ASC PRIMER

Updating of an earlier primer by Julia Lee and Dr. Chandra Asthagiri

based on Dr. Jerry Potter’s work

Current version due to Dr. Oberta Slotterbeck

ASC is a high level parallel language developed originally by Dr. Jerry Potter. ASC stands for Associative Computing. While ASC has been implemented on several computers, all but one implementation on a ClearSpeed machine no longer exists. An emulator that runs on Windows,

however, allows a professor to introduce a different style of parallel computing, have a virtual machine on which to run, and calculate how many scalar and how many parallel operations are utilized on a run. ASC has been used in classes at Kent State University and Hiram College.

A recent article in Computer states:

“Chipmakers … are studying the efficiency of novel advanced parallel architectures for many core processors. However, software researchers still struggle to achieve new technology breakthroughs in the design of software architecture to make the programming of future many-core processors feasible.

Software architects are looking for new approaches to deal with various pitfalls and issues such as data dependency, race conditions, load balancing, nondeterminism, and deadlocks. In addition they are researching advanced parallel programming paradigms that will allow the construction of easy-to-use, productive, energy-efficient, scalable, and portable parallel applications.”[5]

Associative computing is an excellent paradigm that meets the criteria above. As programming an associative computing machine requires different thought processes than those used for other parallel machines, it would be useful educationally to introduce associative computing as one style of parallel programming. Moreover, experiences at Kent State University and Hiram College show that very little time is required to have students start working with the associative computing environment.

The purpose of this primer is to show the basics of the ASC language and show how the emulator works. Most of the material for this primer is taken from Dr. Potter’s publications [6],[7],[8], Hioe’s thesis [2], and Lee’s thesis [3]. There are 6 chapters. The first chapter discusses the background of ASC; the second chapter shows how to get started. Chapters 3, 4, and 5 discuss the parallel operations of ASC; Chapter 6 presents additional features of ASC.

Acknowledgement: This document was originally prepared through the efforts of many people, in particular Julia Lee and Chandra Asthagiri. The drawings are from Lee’s master thesis. Recently Dr. Slotterbeck has eliminated references to running the emulator on The Connection Machine and the WaveTracer as these are no longer existing machines and new programs were added.

Chapter 1

ASC Background

1.1 Introduction

ASC is a parallel computer language which can be used as an algorithmic language or as a programming language that will run on the ASC emulator on a sequential machine. Although timings don’t make sense, ASC can count scalar operations and parallel operations so comparisons between algorithms can be made. The STARAN Computer (Goodyear Aerospace, early 1970’s) and later the ASPRO provided an architectural model for associative computing embodied in the ASC model. Associative computing extends the data parallel paradigm to a complete computational model. It provides a practical model that supports massive parallelism.

The first pass with asc1.exe compiles an ASC program and produces an .iob file which contains intermediate code and a .lst file which shows the relationship between the source code and the triple address intermediate code. The second pass with asc2.exe generates an output file, if it is requested, showing print statements.

1.2 SIMD Machines the Target Architecture

There are several ways to exploit parallelism. One of these is to partition the data and let each block of data be processed by a particular processing element (PE). Each PE has its own local data, and all the PEs can execute at the same time to process the data. Thus, there is no *CPU memory* *bottleneck* that exists with sequential computers. Since there is a one to one relationship between the processors and memories, large amounts of data can be passed between the processors and memory. The same instruction is applied to all the PEs by *broadcasting* the instruction to all the PEs. All *active* PEs will execute the same instruction at the same time on its own data because each processor has its own local memory. This type of computer is called a Single Instruction Multiple Data (SIMD) computer. A SIMD computer consists of a Control Unit, simple processing units, each with its own private memory, and an interconnection network. The emulator supports some interconnection networks, but we won’t show that here.

* SIMDs use less hardware than MIMDs as they have only one control unit and each PE is essentially an ALU.
* SIMDs use less memory than MIMDs because
  + Only one copy of the instructions need to be stored
  + This allows more data to be stored in memory, which reduces movement of data between primary and secondary storage.

The control unit compiles the program and broadcasts the machine language instructions to execute to all PEs simultaneously,, eliminating all control communication between PEs. Only data is communicated between PEs. .

* Some people believe the single instruction stream and synchronization of PEs make SIMD applications easier to program, understand, and debug. (Unfortunately, because the paradigm is unfamiliar to some, there are nay-sayers that believe it is harder to program a SIMD than a MIMD.
* Control flow operations and scalar operations can be executed on the control unit while PEs are executing the PE instructions.
* MIMD architectures require message passing or shared memory communications between the PEs, including explicit synchronization and other control primitives, which create a substantial; amount of additional overhead.
* During a data communication operation between PEs,
  + PEs send data to a neighboring PE in parallel and in lock step
  + The data is moved along an interconnection network following the steps in a communication algorithm.
  + the entire communication operation is executed synchronously
* Less complex hardware is needed in a SIMD since no message decoder is needed in PEs
  + MIMDs need a message decoder in each PE.

1.3 Associative Computing

An associative processor (such as the STARAN-E [10]), consists of a conventional sequential computer called the host, a sequential control, and an array of processors. Each processor in the array has its own memory. So, the machine can be visualized as a 2-dimensonal array with each “row” controlled by 1 PE and each “column” naming a parallel variable. Thus, each column’s values represents collectively a single parallel value.

The 3 basic components of an associative SIMD computer are:

1. Memory

(a) Sequential Associative Processor Control Memory

(b) Parallel Associative Array Memory

2. A set of Processing Elements (PE)

3. Communication Flip Network

The Control Memory stores the assembled instructions of the program and the array memory stores the data. The array memory, the local memory for each of the processors, provides content addressable and parallel processing capabilities. Each PE is associated with a row of memory. The PEs have three 1-bit registers M, X, and Y. The M register is used to mask and to select which processors will carry out the broadcast instruction. The X register is used for temporary storage and the Y register stores the result of a search operation.

1.4 The ASC Emulator

The ASC emulator is written in C. Currently it emulates 300 processors and a parallel array memory 8000 bits wide. However, a few of the examples in this primer have not been checked out and may differ when run on the emulator. Questions and suggestions should be directed to [potter@cs.kent.edu](mailto:potter@cs.kent.edu), [jbaker@cs.kent.edu](mailto:jbaker@cs.kent.edu) or [obie@hiram.edu](mailto:obie@hiram.edu).

1.5 The Associative Computing Model

ASC is based on the Associative Computing Model of Potter [8]. Data items which are *related* or *associated* are stored as one record and one such record is stored in the memory allocated to one processor in the array. All similar records, one record per PE, may be accessed in parallel. In associative computing, data records are referenced by describing their contents and not by their address. Since each memory cell has its own processor, there is no need to move data from memory to a CPU. Thus, associative computing promotes parallelism by maximizing the amount of data parallel processing.

The Associative Computing Model consists of a basic cycle of three phases:

1. search 2. process 3. retrieve

The *search* operation broadcasts the description of the desired object(s) of an association to all the active processors. This allows all processors to search for the desired object(s) in parallel. The processors which locate the desired object in the search phase are called the *active responders.*

The *process* phase consists of a sequence of operations which are executed by the active responders.

