Hypercube Multiprocessors 1987

EDITED BY
Michael T. Heath
Oak Ridge National Laboratory

siam
Philadelphia 1987

A High Performance Operating System for the NCUBE[*]

T. N. MUDGE[†], G. D. BUZZARD[†] AND T. S. ABDEL-RAHMAN[†]

Abstract. This paper examines the existing hypercube array processing node operating system for the NCUBE. Extensions to the operating system that lead to performance increases for important classes of algorithms are described. Performance results are given and are compared to those of other systems.

1 Introduction and Motivation

NCUBE produces a family of commercial hypercube multiprocessors that range in size from 64 processors (NCUBE/six) to 1024 processors (NCUBE/ten). The hypercube array of processing nodes is managed by a host computer (an Intel 80286 based system). Software development for the hypercube is carried out on the host using a multiuser Unix-like operating system called Axis. It allows the hypercube array to be partitioned into subcubes that may be allocated to different users. These subcubes are logically independent of each other and the processing nodes in each subcube are numbered logically beginning at zero. In addition, there is also a run-time executive program called Vertex that executes on the processors of the hypercube array. This paper discusses the design of high performance extensions to Vertex.

The array processors are VAX-class 32-bit microprocessors with IEEE standard floating point capability. Each node has 128 K-bytes of local memory and 22 high speed (1 M-byte per second) unidirectional DMA channels. The channels are paired to provide 11 bidirectional links. Ten of these links can be connected to neighboring processors in the hypercube, thus allowing for systems of 1024 processors. The eleventh link connects to the

[*] NCUBE Corporation, Tempe, Arizona
[†] Advanced Computer Architecture Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan.

I/O subsystem which is also connected to the host. More architectural details can be found in [1].

Algorithms that have been proposed for or implemented on hypercube multiprocessors can be classified by their primary mode of communication. In particular, there are three common modes of communication: nearest neighbor, broadcast, and random access. Nearest neighbor (NN) communications are between adjacent processors in the hypercube array. Broadcast (BC) communications originate from a single source and are sent to the remaining processors. Finally, random access (RA) communications, as the name implies, may originate or terminate at any of the processors. Clearly RA communications are the most general. NN communications are subsumed by RA, and BC communications can be achieved by multiple RA communications.

The present version of Vertex implements only RA communications. While this is clearly sufficient, performance of many algorithms can be greatly enhanced by employing a broad set of more specialized communication primitives. In this paper we discuss our initial investigative efforts in this area.

The remainder of this paper is organized as follows. Section 2 contains a concise description of the Vertex run-time executive program. The points of inefficiency inherent in the standard Vertex communication scheme are pointed out in Section 3. Our high performance extensions to Vertex are described in Section 4. Section 5 describes the experiments that we have performed and discusses their results. Finally, concluding comments are given in Section 6.

2 Explanation of Vertex

Vertex is a run-time executive program that executes on the NCUBE array processors. The services provided by Vertex are user program loading, low-level error handling, low-level debugger support, node identification and time call handlers, and communications support that includes message buffer handling. Vertex is compact, requiring only about 5 K-bytes of memory for both code and data (excluding the communication buffers). Entry to Vertex from high level language libraries and user written assembly language routines is via an operating system trap call that saves the current program status word and program counter and branches into Vertex code via an interrupt jump table. A similar mechanism may also be invoked by the hardware in response to an execution exception, or an external interrupt request.

User programs may query Vertex to learn the logical (with respect to the currently allocated sub-cube) node address on which they are executing, the host interface processor address, and currently allocated sub-cube dimension. They may also request the current processor time, which is kept in multiples of 1024 clock cycles since node initialization. These two call handlers, and those of the communication system that will be discussed later, comprise the primary operating system trap call services.

The low-level error handlers, which are invoked by hardware recognized program exceptions, save processor state information and suspend execution of the user process. This allows the low-level debugger, invoked by communication system interrupt handler upon receiving a debugger message, to examine the state of the node as it was when the exception was detected. The debugger then allows the examination and setting of registers and memory, and the setting of breakpoints on any allocated node.

The memory map of an array node divides the 128 K-bytes into four areas: the interrupt jump table; user code and data space; Vertex code and data space; and the communication buffer pool. The interrupt jump table occupies a fixed 2 K-byte region of memory beginning at address zero. The Vertex code and data space occupies about 5 K-bytes of memory. It is statically relocatable and lies between the user code and data space and the communication buffers. Typically about 27 K-bytes of memory is reserved for communications buffers, leaving the remaining 95 K-bytes for the user code and data.

