Parallel FORTH

John E. Dorband

Image Analysis Facility/Code 635
NASA/Goddard Space Flight Center
Greenbelt, MD 20771

Abstract

The extension of Forth into the realm of parallel processing on the Massively Parallel Processor [1] (MPP) is described. The extended language, MPP Parallel FORTH, is a derivative of Forth-83 [2-4] with extensions designed by the author as philosophically similar to serial Forth as possible. This paper first discusses the MPP hardware characteristics, as viewed by the Forth programmer, and then presents a description of MPP Parallel FORTH along with a detailed example developed by the author showing how the bitonic sort [5] is implemented in this language.

Introduction

The parallelism that can be exploited on the MPP is of a single-instruction-stream multiple-data-stream (SIMD) nature (Figure 1). The SIMD architecture contains one control unit, which is a typical serial processor, and many arithmetic logic units (ALUs).

The SIMD machine is capable of two types of processing, occuring concurrently: serial or scalar processing, and parallel processing. The parallel processing of a SIMD machine is the same as the serial processing except it happens in many different places. The SIMD approach is very different from the parallelism found in most multiprocessor systems which are typically multiple-instruction-stream multiple-data-stream (MIMD) in nature:

- In a MIMD architecture, each processor has its own code to execute so each can execute a totally different set of programs. The control of a SIMD machine on the other hand is much simpler because there is only one set of control code.
- The processors and the languages to support a MIMD machine must include all the necessary operating system routines to facilitate multitasking and network-like communications. By contrast, neither the SIMD hardware nor its languages need to support multitasking or network-like communications.

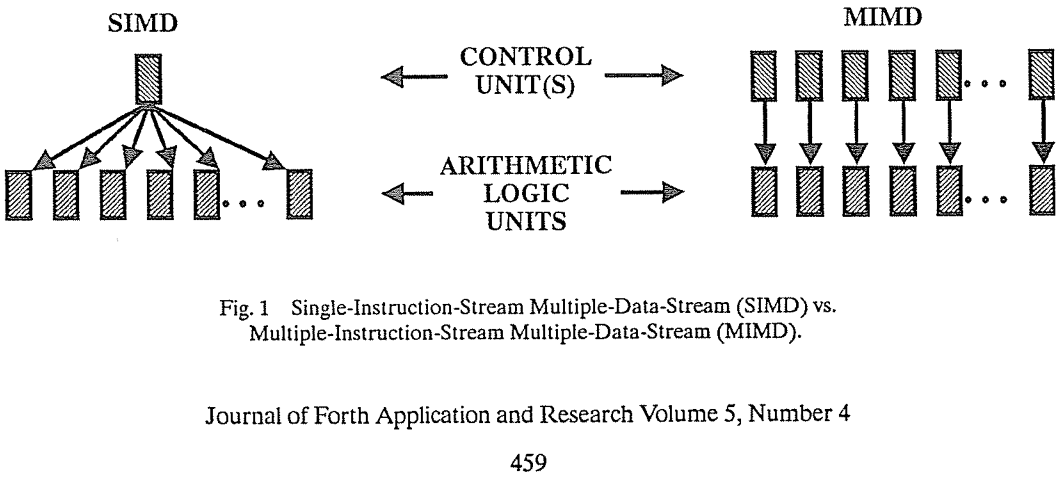


Fig. 1 Single-Instruction-Stream Multiple-Data-Stream (SIMD) vs. Multiple-Instruction-Stream Multiple-Data-Stream (MIMD).

MPP Environment

The MPP contains an array of 16,384 processing elements(PEs), the array unit, arranged in a 128 by 128 square grid. Each PE (Figure 2) is a bit serial arithmetic logic unit (ALU) with 1024 bits of random access memory (RAM). Thus, the entire array contains 2 million bytes of memory which can be viewed as 1024 bit planes of 128×128 bits each. [In the general MPP design, both the number of PEs and the number of bit planes are expandable depending on computational needs and engineering considerations.] All PEs are given the same instruction at the same time; thus computing in this array can be viewed by the programmer as serial processing on a single PE. Yet, processing is actually happening on all 16,384 PEs at the same time. Because of this, the MPP and the languages that are used to program it need not involve extremely complex programing concepts, but only two nearly identical modes of operation, serial and parallel. This similarity of modes is used extensively in MPP Parallel FORTH.

