A Multiprocessor Tree-searching System Design

T.A. Marsland
and
F. Popowich

Technical Report TR83-6

July 1983

A Multiprocessor Tree-searching System Design

T.A. Marsland
and
F. Popowich

Computing Science Department
University of Alberta
EDMONTON T6G 2H1
Canada

07-24-83
TR83-6

Abstract

Sequential versions of the alpha-beta search algorithm have been improved by refinements such as iterative deepening, narrow window searching and the use of memory tables. The design issues affecting a parallel implementation are discussed with emphasis on a tree decomposition scheme which is well suited for use with well ordered trees. When dealing with parallel processing systems, inter-unit communication is perhaps the most important factor. Therefore, an implementation of this tree decomposition based algorithm is presented that can operate with a limited amount of inter-unit communication on a network of processors.

Keywords: multiprocessors, concurrent programming, message sending, graph and tree search strategies, tree decomposition, alpha-beta search.

Acknowledgement

The assistance of Steve Sutphen and Jan Rus in configuring and maintaining the system hardware was much appreciated. Without their modification of the Central Data Communication Interface and the SUN Workstations, a working system would have been impossible. Financial support for this project was provided by the Canadian National Science and Engineering Research Council grants A5556 and A7902.

1. Introduction

Much time and effort has been devoted to the study of parallel evaluation of game trees. The approaches to the problem differ, varying from partitioning the search window between processors [BAUD78], or assigning individual processors to separate subtrees [FISH80], to managing a pool of processors [AKL82][LIND83]. Since the Principal Variation Search has illustrated its effectiveness in strongly ordered game trees [MARS83], a parallel version of this algorithm, dubbed PV Splitting, has been devised that follows the tree splitting approach [MARS82]. We have implemented a version of PV Splitting on a loosely coupled network of processors.

2. Principal Variation Searching (PVS)

The effectiveness of the PVS algorithm relies on strong move ordering. In this algorithm, alternatives to the first move are assumed inferior until proven otherwise. They are examined with a zero-width window, based on a bound obtained from the search of the first move. An alpha-beta algorithm suitable for doing this search, since it can return values outside of the given window [FISH81], is shown in Figure 1. It uses predefined functions *generate*, to form an array of successors to the given position, *empty,* to determine if this array is empty, and *sizeof* to calculate the size of the array. *Make* and *undo* are used to make and retract the given move respectively, while *evaluate* is used to perform a leaf node evaluation of a position. These functions are very application dependent, and in our case-study form part of a simple chess-playing program. The algorithm of Figure 1 ensures that the best available score is returned whenever all the moves have a value that is less than the alpha bound. The mode of presentation is a Pascal-like pseudo code, extended with a *return* statement for function exit, although our actual implementations are done in the C language.

The alphabeta function may either be used directly, or be called upon to perform an aspiration or a minimal window search (MWS). A zero width window is used to determine whether the true value lies above the window searched. If so, the search is repeated on this new principal variation with a window which will contain the true value. The *mws* function of Figure 2 is essentially the same as **Calphabeta** [FISH81], which in turn is similar to **SCOUT** [PEAR80][CAMP83]. In the presentation of the MWS

```
FUNCTION alphabeta(p : position;
                   alpha, beta, depth : integer) : integer;

  VAR score, i, value : integer;
      posn : ARRAY[1..MAXWIDTH] OF position;

BEGIN
  IF depth <= 0 THEN
    return(evaluate(p));        { a terminal node }

  posn := generate(p);          { generate successors }
  IF empty(posn) THEN
    return(evaluate(p));        { no legal moves }

  score := -MAXINT;
  FOR i := 1 TO sizeof(posn) DO BEGIN
    make(posn[i]);
    value := -alphabeta(posn[i], -beta, -max(alpha,score), depth-1);
    undo(posn[i]);

    IF value > score THEN
      score := value;          { an improvement }
    IF score >= beta THEN
      return(score);           { a cutoff }
  END;
  return(score);
END;
```

Figure 1. Negamax implementation to return best available score.

algorithm, reference to the utility functions for making and retracting moves has been omitted for clarity. In our case we prefer to combine *alphabeta* and *mws* into a single function, *pvs*, which is called Principal Variation Search and appears in Figure 3.