Vertex communications are driven by two events: requests from the user program, via operating system traps; and responses to communication channel interrupts. A brief description of the three calls that comprise the interface to the user program is given here, a more detailed discussion appears later. The node write call is named *nwrite*. It sends a message of a specified type to a designated destination. The *nread* call examines the unclaimed received message queue for a message from the specified source of the desired type. When the desired message is found, it is returned to the caller. The *ntest* call performs a function similar to nread, except that it returns immediately after checking the existing messages and only reports on the success of the search — located messages are not returned.

A pool of communication buffers is maintained by Vertex. These buffers are used on both the sending and receiving nodes. They allow callers of nwrite to be released after a communication buffer is allocated and the user data is copied into it. Therefore, callers are not held up while the system is waiting for access to a communication channel. Furthermore, if the data to be sent is allocated from dynamic stack-based storage, the caller may release that storage immediately after returning from the nwrite call. Similarly, callers of nread are not required to make their calls before messages arrive since all arriving messages are first stored in a communication buffer.

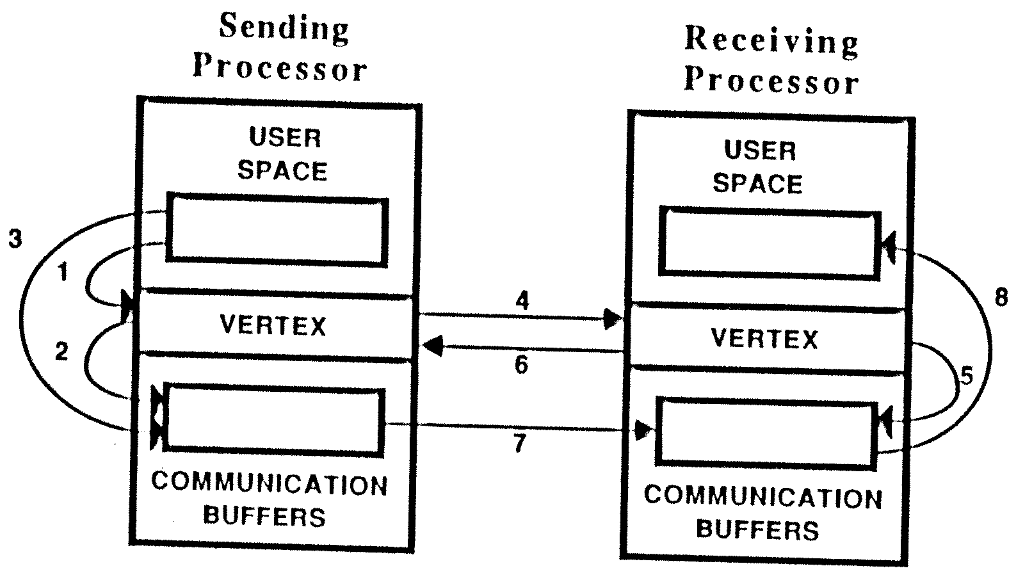
The communication buffer pool begins as a single free buffer. Requests for buffers are satisfied by searching linearly through the doubly linked buffer pool for free buffers of a sufficiently large size. When one is located it is further checked to see if it exceeds the requested size by 32 or more bytes; if so, the free buffer is split so that a buffer of the requested size can be allocated from the end furthest from the front of the list. This policy tends to concentrate free buffers near the beginning of the list. If no buffers of sufficient size are located the request may be queued to search again when another buffer is returned. In addition to attempting to satisfy queued buffer requests, the buffer deallocation routine also collapses adjacent free buffers into a single buffer. Buffers that are in use may be queued on other system queues via a second pair of link fields. Thus, for its own internal use Vertex provides buffer allocate, deallocate, enqueue, and dequeue procedures.

The specific operation of nwrite is to check for valid message length and destination, allocate a communication buffer and copy the user data to it, calculate the outbound channel to use, and, if the channel is inactive, initiate the message transfer protocol. In the case that the channel is busy, the message is queued to be sent as soon as the channel becomes available. In either case, the caller is released to continue execution. When the final step of the DMA controlled message transfer has been completed an interrupt is signalled. The service of this interrupt includes the releasing of the communication buffer. Incoming messages are placed in communication buffers and then queued in an unclaimed message list. Nread checks this queue for a message with a specific source and type, if it is not found it will wait until an appropriate message arrives. When the requested message is located it will be copied to the user data space, dequeued from the unclaimed message list, and the communication buffer will be released. Callers to nread and ntest may specify that messages of any incoming source or type are desired.