In the retrieve phase, values of specific items of the active responders are retrieved and used in subsequent cycles. The basic search, process retrieve cyclecan be nested and in any one cycle the process and/or retrieve phase may be omitted [8]. When exiting a nested level, the group of active responders is restored to the previous state.

ASC associative computing offers many advantages over message passing or common memory programming. Although not discussed further here, these claims have been shown elsewhere in various papers. (See Additional References at the end of this document.)

ASC programs are often simple and similar to sequential programs. Some advantages of an associative computing approach being taught:

* Consistent use of data parallel programming
* Consistent use of global associative searching & responder processing
* Usually, frequent use of the constant time global reduction operations: **AND, OR, MAX, MIN**
* Broadcast of data using IS bus allows the use of the PE network to be restricted to parallel data movement.
* Tabular representation of data
* Use of searching instead of sorting
* Use of searching instead of pointers
* Use of searching instead of the ordering provided by linked lists, stacks, queues
* Promotes an highly intuitive programming style that promotes high productivity
* Uses structure codes (i.e., numeric representation) to represent data structures such as trees, graphs, embedded lists, and matrices.
  + See Nov. 1994 IEEE Computer article.
  + Also, see “Associative Computing” book by Potter.

The usual problems with parallel computing, namely deadlock, race conditions, load balancing, time spent in message passing and data movement, data dependency, and nondeterminism can’t occur.

On the other hand, students with the Associative Computing Lab at Kent State under Dr. Jerry Potter and Dr. Johnnie Baker and students at Hiram College in Dr. Slotterbeck’s classes have programmed many ASC examples such as:

* **A wide range of algorithms implemented in ASC without use of PE network**
  + Graph Algorithms
    - minimum l spanning tree
    - shortest path
    - connected components
  + Computational Geometry Algorithms
    - convex hull algorithms (Jarvis March, Quickhull, Graham Scan, etc)
    - Dynamic hull algorithms
  + String Matching Algorithms
    - all exact substring matches
    - all exact matches with “don’t care” (i.e., wild card) characters.
  + Algorithms for NP-complete problems
    - traveling salesperson
    - 2-D knapsack.
  + Data Base Management Software
    - associative data base
    - relational data base
  + A Two Pass Compiler for ASC
    - first pass
    - optimization phase
  + Two Rule-Based Inference Engines
    - OPS-5 interpreter
    - PPL (Parallel Production Language interpreter)
  + A Context Sensitive Language Interpreter
    - (OPS-5 variables force context sensitivity)
  + An associative PROLOG interpreter
* **Numerous Programs in ASC using a PE network**
  + 2-D Knapsack Algorithm using a 1-D mesh
  + Image Processing algorithms using 1-D mesh
  + FFT using Flip Network
  + Matrix Multiplication using 1-D mesh
  + Smith-Waterman Sequence Alignment using a linear network
  + An Air Traffic Control Program (using Flip network connecting PEs to memory)
    - Demonstrated using live data at Knoxville in mid 70’s.

Chapter 2

Getting Started

2.1 How to execute ASC programs on the emulator.

In a Windows environment, open a command prompt window and move to the directory where you stored asc1.exe, asc2.exe, and your program with an .asc extension. First compile with

>> asc1.exe –e Myprog.asc

The –e says output is to be 3 address code for the emulator. Other switches could be used to generate assembly code for the Connection Machine or the Wavetracer. The asc1 command outputs a .iob and a .lst file. The .iob file is the intermediate 3 address code produced for the emulator.

To execute Myprog.asc, use

>> asc2.exe –e Myprog.iob <Myprog.dat >Myprog.out

where the .dat and .out files are optional. Omitting the .dat file means data must be entered interactively. Omitting the .out file means the output will be to the screen. ALL .dat FILES (AND INTERACTIVE INPUT) MUST END IN A BLANK LINE.

2.2 Program Structure

Only in a few cases (clearly marked) is case significant. However, caps will often be used to highlight actual code.

An ASC program can be identified as a main program or a subroutine (see Chapter 6 for subroutines). The order of the program structure is important. All command lines end with a semi-colon.

MAIN program-name

defining constants;

equivalencing variables;

declaring associations;

body of program;

END;

2.3 Defining Constants

The DEFINE and DEFLOG statements are used to define constants.

define(identifier, value);

deflog(identifier, value);

The DEFINE keyword is used to declare scalar constants, whereas the DEFLDEFLOG OG keyword is used to declare the logical constants. Scalar constants can be decimal, hexadecimal (X), octal(O) or binary(B).

Example: define(Maxnum, 200);

define(MyHex, X’3F’);

define(MyOct, O’765’);

define(MyBin, B’11001’);

deflog(TRUE, 1); /\* There is no logical scalar\*/

deflog(FALSE,0);

2.4 Reserved Words

Reserved words cannot be redefined, so they must not be used as variable names. The reserved words are listed in an appendix.

2.5 Declaring Variables

To declare a variable in ASC, the data mode and the data type of the variables must be indicated. ASC supports two modes of data items, namely scalar and parallel. ASC supports eight data types.

Scalar data items are stored in sequential memory of the computer and the parallel items are stored in the parallel array memory. Scalar data items are stored in fixed length words depending on the word size of the host) whereas parallel data items have varying lengths depending on the type of data. The keywords SCALAR and PARALLEL define the mode of the variable. The character $ indicates parallel mode or parallel operation.

ASC supports nine data types, namely integer (INT), hexadecimal (HEX), octal (OCT), binary (BIN), unsigned (CARD), character (CHAR), single precision (REAL), logical (LOGICAL), and index. (INDEX). Integers may occupy from 2 to 2256 bits. When declaring integer, users must determine the range of the variable since overflow of integer arithmetic is not detected by ASC. A logical variable is 1 bit long and is used to store values of 1 or 0, TRUE or FALSE. An index variable also occupies 1 bit of memory in parallel mode only. Index variables are important in ASC because they link the search, process, and retrieve phases [8]. These variables are used for parallel to scalar reduction (or indexing) purposes when using any control structure dealing with parallel variables. When used with a parallel field, the index variable refers to only one specific association at a time.

Variable names start with a letter and are at most 32 characters long. Lower case and upper case are normally indistinguishable although the performance monitor requires upper case. The width of a variable can be specified by using a colon after the variable name followed by a number indicating the field width. The general form is

FORMAT: data-type data-mode var1, var2;

For example:

int scalar aa,b;

int parallel tail[$], head[$], weight[$];

int found:8[$]; /\*8 bit field \*/

char parallel node[$];

hex parallel code:4[$]; /\*4 bit field\*/

logical parallel found[$];

index parallel xx[$], yy[$];

Commas and spaces separate identifiers, but either can be used.

2.6 Multidimensional Variables

ASC allows multidimensional variables up to three dimensions.

int parallel aa[$, 5]; /\* two dimensional parallel \*/

int parallel b[$, 5 , 5] /\* three dimensional parallel \*/

int scalar c[3,5] /\* two dimensional scalar array \*/

The two dimensional parallel array aa, above, consists of 5 parallel arrays: aa[$,0], aa[$,1], aa[$,2], aa[$,3], and aa[$,4]. Array indices start at zero and go to n-1, where n = 5.