There are several enhancements to *pvs* that improve its performance dramatically. When **iterative deepening** [GILL78] is used, the moves are sorted after successively deeper searches, thus providing a more strongly ordered movelist for the next search. After each D-ply search, a sequence of moves from the root to a leaf node is stored in a **refutation table** [MARS83]. This table contains the best variation for the selected move, while for the others, a sequence sufficient to refute an alternative is stored. Upon a D+1 ply search, the table is used to initially direct each move along a potential refutation and thus dramatically reduce the search time. This knowledge table is of modest size, being linear with search depth, and is

```
FUNCTION mws(p : position; depth : integer) : integer;
  VAR score, i, value : integer;
      posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
  IF depth = 0 THEN
    return(evaluate(p));

  posn := generate(p);      { generate successors }
  IF empty(posn) THEN
    return(evaluate(p));    { no successors }

  score := -mws(posn[1], depth-1);
  FOR i := 2 TO sizeof(posn) DO BEGIN
    value := -alphabeta(posn[i], -score-1, -score, depth-1);
    IF value > score THEN
      score := -alphabeta(posn[i], -MAXINT, -value, depth-1);
  END;
  return(score);
END;
```

Figure 2. Minimal Window Search.

```
FUNCTION pvs(p : position; alpha,beta,depth : integer) : integer;
  VAR score, i, value : integer;
      posn : ARRAY[1..MAXWIDTH] OF position;

BEGIN                          { assert depth positive }
  IF depth = 0 THEN            { leaf, maximum depth?  }
     return(evaluate(p));
                          { determine successors  }
  posn := generate(p);
  IF empty(posn) THEN          { leaf, no moves? }
     return(evaluate(p));

  score := -pvs(posn[1], -beta, -alpha, depth-1);
  FOR i := 2 TO sizeof(posn) DO BEGIN
     IF (score >= beta) THEN        { cutoff? }
        return(score);
     alpha := max(score, alpha);
     value := -pvs(posn[i], -alpha-1, -alpha, depth-1);
     IF (value > score) THEN
        score := -pvs(posn[i], -beta, -value, depth-1);
  END {forloop};
  return(score);
END {pvs};
```

Figure 3. Depth-Limited Principal Variation Search.

cheap to maintain. One can also implement a larger table, which holds for future retrieval not only the preferred move in a position, but also a bound on the value of the emanating subtree. This **transposition table** can be used to improve search windows, provide more cutoffs, and to extend the effective search depth [SLAT77][MARS82].

During our preliminary experiments with iterative deepening on multiprocessor systems, we discovered that the method was very sensitive to the sorting algorithm used. One may compare our uniprocessor results [MARS83], using quicksort, with Figure 4 where heapsort was employed. With the latter sort, although the relative performance was not altered, an overall improvement was noted. From this experience we conclude that a stable sort (e.g., a bubble sort) should be used between the iterations. Although we have not tried this ourselves, it is reasonable to assume that preservation of the initial partial ordering is important, since considerable effort is usually put into this aspect. The need for this preservation is apparent since the search itself may not discriminate effectively between approximately equal moves where the material balance is unaltered. This is because there are so many leaf nodes that an expensive evaluation

Figure 4. Performance of alpha-beta enhancements.

function cannot be used.

3. The Principal Variation Splitting Algorithm

The PV Splitting algorithm, based on PV Search, incorporates tree decomposition by having independent subtrees examined by individual processors arranged in a processor tree. A processor tree consists of computers and communication lines, corresponding to the tree nodes and branches respectively, Figure 5. Any processor can communicate directly with only its master and its slaves. It should be noted that, by supplying appropriate software, an individual processor can behave as a both a master and a slave.