Message transmission is controlled via a three way handshake protocol. Simply stated, the sender sends a two byte message to the receiver indicating the number of bytes it wishes to transmit (all inactive channels are set to receive two byte transmissions). The receiver allocates a communication buffer, sets the appropriate DMA channel to receive the indicated number of bytes, and then transmits a two byte acknowledgement back to the sender. Upon receiving this acknowledgement the sender transmits the entire message. This protocol is initiated by a call to nwrite. However, the remaining steps are handled by a sequence of interrupt events. Specifically, the reception of the two byte length message by the receiver and the reception of the two byte acknowledgement by the sender both signal interrupts. An interrupt is also generated on both the sending and receiving nodes when the DMA transfer of the message is completed.

A more detailed description of a message send/receive transaction between two adjacent nodes is given below. Figure 1 expresses this transaction pictorially. In step 1 the sender issues an nwrite call, which generates an operating system trap event. In step 2 the Vertex nwrite trap handler allocates a communication buffer, waiting on a buffer request queue if none are presently available. Once allocated (step 3), the message data is copied from the user process to the communication buffer and the buffer is queued on the send list. The nwrite caller is released at this point. In step 4 the two byte transmission request, indicating message length, is sent. If the channel to the requested destination is not busy this action occurs as the final action of step 3. Otherwise, this transmission request is queued pending an end of message transfer interrupt from the DMA for the message currently using the channel. Reception of the two byte transmission request generates an interrupt on the receiving node. The handler for this interrupt (step 5) attempts to allocate a communication buffer of the requested length. If it is successful, an acknowledgement message is sent back to the sender (step 6). If a buffer is unavailable a buffer request is queued and the acknowledgement message is postponed until after this request is satisfied. It may also be necessary to wait for the channel from the receiver to the sender to become free during step 6 (if a previous message send is still in progress). The reception of the two byte acknowledgement generates an interrupt for the sender. In this interrupt handler (step 7) the DMA transfer of the message is begun. Upon completion of this DMA transfer an interrupt is generated for both the sender and the receiver. On the sender the communication buffer is dequeued from the send queue and released. If another message is ready to be sent on the same channel this procedure is repeated from step 4, otherwise the channel is reset to the inactive state. On the receiver the message is checked to see if this node is the final destination and if this message is a user message. If so, the communication buffer is queued on the incoming unclaimed message list and the channel is reset to its inactive state. If an nread call has been issued for the incoming message, it is dequeued from the incoming unclaimed message list, copied to the user data space, and the communication buffer is released, this corresponds to step 8.

Multi-hop messages are handled in a manner very similar to the above. The following actions, beginning with step 8, occur on each of the intermediate nodes. The receiver



Actions:

1.	Sender:	nwrite call
2.	Sender:	allocate communication buffer
3.	Sender:	copy buffer, release nwrite caller
4.	Sender:	transmission request
5.	Receiver:	allocate communication buffer
6.	Receiver:	transmission request accepted
7.	Sender:	transmit data
8.	Receiver:	if received pending, copy buffer release nread caller

Interrupts:

after 4 on receiver
after 6 on sender
after 7 on sender and receiver

Potential Waits:

at 2 for local buffer
at 4 for use of channel
at 5 for remote buffer and use of channel

Figure 1: Node to Node Communication.

interrupt handler inspects the destination field of the message; calculates the next channel to send it out on in order to get it another step closer to its final destination; queues the communication buffer on the send list; and, if the channel is not busy, sends the two byte transmission request message thus assuming the role of the sender at step 4. This procedure is repeated on each of the intermediate nodes.

System messages (primarily for interactions with the low-level debugger) are also handled in a very similar manner. On the destination receiving node at step 8 the message is checked to see if it is a system message, if it is the appropriate system message handler is invoked.

3 Points of Inefficiency

Two major points of inefficiency can be identified within the communication scheme implemented in Vertex with respect to NN communications. One, is the use of buffer copies, the second is the overhead incurred by the three-way handshake protocol. In many cases, algorithms that fall into the NN classification have *a priori* knowledge of all of their communication requirements; presently this knowledge is unused. With simple extensions to Vertex this knowledge can be exploited to yield much more efficient communications.

While the cost incurred by copying data is obvious, the cost of requiring a more complex communication protocol than is necessary is much harder to quantify. This cost can be traced to two major sources. One, is the overhead incurred by servicing interrupts generated by unnecessary protocol messages. The second, is the loss of the use of the channel from the receiving node to the sending node for the duration of time between the initial transmission length message arriving at the receiver (step 4 in Figure 1) and the acknowledgement arriving back at the sender (step 6 in Figure 1). Notice that this duration may be arbitrarily long if the receiving node does not have sufficient communication buffer space immediately available. These effects are reflected in the experimental results that are be presented in Section 5.

