction function (and, or, min, max) is performed with the current buffer data. Step 23 executes local operations for the current set of VPEs. It is assumed that there is a finite number of possible instructions that can be executed on a PE, and in fact the number of opcodes should be very small given the simplicity of an associative cell.

After steps 17-23 are completed, data that is being moved from VPEs to VISs is located in a temporary array for each PE. Steps 24-36 will complete the movement of data from the holding area inside PEs to simulated data registers of the VISs. Lines 24-36 also begins the process of simulating broadcasting VIS data to VPEs by moving data from VIS registers to the temporary data arrays in each PE. The same VIS lists gathered in phase 1 (see Figure 4) are used again in step 24 by looping through each VIS list. In step 25 the next entry of $LIST_i$ is selected in each PE. At line 26 the PEs connect to the ISs that contain the VIS of the selected list entry. Line 27 is an inner while loop that loops until all PEs are considered, or in other words have had their current instruction request performed. Line 28 starts ISs acting in parallel. Each IS selects an arbitrary PE_i in step 29, and in step 30 all other PE_is containing the same current VIS request as the arbitrarily selected PE are also made active.

A case statement for the data movement operations to handle is on line 31. For step 32, where VISs send data to the VPEs, the data is broadcast to selected PE_is from the data registers of the VISs. This data is stored in the J sized temporary data array in each PE. In line 33 a VIS reading from a single VPE is processed by simply moving the data from the temporary data array in the selected PE to the VIS. Reductions are performed in line 34 by using the ASC parallel reduction operation on the data in the selected set of PEs. The data to perform this reduction is in the temporary arrays of the PEs. Finally, the the reduction operation is performed again the value reduced from the PEs on any previously reduced value stored in the VIS. Recall that this ASC operation is assumed to take constant time. The selected set of PEs which have had their operations performed this iteration are made temporarily inactive in line 35. Line 36 makes all PEs active and listening to one IS after the inner loop is finished satisfying one entry $LIST_i$. After lines 24-36 are finished executing, all operations and data movements have been completed for one cycle of execution except the movement of broadcast data from the temporary arrays in each PE to the VPE storage areas. Step 37 finishes the final instructions for broadcasting data from VISs to VPEs. A $\lceil N/n \rceil$ loop is performed where all VPEs that expect a data broadcast from a VIS copy data from the temporary PE_i array into their own simulated data registers. One cycle of ASC execution is simulated, after the instructions in lines 1-39 have run.

```
PHASE 2: PERFORM ALL INSTRUCTIONS PHASE
*** In each PE, collect data being sent to VISs ***
17 FOR ALL $PE_i$ do in parallel
18     ALL $PE_i$ are now ACTIVE and LISTENING to $IS_0$
19     For $k = 1$ to $\lceil N/n \rceil$ do
20        CASE of (instruction) for $VPE_i[k]$
21            [VIS READS from 1 PE]: IF ($VPE_i$ self address = opcode address)
                  COPY the data into temporary $PE_i$ buffer indexed by the VIS number
22            [REDUCE data into VIS]: Perform reduction operation on $PE_i$ temporary
                  buffer and data at $VPE_i[k]$
23            [EXECUTE local operations]: Perform finite number of $PE_i$ local operations

*** Process parallel list of VIS requests by looping through responders ***
24     WHILE (any VIS requests in $LIST_i$ are unprocessed)
25        SELECT NEXT $VIS_i$ request in $LIST_i$
26        $PE_i$ LISTENS to the IS that simulates $VIS_i$
27        WHILE (any $PE_i$ has unsatisfied VIS requests)
28            FOR ALL $IS_h$ do in parallel (ISs act in parallel on responding PEs)
29                SELECT arbitrary $PE_i$ from active cells of $IS_h$
30                SELECT all cells requesting same the VIS as the arbitrary cell
31                CASE of (instruction) for current $VPE_i$
32                    [SEND data to $PE_is$]: Move data from $IS_h$ to temp $PE_i$ buffer
33                    [VIS READS at 1 PE]: Move data from temp $PE_i$ buffer
                          to the VIS (at $IS_h$)
34                    [REDUCE data into VIS]: Do ASC reduction operation on the $PE_i$
                          temp buffers and the data currently in the VIS
35                MAKE the selected (satisfied) PEs temporarily INACTIVE
36        ALL $PE_i$ are now ACTIVE and LISTENING to $IS_0$

*** Move data written from VISs to VPEs ***
37     FOR $k = 1$ to $\lceil N/n \rceil$ do
38         IF (instruction for $VPE_i[k]$ = Broadcast data to $PE_i$)
39            COPY data from temporary $PE_i$ buffer to $VPE_i[k]$
```