Tree decomposition is not performed until after all the processors have searched the principal variation. Using this approach, the amount of search overhead can be reduced. Normally, when tree decomposition is used for parallel implementations of alpha-beta algorithms, some tree cut-offs may not occur, since more moves will be examined without the benefit of a new cutoff value [BAUD78]. This becomes

Figure 5. Processor Tree Structure.

increasingly evident with larger processor tree widths. In our PV Splitting algorithm, on the other hand, since the best move is first analyzed by all processors, not only is the first variation searched faster but the subsequent decomposition will also have all the slaves starting with a good window value, thus providing more cutoffs and allowing the balance of the search to proceed more quickly.

Figure 6 illustrates a version of the PV Splitting algorithm [CAMP81] enhanced with the following constructs adapted from Fishburn [FISH81].

1. *j.treesplit,* to indicate the execution of the procedure treesplit on processor "j".
2. PARFOR, a parallel loop which conceptually creates a separate process for each iteration of the loop. The program continues as a single process when all iterations are complete.
3. WHEN, to wait until its associated condition is true before proceeding with the body of the statement.
4. CRITICAL, to allow only one process at a time into the next block of code.
5. *terminate,* to kill all currently active processes in the PARFOR loop.

For illustrative purposes, we have assumed in Figure 6 that the processor tree length is less than the depth of the search tree.

Our current implementation of a PV Splitting algorithm uses a processor tree of depth one. Consequently, the *treesplit* and *pvsplit* routines may be simplified into one called *pmws,* Figure 7. *Pmws,* which stands for Parallel Minimal Window Search, follows the design of *mws* (Figure 2) quite closely, since the search performed on the alternate variations uses the minimum window, instead of the normal window search of *pvsplit.* By omitting some of the details concerning move updates from the *pmws* function we have managed to not only include code for iterative deepening, but also to indicate where inter-processor communication occurs. The *transmit* and *receive* functions are used to pass information between processors, with *j.slaveab* being the routine executed by processor "j". The major disadvantage of this method is that only a single processor is available to search any new principal variation which emerges, since no tree splitting occurs at that time. This could be overcome if all processors were re-applied to a new candidate, as they are for the initial principal variation.

```
FUNCTION pvsplit(p : position;
                 alpha, beta, depth, length : integer) : integer;
  VAR i, value: integer; j: processor;
      posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
  IF length = 0 THEN                  { end of processor tree }
    return(alphabeta(p, alpha, beta, depth));

  posn := generate(p);                { generate successors   }

  alpha := -pvsplit(posn[1], -beta, -alpha, depth-1, length-1);
  IF alpha >= beta THEN
    return(alpha);

  PARFOR i := 2 TO sizeof(posn) DO     { loop through successors }
    WHEN (a slave j is idle) BEGIN
      value := -j.treesplit(posn[i],-beta,-alpha,depth-1,length-1);
      CRITICAL IF value > alpha THEN
        alpha := value;

      IF alpha >= beta THEN BEGIN
        terminate();
        return(alpha);
      END;
    END;
  return(alpha);
END;


FUNCTION treesplit(p : position;
                   alpha, beta, depth, length : integer) : integer;
  VAR i: integer; posn : ARRAY[1..MAXWIDTH] OF position;
      j: processor;
BEGIN
  IF length = 0 THEN                  { end of processor tree }
    return(alphabeta(p, alpha, beta, depth));

  posn := generate(p);                { generate successors   }

  PARFOR i := 1 TO sizeof(posn) DO     { loop through successors }
    WHEN (a slave j is idle) BEGIN
      value := -j.treesplit(posn[i],-beta,-alpha,depth-1,length-1);
      CRITICAL IF value > alpha THEN
        alpha := value;

      IF alpha >= beta THEN BEGIN
        terminate();
        return(alpha);
      END;
    END;
  return(alpha);
END;
```