Parallel array aa: (number of rows is set by the data input.)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| aa[$,0] | aa[$,0] | aa[$,0] | aa[$,0] | aa[$,0] |
| 1 | 4 | 3 | 2 | 1 |
| 2 | 5 | 4 | 2 | 6 |
| 3 | 1 | 2 | 2 | 3 |
| 4 | 3 | 4 | 5 | 1 |

2.7 The DEFVAR Statement

The DEFVAR statement allows the user to define the location of a variable in memory by defining the beginning of the helpful. This is helpful when the user needs to overlap two or more variables. The DEFVAR statement must come before the variable declaration and the second variable must be declared before the first variable.

FORMAT: defvar(var1, var2);

For example: defvar(aa,b); /\* define aa in terms of b \*/

2.8 The scalar IF Statement

ASC has two IF statements, a scalar one and a parallel one. The scalar IF is similar to the IF found in conventional languages. When the IF statement evaluates to TRUE the body is executed, otherwise the body of the THEN is executed. As usual, the THEN is optional.

For example: if aa .eq. 5 then

sum = 0;

else

b = sum;

endif;

The parallel IF is used for parallel searching and is discussed in section 4.2.

2.9 Establishing Associations Between Variables

Associations can be established for parallel variables by using a logical parallel variable, Variables that are associated are related and can be referenced as a group using the logical parallel variable. The parallel variable must be declared prior to establishing the association.

FORMAT: associate var1, var2 with logical-parallel-variable;

Helpful examples will appear in some of the code in the appendices.

2.10 Operators in ASC

2.10.1 relational operators are usual 2.10.2 logical operators are usual

.lt. or < .not. or !

.gt. or > .or. or ||

.le. or <= .and. or &&

.ge. or >= .xor. or ^^

.eq. or ==

.ne. or !=

2.10.3 arithmetic operators

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operator | Opearand type | Result type |
| mult | \* | real | real |
| mult | \* | int | int |
| mult | \* | real, int | real |
| div | / | real, int | real |
| div | / | int | int |
| div | / | real, int | real |
| add | + | real, int | real |
| add | + | int | int |
| add | + | real, int | real |
| sub | - | real | real |
| sub | - | int | int |
| sub | - | real, int | real |

2.11 Parallel Arithmetic Operations

Arithmetic operations can be performed in parallel, which means that the operation applies to all the active responders in one operation. In a sequential computer, this would require a loop.

For example: aa[$] = b[$] + c[$];

Before: After: U=unchanged

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mask | aa[$] | b[$] | c[$] |  | mask | aa[$] | b[$] | c[$] |
| 1 | 2 | 4 | 5 |  | U | 9 | U | U |
| 1 | 6 | 8 | 8 |  | U | 16 | U | U |
| 1 | 3 | 10 | 1 |  | U | 11 | U | U |
| 0 | 5 | 4 | 3 |  | U | 5 | U | U |
| 0 | 6 | 7 | 2 |  | U | 6 | U | U |
| 1 | 5 | 6 | 7 |  | U | 13 | U | U |

A mask of 1 means an active responder (set by another operation as discussed later.)

2.12 The Assignment Statement

The assignment operator is an equal sign. There are three types of assignment in ASC, namely assigning to a scalar variable, assigning to a parallel variable, and assigning to a logical parallel variable. Any arithmetic expression that evaluates into a scalar variable can be assigned to a scalar variable; any parallel arithmetic expression can be assigned to a parallel variable; and any logical parallel expression can be assigned to a logical parallel expression. The data types must be the same on the left and the right hand sides of the equal oprerator.

FORMAT:

scalar-variable = scalar-expression

parallel-variable = scalar-expression or parallel-expression

logical-parallel-variable = logical-parallel-expression

For example: int scalar k;

int parallel aa[$], b[$], c[$];

logical parallel used[$];

index parallel xx[$];

k = aa[xx] + 5;

b[$] = aa[$] + 5;

c[$] = 3 + 5;

used[$] = aa[$] .eq. 5;

Before: After (U = unchanged)

k is set to aa[xx] + 5 = 9 +5 = 14

xx refers to PE with arrow marked.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mask[$] | aa[$] | b[$] | c[$] | used[$] | xx[$] |  | mask[$] | aa[$] | b[$] | c[$] | used[$] | xx[$] |
| 1 | 7 | 1 | 3 | 1 | 0 |  | U | U | 12 | 8 | 0 | U |
| 1 | 5 | 2 | 1 | 1 | 0 |  | U | U | 10 | 8 | 1 | U |
| 1 | 3 | 1 | 4 | 1 | 0 |  | U | U | 8 | 8 | 0 | U |
| 1 | 9 | 6 | 7 | 0 | 1 <- |  | U | U | 14 | 8 | 0 | U |
| 1 | 5 | 8 | 2 | 0 | 0 |  | U | U | 10 | 8 | 1 | U |
| 1 | 5 | 9 | 4 | 0 | 0 |  | U | U | 10 | 8 | 1 | U |
| 0 | 0 | 4 | 5 | 0 | 0 |  | U | U | U | U | 0 | U |

2.13 Comments, Delimiters, and Program Lines

Comments use C’s syntax, namely /\* and \*/ surrounding comments. Delimiters that separate language elements are blanks, new lines, and comments. Furthermore, ASC program lines must not exceed 132 characters or the result will be unpredictable. Comments may be nested, but as usual.

2.14 Embedded Assembler Code

Dropper from this primer as no code for an existing machine is generated.

Chapter 3

Parallel Input and Output

3.1 The Parallel READ Statement

The READ statement deals with the contents of the parallel array memory. Thus it works only with parallel variables. The association of variables to a logical parallel variable must be established before the READ statement is executed because the parallel variables are read as a group.

FORMAT: read parvar1, parvar2,…, in logical-parallel-variable;

For example: main try1

char parallel tail[$], head[$];

int parallel weight[$];

logical parallel weight[$];

associate tail[$], head[$], weight[$] with graph[$];

read tail[$], head[$], weight[$] in graph[$]’

end;

3.2 The Input File

The input file for the parallel array is organized into columns. The position of the columns corresponds with the position of the parallel variable in the READ statement.

For example: Consider an input file below for try1:

a b 40

a c 30

b a 38

b c 24

c a 26

c b 20

<blank line>

It will produce:

|  |  |  |  |
| --- | --- | --- | --- |
| tail[$] | head[$] | weight[$] | graph[$] |
| a | b | 40 | 1 |
| a | c | 30 | 1 |
| b | a | 38 | 1 |
| b | c | 24 | 1 |
| c | a | 26 | 1 |
| c | b | 20 | 1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 |

Note that when data is read, the row entry in graph[$] is set to 1 if input was found for that row.

3.3 The Parallel PRINT Statement

The PRINT statement deals with the contents of the array memory and not with scalar variables. Printing strings of text and scalar variables is done with the MSG statement.

FORMAT: print parvar1, parvar2 in logical-parallel-variable.

For example:

main try2

char parallel tail[$], head[$];

int parallel weight[$];

logical parallel graph[$];

associate tail[$], head[$], weight[$] with graph[$];

read tail[$], head[$], weight[$] in graph[$];

print tail[$], head[$], weight[$] in graph[$];

end;

The output file begins with the message dump of association followed by the contents of the parallel variables printed in columns.