Random communications do not incur the buffer copy overhead at intermediate nodes. However, they do have the same protocol overhead on every node that the message traverses. They also incur the additional cost of storing the entire message at each intermediate node before beginning to forward it to the next node.

BC communications are not directly supported in vertex. Broadcasts are typically implemented by a sequence of NN communications arranged in a spanning tree order [2]. Each node (except the root) receives a message from its parent node, then serially relays the message to each of its child nodes. The root node begins the operation by serially sending the message to each of its child nodes. Thus, in addition to the inefficiencies already noted for near neighbor communications, broadcasts also incur a store and forward (with two buffer copies) overhead and a serialization cost associated with sending to only one child node at a time on each of the intermediate (i.e., non-terminal) nodes in the spanning tree. Serialization is unnecessary as a broadcast instruction is implemented in the instruction set of the node processors. The broadcast instruction allows a message to be sent on any subset of the output channels of a node in a single DMA action.

4 Extensions to Vertex

The Vertex operating system has been extended to address both of the major inefficiencies present in NN communications. The new send call, named *send*, and new receive call, named *rcvreq*, do not use communication buffers and are asynchronous. Since return from the send or rcvreq call does not indicate their completion, the caller passes a pointer to a flag variable which is set for this purpose. The indication of completion is signalled by an end-of-DMA interrupt for both send and rcvreq. This protocol places a burden on the caller of send to ensure that the message data is not corrupted before the completion flag is set. It also requires that the receiver make a call to rcvreq before the anticipated message arrives. The latter constraint could be relaxed by allocating a communication buffer if the call to rcvreq has not yet occurred. However, to do so would lead to incompatibilities with the next extension to be discussed. The primary benefit of bufferless communication is that the incremental (i.e., per byte) cost is substantially reduced.

The send and rcvreq call also exploit the *a priori* knowledge that is inherent in most NN class programs by eliminating the three-way handshake protocol. This extension places the additional requirement on the caller of rcvreq to specify the exact number of bytes expected to be received. The chief benefit of this change is a substantial reduction in the fixed overhead of NN communications.

There are three other calls that are presently in available in extended Vertex. Two routines have been added to allow the communication scheme to be switched dynamically. In fact, the communication scheme in use can be selected on a channel by channel basis. These calls are necessary because the extended Vertex communication scheme relies upon different interrupt handlers than standard Vertex. The final extension is a broadcast call that allows a node to send the same message to any set of nearest neighbors simultaneously.

5 Experiments and Results

For programs with simple NN communications we expect the time added by the communications to be approximated by the formula $t_{comm} = mx + f$, where t_{comm} is the total message communication time, m is the length of the message in bytes, x is the incremental (per byte) message cost, and f is the fixed overhead (protocol) cost. Of course, this formula will only hold for simple communications. It does not take into account the positive effect of overlapping message transmission with computation, nor the negative effect of waiting on resources (e.g., communication buffers or channels). Nonetheless, the parameters of this formula are frequently quoted as they do provide a partial indication of expected performance and they are easily measured.

A common technique for measuring the aforementioned parameters is to configure the array nodes into a logical ring, then record the amount of time required to pass a fixed size message around the ring a specified number of times. When this experiment is repeated for messages of differing sizes the parameters m and f can be easily extracted. The results of such an experiment using both standard and extended Vertex communication primitives are given in Table 1. It can be seen that the use of extended Vertex primitives yielded an improvement of 5.6 in fixed overhead and 2.43 in incremental cost over standard Vertex.

	Standard Vertex	Extended Vertex	Speedup
Fixed Overhead	466.40 (μsec/message)	83.20 (μsec/message)	5.60
Incremental Cost	3.14 (μsec/byte)	1.29 (μsec/byte)	2.43

Table 1: Ring Message Results.

Number	NCUBE		iPSC		MarkII(8Mhz)
of bytes	Std. Vtx.	Ext. Vtx.	CrOS	IHOS	CrOS
8	245.76	46.76	160	5960	86.0
64	41.79	10.37	80	777	45.5
256	19.88	6.45	79	202	41.4

Table 2: Values of t'_{comm} (μsec) for the NCUBE, iPSC and MarkII.