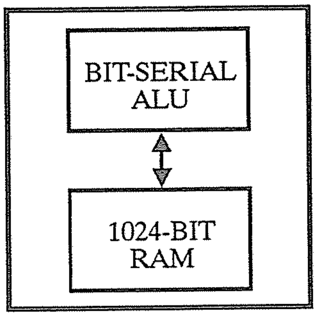
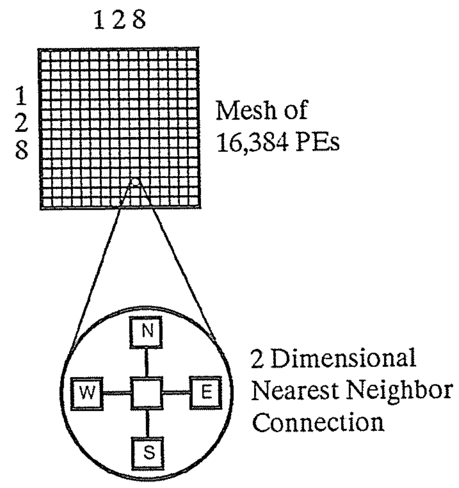
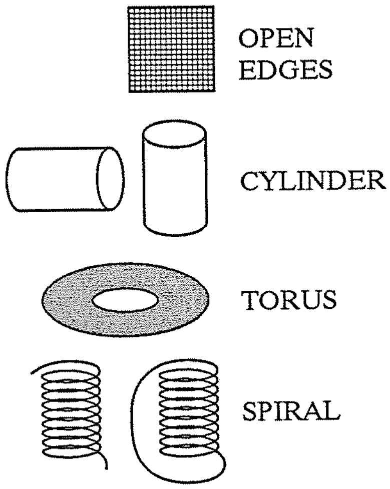


Fig. 2 MPP Processing Element (PE)

The difference between programming in serial and in parallel on a SIMD architecture is not in programming thousands of processors but in communicating between thousands of processors simultaneously. The PEs of the MPP array communicate with each other using a square mesh of data paths (Figure 3(a)), where each PE can pass data only to its four adjacent PEs. The edges



(a) Communication among MPP PEs is over a square mesh of data paths.



(b) Data path Topologies.

Fig. 3 Massively Parallel Communication.

of the mesh can be connected in various ways to form topologies (Figure 3(b)), such as a simple square, cylinders, a torus, or a helix. This communication arrangement allows the programmer to move, simultaneously, as many as 16,384 bits of data, as far as 128 rows or columns away from their original source PEs.

As well as having a main control unit (MCU) for scalar processing and an array unit for processing 16,384 elements of data in parallel, the MPP has a staging memory (STG) (Figure 4). This memory is in the path over which data is moved from the host computer (VAX-11/780) into the array unit memory. The staging memory contains 32 megabytes of storage capacity, allowing it to be configured as 16,384 bit planes of 128×128 bits each.