For example:

DUMP OF ASSOCIATION RESULT FOLLOWS:

TAIL, HEAD, WEIGHT

a b 40

a c 30

b a 38

b c 24

c a 26

c b 20

3.4 The MSG statement

This statement is used to display scalar variables and messages. Variables are displayed on the line following the message. The MSG statement may be used to dump parallel variables for debugging. When a parallel variable is specified, the contents of the field for the entire array is printed regardless of the status of the active responders or association variables.

FORMAT: msg “ string ” list;

Chapter 4

Parallel Searching

Many of the typical operations in other language requiring looping and pointers, can be accomplished in ASC by searching. Several statements provide strong support for parallel searches, namely the SETSCOPE statement, the different IF statement, and the ANY statement.

4.1 The SETSCOPE Statement

This provides the simplest way to mark the set of active PEs.

FORMAT: setscope logical-parallel-variable;

body

endsetscope;

For example: used[$] = aa[$] .eq. 5;

setscope used[$];

tail[$] = 100;

endscope;

Before: (N=irrelevant) After: (U = unchanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| aa[$] | used[$] | tail[$] |  | aa[$] | used[$] | tail[$] |
| 5 | N | 7 |  | U | 1 | 100 |
| 23 | N | 6 |  | U | 0 | U |
| 5 | N | 9 |  | U | 1 | 100 |
| 41 | N | 7 |  | U | 0 | U |

All PEs whose USED bit is 1 will set its tail field to 100.

4.2 The Parallel IF-THEN-ELSE Statement

The parallel IF-THEN-ELSE statement is different from the conventional IF statement, because it is actually a masking statement and not a branching statement. This IF statement refers to the active responders of the search process and both parts of the IF statement are executed. The IF-THEN-ELSE statement executes as follows:

1. Save the mask bit of processors that are currently active.

2. Broadcast code to the processors to calculate the IF operation.

3. Set the individual cell mask bit of the active responders to TRUE if its local condition is TRUE. Set the mask bit of the active processors to FALSE.

4. Broadcast code for the TRUE portion of the IF statement.

5. Compliment the mask bits that are obtained in step 3.

6. Broadcast code for the FALSE portion of the IF statement.

FORMAT: IF (logical-parallel-expression) THEN

body of then

<ELSE

body of else>

ENDIF;

For example:

IF (t[$] .eq. 1) THEN /\* search for t ==1\*/

t[$] = 0; /\* process \*/

ELSE /\* search for t != 1 \*/

t[$] = -1; /\* process \*/

Before: After:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| t[$] | original mask |  | t[$] | THEN mask | ELSE mask |
| 1 | 1 |  | 0 | 1 | 0 |
| 7 | 1 |  | -1 | 0 | 1 |
| 2 | 1 |  | -1 | 0 | 1 |
| 1 | 1 |  | 0 | 1 | 0 |
| 3 | 0 |  | 3 |  |  |

4.3 The IF-NOT-ANY Statement

This is different from the IF-THEN-ELSE statement in that only one body part is executed. The IF-NOT-ANY statement evaluates the conditional expression and if there are one or more active responders, the THEN statement block is executed. On the other hand, if there is not even one active responder. the ELSE-NOT-ANY statement block is executed. The IF-NOT-ANY statement is ELSENANY part, the mask used is the original mask existing prior to the IF-NOT-ANY statement.

FORMAT: IF (logical-parallel-expression) THEN

body of IF

ELSENANY

body of not any

ENDIF;

For example:

if (aa[$] >= 2 && aa[$] < 4) then /\* set mask \*/

if b[$] == 12 then /\*search for b equal 12 \*/

c[$] = 1; /\*process \*/

elsenany /\*search for b != 12 \*/

c[$] = 9; /\*process \*/

endif;

before: after: (U=unchanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| aa[$] | b[$] | c[$] |  | aa[$] | b[$] | c[$] |
| 1 | 17 | 0 |  | U | U | U |
| 2 | 13 | 1 |  | U | U | 9 |
| 2 | 8 | 2 |  | U | U | 9 |
| 3 | 11 | 3 |  | U | U | 9 |
| 2 | 9 | 4 |  | U | U | 9 |
| 4 | 67 | 5 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | U |

4.4 The ANY Statement

The ANY statement is used to search for all data items that satisfy the conditional expression. There must be one responder for the body statement to be performed. If there are no responders, the ANY statement does nothing unless an ELSENANY is used. The mask that is used to execute the body part is the original mask prior to the ANY statement. Thus, all active responders are affected if the conditional expression of the ANY evaluation is TRUE.

FORMAT: any logical-parallel-expression

body

< elsenany

body

endany;

For example: if (aa[$] > 7) then /\* set mask \*/

any aa[$] == 10

b[$] = 11;

endany;

endif;

before: after: (U=unchanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| mask[$] | aa[$] | b[$] |  | mask[$] | aa[$] | b[$] |
| 0 | 3 | 0 |  | U | U | U |
| 0 | 5 | 0 |  | U | U | U |
| 1 | 16 | 0 |  | U | U | 11 |
| 1 | 10 | 0 |  | U | U | 11 |
| 1 | 8 | 0 |  | U | U | 11 |
| 0 | 7 | 0 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | U |

There are 3 responders to the IF statement in row 3, row 4, and row 5. Therefore, the b columns is changed from 0 to 11 at those rows. The ANY statement can be used with the ELSENANY clause as in section 4.3. If there are no responders, then the ELSENANY body part is executed.

FORMAT: (any logical-parallel-expression)

body

elsenany

body

endany.

For example:

if (aa[$] > 7) then /\* set mask \*/

any aa[$] ==10

b[$] = 11;

elsenany

b[$] = 100;

endany;

endif;

before: after: (U=unchanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| mask[$] | aa[$] | b[$] |  | mask[$] | aa[$] | b[$] |
| 0 | 3 | 0 |  | U | U | U |
| 0 | 5 | 0 |  | U | U | U |
| 1 | 16 | 0 |  | U | U | 100 |
| 1 | 10 | 0 |  | U | U | 100 |
| 1 | 8 | 0 |  | U | U | 100 |
| 0 | 7 | 0 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | U |
| 0 | 0 | 0 |  | U | U | u |

Chapter 5

Looping and Retrieving

5.1 The LOOP-UNTIL Statement

ASC supports a variety of control statements. The LOOP statement, used for looping, resembles the REPEAT\_UNTIL in some other languages, but is more flexible since the UNTIL conditional test can appear anywhere in the loop body.

FORMAT: first

initialization

loop

body1

until

logical-scalar-expression

logical-parallel-expresion

nany logical-parallel-expression

body2

endloop;

For example: first

i = 0;

loop

if (aa[$] == I ) then

b[$] = aa[$] + 2;

endif;

i = i + 1;

until I > 10

endloop;

In this example, the variable i is initialized to zero and incremented each time it enters the loop. At each iteration of the loop variable i is tested and if the test evaluates to true, then the loop terminates.