Figure 6. Principal Variation Splitting

```
FUNCTION main(p : position; maxdepth : integer) : integer
  VAR score, i, j, value, depth : integer;
      posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
  posn := generate(p);
  IF empty(posn) THEN
    return(evaluate(p))
  for depth := 1 to maxdepth DO BEGIN   { iterative deepening }
    score := -pmws(posn[1], depth-1);
    PARFOR i := 2 TO sizeof(posn) DO
      WHEN (a slave j is idle) DO BEGIN
        transmit(j, posn[i], score, depth); { send parameters }
        j.slaveab;                          {  to "j"          }
        receive(j, posn[i].value);
        CRITICAL IF posn[i].value > score THEN
              score := posn[i].value;
      END;
      sort(posn);
  END;
  return(score);
END;

function pmws(p : position; depth : integer) : integer;
  VAR score, i, j, value : integer;
      posn : ARRAY[1..MAXWIDTH] OF position;
BEGIN
  IF depth = 0 THEN
    return(evaluate(p));
  posn := generate(p);
  IF empty(posn) THEN
    return(evaluate(p));
  score := -pmws(posn[1], depth-1);
  PARFOR i := 2 TO sizeof(posn) DO
    WHEN (a slave j is idle) DO BEGIN
      transmit(j, posn[i], score, depth);  { send to "j" }
      j.slaveab;
      receive(j, posn[i].value);
      CRITICAL IF posn[i].value > score THEN
        score := posn[i].value;
    END;
  return(score);
END;

procedure slaveab;
  VAR value, score, depth : integer; p : position;
BEGIN
    receive(MASTER, p, score, depth);
    value := alphabeta(p, -score-1, -score, depth-1);
    IF value > score THEN
      value := alphabeta(p, -MAXINT, -value, depth-1);
    transmit(MASTER, value);
END
```

Figure 7. Parallel Minimal Window Search

4. Inter-Processor Communication

One of the features of our implementation is the small amount of necessary inter-processor communication. Even so, there are several categories of inter unit communication, of which the fundamental type is **inter-node** communication.

When a tree is to be searched, it is necessary that each processor generate identical move lists for the current position. As a consequence, they can all recursively search the principal variation in the same manner, and so will have the same value after the first move has been examined. For the remaining moves, the master informs an idle slave of the best score, and indicates which is the next subtree to search. In turn, the slave tells the master what value it found for the subtree it was searching. These **inter-node** messages are short, only 4 bytes in length, and are exchanged only during searches at type 1 nodes [KNUT75], that is, the nodes along the path of the first principal variation, Figure 8.

Inter-level communication occurs after the search of a type 1 node is completed. This message is required whenever a refutation table is implemented, in order to pass the best refutation line back to the master processor. The length of the message depends only on the length of the refutation line transferred.

When using the progressive deepening refinement, **inter-iteration** communication at the root node is also required. Between iterations, the move list is first resorted by the master and then passed on to the slaves. The master refutation table is also updated and subsequently distributed to the slave processors. A large amount of information transfer, proportional to the width of the search tree, must take place at this point, but this communication overhead can be tolerated since only a handful of iterations are usually performed.

If a global transposition table is used, all the processors must have access to this information and be able to update this large direct access hash table. In our current implementation, the transposition table can be managed by the master processor, with the slaves accessing the table via **intra-node** communication. Whenever a slave processor is about to execute *alphabeta* on a node within its assigned search tree, a 6 byte request message, which contains a hash key, is sent to the master to see if a table entry exists for that node. If the search is successful the master passes back 6 bytes consisting of the move found and its score, along

Figure 8. Structure of an alphabeta search tree.

with information describing the reliability of the score [MARS82]. With this information, the slave can narrow its search window or even avoid searching the subtree altogether. After completing its search, the slave transmits to the master a 12 byte update message consisting of the information just described. Unlike the other forms of communication, **intra-node** messages can be very frequent and lengthy. Moreover, the slave will be idle when it is waiting for a response to its request. As a result, it is usually prudent to suppress most of this communication and perhaps implement a transposition table local to the individual processor [POPO83]. Another possibility would be for the master to interrupt the slave only if it finds the position in the hash table. This would eliminate the need for the slave to wait for a response.