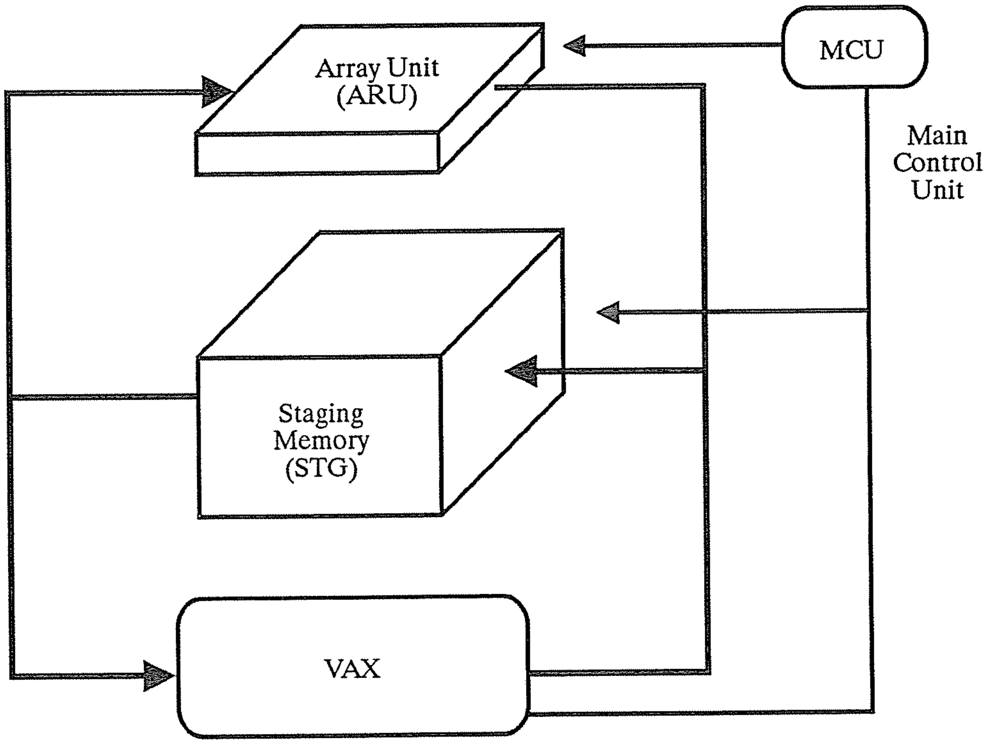


Fig. 4 MPP Overview.

Therefore, the MPP, as viewed by the Forth user, consists of three principal components:

ARU	Array Unit	for parallel processing of data
MCU	Main Control Unit	for scalar processing and control of the ARU
STG	Staging Memory	for I/O and as a large external bit-plane memory

A mask capability in each PE in the ARU allows the programmer to perform conditional processing on a PE by PE basis. Conditional processing (such as the execution of an **IF ... ELSE ... THEN** statement) on the array divides the ARU into two groups of PEs — those for which the condition is true and those for which the condition is false. Since PEs can be individually told not to execute the current instruction, the PEs for which the condition is true will only execute those instructions between the **IF** and the **ELSE** and those processors for which the condition is false will only execute those instructions between the **ELSE** and the **THEN**. Thus, through prudent use of conditional statements, groups of processors can be programmed to perform different functions within the same block of code.

Parallel FORTH Implementation

Parallel FORTH has been implemented by the author as simply and in as straightforward a way as possible. A serial Forth system is implemented on the MCU. Parallel extensions have been added to the kernel under a new vocabulary called **PARALLEL**. Context switching has been simplified so that the Forth word '**{**' switches to the parallel vocabulary and '**}**' switches back to the vocabulary that was in use before the switch to the parallel vocabulary. This allows the user to redefine serial words as analogous parallel words in a parallel context, making it easier for the user to remember the new parallel words, giving Parallel FORTH the facility to treat parallel programming as though it were serial. For example, **+** normally means to add two numbers that are on the *data stack*, but in the parallel context (eg. '**{ + }**') + means add two 128×128 arrays of numbers on the *array stack*, which is in the ARU memory (Figure 5).

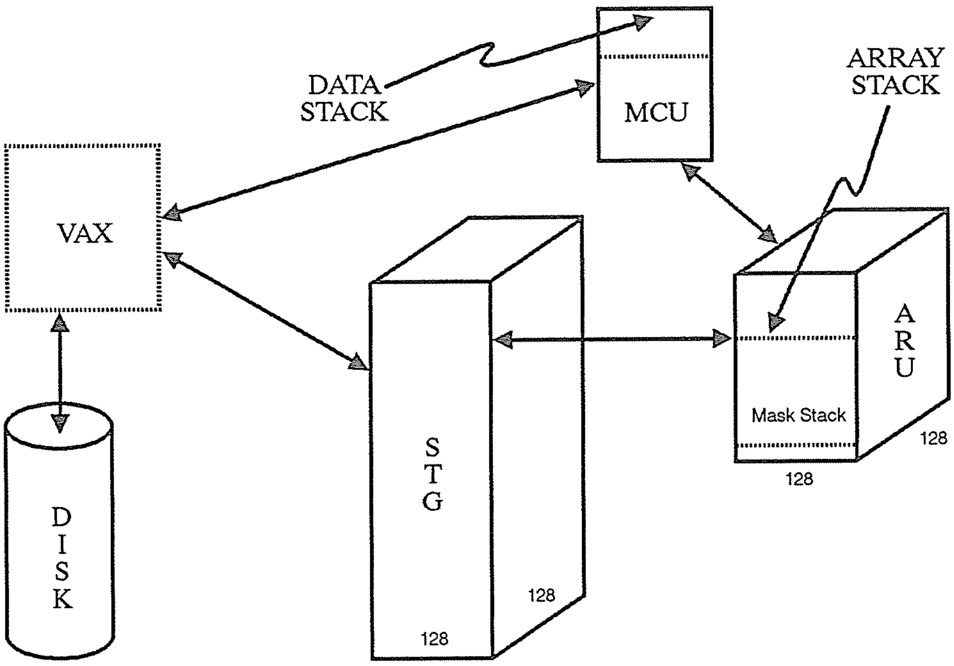


Fig. 5 Organization of MPP memories.