The UNTIL expression may be scalar, as above, or parallel. If parallel, UNTIL exits based on a particular responder test. It may contain an optional NANY keyword. For example, with NANY, the UNTIL may be used to loop until all the responders have been processed as in

UNTIL nany c[$];

where c[$] represents a logical parallel expression. On the other hand,

UNTIL c[$];

stops the loop whenever a responder is detected. . Without NANY a parallel UNTIL will exit when responders are first detected. Both parallel and scalar UNTILs may be used anyplace in the same loop.

5.2 The Parallel FOR-LOOP Statement

The FOR statement is used for looping and retrieving. It evaluates the conditional expression and stores the resulting active responders in an index variable. This index variable can then be used to retrieve a data item of an association. Active responders are processed one after another until all the active responders have been processed. Note that the logical expression must be in the parallel mode.

For example: sum = 0;

for xx in tail[$] .ne. 999

sum = sum + val[xx];

endfor xx; /\* On exit with data below sum = 70 \*/

|  |  |  |
| --- | --- | --- |
| tail[$] | xx[$] | val[$] |
| 3 | 1 | 10 |
| 5 | 1 | 20 |
| 999 | 0 | 30 |
| 6 | 1 | 40 |

Here the index variable xx controls the loop. The loop is repeated only where xx contains the digit 1 starting from the top to the bottom. At each iteration xx is used as a subscript to retrieve the parallel variable val. The contents of xx is updated at the bottom of the loop. The first one is changed to 0. So, the values of each of the parallel variable val is accumulated in the scalar variable sum for all the active responders.

5.3 The Parallel WHILE Statement

The WHILE statement is similar to the FOR except that the WHILE reevaluates the conditional rxpression befor each iteration.

FORMAT: while index-variable in logical-expression

body

endwhile index-variable

For example: sum = 0;

while xx in aa[$] .eq. 2

sum = sum + b[$];

if c[$] .eq. 1 then

if aa[$] .eq. 2 then

aa[$] = 5;

endif;

else

aa[xx] = 7;

endif;

endwhile xx;

Before: After: (U = unhanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| aa[$] | b[$] | c[$] |  | aa[$] | b[$] | c[$] |
| 1 | 17 | 0 |  | 1 | U | U |
| 2 | 13 | 0 |  | 7 | U | U |
| 2 | 8 | 1 |  | 5 | U | U |
| 3 | 11 | 1 |  | 3 | U | U |
| 2 | 9 | 0 |  | 5 | U | U |
| 4 | 67 | 0 |  | 4 | U | U |

As shown, the parallel variable aa is tested at the beginning of each iteration. Inside the loop the parallel variable aa is changed. The iteration is terminated when the conditional expression evaluates to false or no responders. As with any loop such as this one, the programmer must be sure the logical expression will eventually evaluate to false or no responders or the while loop will not terminate. It is advisable for undebugged while statements to include a counter to limit the number of iterations.

For example: i = 5;

sum = 0;

while xx in aa[$] ==2 && i > 0

i = i -1;

sum = sum + b[xx];

if c[xx] ==1 then /\* I forgot to update aa[xx] \*/

else

aa[xx] =7;

endif;

endwhile xx;

if i <= 0 then

MSG “BUG IN WHILE STATEMENT”;

endif;

5.4 The NEXT Statement

The NEXT statement is used to retrieve the “topmost” (lowest row number which represents the PE number) memory word of the active responders masked by a logical parallel variable. After retrieving, the topmost bit of the logical parallel variable is set to zero. NEXT is almost always used within a looping structure. NEXT may contain an ELSENANY statement.

FORMAT: next parallel-index-variable in logical-parallel-variable

body

<elsenany

body>

endnext parallel-index-variable;

For example:

int parallel aa[$], b[$];

logical parallel used[$];

index parallel xx[$];

used[$] = aa[$] .eq. 4;

next xx in used[$]

b[xx] = -1;

endnext xx;

Before: After: (U = unhanged)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| aa[$] | used[$] | b[$] |  | aa[$] | used[$] | b[$] |
| 1 | 0 | 2 |  | U | U | U |
| 4 | 1 | 2 |  | U | 0 | -1 |
| 4 | 1 | 2 |  | U | U | U |
| 19 | 0 | 2 |  | U | U | U |
| 4 | 1 | 2 |  | U | U | U |
| 4 | 1 | 2 |  | U | U | U |
| 17 | 0 | 2 |  | U | U | U |
| 4 | 1 | 2 |  | U | U | U |

In this example, used is the logical parallel variable and the topmost bit that contains a 1 is in the 2nd

row. Thus, at the second row the contents of variable b is changed to -1 and the digit of the variable used is set to 0. The NEXT statement will reference the 3rd row.

Note that

next xx in aa[$] ==4

is not valid.

5.3 The GET Statement

To access the value of a specific item in the memory of an active processor, the GET statement is used. It evaluates the logical parallel expression and uses the index parallel variable to mark the first of the active responders which meet the specified expression. In the body of the GET statement the index parallel variable is used to access the specific item. If there are no responders, the body of the GET statement is not executed. GET may also contain an ELSENANY statement.

FORMAT: get index-parallel-variable in logical-parallel-expression

body

<elsenany

body>

endget index-parallel-variable

For example: get xx in tail[$] .eq. 1

val[xx] =0;

Before: After: (U = unhanged)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| tail[$] | val[$] |  | tail[$] | val[$] |
| 10 | 100 |  | U | U |
| 2 | 90 |  | U | U |
| 1 | 77 |  | U | 0 |
| 1 | 83 |  | U | U |

i

The first responder is the third entry, so the 3rd entry for val[$] is changed. The index parallel variable is not updated y GET, so if GET is in a loop, it will select the same responder on every iteration unless the values of the logical parallel expression is explicitly changed.

Chapter 6

Programming At Large

6.1 Modular Programming: The CALL statement

The subroutine protocal in ASC differs from other languages. In most languages the variables in the calling routine are mapped onto variables in the called routine by their position. In ASC, the actual mapping between fields are specified explicitly. The notation is similar to the UNIX/C file redirection. The ASC subroutine is quite simple. In truth, for teaching purposes, anything more complicated is not needed.

For example: subroutine x

/\* Declare variables aa, b, and c \*/

call y m[$] < aa[$] n[$] < b[$] d[$] > c[$] /\* input parameters marked with < \*/

end; /\* output parameters marked with > \*/

Any number of variables/fields may be mapped and they may be specified in any order.

FORMAT: call subroutine called-field1 < calling-field2 …calling-fieldn > called-fieldn

6.2 Using Subroutines

Subroutines are identified by the keyword SUBROUTINE in the program heading and the structure for the subroutine is the same as the main program. The emulator requires the user to put the subroutines and the main program in separate files. All the subroutines must be named with the extension .asc and must be compiled before the main program and reside in the same directory.

Variables are shared between the main program and the subroutine using the INCLUDE capability (see section 6.3). In this case the main program and the subroutines allocate the same memory fields to the shared variables like COMMON variables in FORTRAN. In ASC, all variables must be declared in th main program. There are no variables local to a subroutine.

FORMAT: subroutine subroutine-name

body

<return>;

end;

For example: program main

defvar(aa,b);

int parallel b[$];

int parallel aa[$];

body

end;

subroutine sub1 /\* in a separate file \*/

defvar(aa,b);

int parallel b[$];

int parallel aa[$];

body

end;

Subroutines conclude with the END statement, but may have an optional RETURN statement. The END statement automatically generates a RETURN.