Another frequently quoted communication performance parameter is given in [3]. It is also called t_{comm}, but we will henceforth refer to it as t'_{comm} to avoid confusion with our earlier definition. The value of t'_{comm} is conventionally defined as the transfer time of a 32 bit word, and is often given for a range of total message sizes. It can easily be related to t_{comm} by $t'_{comm} = \frac{4t_{comm}(m)}{m}$, where m is the length of the message in bytes. To facilitate comparison with other systems, values of t'_{comm} are given in Table 2. The values for the Intel iPSC and Caltech MarkII were taken from [3]. The iPSC times are reported for both the Intel Hypercube Operating System (IHOS) and the Caltech Crystalline Operating System (CrOS). The values stated for the NCUBE were measured directly, though they could have been derived from the information given in Table 1. As we would expect, the improvement of extended Vertex over standard Vertex ranges from the speedup for fixed overhead towards the speedup for incremental costs as the total message size increases.

The effects of computational overlap and resource contention, mentioned earlier, are often factored into the formula for overall execution time. This formula is given by,

$$T_{exec} = T_{calc} + (1 - \gamma)T_{comm}$$

where $\gamma \in (-\infty, 1]$ is the degree of communication transparency, T_{calc} is the total computation time of the algorithm, and $T_{comm} = \sum t_{comm}$ is the total communication time neglecting the effects of computational overlap and resource contention [4]. When $\gamma = 1$, the effect of T_{comm} is completely hidden by computational overlap. Conversely, negative effects of waiting on resources are expressed by $\gamma < 0$. The parameter γ is very highly dependent on both the communication structure of a given algorithm and the communication support provided by the system.

As a trial application, both sets of communication primitives were used on a common image processing algorithm, the Sobel edge detector. The parallel Sobel algorithm divides an image into a grid of equal size subimages which are distributed on the array nodes (the assignment technique is described in [4]). A 3×3 pixel window operator is convolved with each of the pixels in the subimage. This requires that the pixel values for the 8 neighboring

image size	subimage size	Ext. Vertex speedup over Std. Vertex		Fraction of total execution time spent calculating		Total execution speedup over single node algorithm	
		total	comm.	Ext. Vtx.	Std. Vtx.	Ext. Vtx.	Std. Vtx.
128 × 128	32 × 32	1.02	3.83	0.99	0.97	15.9	15.6
64 × 64	16 × 16	1.08	5.24	0.98	0.91	16.0	14.8
32 × 32	8 × 8	1.34	9.79	0.96	0.72	13.7	10.2

Table 3: Sobel Edge Detector Experiment (16 processors).

pixels (N, S, E, W, NE, NW, SE, and SW) be available. Thus, when calculating the values for pixels on the edge of a subimage, the calculating processor must use pixel values from the edge of the neighboring subimage. A similar requirement exists for the corners. The possibility for computation/communication overlap exists since the interior of the subimage may be processed while awaiting reception of the data needed for the border calculations. Also, the potential for a large number of messages to be active in the system simultaneously leads to some resource contention.

The value of γ is given for both standard Vertex and extended Vertex. The results of the Sobel edge detector experiment are given in Table 3. The advantages of prescheduling resources and extracting as much computation/communications overlap as possible are evidenced by the large speedups achieved with the extended Vertex primitives. For a small subimage size the total program execution speedup was 1.34. As expected, the improvement increases as the communications account for a larger portion of total processing effort.

6 Conclusion

The results that we have produced so far have been encouraging. As shown in Table 3, the use of extended Vertex communication primitives allows the array processors to spend a much higher fraction of their total program execution time calculating results, rather than communicating or waiting for resources. To date, these results have been achieved only for algorithms where the communication mode is nearest neighbor. Work is presently underway to achieve the same type of results for algorithms where the primary communication mode is broadcast or random access.

References

[1] John P. Hayes, Trevor N. Mudge, Quentin F. Stout, Steve Colley, and John Palmer. A microprocessor-based hypercube supercomputer. *IEEE MICRO*, :6–17, October 1986.

[2] Joseph E. Brandenburg and David S. Scott. *Embeddings of Communication Trees and Grids into Hypercubes.* Technical Report 1, Intel Scientific Computers, 15201 N.W. Greenbrier Parkway, Beaverton, OR 97006, 1986.

[3] Geoffrey C. Fox and A. Kolawa. Implementation of the high performance crystalline operating system on Intel iPSC hypercube. In Michael T. Heath, editor, *Hypercube Multiprocessors 1986*, pages 269–271, SIAM, Society for Industrial and Applied Mathematics, 1400 Architects Bldg., 117 South 17th Street, Philadelphia, PA 19103, August 1985.

[4] Trevor N. Mudge and Tarek S. Abdel-Rahman. Vision algorithms for hypercube machines. *Journal of Parallel and Distributed Computing*, 4(2), March 1987 (to appear).