Finally, there is **inter-move** communication. This allows the startup of the individual processors, the setting of system configuration variables, and other forms of external communication. Input to the master processor is echoed to all of the slaves. **Inter-move** communication involves tasks such as updating the board configuration, obtaining the opponent's move, and even setting the maximum search depth. Typically, these are interactive messages to the user's console which do not affect the performance of the system.

The different types of communication, along with their frequency and length are summarized in decreasing order of frequency in Table 1.

Type	Frequency	Length
intra-node	before and after each node in the tree (optional)	6 or 12 bytes
inter-node	before and after each successor of a type 1 node	4 bytes
inter-level	after each type 1 node	2 bytes for each ply from leaf
inter-iteration	before and after each iteration	$4w + 2iw$ bytes, w is tree width at root, i is iteration number
inter-move	interactive, occur in response to user requests	varies

Table 1. Types of Inter-Processor Communication

5. System Configuration

The system used for our implementation of *pmws* is composed of four identical SUN Workstations [SUN82], which are each equipped with a Motorola 68000 microprocessor and 256K of random access memory, Figure 9. The supervisor processor, which behaves as both a master and a slave, possesses an extra 128K (at the present time) of memory to aid in transposition table storage and to facilitate usage of an opening book for the chess program. Preliminary simulations of PV Splitting have shown that, with a system of four processors, a processor tree of depth one is slightly superior to one of depth two [MARS82].

Communication between the processors is channeled though an eight-port Serial Communication Interface (SCI) [CDC81], as described in the Appendix. This device currently resides in the supervisor workstation where direct memory data transfer can occur between it and the SUN processor. Communication between the SCI and the other units is done over RS-232 data lines. Additional SCI units could be installed to allow processor trees of depth two and greater. Since the SCI takes care of the character by character transmission of data, the host supervisor processor need only write to, and read from, the appropriate buffers to communicate with the individual processors. For each processor, the SCI provides approximately 2K of buffer space (1024 bytes output, 992 bytes input). After checking a status bit, the host can fill the output buffer and then interrupt the SCI to tell it to transmit the data. The SCI can also interrupt the host whenever data is available and consequently allow the host to act as a master processor, when it is handling an interrupt, or as a slave when it is in normal operating mode.

Whereas the supervisor processor has its input buffered by the SCI, the bare SUN workstation can only buffer two characters. As a result, the supervisor must not be allowed to transmit to the slaves unless it can be sure that no data will be lost. One way that this problem is remedied is by having the slave transmit the Request-To-Send (RTS) signal when it desires input. The SCI will not transmit unless it receives this signal (Clear-To-Send). This extra protocol is not required though if the slave performs a read immediately after a write. Since the **inter-node** and **intra-node** communication, which accounts for most of the message passing, is of this type, there is no need to generate the RTS/CTS signal during these communication phases.

Figure 9. System Configuration.

Since all external input is processed by the supervisor processor, there must be facilities available for external access to the slave processors. This is handled by a network transparent mode. When the supervisor is loaded and subsequently started, transparent mode is entered allowing communication with any slave processor. The slaves can then be loaded and started after which they will wait until the supervisor has left transparent mode. When this is complete, regular **inter-move** communication can occur to setup the initial configuration.

6. Summary

Using this system with a local transposition table for five ply chess game trees, effective speedups of up to 1.89, 2.59 and 3.10 have been achieved with 2, 3, and 4 processors respectively. The elaboration of these results appears in a separate report [POPO83].

We have presented the outline of a multi-processor based system for use in minimax game tree searches. The use of memory tables has been examined along with the problems involved in their local and global implementations. The system development has also addressed the issues of processor management and inter-unit communication, which can be related to other parallel systems. Although we have designed our parallel processing system for a specific application, it is expressed in general terms so that the ideas employed may be suitable for any minimax tree search application.