6.3 The INCLUDE File

The INCLUDE file statement is especially useful when using subroutines. All the DEVARs and the declaration of common variables can be put into a file. Then this file can be included in the ASC program and subroutines by using the INCLUDE statement

FORMAT: #include filename.h /\* filename.h holds defvars and variable declarations \*/

6.4 The MAXVAL and MINVAL Functions

MAXVAL (MINVAL) returns the maximum (minimum) value of the specified item among active responders.

For example: if (tail[$] .ne. 1) then /\* set mask \*/

k = maxval(value[$]);

endif;

|  |  |  |
| --- | --- | --- |
| tail[$] | value[$] | mask[$] |
| 7 | 18 | 1 |
| 1 | 40 | 0 |
| 6 | 4 | 1 |
| 1 | 30 | 0 |
| 1 | 2 | 0 |
| 5 | 10 | 1 |

The value of k is 18, the max of 18, 4, and 10. If maxval is replaced with minval, the value of k is 4, the min of 18, 4, and 10.

6.5 The MAXDEX and MINDEX Function

The MAXDEX (MINDEX) function returns the index of an entry where the maximum (minimum) value of the specified item occurs, among the active responders. This index is also used to retrieve the related fields.

For example: if (tail[$] .ne. 1) then /\* set mask \*/

head[mindex(val[$])] = -1; /\* get index and use it \*/

endif;

Before: After; (U = unchanged)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| mask[$] |  | tail[$] | head[$] | val[$] |  | tail[$] | head[$] | val[$] |
| 1 |  | 8 | 4 | 100 |  | U | U | U |
| 1 |  | 5 | 3 | 80 |  | U | U | U |
| 1 |  | 7 | 5 | 20 |  | U | -1 | U |
| 1 |  | 6 | 9 | 70 |  | U | U | U |
| 0 |  | 1 | 5 | 10 |  | U | U | U |
| 0 |  | 1 | 4 | 1 |  | U | U | U |

In this example, a minimum value for the VAL parallel values with active responders is the third row. So the contents of the HEAD variable in the third row is changed to -1.

6.6 The COUNT Function

This function returns the number of active responders.

For example: if (tail[$] == 1) then /\*set mask \*/

k = count(them[$]); /\* get the number –note use of word them\*/

end;

|  |  |
| --- | --- |
| tail[$] | mask[$] |
| 5 | 0 |
| 1 | 1 |
| 3 | 0 |
| 1 | 1 |
| 1 | 1 |

k is set to 3.

6.7 The NTHVAL and the NTHDEX Functions xxxxxxx

The NTHVAL function returns the Nth value of an item and NTHDEX returns the index variable of that entry. The smallest is the 1st .

For example: num = nthval(a[$],3); /\* third smallest requested \*/

entry = b[nthdex(a[$],3)]; /\* associated entry \*/

|  |
| --- |
| a[$] |
| 5 |
| 1 |
| 4 |
| 1 |
| 1 |

num is set to 1; entry is set to 2.

6.8 Inter-Process Communication

The current version of ASC supports a one dimensional CPU, and a two dimensional memory configuration. So a two-dimensional array declaration

int parallel arr[$, 512]

would be mapped onto memory in rows and columns. The inter-column communication is achieved by specifying a variable or constant expression as the second column index.

For example: Adding adjacent columns, element by element in parallel

arr[$,i] + arr[$,i+1]

Inter-row communication is achieved by specifying a variable or constant expression in the first dimension index. A + is a shift down; - is a shift up. For more inter-process communication see Chapter 3 of Associative Computing [8].

6.9 ASC Pronouns and Articles

To be closer to natural language ASC supports the use of associative pronouns and articles. The associative pronouns are THEM, THEIR, and ITS and the associative articles are A and THE.

THEM refers to the most recent set of responders to a logical parallel expression from a search; THERE is a possessive form of the $ notation; and ITS is an automatically declared parallel index variable. ITS is updated to the most recent index assignment in a FOR, WHILE, NEXT, or GET statement. The generic index variable EACH can be used in non-nested statements.

For example: if aa[$] .gt. 100 then

x = count(them[$]);

endif;

if node[$] .gt. 100 then

if their leftchild[$] .lt. 50 then

. . .

endif;

endif;

for xx in node[$] .gt. 100

if their leftchild[$] .lt. 50 then

. . .

endif;

endif;

for each node[$] .gt. 100

if its leftchild .lt. 50 then

. . .

endif;

endif;

THE refers to the last value of a parallel variable which was reduced to a scalar. Suppose aa[xx] is reduced most recently then THE aa may be used as an alternate form of aa[xx]. The indefinite article A refers to the first entry of an association and its used is similar to a get. (Thus, A is a reserved word and that is why variables are named aa[$], not a[$].)

For example: get xx in node[$] is the same as

k = node[xx]; k = a node; or

endget xx; k = node[a];

6.10 Dynamic Storage Allocation

ASC provides dynamic storage allocation by the ALLOCATE and the RELEASE statement. When a new association entry is needed, memory is allocated by the ALLOCATE statement. The RELEASE statement returns the cell to idle status. In general, several association entries may be released simultaneously.

FORMAT: allocate index-variable in association-name;

statement-block

endallocate index-variable;

release conditional-expression from association-name;

For example: allocate xx in t[$]

aa[xx] = 100;

endallocate xx;

release aa[$] .eq. 100 from t[$];

Note that the scope of the index variable is limited to the statement-block. READs automatically allocate memory, one cell for each row input.

6.11 The ASC Performance Monitor

ASC’s performance monitor calculates the number of scalar and the number of parallel operations performed during a run. Any of the combinations parallel-parallel, scalar-parallel, or parallel-scalar are counted as parallel operations. The monitor can be turned on and off anywhere in the program with the following statements:

PERFORM = 1; /\* monitor on \*/

PERFORM = 0; /\* monitor off \*/

These commands require upper case letters. The values of the scalar and parallel counts can be printed anywhere using the MSG statement as follows:

MSG “scalar count “ SC\_PERFORM;

MSG “parallel count “ PA\_PERFORM;

The statement enclosed with quotes is a message display. The SC\_PERFORM is the counter for scalar operations and the PA\_PERFORM counts parallel operations. These values will be printed on the line following the message. The monitor is turned off automatically during I/O operations. If the performance monitor is on when the program terminates, the counts will be printed automatically.

For example:

main try1

int scalar k;

int parallel aa[$], b[$], used[$];

index parallel xx[$];

logical parallel lg[$];

associate aa[$], b[$] with lg[$];

read aa[$], b[$] in lg[$];

k = 0;

PERFORM = 1; /\* monitor on \*/

while xx in aa[$] .gt. 0 do

if aa[$] .eq. b[$] then

aa[$] = b[$] -5;

endif;

k = k + 1;

endwhile xx;

PERFORM = 0; /\* monitor off\*/

MSG “scalar count :” SC\_PERFORM;

MSG “parallel count :” PA\_PERFORM;

print aa[$], b[$] in lg[$];

end;

6.12 ASC Recursion: The STACKWHILE-RECURSE Construct

ASC does not support recursion. However, the STACKWHILE-RECURSE construct is useful when layers of nesting of logical parallel expressions are needed. A recursive while construct allows nesting to a level as deep as the data requires and the internal stack will allow. Moreover, the compactness of the recursive form greatly reduces the amount of repetitious programming effort. For more explanation about this construct see Chapter 5 [8].