Two new stacks have been added to Parallel FORTH that are not in serial versions of Forth. These two stacks are the array stack (A) and the mask stack (M). The mask stack is not normally used or seen by the Parallel FORTH programmer. It is used to facilitate nested conditional statements, such as '**IF ... ELSE ... THEN**' or '**BEGIN ... UNTIL**'. The array stack is extensively used by the Parallel FORTH programmer, since it is the parallel equivalent of a serial data stack. Most operations that can be performed on elements of the serial data stack have corresponding operations that can be performed in parallel on the array stack, such as **+**, *****, **DUP**, **DROP**, and **ROT**. There are a few other operations that are unique to the array stack. These primarily deal with interprocessor communications and globally accumulative results.

The following sections discuss in more depth the parallel operations designed and implemented by the author to extend Forth into the realm of parallelism.

Vocabulary and Data Definition

In Parallel FORTH there is a vocabulary called **PARALLEL**. All new parallel words are in this vocabulary (Figure 6). As pointed out previously '{' and '}' are used to enter and exit the parallel vocabulary. The following is a definition that will manipulate the MCU data stack:

```
: MULTADD  * + ;
```

While the next definition manipulates the ARU array stack:

```
: MULTADD { * + } ;
```

Parallel variables can be allocated in either the array or the staging memory. If a user wants to allocate a 128×128 array of 7-bit values named **AR1** in the staging memory, the following is used:

```
7  STG  VARIABLE  AR1
```

where **STG** designates the allocation of storage in the staging memory. If a user wants to allocate a 128×128 array of 11-bit values in the array memory named **AR2**, the following is used:

```
11 ARU VARIABLE  AR2
```

where **ARU** designates the allocation of storage in the array memory. The definition of parallel constants is similar to defining variables, except the user puts an array on top of the array stack and then executes the statement:

```
13 ARU CONSTANT CON1
```

to create a 128×128 array of 13-bit constants. Likewise vectors and arrays of 128×128 arrays may be defined with **VECTOR** and **MATRIX**, respectively. A vector of 20 8-bit 128×128 arrays can be defined with the following statement:

```
2Ø 8 ARU VECTOR VEC1
```

Again parallel mode is very similar to serial mode. If a variable, vector, or matrix is allocated in parallel mode, it is allocated simultaneously in every PE.