AKL82 S. Akl, D. Barnard and R. Doran, "Design, Analysis, and Implementation of a Parallel Tree Search Algorithm", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* **PAMI-4.** 2 (1982), 192-203.

BAUD78 G. Baudet, "The Design and Analysis of Algorithms for Asynchronous Multiprocessors", Ph.D. thesis, Computer Science Dept., Carnegie-mellon Univ. Pittsburgh, April 1978.

CAMP81 M. Campbell, "Algorithms for the Parallel Search of Game Trees", M.Sc. thesis, TR81-8, Computer Science Dept., Univ. of Alberta, Edmonton, August 1981.

CAMP83 M. Campbell and T.A. Marsland, "A Comparison of Minimax Tree Search Algorithms", *Artificial Intelligence,* (to appear) 1983.

CDC81 Central Data Corporation, "Intelligent Serial Octal Serial Interface", (1981).

FISH80 J. Fishburn and R. Finkel, "Parallel Alpha-Beta Search on Arachne", TR 394, Computer Science Dept., Univ. Wisconsin, Madison, July 1980.

FISH81 J. Fishburn, "Analysis of Speedup in Distributed Algorithms", Ph.D. thesis, TR 421, Computer Science Dept., Univ. Wisconsin, Madison, May 1981.

GILL78 J. Gillogly, "Performance Analysis of the Technology Chess Program", Ph.D. dissertation, Computer Science Dept., Carnegie-mellon Univ., Pittsburgh, March 1978.

KNUT75 D. Knuth and R. Moore, "An Analysis of Alpha-Beta Pruning", *Artificial Intelligence* **6,** 293-326, (1975).

LIND83 G. Lindstrom, "The Key Node Method: A Highly-Parallel Alpha-Beta Algorithm", UUCS 83-101, Dept. of Computer Science, Univ. of Utah, Salt Lake City, March 1983.

MARS82 T.A. Marsland and M. Campbell, "Parallel Search of Strongly Ordered Game Trees", *Computing Surveys*, **14,** 4 (1982), 533-551.

MARS83 T.A. Marsland, "Relative Efficiency of Alpha-Beta Implementations", IJCAI Conference Proceedings, Karlsruhe, August 1983.

PEAR80 J. Pearl, "Asymptotic Properties of Minimax Trees and Game Searching Procedures", *Artificial Intelligence* **14** (1980), 113-138.

POPO83 F. Popowich and T.A. Marsland, "Parabelle: Experience with a Parallel Chess Program", TR 83-7, Computing Science Dept., Univ. of Alberta, Edmonton, August 1983.

SLAT77 D. Slate and L. Atkin, "CHESS, 4.5 — The Northwestern University Chess Program", In P.Frey(Ed.), *Chess Skill in Man and Machine*, chap 4. Springer Verlag, New York, 1977, pp.82-118.

SUN82 SUN Microsystems Inc., "SUN-1 System Reference Manual", July 1982.

The Intelligent Octal Serial Communication Interface is based on the Signetics 2650 (8 bit) microprocessor. It possesses 16K of dual port RAM, along with 1K of RAM, 4K of PROM and eight 2651 USARTs for the serial ports.

The original ROM driver program provided insufficient input buffer space (only 64 bytes), and resulted in a communication bottleneck. Originally, the SCI would give the host processor only one character at a time from the input buffer. As a result, the M68000 would be idle while the slower 2650 was getting the next character to be read. Furthermore, the 2650 would have to be interrupted each time the host desired a character, and this resulted in the loss of the lower priority USART interrupts. Consequently, data transmission rates had to be reduced to avoid losing characters, and large quantities of data had to be broken up into 64 byte chunks for transmission to the SCI.