6.13 Complex Searching: The ANDIF and ANDFOR Statements

Complex searching addresses searching techniques needed to accommodate rule-based pattern matching. A matching rule can be expressed in an associative parallel form using ASC ANDIF and ANDFOR statements. The nested ANDIF and ANDFOR constructs can be envisioned as tree searches to a fixed depth, with different search and action specification at every level.

FORMAT: andif logical-parallel-expression then

body

endandif;

andfor logical-parallel-expression then

body

endandfor;

In the body no control statements are allowed. Note there is also no else sub-statements.For more explanation and examples, see chapter 6 of [8].

6.14 ASC Debugger

Unfortunately, this is not implemented. But this normally doesn’t seem to be a problem.

Bibliography for Primer

[1] Eisenberg, Ann, (1982) *Effective Technical* Communication, Mc-Graw Hill Inc.

[2] Hioe, K.H., *ASPROL (Associative Programming Language),* Master’s Thesis, Kent State University, August, 1986.

[3] Lee, J.J., *The Design and Implementation of Parallel SIMD Algorithms for the Traveling Salesperson Problem,* Master’s Thesis, Kent State University, December 1989.

[4] Michalakes, John, *ASP-VMS Handbook,* Kent State University.

[5] Marowka, Ami, *Back to Thin-Core Massively Parallel Processors,* IEEE Computer, December 2011, pp 49-54.

[6] Potter, J.L. (1985) *The Massively Parallel Processor,* MIT Press.

[7] Potter, J.L. (1987) “An Associative Model of Computation”, *Proceedings of the Second International Conference on Supercomputing,* Volume III May 4-7, 1987, pp 1-8.

[8] Potter, J.L. (1992) “*Associative Computing”,* Plenum Publishing Inc. New York*.*

[9] Price, Jonathan (1984) *How to Write a Computer Manual,* The Benjamin/Cummins Publishing Company Inc.

[10] STARAN-E Reference Manual, Ger – 16422, Goodyear Aerospace Corp., November 1977.