CONTEXT SWITCH OPERATIONS

```
{
}
```

I/O OPERATIONS

```
OPEN
GET
PUT
```

MEMORY OPERATIONS

```
@
!
SCALAR
GMAX
GMIN
GOR
```

ARRAY STACK OPERATIONS

```
DUP
DROP
SWAP
OVER
ROT
PICK
ROLL
-NDROP
NDROP
A@
>A
ZERO
EXTRACT
SLIDE
TOPOLOGY
```

FIXED PRECISION ARITHMETIC OPERATIONS

```
+
-
*
/
MOD
/MOD
MAX
MIN
ABS
NEGATE
1+
1-
2/
2*
AND
OR
XOR
NOT
```

VARIABLE PRECISION ARITHMETIC OPERATIONS

```
~+
~-
~*
~/
~MOD
~/MOD
```

COMPARISON OPERATIONS

```
<
=
>
Ø<
Ø=
Ø>
```

CONTROL OPERATIONS

```
IF…ELSE…THEN
BEGIN…UNTIL
BEGIN…WHILE…REPEAT
```

Fig. 6 Words Implemented in MPP Parallel FORTH

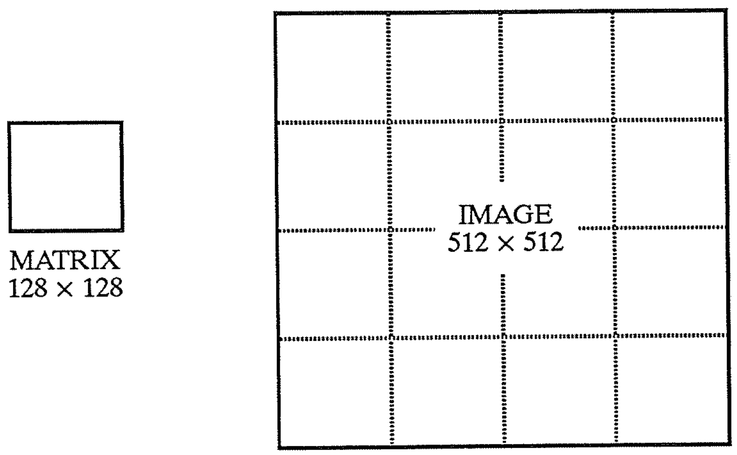


Fig. 7 Matrix and image file formats.

Parallel I/O

Parallel files can be stored on the host in either *matrix* or *image* format (Figure 7). Each format allows for 8-bit, 16-bit and 32-bit values. A matrix format file contains multiple arrays of 128×128 values. An image format file contains multiple images of 512×512 values. The following command:

```
CHANA IMAGE8 OPEN WHAT.DAT
```

opens the file 'what.dat' as an image file of images with 8-bit values. **GET** is then used to read the matrix or image into a previously defined array variable. An image from an image file of 8-bit values should only be loaded into a **VECTOR** or **MATRIX** that has at least 8×16 or 128 bits allocated to it. The command to read a matrix into a stager array is:

```
V1 3  GET
```

This loads the third matrix of the current file into variable **V1**. To store an image into a file, the word **PUT** is used (i.e., **V1 3 PUT**).

Memory Operations

Memory operations are used to move data between the three MPP memories: the MCU, the ARU, and the STG. The word '**@**' fetches arrays from array variables in the STG and the ARU memory and puts them on the array stack. The word '**!**' stores an array from the array stack into an array variable in the STG or ARU memory. The word '**SCALAR**' takes a value from the data stack in the MCU memory, broadcasts it to all PE's, and produces an array on top of the array stack that has the same value for all elements of the array. Operations such as **GMAX** (global maximum), **GMIN** (global minimum), and **GOR** (global OR) can reduce an array of values into a scalar value that can be put onto the data stack.

Array Stack Operations

Most array manipulation occurs on the array stack. The PEs of the ARU are bit serial processors. This means they can perform arithmetic on almost any size numbers, thus the elements on the array stack can be almost any size in terms of bit lengths. A stack in the MCU is used to keep track of the sizes and number of values on the array stack in the ARU. The arrangement of data elements on the array stack is the same for all PEs. The array stack is manipulated by operations very similar to those used on the data stack. These operations consist of words such as DUP, DROP, SWAP, OVER, ROT, PICK, and ROLL. In addition to the standard stack operations there are also operations that are unique to the array stack. They consist of the following words: -NDROP, NDROP, A@, >A, ZERO, EXTRACT, SLIDE, and TOPOLOGY.

NDROP drops the top *n* elements of the array stack. -NDROP skips the first *n*1 elements of the array stack and drops the next *n*2 elements of the array stack. 'A@' copies the descriptor from the array stack onto the data stack. A parallel array descriptor consists of two values: the address of the least significant bit plane of the array(LSB) and the number of bit planes in the array(LEN). '>A' creates an array of *n*-bit data values on the array stack, where *n* is taken from the top of the data stack. ZERO is the same as '>A' except the *n*-bit data values are initialized to zero. EXTRACT extracts a field of bits from the top element of the array stack and leaves it as the top element of the array stack. SLIDE slides the top element of the array stack across the array of PE's. The TOPOLOGY operation changes the topology of the ARU.

Arithmetic, Logic, and Comparison Operations

All the operations in this section deal primarily with the elements on the top of the array stack and are analogous to corresponding operations that operate on the top of the data stack. The difference is that operations on the array stack perform 16,384 operations at the same time instead of one at a time and values on the array stack can have variable numbers of bits instead of a fixed number such as 8, 16, or 32.

Normally operations on the data stack are either single or double precision. On the array stack, however, operations are classified as either fixed or variable precision. A fixed precision operation requires that both operands and their result have the same length. A variable precision operation may operate on operands whose lengths are different. The result of such operations has a length that is dependent on both the specific operation and the length of the operands. All basic operations discussed here have a fixed precision form. Some operations have both a fixed and a variable precision form. These are +, -, *, /, MOD, and /MOD. Their variable precision forms are ~+, ~-, ~*, ~/, ~MOD, and ~/MOD.