Modification to the input portions of the driver program involved in an increased input buffer size of 992 bytes, a decreased output buffer size of 1024 bytes, the permission for USART interrupts to occur during the processing of host interrupts, and the capability for the host to read the buffer directly from the dual port RAM. A minor hardware change directing USART interrupts and host interrupts to different locations allowed a major streamlining of the interrupt handling routine. Each of the eight ports on the board is still allocated 2K of memory in the dual port RAM, with 2016 bytes being used as buffer space while the remaining 32 bytes are reserved for host-SCI communication. All data associated with any particular port is stored in its associated memory block which is divided in the following manner:

OFFSET	LENGTH	DESCRIPTION
0x000	1024	Output Buffer (to device)
0x400	992	Input Buffer (from device)
0x7E0	1	Status Byte
0x7E1	3	Unused
0x7E4	2	Offset of Byte After Last Character (output)
0x7E6	2	Mode Control Bytes
0x7E8	1	FF = Mode Setup Command
0x7E9	1	Unused
0x7EA	2	Number of Characters Available in Input Buffer
0x7EC	2	Displacement of first character to be read in Input Buffer
0x7EE	2	Number of Characters read by host
0x7F0	16	Unused - reserved for program expansion

The output buffer is where the host computer places the data to be sent to the device, while the input buffer is a wrap-around buffer containing the characters received from the serial port. Only the most recent 992 bytes received from a device are kept. The status byte has its least significant bit set if the output buffer is empty, and the next most significant bit set if there is at least one character available in the input buffer.

Output to a particular port is placed in the appropriate buffer by the host (after checking to see if the low bit of the status byte is set) and then the number of characters entered is placed in the offset bytes. To read from the input buffer, the address of the first character to be read is calculated by adding the displacement to the address of the appropriate input buffer. The host then places the number of characters to be read into its two byte location. The host can then read these characters from the buffer. The SCI will update the number of bytes available and the displacement.

There is one I/O port on the board which is used to tell the 2650 processor that either the host processor has filled the output buffer, or that it has read a specified number of characters from the input buffer. This I/O port's data bits are used in the following manner:

BIT	FUNCTION
7	1=Input, 0=Output
6-4	Port Number(0-7)
3	1=Interrupts On, 0=Off
2-0	Interrupt Level(0-7)

If the host sends the SCI a command when offset 7E8 of the corresponding memory block is set to FF, the 2650 will use the mode control bytes from that block to set up the USART for the specified baud rate, character length, number of stop bits, and parity. Otherwise, the 2650 will interpret the data as a input or output command, depending on the value of bit 7.

An input command tells the SCI that the host has placed the number of characters that it has read at offset 7EE, and that the 2650 can update the number of bytes available and the displacement of the first character in the input buffer. If there are no more characters available to be read, then bit 1 of the status byte is cleared. If the interrupt bit is set, then an interrupt will be generated on the specified Multibus vectored interrupt line whenever bit 1 of the status byte is set (ie. when the first character arrives to an empty input buffer), or if the buffer is not empty after a buffer read. This interrupt can be initialized by doing a read of zero bytes from the input buffer.

An output command word tells the 2650 that the output buffer has been loaded with the number of characters specified in the offset count. The 2650 will then start sending the characters to the appropriate port and set the low bit of the status byte when this task is completed, and interrupt the host if an interrupt was requested by the host.

This communication protocol can be transparent to the programmer through the use of the communication support software. The *readx* routine is used to read a specified number of characters from a given I/O port. If zero is given as the number of bytes requested, all the buffered data are transferred, with the actual number read being returned by the function. This function can also be used to enable or disable input interrupts. *Ischarx* determines if a character is available from the device specified as an argument. This routine is useful when operating in polling mode rather than interrupt mode. The *testx* routine augments these functions by sampling data from the input buffer, rather than reading it as the *readx* function does. There is also the *writex* function which allows a specified number of bytes to be transmitted to a specified port.

As a result of the system modifications, communication rates have been increased (9600 baud for up to three processors, 4800 with four processors) with less idle time for both the master and slave processors. The host processor can now read input data directly from the input buffer, and the decomposition of large blocks into small data chunks is no longer required.