Figure 5: Part 2 of ASC(n,j) algorithm to simulate ASC(N,j) where $N \geq n$ and $J \geq j$

The many sets of steps needed to complete this algorithm are necessary to insure that data is not transfered more than one time in terms of the VPE and VISs. It may be possible to further improve the simulation results if a data movement for any given VIS is handled not only for VPEs inside a PE listening to this VIS but for all VPEs distributed among the PEs in constant time. Thus this implies a preordering among the VPEs in each PE to line up requests for VISs. Further work needs to be done to understand if there is an efficient way to lower the running time.

6 Time and space complexities

The worst case time is derived by first noting in Figure 4 and Figure 5 that lines 2-6, 17-23, and 37-39 need $O(N/n)$ time each, and step 1 requires $O(J/j)$ time to simulate the VISs. Both phases one and two of the simulation algorithm asymptotically take the same amount of time to execute. The VIS lists formed in phase one are of size $O(min(N/n, J))$. The while loops at lines 7 and 24 iterate $O(min(N/n, J))$ times based on the size of the VIS request lists. The while loops at lines 10 and 27 iterate $O(min(n, J/j))$ times since the worst case dictates that all n PEs listen to VISs simulated by one IS, but the maximum number of VISs in any IS cannot be greater than $\lceil J/j \rceil$. All other operations inside these loops take constant time. Furthermore, each PE needs $O(N/n)$ space and each IS needs $O(J/j)$ space to simulate the extra processors, while the temporary PE_i arrays for moving data between VISs and VPEs are of size $O(J)$. Thus the total extra time needed to simulate ASC(N,J) with ASC(n,j) where $N \geq n$ and $J \geq j$ is $O(\frac{N}{n} + \frac{J}{j} + min(\frac{N}{n}, J) \times min(n, \frac{J}{j}))$ while the space required per PE is $O(\frac{N}{n} + J)$ and the space needed per IS is $O(J/j)$. These results are for any possible configuration of ASC self simulation. Table 1 shows the general case time as well as more realistic or specific examples of self simulation using this algorithm. This table also includes an average simulation time if it is assumed that the PE distribution among ISs in steps 9 and 26 is roughly uniform.

Table 1: Self Simulation Times where $N \geq n$ and $J \geq j$ for the General and Average Cases

SIMULATE	With		Extra Time	Extra Memory
ASC(N,J)	ASC(n,j)		$O(\frac{N}{n} + \frac{J}{j} + min(\frac{N}{n}, J) \times min(n, \frac{J}{j}))$	$O(\frac{N}{n} + J)$ per PE
avg. ASC(N,J)	ASC(n,j)		$O(\frac{N}{n} + \frac{J}{j} + min(\frac{N}{n}, J) \times min(\frac{n}{j}, \frac{J}{j}))$	$O(\frac{N}{n} + J)$ per PE
ASC(N,J)	ASC(1,1)		$O(N + J)$	$O(N + J)$ per PE
ASC(N,J)	ASC(n,1)		$O(\frac{N}{n} + J + min(\frac{N}{n}, J) \times min(n, J))$	$O(\frac{N}{n} + J)$ per PE
ASC(N,J)	ASC(1,j)		$O(N + \frac{J}{j})$	$O(N + J)$ per PE
ASC(N,J)	ASC(n,J)		$O(\frac{N}{n})$	$O(\frac{N}{n} + J)$ per PE
ASC(N,J)	ASC(N,j)		$O(\frac{J}{j})$	$O(J)$ per PE

7 Overview of ASC simulations with PRAM

This brief section is included to describe a related work in progress of ASC simulating and being simulated by PRAM. The simulations with PRAM are important because they give an indication of the power of ASC, automatically provide algorithms for ASC, and show how well ASC maps to other models and hardware. For this section it is assumed that n is the number of PEs for ASC or PRAM, j is the number of ISs, and m is the number of PRAM shared memories. It is further assumed that all the PRAM models are synchronous. Table 2 shows the results of the simulations. The reader should refer to [15] for further information on this subject.