The result of a ~+ or a ~- operation has a length equal to 1 plus the maximum of the two operand's lengths. The result of a ~* operation has a length equal to the sum of the length of the two operands. The length of result of a ~/ operation is the length of the dividend operand. Note that the length of the dividend must be larger than that of the divisor. The result of the ~MOD operation has a length equal to the length of the divisor operand. Since the result of the ~/MOD operation is the result of the ~/ operation followed by the ~MOD operation, the lengths of the results are the same as described for ~/ and ~MOD.

The fixed precision only operations are MAX, MIN, ABS, NEGATE, 1+, 1-, 2/, 1*, AND, OR, XOR, and NOT. Three special operations find the aggregate result of all the top elements of the array stack and place it on the data stack. These global operations are global maximum(GMAX), global minimum(GMIN), and global OR(GOR).

Comparison operations differ slightly from the other operations in this section in that they result in a value of length 1. These operations are <, =, >, Ø<, Ø=, and Ø>.

Control Operations

Control operations cause certain portions of code to be executed on some data and not on others. Parallel control is quite different from serial control. In serial control, condition evaluation determines whether or not a certain piece of code will be executed. In parallel control, the code corresponding to both the true condition and the false condition may have to be executed. Some of the PEs must be turned off during the execution of the code for the true condition, then turned on for the execution of the code for the false condition. This is accomplished with a mask bit. It is set to one in PEs whose data satisfy the condition, and to zero in those whose data does not. Thus only those PEs that satisfy the condition execute the code for the true condition. The mask bit is then complemented and only those PEs that did not satisfy the condition will execute the code for the false condition. As with execution of serial conditions, parallel conditions can be nested. Therefore, there is a mask stack. Mask stack primitive operations are used to implement the operations in this section.

The basic conditional structure is the `IF ... ELSE ... THEN` statement. The `IF` word takes the least significant bit of the top element of the array stack and puts it on the top of the mask stack. Then it takes the `AND` of the top two bits of the mask stack, and puts it on the top of the mask stack. The `ELSE` word drops the top bit of the mask stack, complements the top element of the mask stack, then it takes the `AND` of the top two bits of the mask stack, and puts it on the top of the mask stack. The `THEN` word drops the top two elements of the mask stack.

The parallel conditional loop structure is also somewhat unusual. It continues to execute as long as there is a PE that has not met the condition to terminate the loop. The two types of loops are the `BEGIN ... UNTIL` and the `BEGIN ... WHILE ... REPEAT`. The `BEGIN` word duplicates the top element of the mask stack. The `REPEAT` word marks the end of the loop. The `WHILE` word `AND`s the least significant bit of the top element of the array stack to the top element of the mask stack and terminates the loop if no PE has the top element of the mask stack equal to one. The `UNTIL` word is the same as `WHILE` except the least significant bit of the top element of the array stack is complemented before it is `AND`ed to the top element of the mask stack.

Note that only certain operations are maskable. Therefore, one should be aware that operations may execute when the processor was masked because the operation was not maskable. Generally, only operations that do not change the number of elements on the array stack or the order of the elements on the array stack are maskable. Thus, most stack manipulation operations and two operand operations are not maskable.

A Bitonic Sort using Parallel FORTH

The following example shows Parallel FORTH code that runs on the MPP. There are two sorts: one that sorts data into shuffled row major order and one that sorts data starting in the northwest corner toward the southeast corner of the array in progressively larger square regions. These sorts sort the top two elements on the array stack across all processors (see Listing).

Conclusion

The development of Parallel FORTH has demonstrated how easy and straight forward it can be to extend a serial language (i.e. Forth) to support parallel processing on a SIMD processor such as the MPP. A more detailed description of MPP Parallel FORTH [6] can be obtained from the MPP User Support Office, Code 635, NASA/Goddard Space Flight Center, Greenbelt, MD 20771, phone (301) 286-9412.

Acknowledgements

I would like to express my appreciation to Dick Fahey, Arne Henden, Dan Klinglesmith, and Archie Warnock for their advice and consultation on the Forth philosophy and to Lisa Hamet for her help in the coding and implementation of the UNIFORTH™[1] kernel on the MPP.