Additional References

1. Nael B. Abu-Ghazaleh, Philip A. Wilsey, Jerry Potter, Robert Walker, and Johnnie Baker, "Flexible Parallel Processing in Memory: Architecture + Programming Model ", Proc. of the Third Petaflop Workshop, held in conjunction with Frontiers on Massively Parallel Computing, presented at Annapolis, MD, February 1999.
2. Maher M. Atwah and Johnnie W. Baker, "An Associative Dynamic Convex Hull Algorithm", Proc. of the Tenth IASTED International Conference on Parallel and Distributed Computing and Systems, pages 250-254, presented at Las Vegas, NV, October 1998.
3. Maher M. Atwah and Johnnie W. Baker "An Associative Static and Dynamic Convex Hull Algorithm", Proc. of the 16th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), abstract on page 249, full text on CDROM, presented Ft. Lauderdale, FL, April 2002.
4. Maher M. Atwah and Johnnie W. Baker, "An Associative Implementation of a Parallel Convex Hull Algorithm", Proc. of the 15th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), full text on CDROM, presented San Francisco, CA, April 2001.
5. Maher Atwah, Johnnie Baker, and Selim Akl, *An Associative Implementation of Classical Convex Hull Algorithms*, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, 1996, 435-438.
6. Maher M. Atwah, Johnnie W. Baker, and Selim Akl, "An Associative Implementation of Graham's Convex Hull Algorithm", Proc. of the Seventh IASTED International Conference on Parallel and Distributed Computing and Systems, pages 273-276, presented at George Mason University, October 1995.
7. Johnnie Baker and Mingxian Jin, *Simulation of Enhanced Meshes with MASC, a MSIMD Model*, Proc. of the Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems, Nov. 1999, 511-516.
8. Johnnie W. Baker and Mingxian Jin, "Simulations Between Enhanced Meshes and the Multiple Associative Computing (MASC) Model", Proc. of the 1999 Midwest Workshop on Parallel Processing (MWPP'99), presented at Kent, OH, August 1999.
9. Johnnie W. Baker and Andrew Miller, "A Parallel Production System Extending OPS5", Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation, edited by Joseph JaJa, pages 110-118, paper presented at The University of Maryland at College Park, October, 1990.
10. Wittaya Chantamas, Johnnie Baker, and Michael Scherger, "An Extension of the ASC Language Compiler to Support Multiple Instruction Streams in the MASC Model using the Manager-Worker Paradigm,", Proc. of the 2006 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2006), 7 pages in proceedings CD, June 2006.
11. Wittaya Chantamas and Johnnie Baker, "A Multiple Associative Model to Support Branches in Data Parallel Applications using the Manager-Worker Paradigm", Proc. of the 19th International Parallel and Distributed Processing Symposium (IEEE WMPP Workshop), 8 pages, April 2005.
12. Paul Durand, Rohit Pasari, Johnnie W. Baker, and Chun-che Tsai, "An Efficient Algorithm for Similarity Analysis of Molecules", Internet Journal of Chemistry (IJC), Vol. 2, Article 17, 16 pages,, Online by subscription at http://www.ijc.com/article/1999v2/17/. Unofficial copy at http://www.cs.kent.edu/~jbaker/paper, 1999.
13. Mary Esenwein and Johnnie W. Baker, "VLCD String Matching for Associative Computing and Multiple Broadcast Mesh", Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems, pages 69-74, presented at George Washington University, October, 1997.
14. Mingxian Jin and Johnnie W. Baker "Two Graph Algorithms on an Associative Computing Model", 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'07), Las Vegas, 7 pages, June 25-28, 2007.
15. Mingxian Jin, Johnnie Baker, and Kenneth Batcher, *Timings for Associative Operations on the MASC Model*, Proc. of the 15th International Parallel and Distributed Processing Symposium, Workshop on Massively Parallel Processing, San Francisco, April 2001.
16. Mingxian Jin, Johnnie W. Baker, and Will C. Meilander, "The Power of SIMDs vs. MIMDs in Real-Time Scheduling", , in Proc. of the 16th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), abstract on page 248, full text on CDROM, presented Ft. Lauderdale, FL, April 2002.
17. Will C. Meilander and Johnnie W. Baker, "ATC Architecture Computers -Yesterday, Today, Tomorrow", 43rd Annual Air Traffic Control Association Fall Conference Proceedings, pages 91-95, 1998.
18. Will Meilander, Johnnie Baker, and Mingxian Jin, "Importance of SIMD Computation Reconsidered", Proc. of the 17th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), abstract on page 266, full text on CDROM , presented at Nice, France, April 2003.
19. Will Meilander, Johnnie Baker, and Mingxian Jin, "Predictable Real-Time Scheduling for Air Traffic Control", Proc. of the 15th International Conference of Systems Engineering, pages 533-539, presented at Las Vegas, NV, August 2002.
20. Will C. Meilander, Mingxian Jin, and Johnnie W. Baker, "Tractable Real-Time Air Traffic Control Automation", Proc. of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems, pg 483-488, Cambridge, MA, November 2002.
21. Will C. Meilander, Johnnie W. Baker, and Jerry Potter, "Predictability for Real-Time Command and Control", Proc. of the 15th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), abstract on page 194, full text on CDROM, presented San Francisco, CA, April 2001.
22. Will C. Meilander, Jerry L. Potter, Kathy J. Liszka, and Johnnie W. Baker, "Real-Time Scheduling in Command and Control", in Proc. of the 1999 Midwest Workshop on Parallel Processing, (MWPP'99), presented at Kent, OH, August 1999.
23. Mark Merry and Johnnie W. Baker, "A Constant Time Algorithm for Computing the Hough Transform on a Reconfigurable Mesh", Image and Vision Computing Journal, vol. 14, 1996, pages 35-37, 1996
24. Mark S. Merry and Johnnie W. Baker, "A Constant Time Sorting Algorithm for a Three-Dimensional Mesh and Reconfigurable Network, Parallel Processing Letters, 5(3), pages 401-412, 1995.
25. Mark S. Merry and Johnnie W. Baker "A Constant Time Algorithm for the Channel Assignment Problem Using the Reconfigurable Mesh", in Journal of Parallel Algorithms and Applications, Vol 6, pages 259-271, 1995.
26. Yi Pan , Selim Akl, and Johnnie W. Baker, Co-editors, "Computing on Bus-Based Architectures", Parallel Processing Letters, Special issue, 8(2), 1998.
27. Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, *An Associative Computing Paradigm*, Special Issue on Associative Processing, IEEE Computer, 27(11):19-25, Nov. 1994.
28. Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, "ASC: An Associative Computing Paradigm, Associative Processing and Processors", Associative Processing and Processors, editors: A. Krikelei and C.C.Weems, IEEE Computer Society Press, pages 188-194, 1997.
29. Stewart Reddaway, Will Meilander, Johnnie Baker, and Justin Kidman, "Overview of Air Traffic Control Using an SIMD COTS System", International Parallel and Distributed Processing Symposium, 9 pages, Published on CD, April 2005.
30. Michael Scherger, Johnnie Baker, and Jerry Potter, "Multiple Instruction Stream Control for an Associative Model of Parallel Computation", Proc. of the 17th International Parallel and Distributed Processing Symposium (IEEE Workshop on Massively Parallel Processing), abstract on page 266, full text on CDROM, presented at Nice, France, April 2003.
31. Michael Scherger, Johnnie Baker, and Jerry Potter, "An Object Oriented Framework for and Associative Model of Parallel Computation", Proc. of the 17th International Parallel and Distributed Processing Symposium (Workshop in Advances in Parallel and Distributed Computational Models), abstract on page 166 , full text on CDROM, presented at Nice, France, April 2003.
32. Michael Scherger, Jerry Potter, and Johnnie W. Baker, "On Using the UML to Describe the BSP Model of Parallel Computation", in Proc. of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2002), volume II, pages 578-583, presented Las Vegas, NV, June 2002.
33. Michael Scherger, Jerry Potter, and Johnnie Baker, "On Using UML to Describe the MASC Model of Parallel Computation", Proc. of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '2000), volume V, pages 2639-2645, presented Las Vegas, NV, June 2000.
34. Stephen Scott and Johnnie W. Baker , "Embedding the Hypercube into the 3-Dimensional Mesh" , Proceedings of the 4th Symposium on the Frontiers of Massively Parallel Computation, Edited by H. J. Siegel, pages 577-8, paper presented at Goddard Space Flight Center in McLean VA, October, 1992.
35. Shannon Steinfadt and Johnnie. W. Baker, "SWAMP: Smith-Waterman using Associative Massive Parallelism", IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, 2008 International Parallel and Distributed Processing Symposium (IPDPS) at Miami, 8 pages, published on CD, April 14-18, 2008.
36. Shannon Steinfadt, Michael Scherger, and Johnnie W. Baker, "A Local Sequence Alignment Algorithm Using an Associative Model of Parallel Computation", Proc. of IASTED Computational and Systems Biology (CASB 2006), Dallas, pp. 38-43, Nov. 13-14, 2006.
37. Jerry Trahan, Mingxian Jin, Wittaya Chantamas, and Johnnie Baker, "Relating the Power of the Multiple Associative Computing Model (MASC) to that of Reconfigurable Bus-Based Models", Journal of Parallel & Distributed Computing 70(2010), pgs 458-466, Elsevier Publishers, Available online at ScienceDirect: [www.sciencedirect.com](http://www.cs.kent.edu/~jbaker/www.sciencedirect.com), May, 2010.
38. Darrell Ulm and Johnnie W. Baker, Virtual Parallelism by Self Simulation of the Multiple Instruction Stream Associative Model", in Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 1421-1430, presented Sunnyvale, CA, August, 1996.
39. Darrell Ulm and Johnnie W. Baker, "Solving a 2D Knapsack Problem on an Associative Computer Augmented with a Linear Network", Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications, pages 29-32, presented Sunnyvale, CA, August, 1996.
40. Darrell Ulm and Johnnie Baker, "Solving a Two-Dimensional Knapsack Problem on a Mesh with Multiple Buses", Proc. of the International Conference on Parallel Processing, vol. 3, pages 168-171, paper presented in Wisconsin, August, 1995.
41. Darrell Ulm and Johnnie Baker, "Simulating PRAM with a MSIMD Model (ASC)", Proc. of the International Conference on Parallel Processing, pages 3-10, presented at Minneapolis, MN, August, 1998
42. Darrell Ulm, Johnnie Baker, and Michael Scherger, "Solving a 2D Knapsack Problem Using a Hybrid Data-Parallel/Control-Parallel Style of Computing", Proc of the 18th International Parallel and Distributed Processing Symposium, (IEEE Workshop on Massively Parallel Processing), Santa Fe, NM,, May 2004.
43. Mike Yuan, Johnnie Baker, Frank Drews, Lev Neiman, and Will Meilander, "An Efficient Associative Processor Solution to an Air Traffic Control Problem", Large Scale Parallel Processing IEEE Workshop at the International Parallel and Distributed Computing Symposium (IPDPS2010), published on IPDPS-2010 CDROM and in the IEEE Digital Library with other IPDPS-2010 publications, 8 pages, April 2010.
44. Mike Yuan, Johnnie Baker, Frank Drews, Will Meilander, and Kevin Shaffer, "System Design and Algorithms for an Air Traffic Control Prototype using an Associative Processor with Timings and Comparisons", IEEE Transactions of Parallel and Distributed Systems (TPDS), Currently revising to meet reviewers' requests
45. Mike Yuan, Johnnie Baker, Frank Drews, Lev Neiman, and Will Meilander, *Scalable and Efficient Associative Processor Solution to Guarantee Real-Time Requirements for Air Traffic Control Systems,* Large Scale Parallel Processing IEEE Workshop at the International Parallel and Distributed Computing Symposium (IPDPS2011), published on IPDPS-2011 CDROM and in the IEEE Digital Library with other IPDPS-2011 publications, 8 pages,