8 Conclusions

It has been shown how the ASC model can simulate itself. The methods used in this paper allow the number of PEs and ISs to be unbounded for a flexible evaluation of the self simu-

Table 2: ASC Simulation Times for Several PRAM Models

SIMULATE	With	Extra Time	Extra Memory
ASC(n,j)	RAM	$O(n+j)$	$O(n+j)$
nPRAM(n,m)	ASC(n,j)	$O(min(m/j,n))$	$O(m/j)$-PE
ASC(n,1)	cCRCW(n,m)	$O(1)$	constant
ASC(n,1)	CREW(n,m)	$O(n/m+\log n)$	constant
ASC(n,1)	EREW(n,m)	$O(n/m+\log n)$	constant
ASC(n,j)	cCRCW(n,m)	$O(j)$	$O(j/n-PE)$
ASC(n,j)	CREW(n,m)	$O(\frac{jn}{m}+j*\log n)$	$O(j/n-PE)$
ASC(n,j)	EREW(n,m)	$O(\frac{jn}{m}+j*\log n)$	$O(j/n-PE)$

lation. If the current technology in terms of hardware implementation is taken into account, then certain resources would definitely be restricted for real ASC machines. The ASC model is intended to encompass associative machines that are buildable today with reasonable resources. The current upper bound on the number of implementable PEs is no doubt in the tens of thousands, while the number of instruction steams build-able on an ASC like machine is probably at least $\log n$ where n is the number of PEs[1]. The self simulation algorithm is a reasonable solution to provide more flexibility in terms of programming actual machines in a data parallel style and, more generally, associative programming.

References

[1] J. Potter J. Baker S. Scott A. Bansal C. Leangsuksun C. Asthagiri. Asc: An associative computing paradigm. *IEEE Computer*, pages 19–25, November 1994.

[2] B. Svensson C. Fernstrom, I. Kruzela. *LUCAS Associative Array Processor.* Springer-Verlag, Berlin-Heidelberg, 1986.

[3] Tom Blank and John R. Nickolls. A grimm collection of mimd fairy tales. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 448–457, McLean, VA, October 19-21, 1992. IEEE Computer Society Press.

[4] J.L. Potter. *Associative Computing — A Programming Paradigm for Massively Parallel Computers.* Plenum Publishing, N.Y., 1992.

[5] H. Li M. Maresca. Connection autonomy and simd computers: a vlsi implementation. *Journal of Parallel and Distributed Computing*, 7:302–320, 1989.

[6] Tse-yun Feng Chuan-lin Wu, editor. *Tutorial: Interconnection networks for parallel and distributed processing.* IEEE Computer Society Press, Silver Spring, MD, 1984.

[7] R.A. Heaton J.M. Jennings, E.W. Davis. Comparative performance evaluation of a new simd machine. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 255–258, College Park, MD, October 8-10, 1990. IEEE Computer Society Press.

[8] Yonatan Aumann and Assaf Schuster. Deterministic pram simulation with constant memory blow-up and no time-stamps. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 22–29, College Park, MD, October 8-10, 1990. IEEE Computer Society Press.

[9] J. Zhang R. Lin, S. Olariu. Simulating enhanced meshes with applications. *Parallel Processing Letters*, 3(1):59–70, 1993.

[10] S. Rajasekaran M. Palis. Packet routing and pram emulation on star graphs and leveled networks. *Journal of Parallel and Distributed Computing*, (20):145–157, 1994.

[11] J. Trahan R. Vaidyanathan. Optimal simulation of multidimensional reconfigurable meshes by two-dimensional reconfigurable meshes. *Information Processing Letters*, (47):267–273, October 1993.

[12] R. Staraman A. Rosenberg, V. Scarano. The reconfigurable ring of processors: Efficient algorithms via hypercube simulation. *Parallel Processing Letters*, 5(1):37–48, 1995.

[13] F. Annexstein M. Baumslag M. Herbordt B. Obrenic A. Rosenberg C. Weems. Achieving multigauge behavior in bit-serial simd architectures via emulation. In *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pages 186–195, College Park, MD, October 8-10, 1990. IEEE Computer Society Press.

[14] Yosi Ben-Asher, Dan Gordon, and Assaf Schuster. Efficient self-simulation algorithms for reconfigurable arrays. *Journal of Parallel and Distributed Computing*, (30):1–22, 1995.

[15] D. R. Ulm and Johnnie W. Baker. The power of the associative model by comparison to pram. Technical report, Kent State University, 1996.