References

[1] Potter, J.L., ed., *The Massively Parallel Processor*, MIT Press, Cambridge, MA, 1985.

[2] Brodie, L., *Starting Forth*, Prentice-Hall, Inc., Englewood Cliffs, NJ., 1981.

[3] Brodie, L., *Thinking Forth*, Prentice-Hall, Inc., Englewood Cliffs, NJ., 1984.

[1] UNIFORTH is a trademark of Unified Software Systems.

[4] Henden, A., *UNIFORTH User's Guide*, Unified Software Systems, Columbus, OH, 1985.

[5] Nassimi, D. and Sahni, S., "Bitonic Sort on a Mesh-Connected Parallel Computer," *IEEE Trans. on Computers*, C-27, 1 (January 1979), pp. 2-7.

[6] Dorband, J.E., *MPP Parallel FORTH User's Guide*, September 1986.

```
SCR # Ø



                A SHUFFLE ROW MAJOR BITONIC SORT (SORTSH)
                    AND A SORT THAT SORTS TO THE NORTHWEST
                         CORNER OF THE MPP ARRAY (SORTCR)




               NOTE: THESE SORTS ONLY SORT THE TOP 2 LAYERS
                                 ON THE ARRAY STACK.




SCR # 1
( SORTF - MASK ALLOCATION AND INITIALIZATION )

14 VECTOR NS  14 VECTOR WE
14 VECTOR SN  14 VECTOR EW
{  15 1 ARU VECTOR MSK  1 ARU VARIABLE OF }
: INIT_MASK
    14 Ø DO Ø I WE ! Ø I NS ! LOOP  ( CLEAR NS & WE )
    14 Ø DO Ø I EW ! Ø I SN ! LOOP  ( CLEAR SN & EW )
    1 14 Ø DO DUP I WE ! 2* 2 +LOOP DROP ( LOAD WE )
    -1 14 Ø DO DUP I EW ! 2* 2 +LOOP DROP ( LOAD EW )
    1 14 1 DO DUP I NS ! 2* 2 +LOOP DROP ( LOAD NS )
    -1 14 1 DO DUP I SN ! 2* 2 +LOOP DROP ( LOAD SN )
    { 1 ZERO 14 MSK ! }              ( BUILD MASK )
    Ø 14 Ø DO DUP { C 1 EXTRACT I MSK ! } 1+ 2 +LOOP DROP
    Ø 14 1 DO DUP { R 1 EXTRACT I MSK ! } 1+ 2 +LOOP DROP
; -->
```

```
SCR # 2
( SORTF - COMPARISON AND EXCHANGE ROUTINES )

: COMPARE ( - A: <EXCHANGE FLAG> )
     { OVER OVER < } ;
: SORT_EXG  {  IF OVER OVER Ø INSERT
             ROT ROT SWAP Ø INSERT THEN } ;
: COM/EXG ( A: <EXCHANGE ORDER> - )
     COMPARE { OF @ XOR SORT_EXG }   ;

: CROSS ( S: NS WE MSK - )
          { @ } OVER OVER { IF OVER OVER SLIDE Ø INSERT
          ELSE ROT ROT SWAP } NEGATE SWAP NEGATE SWAP
          { SLIDE Ø INSERT THEN SWAP } ;
-->


SCR # 3
( SORTF - SHUFFLE ROW MAJOR BITONIC SORT )

: SORTSH  ( A: <VALUE> <VALUE> - <VALUE> <VALUE>  )
      INIT_MASK
      Ø MSK { @ OF ! } COM/EXG
         Ø DO
            I 1+ MSK { @ OF ! }
            I NS @ I WE @ I MSK CROSS COM/EXG
            I 1+ Ø DO
                 I NS @ I WE @ I MSK CROSS COM/EXG
            LOOP
      LOOP
      ;
-->


SCR # 4
( SORTF - SORTS INTO THE NORTHWEST CORNER )

: SORTCR  ( A:  <VALUE> <VALUE> - <VALUE> <VALUE>  )
      INIT_MASK
      Ø MSK { @ OF ! } COM/EXG
         Ø DO
           I 1+ MSK { @ OF ! }
           Ø I DO
                 I NS @ I WE @ I MSK CROSS COM/EXG
                 I NS @ I WE @ I MSK CROSS
           -1 +LOOP COM/EXG
      LOOP
      ;
;S
```